Groovy Language Documentation

Version 6.0.0

Introduction

Groovy…

- is an agile and dynamic language for the Java Virtual Machine
- builds upon the strengths of Java but has additional power features inspired by languages like Python, Ruby and Smalltalk
- makes modern programming features available to Java developers with almost-zero learning curve
- provides the ability to statically type check and statically compile your code for robustness and performance
- supports Domain-Specific Languages and other compact syntax so your code becomes easy to read and maintain
- makes writing shell and build scripts easy with its powerful processing primitives, OO abilities and an Ant DSL
- increases developer productivity by reducing scaffolding code when developing web, GUI, database or console applications
- simplifies testing by supporting unit testing and mocking out-of-the-box
- seamlessly integrates with all existing Java classes and libraries
- compiles straight to Java bytecode so you can use it anywhere you can use Java

Groovy Language Specification

Syntax

This chapter covers the syntax of the Groovy programming language. The grammar of the language derives from the Java grammar, but enhances it with specific constructs for Groovy, and allows certain simplifications.

Comments

Single-line comment

Single-line comments start with `//` and can be found at any position in the line. The characters following `//`, until the end of the line, are considered part of the comment.

```
// a standalone single line comment
println "hello" // a comment till the end of the line
```

Multiline comment

A multiline comment starts with `/*` and can be found at any position in the line. The characters following `/*` will be considered part of the comment, including new line characters, up to the first `*/` closing the comment. Multiline comments can thus be put at the end of a statement, or even inside a statement.

```
/* a standalone multiline comment
   spanning two lines */
println "hello" /* a multiline comment starting
                  at the end of a statement */
println 1 /* one */ + 2 /* two */
```

Groovydoc comment

Similarly to multiline comments, Groovydoc comments are multiline, but start with `/**` and end with `*/`. Lines following the first Groovydoc comment line can optionally start with a star `*`. Those comments are associated with:

- type definitions (classes, interfaces, enums, annotations),
- fields and properties definitions
- methods definitions

Although the compiler will not complain about Groovydoc comments not being associated with the above language elements, you should prepend those constructs with the comment right before it.

```
/**
```

```
 * A Class description
 */
class Person {
    /** the name of the person */
    String name

    /**
     * Creates a greeting method for a certain person.
     *
     * @param otherPerson the person to greet
     * @return a greeting message
     */
    String greet(String otherPerson) {
       "Hello ${otherPerson}"
    }
}
```

Groovydoc follows the same conventions as Java's own Javadoc. So you'll be able to use the same tags as with Javadoc.

In addition, Groovy supports **Runtime Groovydoc** since 3.0.0, i.e. Groovydoc can be retained at runtime.

NOTE	Runtime Groovydoc is disabled by default. It can be enabled by adding JVM option `-Dgroovy.attach.runtime.groovydoc=true`

The Runtime Groovydoc starts with `/**@` and ends with `*/`, for example:

```
/**@
 * Some class groovydoc for Foo
 */
class Foo {
    /**@
     * Some method groovydoc for bar
     */
    void bar() {
    }
}

assert Foo.class.groovydoc.content.contains('Some class groovydoc for Foo') ①
assert Foo.class.getMethod('bar', new Class[0]).groovydoc.content.contains('Some
method groovydoc for bar') ②
```

① Get the runtime groovydoc for class `Foo`

② Get the runtime groovydoc for method `bar`

Shebang line

Beside the single-line comment, there is a special line comment, often called the *shebang* line

understood by UNIX systems which allows scripts to be run directly from the command-line, provided you have installed the Groovy distribution and the `groovy` command is available on the `PATH`.

```
#!/usr/bin/env groovy
println "Hello from the shebang line"
```

NOTE	The `#` character must be the first character of the file. Any indentation would yield a compilation error.

Keywords

Groovy has the following reserved keywords:

Table 1. Reserved Keywords

abstract	assert	break	case
catch	class	const	continue
def	default	do	else
enum	extends	final	finally
for	goto	if	implements
import	instanceof	interface	native
new	null	non-sealed	package
public	protected	private	return
static	strictfp	super	switch
synchronized	this	threadsafe	throw
throws	transient	try	while

Of these, `const`, `goto`, `strictfp`, and `threadsafe` are not currently in use. Using `const`, `goto`, or `threadsafe` is a compilation error: the diagnostic names the reserved keyword and, for `const`, points at `val` (locals) or `static final` (class constants). A `default:` or `default →` clause outside a `switch` is reported as `'default' outside of switch`. Unexpected Unicode spaces (for example a no-break space copied from a word processor) are named with a `\u` escape rather than an empty-looking glyph.

The reserved keywords can't in general be used for variable, field and method names.

A trick allows methods to be defined having the same name as a keyword by surrounding the name in quotes as shown in the following example:

```
// reserved keywords can be used for method names if quoted
def "abstract"() { true }
// when calling such methods, the name must be qualified using "this."
```

```
this.abstract()
```

Using such names might be confusing and is often best to avoid. The trick is primarily intended to enable certain Java integration scenarios and certain DSL scenarios where having "verbs" and "nouns" with the same name as keywords may be desirable.

In addition, Groovy has the following contextual keywords:

Table 2. Contextual Keywords

as	in	permits	record
sealed	trait	var	yields

These words are only keywords in certain contexts and can be more freely used in some places, in particular for variables, fields and method names.

This extra lenience allows using method or variable names that were not keywords in earlier versions of Groovy or are not keywords in Java. Examples are shown here:

```
// contextual keywords can be used for field and variable names
def as = true
assert as

// contextual keywords can be used for method names
def in() { true }
// when calling such methods, the name only needs to be qualified using "this."
in scenarios which would be ambiguous
this.in()
```

Groovy programmers familiar with these contextual keywords may still wish to avoid using those names unless there is a good reason to use such a name.

The restrictions on reserved keywords also apply for the primitive types, the boolean literals and the null literal (all of which are discussed later):

Table 3. Other reserved words

null	true	false	boolean
char	byte	short	int
long	float	double	

While not recommended, the same trick as for reserved keywords can be used:

```
def "null"() { true }  // not recommended; potentially confusing
```

```
assert this.null()     // must be qualified
```

Using such words as method names is potentially confusing and is often best to avoid, however, it might be useful for certain kinds of DSLs.

Identifiers

Normal identifiers

Identifiers start with a letter, a dollar or an underscore. They cannot start with a number.

A letter can be in the following ranges:

- 'a' to 'z' (lowercase ascii letter)
- 'A' to 'Z' (uppercase ascii letter)
- '\u00C0' to '\u00D6'
- '\u00D8' to '\u00F6'
- '\u00F8' to '\u00FF'
- '\u0100' to '\uFFFE'

Then following characters can contain letters and numbers.

Here are a few examples of valid identifiers (here, variable names):

```
def name
def item3
def with_underscore
def $dollarStart
```

But the following ones are invalid identifiers:

```
def 3tier
def a+b
def a#b
```

All keywords are also valid identifiers when following a dot:

```
foo.as
foo.assert
foo.break
foo.case
foo.catch
```

Quoted identifiers

Quoted identifiers appear after the dot of a dotted expression. For instance, the `name` part of the `person.name` expression can be quoted with `person."name"` or `person.'name'`. This is particularly interesting when certain identifiers contain illegal characters that are forbidden by the Java Language Specification, but which are allowed by Groovy when quoted. For example, characters like a dash, a space, an exclamation mark, etc.

```
def map = [:]

map."an identifier with a space and double quotes" = "ALLOWED"
map.'with-dash-signs-and-single-quotes' = "ALLOWED"

assert map."an identifier with a space and double quotes" == "ALLOWED"
assert map.'with-dash-signs-and-single-quotes' == "ALLOWED"
```

As we shall see in the following section on strings, Groovy provides different string literals. All kind of strings are actually allowed after the dot:

```
map.'single quote'
map."double quote"
map.'''triple single quote'''
map."""triple double quote"""
map./slashy string/
map.$/dollar slashy string/$
```

There's a difference between plain character strings and Groovy's GStrings (interpolated strings), as in that the latter case, the interpolated values are inserted in the final string for evaluating the whole identifier:

```
def firstname = "Homer"
map."Simpson-${firstname}" = "Homer Simpson"

assert map.'Simpson-Homer' == "Homer Simpson"
```

Strings

Text literals are represented in the form of chain of characters called strings. Groovy lets you instantiate `java.lang.String` objects, as well as GStrings (`groovy.lang.GString`) which are also called *interpolated strings* in other programming languages.

Single-quoted string

Single-quoted strings are a series of characters surrounded by single quotes:

```
'a single-quoted string'
```

NOTE	Single-quoted strings are plain `java.lang.String` and don't support interpolation.

String concatenation

All the Groovy strings can be concatenated with the `+` operator:

```
assert 'ab' == 'a' + 'b'
```

Triple-single-quoted string

Triple-single-quoted strings are a series of characters surrounded by triplets of single quotes:

```
'''a triple-single-quoted string'''
```

NOTE	Triple-single-quoted strings are plain `java.lang.String` and don't support interpolation.

Triple-single-quoted strings may span multiple lines. The content of the string can cross line boundaries without the need to split the string in several pieces and without concatenation or newline escape characters:

```
def aMultilineString = '''line one
line two
line three'''
```

If your code is indented, for example in the body of the method of a class, your string will contain the whitespace of the indentation. The Groovy Development Kit contains methods for stripping out the indentation with the `String#stripIndent()` method, and with the `String#stripMargin()` method that takes a delimiter character to identify the text to remove from the beginning of a string.

When creating a string as follows:

```
def startingAndEndingWithANewline = '''
line one
line two
line three
'''
```

You will notice that the resulting string contains a newline character as first character. It is possible to strip that character by escaping the newline with a backslash:

```
def strippedFirstNewline = '''\
line one
line two
```

```
line three
'''

assert !strippedFirstNewline.startsWith('\n')
```

Escaping special characters

You can escape single quotes with the backslash character to avoid terminating the string literal:

```
'an escaped single quote: \' needs a backslash'
```

And you can escape the escape character itself with a double backslash:

```
'an escaped escape character: \\ needs a double backslash'
```

Some special characters also use the backslash as escape character:

Escape sequence	Character
\b	backspace
\f	formfeed
\n	newline
\r	carriage return
\s	single space
\t	tabulation
\\	backslash
\'	single quote within a single-quoted string (and optional for triple-single-quoted and double-quoted strings)
\"	double quote within a double-quoted string (and optional for triple-double-quoted and single-quoted strings)

An unknown letter escape such as `\U` in `"C:\Users"` is a compilation error (`Illegal escape character`). Write a literal backslash as `\\`, or use a slashy / dollar-slashy string for Windows paths and regexes.

We'll see some more escaping details when it comes to other types of strings discussed later.

Unicode escape sequence

For characters that are not present on your keyboard, you can use unicode escape sequences: a backslash, followed by 'u', then 4 hexadecimal digits.

For example, the Euro currency symbol can be represented with:

```
'The Euro currency symbol: \u20AC'
```

Double-quoted string

Double-quoted strings are a series of characters surrounded by double quotes:

```
"a double-quoted string"
```

NOTE	Double-quoted strings are plain `java.lang.String` if there's no interpolated expression, but are `groovy.lang.GString` instances if interpolation is present.

NOTE	To escape a double quote, you can use the backslash character: "A double quote: \"".

String interpolation

Any Groovy expression can be interpolated in all string literals, apart from single and triple-single-quoted strings. Interpolation is the act of replacing a placeholder in the string with its value upon evaluation of the string. The placeholder expressions are surrounded by `${}`. The curly braces may be omitted for unambiguous dotted expressions, i.e. we can use just a $ prefix in those cases. If the GString is ever passed to a method taking a String, the expression value inside the placeholder is evaluated to its string representation (by calling `toString()` on that expression) and the resulting String is passed to the method.

Here, we have a string with a placeholder referencing a local variable:

```
def name = 'Guillaume' // a plain string
def greeting = "Hello ${name}"

assert greeting.toString() == 'Hello Guillaume'
```

Any Groovy expression is valid, as we can see in this example with an arithmetic expression:

```
def sum = "The sum of 2 and 3 equals ${2 + 3}"
assert sum.toString() == 'The sum of 2 and 3 equals 5'
```

NOTE	Not only are expressions allowed in between the `${}` placeholder, but so are statements. However, a statement's value is just `null`. So if several statements are inserted in that placeholder, the last one should somehow return a meaningful value to be inserted. For instance, "The sum of 1 and 2 is equal to ${def a = 1; def b = 2; a + b}" is supported and works as expected but a good practice is usually to stick to simple expressions inside GString placeholders.

In addition to `${}` placeholders, we can also use a lone `$` sign prefixing a dotted expression:

```
def person = [name: 'Guillaume', age: 36]
assert "$person.name is $person.age years old" == 'Guillaume is 36 years old'
```

But only dotted expressions of the form `a.b`, `a.b.c`, etc, are valid. Expressions containing parentheses like method calls, curly braces for closures, dots which aren't part of a property expression or arithmetic operators would be invalid. Given the following variable definition of a number:

```
def number = 3.14
```

The following statement will throw a `groovy.lang.MissingPropertyException` because Groovy believes you're trying to access the `toString` property of that number, which doesn't exist:

```
shouldFail(MissingPropertyException) {
    println "$number.toString()"
}
```

NOTE You can think of `"$number.toString()"` as being interpreted by the parser as `"${number.toString}()"`.

Similarly, if the expression is ambiguous, you need to keep the curly braces:

```
String thing = 'treasure'
assert 'The x-coordinate of the treasure is represented by treasure.x' ==
    "The x-coordinate of the $thing is represented by $thing.x"   // <= Not allowed:
ambiguous!!
assert 'The x-coordinate of the treasure is represented by treasure.x' ==
        "The x-coordinate of the $thing is represented by ${thing}.x"  // <= Curly
braces required
```

If you need to escape the `$` or `${}` placeholders in a GString so they appear as is without interpolation, you just need to use a `\` backslash character to escape the dollar sign:

```
assert '$5' == "\$5"
assert '${name}' == "\${name}"
```

Special case of interpolating closure expressions

So far, we've seen we could interpolate arbitrary expressions inside the `${}` placeholder, but there is a special case and notation for closure expressions. When the placeholder contains an arrow, `${→}`, the expression is actually a closure expression — you can think of it as a closure with a dollar prepended in front of it:

```
def sParameterLessClosure = "1 + 2 == ${-> 3}" ①
assert sParameterLessClosure == '1 + 2 == 3'

def sOneParamClosure = "1 + 2 == ${ w -> w << 3}" ②
assert sOneParamClosure == '1 + 2 == 3'
```

① The closure is a parameterless closure which doesn't take arguments.

② Here, the closure takes a single `java.io.StringWriter` argument, to which you can append content with the `<<` leftShift operator. In either case, both placeholders are embedded closures.

In appearance, it looks like a more verbose way of defining expressions to be interpolated, but closures have an interesting advantage over mere expressions: lazy evaluation.

Let's consider the following sample:

```
def number = 1 ①
def eagerGString = "value == ${number}"
def lazyGString = "value == ${ -> number }"

assert eagerGString == "value == 1" ②
assert lazyGString ==  "value == 1" ③

number = 2 ④
assert eagerGString == "value == 1" ⑤
assert lazyGString ==  "value == 2" ⑥
```

① We define a `number` variable containing `1` that we then interpolate within two GStrings, as an expression in `eagerGString` and as a closure in `lazyGString`.

② We expect the resulting string to contain the same string value of 1 for `eagerGString`.

③ Similarly for `lazyGString`

④ Then we change the value of the variable to a new number

⑤ With a plain interpolated expression, the value was actually bound at the time of creation of the GString.

⑥ But with a closure expression, the closure is called upon each coercion of the GString into String, resulting in an updated string containing the new number value.

NOTE	An embedded closure expression taking more than one parameter will generate an exception at runtime. Only closures with zero or one parameter are allowed.

Interoperability with Java

When a method (whether implemented in Java or Groovy) expects a `java.lang.String`, but we pass a `groovy.lang.GString` instance, the `toString()` method of the GString is automatically and transparently called.

```
String takeString(String message) {          ④
    assert message instanceof String         ⑤
    return message
}

def message = "The message is ${'hello'}"    ①
assert message instanceof GString            ②

def result = takeString(message)             ③
assert result instanceof String
assert result == 'The message is hello'
```

① We create a GString variable

② We double-check it's an instance of the GString

③ We then pass that GString to a method taking a String as parameter

④ The signature of the `takeString()` method explicitly says its sole parameter is a String

⑤ We also verify that the parameter is indeed a String and not a GString.

GString and String hashCodes

Although interpolated strings can be used in lieu of plain Java strings, they differ with strings in a particular way: their hashCodes are different. Plain Java strings are immutable, whereas the resulting String representation of a GString can vary, depending on its interpolated values. Even for the same resulting string, GStrings and Strings don't have the same hashCode.

```
assert "one: ${1}".hashCode() != "one: 1".hashCode()
```

GString and Strings having different hashCode values, using GString as Map keys should be avoided, especially if we try to retrieve an associated value with a String instead of a GString.

```
def key = "a"
def m = ["${key}": "letter ${key}"]     ①

assert m["a"] == null                   ②
```

① The map is created with an initial pair whose key is a GString

② When we try to fetch the value with a String key, we will not find it, as Strings and GString have different hashCode values

Triple-double-quoted string

Triple-double-quoted strings behave like double-quoted strings, with the addition that they are multiline, like the triple-single-quoted strings.

```
def name = 'Groovy'
```

```
def template = """
    Dear Mr ${name},

    You're the winner of the lottery!

    Yours sincerly,

    Dave
"""

assert template.toString().contains('Groovy')
```

NOTE	Neither double quotes nor single quotes need be escaped in triple-double-quoted strings.

Slashy string

Beyond the usual quoted strings, Groovy offers slashy strings, which use / as the opening and closing delimiter. Slashy strings are particularly useful for defining regular expressions and patterns, as there is no need to escape backslashes.

Example of a slashy string:

```
def fooPattern = /.*foo.*/
assert fooPattern == '.*foo.*'
```

Only forward slashes need to be escaped with a backslash:

```
def escapeSlash = /The character \/ is a forward slash/
assert escapeSlash == 'The character / is a forward slash'
```

Slashy strings are multiline:

```
def multilineSlashy = /one
    two
    three/

assert multilineSlashy.contains('\n')
```

Slashy strings can be thought of as just another way to define a GString but with different escaping rules. They hence support interpolation:

```
def color = 'blue'
def interpolatedSlashy = /a ${color} car/
```

```
assert interpolatedSlashy == 'a blue car'
```

Special cases

An empty slashy string cannot be represented with a double forward slash, as it's understood by the Groovy parser as a line comment. That's why the following assert would actually not compile as it would look like a non-terminated statement:

```
assert '' == //
```

As slashy strings were mostly designed to make regexp easier so a few things that are errors in GStrings like `$()` or `$5` will work with slashy strings.

Remember that escaping backslashes is not required. An alternative way of thinking of this is that in fact escaping is not supported. The slashy string `/\t/` won't contain a tab but instead a backslash followed by the character 't'. Escaping is only allowed for the slash character, i.e. `/\/folder/` will be a slashy string containing `'/folder'`. A consequence of slash escaping is that a slashy string can't end with a backslash. Otherwise that will escape the slashy string terminator. You can instead use a special trick, `/ends with slash ${'\\'}/`. But best just avoid using a slashy string in such a case.

Dollar slashy string

Dollar slashy strings are multiline GStrings delimited with an opening `$/` and a closing `/$`. The escaping character is the dollar sign, and it can escape another dollar, or a forward slash. Escaping for the dollar and forward slash characters is only needed where conflicts arise with the special use of those characters. The characters `$foo` would normally indicate a GString placeholder, so those four characters can be entered into a dollar slashy string by escaping the dollar, i.e. `$$foo`. Similarly, you will need to escape a dollar slashy closing delimiter if you want it to appear in your string.

Here are a few examples:

```
def name = "Guillaume"
def date = "April, 1st"

def dollarSlashy = $/
    Hello $name,
    today we're ${date}.

    $ dollar sign
    $$ escaped dollar sign
    \ backslash
    / forward slash
    $/ escaped forward slash
    $$$/ escaped opening dollar slashy
    $/$$ escaped closing dollar slashy
/$

assert [
```

```
    'Guillaume',
    'April, 1st',
    '$ dollar sign',
    '$ escaped dollar sign',
    '\\ backslash',
    '/ forward slash',
    '/ escaped forward slash',
    '$/ escaped opening dollar slashy',
    '/$ escaped closing dollar slashy'
].every { dollarSlashy.contains(it) }
```

It was created to overcome some of the limitations of the slashy string escaping rules. Use it when its escaping rules suit your string contents (typically if it has some slashes you don't want to escape).

String summary table

String name	String syntax	Interpolated	Multiline	Escape character
Single-quoted	`'…'`	[check empty]	[check empty]	`\`
Triple-single-quoted	`'''…'''`	[check empty]	[check]	`\`
Double-quoted	`"…"`	[check]	[check empty]	`\`
Triple-double-quoted	`"""…"""`	[check]	[check]	`\`
Slashy	`/…/`	[check]	[check]	`\`
Dollar slashy	`$/…/$`	[check]	[check]	`$`

Characters

Unlike Java, Groovy doesn't have an explicit character literal. However, you can be explicit about making a Groovy string an actual character, by three different means:

```
char c1 = 'A' ①
assert c1 instanceof Character

def c2 = 'B' as char ②
assert c2 instanceof Character

def c3 = (char)'C' ③
assert c3 instanceof Character
```

① by being explicit when declaring a variable holding the character by specifying the `char` type

② by using type coercion with the `as` operator

③ by using a cast to char operation

NOTE	The first option *1* is interesting when the character is held in a variable, while the other two (*2* and *3*) are more interesting when a char value must be passed as argument of a method call.

Numbers

Groovy supports different kinds of integral literals and decimal literals, backed by the usual `Number` types of Java.

Integral literals

The integral literal types are the same as in Java:

- `byte`
- `char`
- `short`
- `int`
- `long`
- `java.math.BigInteger`

You can create integral numbers of those types with the following declarations:

```
// primitive types
byte  b = 1
char  c = 2
short s = 3
int   i = 4
long  l = 5

// infinite precision
BigInteger bi =  6
```

If you use optional typing by using the `def` keyword, the type of the integral number will vary: it'll adapt to the capacity of the type that can hold that number.

For positive numbers:

```
def a = 1
assert a instanceof Integer

// Integer.MAX_VALUE
def b = 2147483647
assert b instanceof Integer

// Integer.MAX_VALUE + 1
def c = 2147483648
```

```
assert c instanceof Long

// Long.MAX_VALUE
def d = 9223372036854775807
assert d instanceof Long

// Long.MAX_VALUE + 1
def e = 9223372036854775808
assert e instanceof BigInteger
```

As well as for negative numbers:

```
def na = -1
assert na instanceof Integer

// Integer.MIN_VALUE
def nb = -2147483648
assert nb instanceof Integer

// Integer.MIN_VALUE - 1
def nc = -2147483649
assert nc instanceof Long

// Long.MIN_VALUE
def nd = -9223372036854775808
assert nd instanceof Long

// Long.MIN_VALUE - 1
def ne = -9223372036854775809
assert ne instanceof BigInteger
```

Alternative non-base 10 representations

Numbers can also be represented in binary, octal, hexadecimal and decimal bases.

Binary literal

Binary numbers start with a `0b` prefix:

```
int xInt = 0b10101111
assert xInt == 175

short xShort = 0b11001001
assert xShort == 201 as short

byte xByte = 0b11
assert xByte == 3 as byte

long xLong = 0b101101101101
```

```
assert xLong == 2925l

BigInteger xBigInteger = 0b111100100001
assert xBigInteger == 3873g

int xNegativeInt = -0b10101111
assert xNegativeInt == -175
```

Octal literal

Octal numbers are specified in the typical format of `0` followed by octal digits.

```
int xInt = 077
assert xInt == 63

short xShort = 011
assert xShort == 9 as short

byte xByte = 032
assert xByte == 26 as byte

long xLong = 0246
assert xLong == 166l

BigInteger xBigInteger = 01111
assert xBigInteger == 585g

int xNegativeInt = -077
assert xNegativeInt == -63
```

Hexadecimal literal

Hexadecimal numbers are specified in the typical format of `0x` followed by hex digits.

```
int xInt = 0x77
assert xInt == 119

short xShort = 0xaa
assert xShort == 170 as short

byte xByte = 0x3a
assert xByte == 58 as byte

long xLong = 0xffff
assert xLong == 65535l

BigInteger xBigInteger = 0xaaaa
assert xBigInteger == 43690g
```

```
Double xDouble = new Double('0x1.0p0')
assert xDouble == 1.0d

int xNegativeInt = -0x77
assert xNegativeInt == -119
```

Decimal literals

The decimal literal types are the same as in Java:

- `float`
- `double`
- `java.math.BigDecimal`

You can create decimal numbers of those types with the following declarations:

```
// primitive types
float  f = 1.234
double d = 2.345

// infinite precision
BigDecimal bd =  3.456
```

Decimals can use exponents, with the `e` or `E` exponent letter, followed by an optional sign, and an integral number representing the exponent:

```
assert 1e3  ==  1_000.0
assert 2E4  == 20_000.0
assert 3e+1 ==     30.0
assert 4E-2 ==      0.04
assert 5e-1 ==      0.5
```

Conveniently for exact decimal number calculations, Groovy chooses `java.math.BigDecimal` as its decimal number type. In addition, both `float` and `double` are supported, but require an explicit type declaration, type coercion or suffix. Even if `BigDecimal` is the default for decimal numbers, such literals are accepted in methods or closures taking `float` or `double` as parameter types.

NOTE	Decimal numbers can't be represented using a binary, octal or hexadecimal representation.

Underscore in literals

When writing long literal numbers, it's harder on the eye to figure out how some numbers are grouped together, for example with groups of thousands, of words, etc. By allowing you to place underscore in number literals, it's easier to spot those groups:

```
long creditCardNumber = 1234_5678_9012_3456L
long socialSecurityNumbers = 999_99_9999L
double monetaryAmount = 12_345_132.12
long hexBytes = 0xFF_EC_DE_5E
long hexWords = 0xFFEC_DE5E
long maxLong = 0x7fff_ffff_ffff_ffffL
long alsoMaxLong = 9_223_372_036_854_775_807L
long bytes = 0b11010010_01101001_10010100_10010010
```

Number type suffixes

We can force a number (including binary, octals and hexadecimals) to have a specific type by giving a suffix (see table below), either uppercase or lowercase.

Type	Suffix
BigInteger	`G` or `g`
Long	`L` or `l`
Integer	`I` or `i`
BigDecimal	`G` or `g`
Double	`D` or `d`
Float	`F` or `f`

Examples:

```
assert 42I == Integer.valueOf('42')
assert 42i == Integer.valueOf('42') // lowercase i more readable
assert 123L == Long.valueOf("123") // uppercase L more readable
assert 2147483648 == Long.valueOf('2147483648') // Long type used, value too large for
an Integer
assert 456G == new BigInteger('456')
assert 456g == new BigInteger('456')
assert 123.45 == new BigDecimal('123.45') // default BigDecimal type used
assert .321 == new BigDecimal('.321')
assert 1.200065D == Double.valueOf('1.200065')
assert 1.234F == Float.valueOf('1.234')
assert 1.23E23D == Double.valueOf('1.23E23')
assert 0b1111L.class == Long // binary
assert 0xFFi.class == Integer // hexadecimal
assert 034G.class == BigInteger // octal
```

Math operations

Although operators are covered in more detail elsewhere, it's important to discuss the behavior of math operations and what their resulting types are.

Division and power binary operations aside (covered below),

- binary operations between `byte`, `char`, `short` and `int` result in `int`
- binary operations involving `long` with `byte`, `char`, `short` and `int` result in `long`
- binary operations involving `BigInteger` and any other integral type result in `BigInteger`
- binary operations involving `BigDecimal` with `byte`, `char`, `short`, `int` and `BigInteger` result in `BigDecimal`
- binary operations between `float`, `double` and `BigDecimal` result in `double`
- binary operations between two `BigDecimal` result in `BigDecimal`

The following table summarizes those rules:

	byte	**char**	**short**	**int**	**long**	**BigInteger**	**float**	**double**	**BigDecimal**
byte	int	int	int	int	long	BigInteger	double	double	BigDecimal
char		int	int	int	long	BigInteger	double	double	BigDecimal
short			int	int	long	BigInteger	double	double	BigDecimal
int				int	long	BigInteger	double	double	BigDecimal
long					long	BigInteger	double	double	BigDecimal
BigInteger						BigInteger	double	double	BigDecimal
float							double	double	double
double								double	double
BigDecimal									BigDecimal

NOTE	Thanks to Groovy's operator overloading, the usual arithmetic operators work as well with `BigInteger` and `BigDecimal`, unlike in Java where you have to use explicit methods for operating on those numbers.

The case of the division operator

The division operators `/` (and `/=` for division and assignment) produce a `double` result if either operand is a `float` or `double`, and a `BigDecimal` result otherwise (when both operands are any combination of an integral type `short`, `char`, `byte`, `int`, `long`, `BigInteger` or `BigDecimal`).

Under static type checking, the inferred type of a division follows the same rule from the static types of the operands. An operand typed as `Number` may hold a `Float` or `Double` at runtime as well as

any of the other types, so a division involving one is inferred as `Number`.

`BigDecimal` division is performed with the `divide()` method if the division is exact (i.e. yielding a result that can be represented within the bounds of the same precision and scale), or using a `MathContext` with a precision of the maximum of the two operands' precision plus an extra precision of 10, and a scale of the maximum of 10 and the maximum of the operands' scale.

NOTE	For integer division like in Java, you should use the `intdiv()` method, as Groovy doesn't provide a dedicated integer division operator symbol.

The case of the power operator

The power operation is represented by the `**` operator, with two parameters: the base and the exponent. The result of the power operation depends on its operands, and the result of the operation (in particular if the result can be represented as an integral value).

The following rules are used by Groovy's power operation to determine the resulting type:

- If the exponent is a decimal value
  - if the result can be represented as an `Integer`, then return an `Integer`
  - else if the result can be represented as a `Long`, then return a `Long`
  - otherwise return a `Double`
- If the exponent is an integral value
  - if the exponent is strictly negative, then return an `Integer`, `Long` or `Double` if the result value fits in that type
  - if the exponent is positive or zero
    - if the base is a `BigDecimal`, then return a `BigDecimal` result value
    - if the base is a `BigInteger`, then return a `BigInteger` result value
    - if the base is an `Integer`, then return an `Integer` if the result value fits in it, otherwise a `BigInteger`
    - if the base is a `Long`, then return a `Long` if the result value fits in it, otherwise a `BigInteger`

We can illustrate those rules with a few examples:

```
// base and exponent are ints and the result can be represented by an Integer
assert    2    **   3    instanceof Integer    //  8
assert   10    **   9    instanceof Integer    //  1_000_000_000

// the base is a long, so fit the result in a Long
// (although it could have fit in an Integer)
assert    5L   **   2    instanceof Long       //  25

// the result can't be represented as an Integer or Long, so return a BigInteger
assert  100    **  10    instanceof BigInteger //  10e20
assert 1234    ** 123    instanceof BigInteger //  17051580621272704287 5...
```

```
// the base is a BigDecimal and the exponent a negative int
// but the result can be represented as an Integer
assert    0.5  **  -2    instanceof Integer    //  4

// the base is an int, and the exponent a negative float
// but again, the result can be represented as an Integer
assert    1    **  -0.3f instanceof Integer    //  1

// the base is an int, and the exponent a negative int
// but the result will be calculated as a Double
// (both base and exponent are actually converted to doubles)
assert   10    **  -1    instanceof Double     //  0.1

// the base is a BigDecimal, and the exponent is an int, so return a BigDecimal
assert    1.2  **  10    instanceof BigDecimal //  6.1917364224

// the base is a float or double, and the exponent is an int
// but the result can only be represented as a Double value
assert    3.4f **   5    instanceof Double     //  454.35430372146965
assert    5.6d **   2    instanceof Double     //  31.359999999999996

// the exponent is a decimal value
// and the result can only be represented as a Double value
assert    7.8  **   1.9  instanceof Double     //  49.542708423868476
assert    2    **   0.1f instanceof Double     //  1.0717734636432956
```

Booleans

Boolean is a special data type that is used to represent truth values: `true` and `false`. Use this data type for simple flags that track true/false conditions.

Boolean values can be stored in variables, assigned into fields, just like any other data type:

```
def myBooleanVariable = true
boolean untypedBooleanVar = false
booleanField = true
```

`true` and `false` are the only two primitive boolean values. But more complex boolean expressions can be represented using logical operators.

In addition, Groovy has special rules (often referred to as *Groovy Truth*) for coercing non-boolean objects to a boolean value.

Lists

Groovy uses a comma-separated list of values, surrounded by square brackets, to denote lists. Groovy lists are plain JDK `java.util.List`, as Groovy doesn't define its own collection classes. The concrete list implementation used when defining list literals are `java.util.ArrayList` by default,

unless you decide to specify otherwise, as we shall see later on.

```
def numbers = [1, 2, 3]         ①

assert numbers instanceof List  ②
assert numbers.size() == 3      ③
```

① We define a list numbers delimited by commas and surrounded by square brackets, and we assign that list into a variable

② The list is an instance of Java's `java.util.List` interface

③ The size of the list can be queried with the `size()` method, and shows our list contains 3 elements

In the above example, we used a homogeneous list, but you can also create lists containing values of heterogeneous types:

```
def heterogeneous = [1, "a", true]  ①
```

① Our list here contains a number, a string and a boolean value

We mentioned that by default, list literals are actually instances of `java.util.ArrayList`, but it is possible to use a different backing type for our lists, thanks to using type coercion with the `as` operator, or with explicit type declaration for your variables:

```
def arrayList = [1, 2, 3]
assert arrayList instanceof java.util.ArrayList

def linkedList = [2, 3, 4] as LinkedList    ①
assert linkedList instanceof java.util.LinkedList

LinkedList otherLinked = [3, 4, 5]          ②
assert otherLinked instanceof java.util.LinkedList
```

① We use coercion with the `as` operator to explicitly request a `java.util.LinkedList` implementation

② We can say that the variable holding the list literal is of type `java.util.LinkedList`

You can access elements of the list with the `[]` subscript operator (both for reading and setting values) with positive indices or negative indices to access elements from the end of the list, as well as with ranges, and use the `<<` leftShift operator to append elements to a list:

```
def letters = ['a', 'b', 'c', 'd']

assert letters[0] == 'a'     ①
assert letters[1] == 'b'
```

```
assert letters[-1] == 'd'    ②
assert letters[-2] == 'c'

letters[2] = 'C'             ③
assert letters[2] == 'C'

letters << 'e'               ④
assert letters[ 4] == 'e'
assert letters[-1] == 'e'

assert letters[1, 3] == ['b', 'd']        ⑤
assert letters[2..4] == ['C', 'd', 'e']   ⑥
```

① Access the first element of the list (zero-based counting)

② Access the last element of the list with a negative index: -1 is the first element from the end of the list

③ Use an assignment to set a new value for the third element of the list

④ Use the << leftShift operator to append an element at the end of the list

⑤ Access two elements at once, returning a new list containing those two elements

⑥ Use a range to access a range of values from the list, from a start to an end element position

As lists can be heterogeneous in nature, lists can also contain other lists to create multidimensional lists:

```
def multi = [[0, 1], [2, 3]]     ①
assert multi[1][0] == 2          ②
```

① Define a list of numbers

② Access the second element of the top-most list, and the first element of the inner list

Arrays

Groovy reuses the list notation for arrays, but to make such literals arrays, you need to explicitly define the type of the array through coercion or type declaration.

```
String[] arrStr = ['Ananas', 'Banana', 'Kiwi']  ①

assert arrStr instanceof String[]    ②
assert !(arrStr instanceof List)

def numArr = [1, 2, 3] as int[]      ③

assert numArr instanceof int[]       ④
assert numArr.size() == 3
```

① Define an array of strings using explicit variable type declaration

② Assert that we created an array of strings

③ Create an array of ints with the `as` operator

④ Assert that we created an array of primitive ints

You can also create multi-dimensional arrays:

```
def matrix3 = new Integer[3][3]         ①
assert matrix3.size() == 3

Integer[][] matrix2                     ②
matrix2 = [[1, 2], [3, 4]]
assert matrix2 instanceof Integer[][]
```

① You can define the bounds of a new array

② Or declare an array without specifying its bounds

Access to elements of an array follows the same notation as for lists:

```
String[] names = ['Cédric', 'Guillaume', 'Jochen', 'Paul']
assert names[0] == 'Cédric'     ①

names[2] = 'Blackdrag'          ②
assert names[2] == 'Blackdrag'
```

① Retrieve the first element of the array

② Set the value of the third element of the array to a new value

Java-style array initialization

Groovy has always supported literal list/array definitions using square brackets and has avoided Java-style curly braces so as not to conflict with closure definitions. In the case where the curly braces come immediately after an array type declaration however, there is no ambiguity with closure definitions, so Groovy 3 and above support that variant of the Java array initialization expression.

Examples:

```
def primes = new int[] {2, 3, 5, 7, 11}
assert primes.size() == 5 && primes.sum() == 28
assert primes.class.name == '[I'

def pets = new String[] {'cat', 'dog'}
assert pets.size() == 2 && pets.sum() == 'catdog'
assert pets.class.name == '[Ljava.lang.String;'

// traditional Groovy alternative still supported
String[] groovyBooks = [ 'Groovy in Action', 'Making Java Groovy' ]
```

```
assert groovyBooks.every{ it.contains('Groovy') }
```

Maps

Sometimes called dictionaries or associative arrays in other languages, Groovy features maps. Maps associate keys to values, separating keys and values with colons, and each key/value pairs with commas, and the whole keys and values surrounded by square brackets.

```
def colors = [red: '#FF0000', green: '#00FF00', blue: '#0000FF']   ①

assert colors['red'] == '#FF0000'    ②
assert colors.green  == '#00FF00'    ③

colors['pink'] = '#FF00FF'           ④
colors.yellow  = '#FFFF00'           ⑤

assert colors.pink == '#FF00FF'
assert colors['yellow'] == '#FFFF00'

assert colors instanceof java.util.LinkedHashMap
```

① We define a map of string color names, associated with their hexadecimal-coded html colors

② We use the subscript notation to check the content associated with the `red` key

③ We can also use the property notation to assert the color green's hexadecimal representation

④ Similarly, we can use the subscript notation to add a new key/value pair

⑤ Or the property notation, to add the `yellow` color

NOTE When using names for the keys, we actually define string keys in the map.

NOTE Groovy creates maps that are actually instances of `java.util.LinkedHashMap`.

If you try to access a key which is not present in the map:

```
assert colors.unknown == null

def emptyMap = [:]
assert emptyMap.anyKey == null
```

You will retrieve a `null` result.

In the examples above, we used string keys, but you can also use values of other types as keys:

```
def numbers = [1: 'one', 2: 'two']
```

```
assert numbers[1] == 'one'
```

Here, we used numbers as keys, as numbers can unambiguously be recognized as numbers, so Groovy will not create a string key like in our previous examples. But consider the case you want to pass a variable in lieu of the key, to have the value of that variable become the key:

```
def key = 'name'
def person = [key: 'Guillaume']      ①

assert !person.containsKey('name')   ②
assert person.containsKey('key')     ③
```

① The `key` associated with the `'Guillaume'` name will actually be the `"key"` string, not the value associated with the `key` variable

② The map doesn't contain the `'name'` key

③ Instead, the map contains a `'key'` key

NOTE You can also pass quoted strings as well as keys: ["name": "Guillaume"]. This is mandatory if your key string isn't a valid identifier, for example if you wanted to create a string key containing a dash like in: ["street-name": "Main street"].

When you need to pass variable values as keys in your map definitions, you must surround the variable or expression with parentheses:

```
person = [(key): 'Guillaume']        ①

assert person.containsKey('name')    ②
assert !person.containsKey('key')    ③
```

① This time, we surround the `key` variable with parentheses, to instruct the parser we are passing a variable rather than defining a string key

② The map does contain the `name` key

③ But the map doesn't contain the `key` key as before

Grammar

This chapter is a complete specification of the Groovy language syntax recognised at compilation phase `PARSING`. It is written for readers who need a closed, citable grammar — language researchers, tool authors, and implementers. Informal syntax, examples, and runtime meaning live in Syntax, Operators, Program structure, and Semantics.

The executable grammar is `src/antlr/GroovyLexer.g4` and `src/antlr/GroovyParser.g4`. This chapter is a complete EBNF projection of those files: every parser rule, lexer token, fragment, and mode appears. The `.g4` files remain the implementation source of truth — if a production here and a rule there disagree, the rule wins until this chapter is updated.

Status and scope

Groovy's surface syntax is **not a pure context-free grammar**. The specification has three layers:

1. a **lexical context machine** (lexer modes, grouping stack, newline hiding, token-type rewrites);
2. a **CFG skeleton** in Groovy Spec EBNF, using the ANTLR parser rule names as nonterminals;
3. a finite list of **restrictions** R1, R2, … that filter or disambiguate that skeleton (the analogue of JLS static restrictions / ECMA-262 early errors).

Lexer alternatives are **ordered** (first match among those whose predicates hold, then longest match). Parser alternatives of `expression` are ordered by precedence (earlier binds tighter). Conversion of the parse tree to AST (`AstBuilder`, compilation phase `CONVERSION`), semantic analysis, type checking, and AST transforms are out of scope. The CFG is an intentional over-approximation: some programs that match it are rejected later. Those cases are listed under Syntactic over-approximation rather than hidden by inventing extra productions.

This chapter does **not** re-introduce parser error alternatives for missing `)`, `]`, or `}` (GROOVY-9588). Recognition failures share the diagnostic path described in `ARCHITECTURE.md`.

Notation

Groovy Spec EBNF

Productions are written in `[source,ebnf]` blocks. The operator set is W3C EBNF; the names are ANTLR 4 rule names.

Construct	Meaning	Example
`N ::= …`	Production for nonterminal `N`	`sep ::= (NL | ';')+`
`A | B`	Alternation (one of)	`'class' | 'interface'`
`( … )`	Grouping	`'static'? qualifiedName`
`X?`	Optional (zero or one)	`packageDeclaration?`
`X*`	Kleene star (zero or more)	`NL*`
`X+`	One or more	`(NL | ';')+`
`'foo'`	Terminal with that exact spelling (see Terminal vocabulary)	`'import'`
`Identifier`	Terminal with no unique spelling (token name)	`GStringBegin`
`/* R12 */`	This alternative is subject to restriction R12. Lexer fragments with such marks are **ordered**: try earlier alternatives first.	`'$' /* R2 fails */`
`fragment F`	Lexer fragment: not a token, only used by other lexer productions	`fragment Digits`

Construct	Meaning	Example
`<MODE>`	Lexer start condition (mode). The production applies only in that mode	`<DQ_GSTRING_MODE>`

Concatenation is juxtaposition (W3C, not ISO 14977 comma). Square brackets and braces are Groovy terminals (lists, closures), so ISO 14977 optional/repetition brackets are not used.

A bare `A | B` in a parser production is unordered alternation (the CFG skeleton). Where the g4 is ordered — lexer rules, lexer fragments with predicates, and predicated parser alternatives — this chapter annotates the restricted alternative with `/* R */`. Reconstructing a lexer from the unordered `|` of a fragment without those comments yields a different language.

There is **no** `nls` nonterminal in `GroovyParser.g4`. Optional newlines are written `NL*` in situ. The statement separator is `sep`.

Left-recursive `expression` is a CFG production. Associativity that ANTLR marks with `<assoc=right>` is recorded in the precedence table, not with a special EBNF operator. Earlier alternatives of `expression` bind tighter (ANTLR 4 left-recursion).

ANTLR parameterized rules (`classBody[int t]`, `expressionList[boolean canSpread]`) are not parameterized EBNF. Syntactic parameters become named specialisations plus Parameterised-rule bijection; parameters that exist only to tag the CST for `AstBuilder` are notes, not syntax.

Parser alternative labels (`#castExprAlt`, …) are visitor names, not nonterminals.

Terminal vocabulary

Keywords and identifier sets

The lexer always emits a dedicated token for a keyword spelling (except `val` when R5 disables it). The parser then **re-accepts** some of those tokens as names.

Four concentric sets:

- `identifier` — unquoted names, labels, type names, unquoted method names.
- `keywords` — after `.` (`namePart`), as map keys, as annotation element names.
- `qualifiedNameElement` — package / import pieces, additionally `'def' 'in' 'as' 'trait'`.
- `methodName` — `identifier` or `stringLiteral` (quoted names of any spelling, including reserved words).

`className` is only `CapitalizedIdentifier`.

Lexeme	Token	In `identifier`?	In `keywords`?	Notes
`abstract assert break case catch class continue default do else enum extends final finally for if implements import instanceof interface native new package return static strictfp super switch synchronized this throw throws transient try volatile while`	matching keyword token	no	yes	reserved
`public protected private`	`PUBLIC` `PROTECTED` `PRIVATE`	no	yes	reserved
`void` and primitives	`VOID` / `BuiltInPrimitiveType`	no	yes	primitives are one token type (see below)
`true false null`	`BooleanLiteral` / `NullLiteral`	no	yes	reserved words, not `identifier`
`def`	`DEF`	no	yes	also `qualifiedNameElement`; a modifier, not a type production
`const goto threadsafe`	`CONST` `GOTO` `THREADSAFE`	no	yes	reserved and unused. Not in `identifier`: a bare use is a **parse** error. In `keywords`, so legal after `.`, as a map key, and as an annotation element name.
`non-sealed`	`NON_SEALED`	no	yes	one token, including the hyphen
`as in trait record sealed permits var yield async await defer module`	matching token	**yes**	`module`: **no**; others: yes	contextual. `module` is in `identifier` only (for `import module`)

Lexeme	Token	In `identifier`?	In `keywords`?	Notes
`val`	`VAL` or `Identifier` (R5)	**yes** (always: token `VAL` or `Identifier`)	**yes** only when the token is `VAL`	feature flag `groovy.val.enabled` (default `true`). When false, `val` is `Identifier` and is therefore not a `variableModifier`.

`BuiltInPrimitiveType` is a **single** token type whose spellings are the fragments `boolean char byte short int long float double`. Those fragments are not independent tokens.

`it` is **not** a token. Implicit closure parameters are semantic.

The tutorial keyword tables in Syntax are a user-facing cut. This chapter follows the g4. In particular the grammar has `yield` (not `yields`), and `async` / `await` / `defer` / `module` / `val` as tokens.

Operators and separators

Lexer tokens with unique spelling (parser productions quote the lexeme):

Lexeme	Token	Role

`.. <.. ..< <..<`	`RANGE_INCLUSIVE`
	`RANGE_EXCLUSIVE_LEFT`
	`RANGE_EXCLUSIVE_RIGHT`
	`RANGE_EXCLUSIVE_FULL`
		ranges (shift precedence)

`*. ?. ??. .& ::`	`SPREAD_DOT`
	`SAFE_DOT`
	`SAFE_CHAIN_DOT`
	`METHOD_POINTER`
	`METHOD_REFERENCE`
		path operators. `'??.'` is syntax even if omitted from the operators tutorial

`?[`	`SAFE_INDEX`	safe index; pushes grouping (R9)
`?: ?=`	`ELVIS` `ELVIS_ASSIGN`	elvis / elvis-assign
`=~ ==~`	`REGEX_FIND` `REGEX_MATCH`	regex find / match
`**`	`POWER` `POWER_ASSIGN`	power

`<=> === !== ==>`	`SPACESHIP`
	`IDENTICAL`
	`NOT_IDENTICAL`
	`IMPLIES`	

`->`	`ARROW`	closure / lambda / switch-expression arrow
`!instanceof !in`	`NOT_INSTANCEOF` `NOT_IN`	R6

Lexeme	Token	Role
( ) { } [ ]	LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK	grouping; each opener pushMode(DEFAULT_MODE)
; , . @ …	SEMI COMMA DOT AT ELLIPSIS	
= > < ! ~ ? : == <= >= != && || +` `--` ` - * / & | ^ %	ASSIGN GT LT NOT BITNOT QUESTION COLON EQUAL LE GE NOTEQUAL AND OR INC DEC ADD SUB MUL DIV BITAND BITOR XOR MOD	/ as an **operator** is DIV, never written '/' in parser EBNF (R1)
+= -= *= /= &= |= ^= %= <<= >>= >>>=	matching *_ASSIGN tokens	shift **assignments** are tokens; shift **operators** are not

There are **no** LSHIFT / RSHIFT / URSHIFT lexer tokens. The parser composes '<' '<', '>' '>', '>' '>' '>' so List<List<T>> can close.

Literals as token names

IntegerLiteral, FloatingPointLiteral, BooleanLiteral, NullLiteral, StringLiteral, GStringBegin, GStringPart, GStringEnd, GStringPathPart, Identifier, CapitalizedIdentifier, NL.

Quote style of an interpolating string is **not** a parser token: TdqGStringBegin, SlashyGStringBegin, and DollarSlashyGStringBegin are rewritten to GStringBegin (similarly GStringPart / GStringEnd).

Lexical structure

Input elements

The lexer produces a stream of default-channel tokens for the parser, plus skipped or hidden trivia. Successful tokenisation never takes the UNEXPECTED_CHAR / unclosed-comment / invalid-octal paths (AbstractLexer); those are diagnostics, not language (Invalid input and diagnostics).

White space and line escapes

```
WS ::= ([ \t]+ | LineEscape+)          /* skip */

fragment LineEscape     ::= '\\' LineTerminator
fragment LineTerminator ::= '\r'? '\n' | '\r'
```

Spaces and tabs are skipped. A backslash immediately followed by a line terminator is also skipped (line continuation) and is **not** NL.

Line terminators, comments, shebang

```
NL ::= LineTerminator                  /* then R9 */

SL_COMMENT ::=
    '//' ~[\r\n\uFFFF]*                /* -> type(NL); then R9 */

ML_COMMENT ::=
    '/*' .*? ('*/' | EOF)              /* -> type(NL); then R10.
                                          unclosed: diagnostic at opener */

SH_COMMENT ::=                         /* skip; R11 */
    '#!' ShCommand (LineTerminator '#!' ShCommand)*

fragment ShCommand ::= ~[\r\n\uFFFF]*
```

There is no comment channel. `//` and `/* … */` are rewritten to token type `NL`, so a comment can act as a statement separator when it stays on the default channel.

Groovydoc (`/** … */` and runtime `/@ … @/`) is **not** a distinct token. It is `ML_COMMENT`; `GroovydocManager` attaches it after parse. Groovydoc `*tag` semantics are excluded.

`SH_COMMENT` is skipped when R11 holds (`tokenIndex == 0`). Leading skipped `WS` (spaces, tabs, line-escapes) does not increment `tokenIndex`, so a shebang after leading spaces is still skipped. A prior **default-channel** token — including a leading `NL` — is not allowed. Extra `#!` lines immediately after a line terminator are part of the same `SH_COMMENT` token.

The grouping stack and newline significance

`(`/`[`/`{`/`?[` record a grouping frame (`enterParen`) and `pushMode(DEFAULT_MODE)`. The matching closer pops both.

Restriction R9: `NL` (and `SL_COMMENT` rewritten to `NL`) is moved to the **hidden** channel when the stack top is:

- `'('` whose **preceding default-channel token** is not `'try'`, or
- `'['`, or
- `'?['`.

Newlines inside `{ … }` are **significant**. Try-with-resources `try ( … )` does **not** hide newlines (`TRY` on the `(` frame). Angle brackets `'<'` / `'>'` do **not** push a grouping frame, which is why `typeArguments` writes `NL*` explicitly.

`$\{` in a GString also calls `enterParen()` (see GString lexical contexts), so the interpolation's `'{'` — not an enclosing `'('` / `'['` — is the stack top while the closure body is tokenised.

Restriction R10: `/* … */` remains a visible `NL` only when it is **not** inside grouping (R9) **and** only whitespace remains until end of line or file. So `a /*c*/ b` is not two statements; `a /c/` then newline then `b` is.

Inside grouping, hidden `NL` is absent from the parser token stream. Parser productions still write `NL*` / `sep`; those match only **visible** `NL`.

`lastTokenType` (used by slashy-string vs `DIV`, R1) is updated only for default-channel tokens. Hidden newlines do not change it.

Identifiers

```
CapitalizedIdentifier ::=
      [A-Z] JavaLetterOrDigit*
    | ~[\u0000-\u007F\uD800-\uDBFF]    /* R7 BMP, uppercase */
      JavaLetterOrDigit*
    | [\uD800-\uDBFF] [\uDC00-\uDFFF]  /* R7 supplementary, uppercase */
      JavaLetterOrDigit*

Identifier ::=
      [a-z$_] JavaLetterOrDigit*
    | ~[\u0000-\u007F\uD800-\uDBFF]    /* R7 BMP, not uppercase */
      JavaLetterOrDigit*
    | [\uD800-\uDBFF] [\uDC00-\uDFFF]  /* R7 supplementary, not uppercase */
      JavaLetterOrDigit*

fragment JavaLetter ::=
      [a-zA-Z$_]
    | ~[\u0000-\u007F\uD800-\uDBFF]    /* R7 BMP start, not ignorable */
    | [\uD800-\uDBFF] [\uDC00-\uDFFF]  /* supplementary start (code point) */

fragment JavaLetterOrDigit ::=
      [a-zA-Z0-9$_]
    | ~[\u0000-\u007F\uD800-\uDBFF]    /* R7 BMP part, not ignorable */
    | [\uD800-\uDBFF] [\uDC00-\uDFFF]  /* supplementary part (code point) */

fragment JavaLetterInGString ::=
    JavaLetter                         /* R8: last consumed char is not '$' */

fragment JavaLetterOrDigitInGString ::=
    JavaLetterOrDigit                  /* R8 */

fragment IdentifierInGString ::=
    JavaLetterInGString JavaLetterOrDigitInGString*
```

ASCII is a predicate-free DFA split (`[A-Z]` vs `[a-z$_]`). Supplementary-plane letters classify capitalized vs not from the decoded code point, not `LA(-1)` (a low surrogate is never uppercase; GROOVY-12398). Identifier-ignorable start and continuation characters are rejected on the **BMP** path (R7); the supplementary-plane path tests `Character.isJavaIdentifierStart` / `Part` and case only. GString identifiers cannot contain `$` (R8).

`$` and are allowed in Identifier. A leading $ or is always `Identifier`, never `CapitalizedIdentifier`.

This is stricter and more precise than the identifier ranges in the syntax tutorial.

`!in` / `!instanceof` / `val`

See R5 and R6.

Numeric literals

```
IntegerLiteral ::=
      (DecimalIntegerLiteral | HexIntegerLiteral
       | OctalIntegerLiteral | BinaryIntegerLiteral)
      Underscore?                    /* trailing '_' is then a diagnostic */

fragment DecimalIntegerLiteral ::= DecimalNumeral IntegerTypeSuffix?
fragment HexIntegerLiteral     ::= HexNumeral     IntegerTypeSuffix?
fragment OctalIntegerLiteral   ::= OctalNumeral   IntegerTypeSuffix?
fragment BinaryIntegerLiteral  ::= BinaryNumeral  IntegerTypeSuffix?
fragment IntegerTypeSuffix     ::= [lLiIgG]
fragment DecimalNumeral        ::= '0' | [1-9] (Digits? | Underscores Digits)
fragment Digits                ::= [0-9] ([0-9_]* [0-9])?
fragment Underscores           ::= '_'+
fragment Underscore            ::= '_'
fragment HexNumeral            ::= '0' [xX] HexDigits
fragment HexDigits             ::= HexDigit ([0-9a-fA-F_]* HexDigit)?
fragment HexDigit              ::= [0-9a-fA-F]
fragment OctalNumeral          ::= '0' '_'* OctalDigits
fragment OctalDigits           ::= [0-7] ([0-7_]* [0-7])?
fragment BinaryNumeral         ::= '0' [bB] BinaryDigits
fragment BinaryDigits          ::= [01] ([01_]* [01])?

FloatingPointLiteral ::=
      (DecimalFloatingPointLiteral | HexadecimalFloatingPointLiteral)
      Underscore?

fragment DecimalFloatingPointLiteral ::=
      Digits? '.' Digits ExponentPart? FloatTypeSuffix?
    | Digits ExponentPart FloatTypeSuffix?
    | Digits FloatTypeSuffix
fragment ExponentPart    ::= [eE] [+-]? Digits
fragment FloatTypeSuffix ::= [fFdDgG]
fragment HexadecimalFloatingPointLiteral ::=
    HexSignificand BinaryExponent FloatTypeSuffix?
fragment HexSignificand ::= HexNumeral '.'? | '0' [xX] HexDigits? '.' HexDigits
fragment BinaryExponent ::= [pP] [+-]? Digits
```

Suffix `G`/`g` is `BigInteger` / `BigDecimal` at conversion, not a distinct token. Invalid octal (`08`, `09`, …) and trailing `_` are lexer diagnostics, not language (Invalid input and diagnostics).

Ordinary string literals (`StringLiteral`)

A string with **no** interpolating `$` is `StringLiteral`:

```
StringLiteral ::=
      '"'  DqStringCharacter*  '"'
    | "'"  SqStringCharacter*  "'"
    | '/'  SlashyStringCharacter+ '/'     /* R1; at least one inner char */
    | '"""' TdqStringCharacter* '"""'
    | "'''" TsqStringCharacter* "'''"
    | '$/' DollarSlashyStringCharacter+ '/$'

fragment DqStringCharacter ::= ~["\r\n\\$] | EscapeSequence
fragment SqStringCharacter ::= ~['\r\n\\]  | EscapeSequence
fragment TdqStringCharacter ::=
      ~["\\$]
    | '"' /* R3 */
    | EscapeSequence
fragment TsqStringCharacter ::=
      ~['\\]
    | "'" /* R3 */
    | EscapeSequence
fragment SlashyStringCharacter ::=
      '\\' '/'
    | '$'                 /* R2 fails */
    | ~[/$\u0000]
fragment DollarSlashyStringCharacter ::=
      '$$'
    | '$/$'               /* R4: LA(-4) != '$' */
    | '$/'                /* R4: LA(1) != '$' */
    | '/'                 /* R4: LA(1) != '$' */
    | '$'                 /* R2 fails */
    | ~[/$\u0000]

fragment EscapeSequence ::=
      '\\' [btnfrs"'\\]
    | OctalEscape
    | UnicodeEscape                   /* '\' 'u' HexDigit{4} -- exactly one u */
    | '\\' '$'
    | LineEscape
fragment OctalEscape ::=
      '\\' [0-7]
    | '\\' [0-7] [0-7]
    | '\\' [0-3] [0-7] [0-7]
fragment UnicodeEscape ::=
    '\\' 'u' HexDigit HexDigit HexDigit HexDigit
```

Empty `//` is `SL_COMMENT`, not an empty slashy string (the slashy alternative requires `SlashyStringCharacter+`). Empty `$/$` is not `StringLiteral` either.

The g4 comment "Groovy allows 1 or more u's after the backslash" does **not** match the production: `UnicodeEscape` is exactly one `u`.

GString lexical contexts

An interpolating string is **not** `StringLiteral`. At the first interpolating `$` the lexer emits `GStringBegin` and enters a mode machine. The parser later sees only:

```
gstring ::=
    GStringBegin gstringValue (GStringPart gstringValue)* GStringEnd

gstringValue ::=
    gstringPath
  | closure

gstringPath ::=
    identifier GStringPathPart*
```

Modes

Mode	Role
`DEFAULT_MODE`	Normal source. Also re-entered for `$\{ ⋯ }` and nested grouping.
`DQ_GSTRING_MODE`	Body of `"⋯"` after the first interpolating `$`
`TDQ_GSTRING_MODE`	Body of `"""⋯"""`
`SLASHY_GSTRING_MODE`	Body of `/⋯/`
`DOLLAR_SLASHY_GSTRING_MODE`	Body of `$/⋯/$`
`GSTRING_TYPE_SELECTOR_MODE`	Just after `$`: either `'{'` or an `IdentifierInGString`
`GSTRING_PATH_MODE`	After a bare identifier: zero or more `'.' IdentifierInGString`, then one-character pushback (`RollBackOne`)

Opening (all in `DEFAULT_MODE`). The last three rules rewrite their token type to `GStringBegin`; each still pushes its own body mode and then `GSTRING_TYPE_SELECTOR_MODE`:

```
GStringBegin ::=
    '"' DqStringCharacter* '$'
    /* push DQ_GSTRING_MODE, GSTRING_TYPE_SELECTOR_MODE */

TdqGStringBegin ::=
    '"""' TdqStringCharacter* '$'
    /* type(GStringBegin);
       push TDQ_GSTRING_MODE, GSTRING_TYPE_SELECTOR_MODE */

SlashyGStringBegin ::=                 /* R1; R2 on the interpolating '$' */
    '/' SlashyStringCharacter* '$'
```

```
    /* type(GStringBegin);
       push SLASHY_GSTRING_MODE, GSTRING_TYPE_SELECTOR_MODE */

DollarSlashyGStringBegin ::=          /* R2 on the interpolating '$' */
    '$/' DollarSlashyStringCharacter* '$'
    /* type(GStringBegin);
       push DOLLAR_SLASHY_GSTRING_MODE, GSTRING_TYPE_SELECTOR_MODE */
```

Quote bodies accumulate with `→ more` (not parser tokens). Every body mode has a character rule:

```
<DQ_GSTRING_MODE>
GStringEnd       ::= '"'              /* popMode */
GStringPart      ::= '$'              /* push GSTRING_TYPE_SELECTOR_MODE */
GStringCharacter ::= DqStringCharacter /* more */

<TDQ_GSTRING_MODE>
TdqGStringEnd       ::= '"""'         /* type(GStringEnd), popMode */
TdqGStringPart      ::= '$'           /* type(GStringPart);
                                         push GSTRING_TYPE_SELECTOR_MODE */
TdqGStringCharacter ::= TdqStringCharacter /* more */

<SLASHY_GSTRING_MODE>
SlashyGStringEnd       ::= '$'? '/'   /* type(GStringEnd), popMode */
SlashyGStringPart      ::= '$'        /* R2; type(GStringPart);
                                         push GSTRING_TYPE_SELECTOR_MODE */
SlashyGStringCharacter ::= SlashyStringCharacter /* more */

<DOLLAR_SLASHY_GSTRING_MODE>
DollarSlashyGStringEnd       ::= '/$'  /* type(GStringEnd), popMode */
DollarSlashyGStringPart      ::= '$'   /* R2; type(GStringPart);
                                         push GSTRING_TYPE_SELECTOR_MODE */
DollarSlashyGStringCharacter ::= DollarSlashyStringCharacter /* more */
```

Selector and path:

```
<GSTRING_TYPE_SELECTOR_MODE>
GStringLBrace     ::= '{'
    /* enterParen(); type(LBRACE); popMode; push DEFAULT_MODE */
GStringIdentifier ::= IdentifierInGString
    /* type(Identifier); popMode; push GSTRING_PATH_MODE */

<GSTRING_PATH_MODE>
GStringPathPart ::= '.' IdentifierInGString
RollBackOne     ::= .
    /* popMode. If EOF after '"' or '/', type(GStringEnd);
       else channel HIDDEN. emit() calls rollbackOneChar()
       only when the type is still RollBackOne. */
```

`RollBackOne` is not a parser terminal. When the type remains `RollBackOne`, `emit()` rolls the input back one character so the parent GString mode can re-lex it. The EOF-after-`"//` path **is** the `GStringEnd` token and does **not** roll back.

`gstringPath` is only `identifier` followed by `'.' ident` segments. A method call in interpolation must be a `closure`: `"$\{foo.bar()}"`, not `"$foo.bar()"`.

How `$\{ … }` contains real Groovy

`GStringLBrace` records a `'{'` grouping frame (`enterParen()`), emits `LBRACE`, and pushes `DEFAULT_MODE`. The parser matches `closure`. So `"$\{ x + 1 }"`, `"$\{ def a = 1; a }"`, and `"$\{ → x }"` are all `gstringValue → closure` syntactically. Conversion may unwrap a no-arrow single expression; that unwrap is not this grammar.

Because `'('`, `'['`, `'{'`, and `'?['` all `pushMode(DEFAULT_MODE)`, an interpolation may contain nested strings and GStrings. That is mode nesting, not a CFG of characters.

Dollar vs interpolation (R2)

In `"…"` and `"""…"""`, every `$` that is not `\$` starts interpolation.

In `/…/` and `$/…/$`, `$` starts interpolation only if R2 holds. Therefore `/$5/` and `/$()/` are legal slashy **strings**. The same text in double quotes is a GString (or a parse failure at the selector).

Slashy string vs division vs comment (R1)

Tokenisation of `/` is a three-way (plus comment):

- `//` → `SL_COMMENT`
- slashy `StringLiteral` / `SlashyGStringBegin` when R1 holds and the next character is not `(` (`/` is `ML_COMMENT`)
- otherwise `DIV`

R1 forbids slashy strings when the previous **default-channel** token is one of `DEC`, `INC`, `THIS`, `RBRACE`, `RBRACK`, `RPAREN`, `GStringEnd`, `NullLiteral`, `StringLiteral`, `BooleanLiteral`, `IntegerLiteral`, `FloatingPointLiteral`, `Identifier`, `CapitalizedIdentifier`.

Examples: `a++ / b` is division; `list[i] / n` is division; `x = /foo/` is a slashy string (`ASSIGN` is not in the set).

Restrictions

Each restriction cites its implementation site. They are not CFG.

Lexical

R1 (`isRegexAllowed`, `LA(1) != '*'`)

`/…/` is a slashy string only if the previous default-channel token is not in the set listed above, and the character after the opening `/` is not `*`.

R2 (`isFollowedByJavaLetterInGString`)

In slashy and dollar-slashy **text** (including the interpolating `$` of `SlashyGStringBegin` / `DollarSlashyGStringBegin` and the `'$'` alternative of `SlashyStringCharacter` / `DollarSlashyStringCharacter`), `$` opens interpolation iff the next character is `{`, `_`, an ASCII letter, or a non-ASCII Java identifier part (surrogate pairs included). Otherwise `$` is a string character. Double-quoted GString `$` (unescaped) always starts interpolation.

R3 (`TdqStringCharacter` / `TsqStringCharacter`)

An interior `"` (resp. `'`) is a character when the g4 lookahead holds:

```
LA(1) != Q || LA(2) != Q
  || LA(3) == Q && (LA(4) != Q || LA(5) != Q)
```

where `Q` is the quote. That is the four-/five-quote exception: a run that looks like a closer is a character plus a closer when a fourth quote follows. It is not merely "not the start of `"""` / `'''`".

R4 (`DollarSlashyStringCharacter`)

Ordered, predicated alternatives:

- `$/$` (escaped closer) only if `LA(-4) != '$'` after the match;
- `$/` (escaped slash) only if `LA(1) != '$'`;
- `/` as a character only if `LA(1) != '$'`;
- `$` as a character only if R2 **fails**.

R5 (`VAL : 'val' {isValEnabled()}?`)

If `groovy.val.enabled` is `false` (default `true`), `val` is `Identifier`, not `VAL`.

R6 (`NOT_INSTANCEOF`, `NOT_IN`)

`!instanceof` is one token only if followed by space / tab / CR / LF. `!in` is one token only if followed by space / tab / CR / LF / `[` / `(` / `{`. Otherwise they are `NOT` plus `INSTANCEOF` / `IN` (`!internal` is not `NOT_IN`).

R7 (`CapitalizedIdentifier` / `Identifier`, `JavaLetter`, `JavaLetterOrDigit`):: ASCII is the DFA split `[A-Z]` vs `[a-z$_]`. BMP non-ASCII starts and continuations require `Character.isJavaIdentifierStart` / `Part` **and not** `Character.isIdentifierIgnorable`. Supplementary-plane letters use the decoded code point (GROOVY-12398) and test start/part plus case only — they do **not** test identifier-ignorable.

R8 (`JavaLetterInGString`)

GString identifiers cannot contain `$`.

R9 (`isInsideParens` / `ignoreTokenInsideParens`)

Newline hiding as in Lexical structure.

R10 (`ignoreMultiLineCommentConditionally`)

Multiline-comment channel as in Lexical structure.

R11 (SH_COMMENT)

`!` is skipped when tokenIndex == 0. Skipped WS does not increment tokenIndex; a prior default-channel token (including NL) does. This is not "the first character of the file must be `#!`".

Parser disambiguation and context gates

R12 (isInvalidMethodDeclaration on scriptStatement)

A script-level methodDeclaration is refused when the next two tokens are (Identifier | CapitalizedIdentifier | StringLiteral | YIELD) immediately followed by '('. That input is a call (or yield(…)), parsed through statement.

R13 (isInvalidLocalVariableDeclaration)

A localVariableDeclaration is refused when the upcoming tokens look like a command rather than a typed local. The predicate:

- walks a dotted name (foo.Bar) and then inspects the last segment;
- treats the construct as a local if that segment is a primitive, a modifier, starts with an uppercase code point, is @ (unless it is an annotated for / while / do), or is followed by '=' / '<' / '[' at the documented LT offsets (string x = 1 is a local; string x is a command).

Consequence: String x and java.lang.String x are locals; string x and java.lang.string x are commands.

R14 (isIdentifierAssign on elementValues)

@Foo(a = 1) is elementValuePairs only. A single elementValue is not attempted when LT(1) is in FIRST(elementValuePairName) and LT(2) is '=' (GROOVY-12398 / AdaptivePredict).

R15 (classBody { $t == 2 }?)

Enum bodies may start with enumConstants. Other classBody values may not.

R16 (methodDeclaration { $ct == 3 }?)

default annotation element values only when the enclosing type is @interface (ct == 3). Script methods use ct == 9 and do not get this alternative.

R17 (isFollowingArgumentsOrClosure)

In commandExpression, the optional argumentList after expression is absent if that expression is a postfix path whose last pathElement has t equal to 2 or 3 (already arguments or closureOrLambdaExpression).

R18 (pathExpression { LT(2) == DOT }? STATIC)

Bare static is a path head only when immediately followed by . (e.g. static.unused = { → }).

R19 (inSwitchExpressionLevel > 0)

yield expression is a statement only while parsing switchExpression.

R20 (inAsyncClosureLevel > 0)

defer statementExpression only while parsing the async closure/lambda alternative of expression.

R21 (`AstBuilder.visitExpressionListElement`)

`'*' expression` (spread) is parsed everywhere `expressionListElement` occurs; **rejected** in `forInit` / `forUpdate` (`canSpread == false`). This is conversion, not recognition.

`<assoc=right>` on `=⇒`, ternary / elvis, and assignment is CFG disambiguation of an ambiguous grammar, recorded in the precedence table, not numbered here.

Compilation units

The parser start rule is `compilationUnit`. One compilation unit is one source file: optional package, then a sequence of script statements, then end of file. Leading `NL` tokens (blank lines, and comments rewritten to `NL`) are permitted before the package.

```
compilationUnit ::=
    NL* (packageDeclaration sep?)? scriptStatements? EOF

scriptStatements ::=
    scriptStatement (sep scriptStatement)* sep?

scriptStatement ::=
      importDeclaration
    | typeDeclaration
    | methodDeclaration                 /* R12; g4 call is methodDeclaration[3, 9] */
    | statement

packageDeclaration ::=
    annotationsOpt 'package' qualifiedName

importDeclaration ::=
    annotationsOpt 'import'
    (   'module' qualifiedName
      | 'static'? qualifiedName ('.' '*' | 'as' identifier)?
    )

sep ::=
    (NL | ';')+
```

A compilation unit is a **script** in the syntactic sense: it may mix imports, type declarations, script-level methods, and statements in one list. Conversion wraps script statements in a generated class; that wrapping is not syntax. There is no Java-style "one public type per compilation unit" production. JEP 445 compatible-script **shape** is conversion, not a second start rule.

Package and import

If a package declaration is present it is the first non-`NL` construct; the CFG already enforces that.

`importDeclaration` has two branches:

- `import module qualifiedName` — JEP 476-style module import. The g4 does not admit `import`

`module p.*` or `import module p as x`.

- `import static? qualifiedName` with optional `.*` or `as identifier`.

`'module'` is token `MODULE` and is also an alternative of `identifier`.

Default imports (`java.lang.*`, ...) are inserted later. They are not productions.

Package names may contain `'def' 'in' 'as' 'trait'` (`qualifiedNameElement`) for integration with existing Java packages.

Script-level methods (R12)

The g4 writes the method alternative as `{ !SemanticPredicates.isInvalidMethodDeclaration(_input) }? methodDeclaration[3, 9]`.

R12: if the next two tokens are a name or string (or `yield`) immediately followed by `'('`, the alternative is not taken.

The parameters `[3, 9]` are not EBNF. `9` records "enclosing type is script" for conversion. The `default` annotation-element alternative is gated by R16 and does **not** apply to script methods.

A script method without a body is accepted by the CFG (the body is optional) and rejected by `AstBuilder`. That check is conversion, not R12.

Newlines between script statements

Adjacent `scriptStatement`s require `sep`: at least one visible newline or semicolon. Two statements on one line need `';'`. A trailing `sep` before `EOF` is allowed.

Inside `'('`, `'['`, or `'?['`, `NL` is hidden (R9), so those newlines do not satisfy `sep`. Inside `'{'` they do. The try-with-resources list uses `sep` between resources **because** `try (` is exempt from R9.

Types and names

```
qualifiedName ::=
    qualifiedNameElement ('.' qualifiedNameElement)*

qualifiedNameElement ::=
    identifier | 'def' | 'in' | 'as' | 'trait'

qualifiedNameElements ::=
    (qualifiedNameElement '.')*

qualifiedClassName ::=
    qualifiedNameElements identifier

qualifiedStandardClassName ::=
    qualifiedNameElements className ('.' className)*

className ::=
```

```
    CapitalizedIdentifier

identifier ::=
      Identifier
    | CapitalizedIdentifier
    | 'as' | 'async' | 'await' | 'defer' | 'in' | 'module'
    | 'permits' | 'record' | 'sealed' | 'trait' | 'val' | 'var' | 'yield'

keywords ::=
      'abstract' | 'as' | 'assert' | 'async' | 'await' | 'break'
    | 'case' | 'catch' | 'class' | 'const' | 'continue' | 'def'
    | 'default' | 'defer' | 'do' | 'else' | 'enum' | 'extends'
    | 'final' | 'finally' | 'for' | 'goto' | 'if' | 'implements'
    | 'import' | 'in' | 'instanceof' | 'interface' | 'native'
    | 'new' | 'non-sealed' | 'package' | 'permits' | 'record'
    | 'return' | 'sealed' | 'static' | 'strictfp' | 'super'
    | 'switch' | 'synchronized' | 'this' | 'throw' | 'throws'
    | 'transient' | 'trait' | 'threadsafe' | 'try' | 'val' | 'var'
    | 'volatile' | 'while' | 'yield'
    | NullLiteral | BooleanLiteral
    | BuiltInPrimitiveType | 'void'
    | 'public' | 'protected' | 'private'

builtInType ::=
    BuiltInPrimitiveType | 'void'

type ::=
    annotationsOpt
    ( 'void'                          /* parsed; rejected at conversion */
    | primitiveType
    | referenceType
    )
    dim0*

primitiveType ::=
    BuiltInPrimitiveType

referenceType ::=                     /* GROOVY-12319 rare types */
    qualifiedClassName
    (typeArguments ('.' identifier typeArguments?)*)?

standardType ::=                      /* restricted form of type */
    annotationsOpt
    ( primitiveType | standardClassOrInterfaceType )
    dim0*

standardClassOrInterfaceType ::=      /* restricted form of referenceType */
    qualifiedStandardClassName
    (typeArguments ('.' className typeArguments?)*)?

matchingType ::=                      /* instanceof pattern (JEP 394) */
```

```
    standardType identifier?

notInstanceofType ::=
    matchingType
  | castParExpression

typeParameters ::=
    '<' NL* typeParameter (',' NL* typeParameter)* NL* '>'

typeParameter ::=
    annotationsOpt className ('extends' NL* typeBound)?

typeBound ::=
    type ('&' NL* type)*

typeList ::=
    type (',' NL* type)*

typeArguments ::=
    '<' NL* typeArgument (',' NL* typeArgument)* NL* '>'

typeArgument ::=
    type
  | annotationsOpt '?' (('extends' | 'super') NL* type)?

typeArgumentsOrDiamond ::=
    '<' '>'
  | typeArguments

nonWildcardTypeArguments ::=
    '<' NL* typeList NL* '>'

dim0 ::=
    annotationsOpt '[' ']'

dim1 ::=
    annotationsOpt '[' expression ']'

intersectionType ::=
    type ('&' NL* type)*

castParExpression ::=
    '(' intersectionType ')'

coercionType ::=
    castParExpression                  /* (T) or (A & B & ...) */
  | type

annotatedQualifiedClassName ::=
    annotationsOpt qualifiedClassName
```

```
qualifiedClassNameList ::=
    annotatedQualifiedClassName (',' NL* annotatedQualifiedClassName)*
```

`'module'` is in `identifier` and **not** in `keywords`. Intersection types appear in **casts and `as`**, not as general types. `void` as a value type is parsed (`type` has a `'void'` alternative marked error in the g4) and rejected later.

`instanceof` takes `matchingType` (optional pattern variable). `!instanceof` takes `notInstanceofType`; conversion rejects intersection on `!instanceof`.

Declarations

Modifiers

```
typeDeclaration ::=
    classOrInterfaceModifiersOpt classDeclaration

classOrInterfaceModifiersOpt ::=
    (classOrInterfaceModifiers NL*)?

classOrInterfaceModifiers ::=
    classOrInterfaceModifier (NL* classOrInterfaceModifier)*

classOrInterfaceModifier ::=
      annotation
    | 'public' | 'protected' | 'private' | 'static' | 'abstract'
    | 'sealed' | 'non-sealed'
    | 'final'                         /* class only, semantically */
    | 'strictfp'
    | 'default'                       /* interface only, semantically */

modifier ::=
      classOrInterfaceModifier
    | 'native' | 'synchronized' | 'transient' | 'volatile'
    | 'def' | 'val' | 'var'

modifiersOpt ::= (modifiers NL*)?
modifiers    ::= modifier (NL* modifier)*

variableModifier ::=
      annotation
    | 'final' | 'def' | 'val' | 'var'
    | 'public' | 'protected' | 'private' | 'static' | 'abstract' | 'strictfp'

variableModifiersOpt ::= (variableModifiers NL*)?
variableModifiers    ::= variableModifier (NL* variableModifier)*
```

Repeat modifiers, illegal combinations (`sealed` + `non-sealed`, `sealed` + `final`, `permits` without `sealed`), and "visibility implies field not property" are conversion / later phases.

Type declarations

One production `classDeclaration` with local `t` set from the keyword:

```
classDeclaration ::=
    ( 'class' | 'interface' | 'enum' | '@' 'interface' | 'trait' | 'record' )
    identifier
    (NL* typeParameters)?
    (NL* formalParameters)?           /* record header; illegal for others at
conversion */
    (NL* 'extends'    NL* typeList)?
    (NL* 'implements' NL* typeList)?
    (NL* 'permits'    NL* typeList)?
    NL* classBody                     /* R15: enum vs others */
```

t	Keyword	Body
0	`'class'`	`classBody` without `enumConstants`
1	`'interface'`	same
2	`'enum'`	R15: may start with `enumConstants`
3	`'@' 'interface'`	same as class; methods may have `default` values (R16)
4	`'trait'`	Groovy-only keyword
5	`'record'`	header via optional `formalParameters`

`@Sealed` / `@RecordType` / `@Trait` **annotation** styles are annotations plus AST transforms, not additional keywords.

```
classBody ::=
    '{' NL*
    (
        enumConstants                 /* R15, t == 2 */
        ( (NL* ',')?
        | ((NL* ',')? NL* ';')? NL*
          classBodyDeclaration (sep classBodyDeclaration)*
        )
      | (classBodyDeclaration (sep classBodyDeclaration)*)?
    )
    sep? '}'

enumConstants ::=
    enumConstant (NL* ',' NL* enumConstant)*

enumConstant ::=
    annotationsOpt identifier arguments? anonymousInnerClassDeclaration?

classBodyDeclaration ::=
```

```
      ('static' NL*)? block           /* instance / static initializer */
    | memberDeclaration

memberDeclaration ::=
      methodDeclaration
    | fieldDeclaration
    | modifiersOpt (classDeclaration | compactConstructorDeclaration)
```

Nested types are `modifiersOpt classDeclaration`. Anonymous inner classes use `classBody` with `t == 0` regardless of the ANTLR parameter on `anonymousInnerClassDeclaration`.

Methods, constructors, fields

```
methodDeclaration ::=
    modifiersOpt typeParameters? (returnType NL*)?
    methodName formalParameters
    (   ('default' NL* elementValue)   /* R16: @interface only */
      | NL* 'throws' NL* qualifiedClassNameList (NL* methodBody)?
      | NL* methodBody
    )?

compactConstructorDeclaration ::=
    methodName NL* methodBody          /* no parameter list */

methodName ::=
    identifier | stringLiteral

returnType ::=
    standardType | 'void'

fieldDeclaration ::=
    variableDeclaration

variableDeclaration ::=
      modifiers NL*
      ( type? variableDeclarators
      | typeNamePairs NL* '=' NL* variableInitializer
      )
    | type variableDeclarators

variableDeclarators ::=
    variableDeclarator (',' NL* variableDeclarator)*

variableDeclarator ::=
    variableDeclaratorId (NL* '=' NL* variableInitializer)?

variableDeclaratorId ::=
    identifier

variableInitializer ::=
```

```
    enhancedStatementExpression

formalParameters ::=
    '(' formalParameterList? ')'

formalParameterList ::=
    (formalParameter | thisFormalParameter) (',' NL* formalParameter)*

thisFormalParameter ::=
    type 'this'

formalParameter ::=
    variableModifiersOpt type? '...'? variableDeclaratorId
    (NL* '=' NL* expression)?

methodBody ::=
    block
```

There is **no** separate constructor production. A constructor is a `methodDeclaration` with no `returnType`, name equal to the class name, and a body — those checks are `AstBuilder`. Compact constructors are syntax; "records only, name equals record name" is conversion.

Properties are **not** a production. The same field production is a property at conversion when it has no visibility modifier.

Default argument values, optional types, and varargs are syntax. Duplicate / non-last varargs, `val` /`var` as a return type, and script abstract methods are conversion.

Annotations

```
annotationsOpt ::=
    (annotation (NL* annotation)* NL*)?

annotation ::=
    '@' annotationName (NL* '(' elementValues? ')')?

annotationName ::=
    qualifiedClassName

elementValues ::=
      elementValuePairs
    | elementValue                      /* R14 */

elementValuePairs ::=
    elementValuePair (',' elementValuePair)*

elementValuePair ::=
    elementValuePairName NL* '=' NL* elementValue

elementValuePairName ::=
```

```
    identifier | keywords

elementValue ::=
      elementValueArrayInitializer
    | annotation
    | expression

elementValueArrayInitializer ::=
      '[' (elementValue (',' elementValue)* ','?)? ']'
    | '{' (elementValue ',')+ elementValue? '}'   /* at least one comma, vs closure */
```

`elementValue` is a general `expression` (closures as annotation values are syntax). Inside annotation parentheses, newlines are already hidden (R9), so R14 can look at `LT(2)`.

Statements

```
block ::=
    '{' sep? blockStatementsOpt '}'

blockStatementsOpt ::=
    blockStatements?

blockStatements ::=
    blockStatement (sep blockStatement)* sep?

blockStatement ::=
    statement

statement ::=
      block
    | conditionalStatement
    | loopStatement
    | tryCatchStatement
    | 'synchronized' expressionInPar NL* block
    | 'return' expression?
    | 'throw' expression
    | breakStatement
    | continueStatement
    | yieldStatement                    /* R19 */
    | 'yield' 'return' NL* expression   /* always; generator */
    | 'defer' NL* statementExpression   /* R20 */
    | identifier ':' NL* statement
    | assertStatement
    | localVariableDeclaration          /* R13 */
    | statementExpression
    | ';'

conditionalStatement ::=
    ifElseStatement | switchStatement
```

```
ifElseStatement ::=
    'if' expressionInPar NL* statement
    ((NL* | sep) 'else' NL* statement)?

switchStatement ::=
    'switch' expressionInPar NL*
    '{' NL* (switchBlockStatementGroup+ NL*)? '}'

switchBlockStatementGroup ::=
    switchLabel (NL* switchLabel)* NL* blockStatements

switchLabel ::=
      'case' expression ':'
    | 'default' ':'

loopStatement ::=
      annotationsOpt 'for' 'await'? '(' forControl ')' NL* statement
    | annotationsOpt 'while' expressionInPar NL* statement
    | annotationsOpt 'do' NL* statement NL* 'while' expressionInPar

forControl ::=
    enhancedForControl | originalForControl

enhancedForControl ::=
    (indexVariable ',')? variableModifiersOpt type? identifier
    (':' | 'in') expression

indexVariable ::=
    (BuiltInPrimitiveType | 'def' | 'val' | 'var')? identifier

originalForControl ::=
    forInit? ';' expression? ';' forUpdate?

forInit ::=
    localVariableDeclaration | expressionList   /* canSpread false: R21 */

forUpdate ::=
    expressionList                              /* canSpread false: R21 */

continueStatement ::= 'continue' identifier?
breakStatement    ::= 'break'    identifier?
yieldStatement    ::= 'yield' expression

tryCatchStatement ::=
    'try' resources? NL* block
    (NL* catchClause)*
    (NL* finallyBlock)?

resources ::=
    '(' NL* resourceList sep? ')'
```

```
resourceList ::=
    resource (sep resource)*

resource ::=
    localVariableDeclaration | expression

catchClause ::=
    'catch' '(' variableModifiersOpt catchType? identifier ')' NL* block

catchType ::=
    qualifiedClassName ('|' qualifiedClassName)*

finallyBlock ::=
    'finally' NL* block

assertStatement ::=
    'assert' expression (NL* (':' | ',') NL* expression)?

localVariableDeclaration ::=
    variableDeclaration                /* R13 */

typeNamePairs ::=
    '(' ( typeNamePair (',' typeNamePair)*
        | keyedPair    (',' keyedPair)*
        ) ')'

typeNamePair ::=
    ('def' | 'val' | 'var' | type)? '*'? variableDeclaratorId

keyedPair ::=
    identifier ':' ('def' | 'val' | 'var' | type)? variableDeclaratorId

variableNames ::=
    '(' variableDeclaratorId (',' variableDeclaratorId)+ ')'
```

Switch **statement** labels are `case expression ':'` / `default ':'` only — no arrow, no `expressionList`, no `yield`. Switch **expressions** are a different nonterminal (Expressions).

`yield expression` is gated by R19. `yield return expression` is a **separate**, always-available alternative and must not be folded into `yieldStatement`.

`defer` is gated by R20. `for await` is syntax (`'await'?` on `for`). Loop annotations (`annotationsOpt` on `for`/`while`/`do`) are syntax; R13 keeps `@Ann for (` as a loop.

`assert` allows `:` **or** `,` before the message. `try { }` with no catch, finally, or resources is parsed and rejected later. TWR resources are separated by `sep` (newline or `;`), not only `;`.

`variableNames` requires **at least two** identifiers. `(a) = ⋯` is ordinary assignment; nested `a` is rejected at conversion.

Expressions

Two facts must not be simplified away:

- The CFG of operators is the left-recursive rule `expression`. ANTLR binds **earlier alternatives tighter**.
- Every **statement** expression is a `commandExpression`. Command syntax is not a side dialect.

Compact form

The alternatives below are those of `expression` in `GroovyParser.g4`, in the same order. Shift and assignment operators are inlined (there are no `LSHIFT` / `RSHIFT` / `URSHIFT` lexer tokens).

```
expression ::=
      castParExpression castOperandExpression          /* cast, tightest */
    | 'async' NL* closureOrLambdaExpression            /* R20 enters */
    | 'await' NL*
          ( '(' expression (',' NL* expression)* ')'
          | expression (',' NL* expression)*
          )
    | postfixExpression
    | switchExpression                                 /* R19 enters */
    | ('~' | '!') NL* expression
    | expression '**' NL* expression                   /* left-assoc */
    | ('++' | '--' | '+' | '-') expression
    | expression NL* ('*' | DIV | '%') NL* expression
    | expression ('+' | '-') NL* expression            /* no NL before +/□ */
    | expression NL*
          ('<' '<' | '>' '>' '>' | '>' '>'
          | '..' | '<..' | '..<' | '<..<')
          NL* expression
    | expression NL* 'instanceof' NL* matchingType
    | expression NL* '!instanceof' NL* notInstanceofType
    | expression NL* 'as' NL* coercionType
    | expression NL* ('<=' | '>=' | '>' | '<' | 'in' | '!in')
                NL* expression
    | expression NL*
          ('===' | '!==' | '==' | '!=' | '<=>') NL* expression
    | expression NL* ('=~' | '==~') NL* expression
    | expression NL* '&'  NL* expression
    | expression NL* '^'  NL* expression
    | expression NL* '|'  NL* expression
    | expression NL* '&&' NL* expression
    | expression NL* '||' NL* expression
    | expression NL* '==>' NL* expression              /* right-assoc */
    | expression NL*
          ( '?' NL* expression NL* ':' NL* | '?:' NL* )
          expression                                   /* right-assoc */
    | variableNames NL* '=' NL* statementExpression    /* prefix; right-assoc */
    | expression NL*
```

```
          ('=' | '+=' | '-=' | '*=' | '/=' | '&=' | '|=' | '^='
          | '>>=' | '>>>=' | '<<=' | '%=' | '**=' | '?=')
          NL* enhancedStatementExpression              /* right-assoc */

castOperandExpression ::=              /* restricted form of expression */
      castParExpression castOperandExpression
    | postfixExpression
    | ('~' | '!') NL* castOperandExpression
    | ('++' | '--' | '+' | '-') castOperandExpression

postfixExpression ::=
    pathExpression ('++' | '--')?

statementExpression ::=
    commandExpression

enhancedExpression ::=
    expression | standardLambdaExpression

enhancedStatementExpression ::=
    statementExpression | standardLambdaExpression

expressionInPar ::=
    '(' enhancedStatementExpression ')'

parExpression ::=
    expressionInPar

expressionList ::=
    expressionListElement (',' NL* expressionListElement)*

expressionListElement ::=
    '*'? expression                    /* R21: spread rejected in forInit/forUpdate */
```

`DIV` is the division token, spelling `'/'`. It is not written `'/'` here because that character also opens slashy strings (R1).

`castOperandExpression` exists so `(int) a + b` is `((int) a) + b` and `(int) ++x` is a cast of a prefix.

`await` takes a full `expression` (or a parenthesised list). So `await a + b` is `await (a + b)`.

Precedence and associativity

Derived from the alternative **order** of `expression` as implemented by the generated `expression(int)` method, not copied from the tutorial table in `core-operators.adoc`. Rows are tightest first. The column "Level (g4 comment)" repeats the comments in the g4; the **(atom)** row is tighter than that numbering's "level 1".

Because `instanceof`, `!instanceof`, `as`, and comparisons are **separate** alternatives, ANTLR assigns them distinct numeric precedences (tighter in that order). The g4 comments still call the group

"level 7"; the table keeps that grouping and notes the split.

Level (g4 comment)	Operators / forms	Alternative	Assoc.
(atom)	`(T) e`, `async { }`, `await e`, path/postfix, `switch (e) { }`	`#castExprAlt` `#asyncClosureExprAlt` `#awaitExprAlt` `#postfixExprAlt` `#switchExprAlt`	n/a
1	`~ !`	`#unaryNotExprAlt`	prefix
2	`**`	`#powerExprAlt`	**left** (not marked `<assoc=right>`)
3	prefix `+` `` `--` `` `-`	`#unaryAddExprAlt`	prefix
4	`* / %`	`#multiplicativeExprAlt`	left
5	`+ -`	`#additiveExprAlt`	left
6	`<< >> >>> .. ..< <..< <..`	`#shiftExprAlt`	left
7	`instanceof`, then `!instanceof`, then `as`, then `< <= > >= in !in`	`#relationalExprAlt` (four alts; earlier is tighter)	left / type RHS
8	`== != <⇒ === !==`	`#equalityExprAlt`	left
8.5	`=~ ==~`	`#regexExprAlt`	left
9	`&`	`#andExprAlt`	left
10	`^`	`#exclusiveOrExprAlt`	left
11	`|`	`#inclusiveOrExprAlt`	left
12	`&&`	`#logicalAndExprAlt`	left
13	`||`	`#logicalOrExprAlt`	left
13.5	`=⇒`	`#implicationExprAlt`	**right**
14	`? : ?:`	`#conditionalExprAlt`	**right**
15	assignment operators	`#assignmentExprAlt` (`_prec(1)`; RHS is `enhancedStatementExpression`)	**right**
(prefix)	`(a, b) = ⋯`	`#multipleAssignmentExprAlt` (not left-recursive; parsed as a prefix alternative of `expression`, like cast)	n/a

Notes the tutorial table does not make:

- `sits between ! and unary minus: !a  b` is `(!a)  b`; `-a  b` is `-(a ** b)` (Python-style). The g4 comments call these levels 1, 2, 3.
- `is` left**-associative in this grammar.

- Path operators `. ?. ??. *. .& :: .@ () {} [] ?[]` live in `pathElement`, inside `postfixExpression`, tighter than the table above.
- Newline **before** `+`, `-`, or `* is *not` part of those operators (`NL*` is only on the right). At script top level that newline is `sep` and the next line starts a new `statement`. Newline before `, /, comparison, =, ⋯ *is` allowed when the `NL` is visible.

Path expressions

```
pathExpression ::=
    ( primary
    | 'static'                          /* R18: only when next token is '.' */
    )
    pathElement*

pathElement ::=
      NL*
      ( '.' NL*
            ( 'new' nonWildcardTypeArguments? creator
            | ('@' | nonWildcardTypeArguments)? namePart
            )
        | ( '*.' | '?.' | '??.' ) NL* ('@' | nonWildcardTypeArguments)? namePart
        | '.&' NL* namePart
        | '::' NL* nonWildcardTypeArguments? namePart
        | closureOrLambdaExpression
      )
    | arguments
    | indexPropertyArgs
    | namedPropertyArgs

namePart ::=
    identifier | stringLiteral | dynamicMemberName | keywords

dynamicMemberName ::=
    parExpression | gstring

indexPropertyArgs ::=
    ('?[' | '[') expressionList? ']'

namedPropertyArgs ::=
    ('?[' | '[') (namedPropertyArgList | ':') ']'
```

`pathElement t` codes (CST tags for R17 / conversion, not EBNF parameters): 1 namePart, 2 arguments, 3 closure/lambda, 4 index, 5 named index, 6 `outer.new Inner(⋯)`.

`arguments` / index / named index do **not** have a leading `NL*` in this production. A newline before `(` at script level is `sep`, not a call. Inside grouping, that newline is hidden (R9).

The unimplemented proposal `a.{foo}` as inline `with` is a comment in the g4, not a production.

Primaries, lists, maps, creation

```
primary ::=
      identifier typeArguments?
    | literal
    | gstring
    | 'new' NL* nonWildcardTypeArguments? creator
    | nonWildcardTypeArguments? 'this'
    | nonWildcardTypeArguments? 'super'
    | parExpression
    | closureOrLambdaExpression
    | list
    | map
    | builtInType

literal ::=
    IntegerLiteral | FloatingPointLiteral | stringLiteral
  | BooleanLiteral | NullLiteral

stringLiteral ::=
    StringLiteral

list ::=
    '[' expressionList? ','? ']'

map ::=
    '[' (mapEntryList ','? | ':') ']'

mapEntryList ::=
    mapEntry (',' mapEntry)*

mapEntry ::=
      mapEntryLabel ':' NL* enhancedExpression
    | '*' ':' NL* enhancedExpression

mapEntryLabel ::=
    keywords | primary

creator ::=
    createdName
    (   NL* arguments anonymousInnerClassDeclaration?
      | dim0+ NL* arrayInitializer
      | dim1+ dim0*
    )

createdName ::=
    annotationsOpt
    ( primitiveType
    | qualifiedClassName
      (typeArgumentsOrDiamond ('.' identifier typeArgumentsOrDiamond?)*)?
    )
```

```
arrayInitializer ::=
    '{' NL*
    ( (arrayInitializer | variableInitializer) NL*
      (',' NL* (arrayInitializer | variableInitializer) NL*)*
    )?
    ','? NL* '}'

anonymousInnerClassDeclaration ::=
    classBody

arguments ::=
    '(' enhancedArgumentListInPar? ','? ')'

enhancedArgumentListInPar ::=
    enhancedArgumentListElement
    (',' NL* enhancedArgumentListElement)*

enhancedArgumentListElement ::=
      expressionListElement
    | standardLambdaExpression
    | namedPropertyArg

argumentList ::=                        /* command / chain args; restricted form */
    firstArgumentListElement
    (',' NL* argumentListElement)*

firstArgumentListElement ::=
    expressionListElement | namedArg

argumentListElement ::=
    expressionListElement | namedPropertyArg

namedArg ::=
      namedArgLabel ':' NL* enhancedExpression
    | '*' ':' NL* enhancedExpression

namedPropertyArg ::=
      namedPropertyArgLabel ':' NL* enhancedExpression
    | '*' ':' NL* enhancedExpression

namedArgLabel ::=
    keywords | namedArgPrimary

namedPropertyArgLabel ::=
    keywords | namedPropertyArgPrimary

namedPropertyArgList ::=
    namedPropertyArg (',' namedPropertyArg)*

namedArgPrimary ::=
```

```
    identifier | literal | gstring

namedPropertyArgPrimary ::=
    identifier | literal | gstring | parExpression | list | map
```

Trailing commas are allowed in lists, maps, and argument lists. `[:]` is the empty map. `* :` is map spread.

Command expressions (R17)

```
commandExpression ::=
    expression
    argumentList?                       /* R17 */
    commandArgument*

commandArgument ::=
    commandPrimary (pathElement+ | argumentList)?

commandPrimary ::=                      /* restricted primary */
    identifier | literal | gstring
```

Every statement expression is a `commandExpression`: an `expression` followed by an optional `argumentList` (R17) and zero or more `commandArgument`s. A bare `foo` is therefore a `commandExpression` with both tails empty. Parentheses around a call may be omitted when R17 allows an `argumentList` (the callee is not already a `(…)` or `{…}` path element). Further words are a **command chain**:

- `a b c()` → `a(b).c()`
- `a b c()()` → `a(b).c().call()`
- `a b c[x]` → `a(b).c[x]`
- `a b c { x }` → `a(b).c({ x })`

R17: `foo(1) bar` does not treat `bar` as a second positional argument of `foo`; `foo 1 bar` may.

`commandPrimary` cannot be `new`, `this`, a parenthesised expression, a list, a map, or a closure. Those remain ordinary `expression` heads.

Parenthesised `arguments` may contain `standardLambdaExpression`; command `argumentList` may not (only `expressionListElement` / `namedArg` on the first element).

Closures and lambdas

```
closure ::=
    '{' (NL* (formalParameterList NL*)? '->')? sep? blockStatementsOpt '}'

lambdaExpression ::=                    /* Groovy lambda: parentheses required */
    lambdaParameters NL* '->' NL* lambdaBody
```

```
lambdaParameters ::=
    formalParameters                    /* single id without () is NOT here */

standardLambdaExpression ::=
    standardLambdaParameters NL* '->' NL* lambdaBody

standardLambdaParameters ::=
    formalParameters | variableDeclaratorId

lambdaBody ::=
    block | statementExpression

closureOrLambdaExpression ::=
    closure | lambdaExpression
```

`{ a → a * 2 }` is a **closure**. The g4 comments out a `variableDeclaratorId` alternative on `lambdaParameters` so that form is not stolen from closures (GROOVY-8991). Java-style `(params) → body` and `x → body` are `standardLambdaExpression`, used in parenthesised argument lists and as `enhancedStatementExpression`.

Switch expressions

```
switchExpression ::=
    'switch' expressionInPar NL*
    '{' NL* switchBlockStatementExpressionGroup* NL* '}'

switchBlockStatementExpressionGroup ::=
    (switchExpressionLabel NL*)+ blockStatements

switchExpressionLabel ::=
    ('case' expressionList | 'default') ('->' | ':')
```

While this production is active, R19 allows `yield expression`. `return` / `break` / `continue` inside a switch expression are conversion errors.

Which production matches is a matter of **context and labels**. `switchStatement` (under `conditionalStatement`) admits only `case expression ':'` / `default ':'`. Arrow labels exist only on `switchExpression`. An arrow `switch` in statement position is therefore `statementExpression` → `switchExpression`, not `switchStatement`.

GString in the parser

Parser `gstring` does not know the quote style. Interpolation is `gstringPath` or `closure` only — not an arbitrary `expression` nonterminal. `gstringPath` cannot include a call: `"$foo.bar()"` is a path `foo.bar` followed by the characters `()`; `"$\{foo.bar()}"` is a closure. Conversion may unwrap a no-arrow single-expression closure.

Parameterised-rule bijection

g4 rule	Parameter	Used in recognition?	Published form
`classDeclaration`	locals `t` 0..5	sets `t` from keyword	one production; keyword choice is syntax
`classBody[t]`	`t == 2` gates `enumConstants`	**yes** (R15)	one production with R15
`classBodyDeclaration[t]` `memberDeclaration[t]`	passed through	only to `methodDeclaration[0, t]`	one production; `t` is AST/context
`methodDeclaration[t, ct]`	`ct == 3` gates `default`	**`ct` yes (R16); `t` no**	one production with R16. Script call is `[3, 9]`
`returnType[ct]`	unused in the production	no	one production; parameter dropped
`variableDeclaration[t]`	unused (0 local / 1 field is `AstBuilder`)	no	one production
`creator[t]` `anonymousInnerClassDeclaration[t]`	unused in the production	no	one production
`expressionList[canSpread]` `expressionListElement[canSpread]`	`MUL?` always parsed	conversion (R21)	`'*'? expression`
`pathExpression` / `pathElement returns [int t]`	CST tag	R17 reads `t`	not an EBNF parameter

Restricted forms (`baseContext`)

ANTLR `options { baseContext = X; }` shares a generated context class. Each is still a distinct production and is listed above:

Rule	Base	Drops / tightens
`standardType`	`type`	no `'void'` alternative; uses `standardClassOrInterfaceType`
`standardClassOrInterfaceType`	`referenceType`	`className` (`CapitalizedIdentifier`) instead of `identifier`
`lambdaExpression`	`standardLambdaExpression`	parameters must be `formalParameters`
`lambdaParameters`	`standardLambdaParameters`	no single `variableDeclaratorId`
`castOperandExpression`	`expression`	only cast, postfix, unary not, unary add/sub
`commandPrimary`	`primary`	`identifier` `literal` `gstring` only

Rule	Base	Drops / tightens
`namedArgPrimary`	`primary`	identifier / literal / gstring
`namedPropertyArgPrimary`	`primary`	plus parExpression / list / map
`namedPropertyArgList`	`mapEntryList`	`namedPropertyArg` elements
`namedPropertyArg` `namedArg`	`mapEntry`	corresponding labels
`namedPropertyArgLabel` `namedArgLabel`	`mapEntryLabel`	corresponding primaries
`argumentList`	`enhancedArgumentListInPar`	no `standardLambdaExpression`; first element may be `namedArg`
`firstArgumentListElement` `argumentListElement`	`enhancedArgumentListElement`	as above

Syntactic over-approximation

The CFG accepts programs that conversion or later phases reject. Representative cases (not an exhaustive semantic-analysis list):

- `'void'` as a value type; `void[]`.
- `formalParameters` on non-records; `extends` / `implements` / `permits` / `typeParameters` on kinds that cannot have them; `var` / `val` as a type name.
- Methods without bodies in classes; abstract methods with bodies; annotation methods with a body or `void`; script abstract methods.
- Compact constructor outside a record / name mismatch.
- `this(…)` / `super(…)` not the first constructor statement (grammar allows them anywhere a path can appear).
- `try { }` with no catch, finally, or resources.
- Switch: more than one `default`; switch **expression** `return` / `break` / `continue`.
- Nested `a = …`; assignment LHS not variable / property / index.
- `!instanceof (A & B)`; non-reifiable `instanceof`.
- Spread `in forInit / forUpdate (R21). Spread is *recognised` wherever `expressionListElement` occurs, including `switchExpressionLabel` (`case *x →`); R21 conversion-checks only for-init/update.
- `for await` is not gated by `inAsyncClosureLevel` (only `defer` is R20).
- Command chains inside `arrayInitializer`.
- Primitive type used as a method name `int()`.

Invalid input and diagnostics

These are **not** language productions:

- Parser error alternatives for missing `)` / `]` / `}` — removed (GROOVY-9588).

`MissingDelimiterDiagnostic` may relocate the caret after recognition has already failed.

- `UNEXPECTED_CHAR` — unexpected character; an unexpected quote is reported as an unclosed string unless a scan-ahead finds an illegal escape in an otherwise closed literal.
- Invalid octal (`'0' [0-9]+` error alternative).
- Number ending with `_`.
- Unclosed `/*` (`requireUnclosedComment` at the opener).
- Shebang when `tokenIndex != 0` (R11).
- `GroovyLangLexer.nextToken`: EOF in `GSTRING_TYPE_SELECTOR_MODE` (`Illegal string body character after dollar sign`).
- `errorIgnored` IDE tokenisation.

Intentionally excluded

Not syntax (not in the g4 as productions of the language):

- AST transforms and their annotations as **language forms** (`@CompileStatic`, `@TypeChecked`, `@Field`, `@Sealed` as the **alternative** to the `sealed` keyword, …).
- Groovydoc tag semantics (lexing of `/**` is in; tags are not).
- Type checking, Groovy Truth, operator overloading, optional `return` of the last expression.
- Default imports.
- Runtime GString evaluation and `hashCode` coercion.
- Implicit closure parameter `it`.
- JEP 445 script classification.
- Trait **implementation** (the `trait` keyword is syntax; mixin semantics are a transform).
- Java `module-info` compilation units; Java text blocks as a distinct form (Groovy `"""` is triple-quoted String/GString); Java 21+ switch type / record / pattern labels and `when` guards (Groovy `case` is `expression` / `expressionList`).

Production index

Every named parser rule in `GroovyParser.g4` appears in this chapter. There is no `nls` rule. Lexer modes are listed under GString lexical contexts. Parser-visible tokens and fragments are in Terminal vocabulary and Lexical structure; rewritten lexer rules follow this index.

- Compilation units:: `compilationUnit`, `scriptStatements`, `scriptStatement`, `packageDeclaration`, `importDeclaration`, `sep`
- Types and names:: `qualifiedName`, `qualifiedNameElement`, `qualifiedNameElements`, `qualifiedClassName`, `qualifiedStandardClassName`, `className`, `identifier`, `keywords`, `builtInType`, `type`, `primitiveType`, `referenceType`, `standardType`, `standardClassOrInterfaceType`, `matchingType`, `notInstanceofType`, `typeParameters`, `typeParameter`, `typeBound`, `typeList`, `typeArguments`, `typeArgument`, `typeArgumentsOrDiamond`, `nonWildcardTypeArguments`, `dim0`, `dim1`, `intersectionType`, `castParExpression`, `coercionType`, `annotatedQualifiedClassName`, `qualifiedClassNameList`

- Declarations:: typeDeclaration, modifier, modifiersOpt, modifiers, classOrInterfaceModifiersOpt, classOrInterfaceModifiers, classOrInterfaceModifier, variableModifier, variableModifiersOpt, variableModifiers, classDeclaration, classBody, enumConstants, enumConstant, classBodyDeclaration, memberDeclaration, methodDeclaration, compactConstructorDeclaration, methodName, returnType, fieldDeclaration, variableDeclarators, variableDeclarator, variableDeclaratorId, variableInitializer, formalParameters, formalParameterList, thisFormalParameter, formalParameter, methodBody, annotationsOpt, annotation, elementValues, annotationName, elementValuePairs, elementValuePair, elementValuePairName, elementValue, elementValueArrayInitializer
- Statements:: block, blockStatementsOpt, blockStatements, blockStatement, statement, conditionalStatement, ifElseStatement, switchStatement, switchBlockStatementGroup, switchLabel, loopStatement, forControl, enhancedForControl, indexVariable, originalForControl, forInit, forUpdate, continueStatement, breakStatement, yieldStatement, tryCatchStatement, resources, resourceList, resource, catchClause, catchType, finallyBlock, assertStatement, localVariableDeclaration, variableDeclaration, typeNamePairs, typeNamePair, keyedPair, variableNames
- Expressions:: expression, castOperandExpression, postfixExpression, statementExpression, enhancedExpression, enhancedStatementExpression, expressionInPar, parExpression, expressionList, expressionListElement, pathExpression, pathElement, namePart, dynamicMemberName, indexPropertyArgs, namedPropertyArgs, primary, literal, stringLiteral, gstring, gstringValue, gstringPath, list, map, mapEntryList, mapEntry, mapEntryLabel, creator, createdName, arrayInitializer, anonymousInnerClassDeclaration, arguments, enhancedArgumentListInPar, enhancedArgumentListElement, argumentList, firstArgumentListElement, argumentListElement, namedArg, namedPropertyArg, namedArgLabel, namedPropertyArgLabel, namedPropertyArgList, namedArgPrimary, namedPropertyArgPrimary, commandExpression, commandArgument, commandPrimary, closure, lambdaExpression, lambdaParameters, standardLambdaExpression, standardLambdaParameters, lambdaBody, closureOrLambdaExpression, switchExpression, switchBlockStatementExpressionGroup, switchExpressionLabel

A unique-spelling keyword or operator `'foo'` is token `FOO`, except: `non-sealed` is `NON_SEALED`; the eight primitives are one token `BuiltInPrimitiveType`; `true`/`false` are `BooleanLiteral`; `null` is `NullLiteral`; `/` as an operator is `DIV`.

Lexer token names in `GroovyLexer.g4` (parser-visible, rewritten, skip, and diagnostic). This list is the coverage checklist; spellings and roles are in Terminal vocabulary and Lexical structure.

AS, DEF, IN, TRAIT, THREADSAFE, ASYNC, AWAIT, DEFER, BuiltInPrimitiveType, ABSTRACT, ASSERT, BREAK, CASE, CATCH, CLASS, CONST, CONTINUE, DEFAULT, DO, ELSE, ENUM, EXTENDS, FINAL, FINALLY, FOR, IF, GOTO, IMPLEMENTS, IMPORT, INSTANCEOF, INTERFACE, MODULE, NATIVE, NEW, NON_SEALED, PACKAGE, PERMITS, PRIVATE, PROTECTED, PUBLIC, RECORD, RETURN, SEALED, STATIC, STRICTFP, SUPER, SWITCH, SYNCHRONIZED, THIS, THROW, THROWS, TRANSIENT, TRY, VAL, VAR, VOID, VOLATILE, WHILE, YIELD, IntegerLiteral, FloatingPointLiteral, BooleanLiteral, NullLiteral, StringLiteral, GStringBegin, GStringPart, GStringEnd, GStringPathPart, TdqGStringBegin, TdqGStringPart, TdqGStringEnd, TdqGStringCharacter, SlashyGStringBegin, SlashyGStringPart, SlashyGStringEnd, SlashyGStringCharacter, DollarSlashyGStringBegin, DollarSlashyGStringPart, DollarSlashyGStringEnd, DollarSlashyGStringCharacter, GStringCharacter, GStringLBrace, GStringIdentifier, RollBackOne, RANGE_INCLUSIVE, RANGE_EXCLUSIVE_LEFT, RANGE_EXCLUSIVE_RIGHT, RANGE_EXCLUSIVE_FULL, SPREAD_DOT, SAFE_DOT, SAFE_INDEX, SAFE_CHAIN_DOT, ELVIS, METHOD_POINTER, METHOD_REFERENCE, REGEX_FIND, REGEX_MATCH, POWER, POWER_ASSIGN, SPACESHIP, IDENTICAL,

IMPLIES, NOT_IDENTICAL, ARROW, NOT_INSTANCEOF, NOT_IN, LPAREN, RPAREN, LBRACE, RBRACE, LBRACK, RBRACK, SEMI, COMMA, DOT, ASSIGN, GT, LT, NOT, BITNOT, QUESTION, COLON, EQUAL, LE, GE, NOTEQUAL, AND, OR, INC, DEC, ADD, SUB, MUL, DIV, BITAND, BITOR, XOR, MOD, ADD_ASSIGN, SUB_ASSIGN, MUL_ASSIGN, DIV_ASSIGN, AND_ASSIGN, OR_ASSIGN, XOR_ASSIGN, MOD_ASSIGN, LSHIFT_ASSIGN, RSHIFT_ASSIGN, URSHIFT_ASSIGN, ELVIS_ASSIGN, CapitalizedIdentifier, Identifier, AT, ELLIPSIS, WS, NL, ML_COMMENT, SL_COMMENT, SH_COMMENT, UNEXPECTED_CHAR.

Lexer rules that are rewritten before the parser sees them:

Rewritten and mode-local lexer rules (not parser terminals)

These names exist in GroovyLexer.g4 and are rewritten or accumulated; the parser never sees the original name.

Lexer rule	Fate
TdqGStringBegin SlashyGStringBegin DollarSlashyGStringBegin	→ type(GStringBegin) and pushMode of the matching body mode plus GSTRING_TYPE_SELECTOR_MODE
TdqGStringEnd SlashyGStringEnd DollarSlashyGStringEnd	→ type(GStringEnd), popMode
TdqGStringPart SlashyGStringPart DollarSlashyGStringPart	→ type(GStringPart), pushMode(GSTRING_TYPE_SELECTOR_MODE)
GStringCharacter TdqGStringCharacter SlashyGStringCharacter DollarSlashyGStringCharacter	→ more (glued into the following GStringPart / GStringEnd)
GStringLBrace	enterParen(); → type(LBRACE); popMode; pushMode(DEFAULT_MODE)
GStringIdentifier	→ type(Identifier); popMode; pushMode(GSTRING_PATH_MODE)
RollBackOne	popMode. Usually HIDDEN **and** rollbackOneChar(). At EOF after " or /, type(GStringEnd) and **no** rollback.

Lexer fragments

Every fragment used by a token:

DqStringCharacter, SqStringCharacter, TdqStringCharacter, TsqStringCharacter, SlashyStringCharacter, DollarSlashyStringCharacter, BOOLEAN, BYTE, CHAR, DOUBLE, FLOAT, INT, LONG, SHORT, DecimalIntegerLiteral, HexIntegerLiteral, OctalIntegerLiteral, BinaryIntegerLiteral, IntegerTypeSuffix, DecimalNumeral, Digits, Underscores, Underscore, HexNumeral, HexDigits, HexDigit, OctalNumeral, OctalDigits, BinaryNumeral, BinaryDigits, DecimalFloatingPointLiteral, ExponentPart, FloatTypeSuffix, HexadecimalFloatingPointLiteral, HexSignificand, BinaryExponent, EscapeSequence, OctalEscape, UnicodeEscape, DollarEscape, LineEscape, LineTerminator, SlashEscape, Backslash, Slash, Dollar, GStringQuotationMark, SqStringQuotationMark, TdqStringQuotationMark, TsqStringQuotationMark, DollarSlashyGStringQuotationMarkBegin, DollarSlashyGStringQuotationMarkEnd, DollarSlashEscape, DollarDollarEscape,

DollarSlashDollarEscape, IdentifierInGString, JavaLetter, JavaLetterInGString, JavaLetterOrDigit, JavaLetterOrDigitInGString, ShCommand.

The fragments used by tokens are defined in Lexical structure (JavaLetter, Digits, HexDigits, OctalEscape, UnicodeEscape, ShCommand, string-character fragments, ...). The remaining punctuation and primitive-spelling fragments are:

```
fragment GStringQuotationMark ::= '"'
fragment SqStringQuotationMark ::= "'"
fragment TdqStringQuotationMark ::= '"""'
fragment TsqStringQuotationMark ::= "'''"
fragment DollarSlashyGStringQuotationMarkBegin ::= '$/'
fragment DollarSlashyGStringQuotationMarkEnd   ::= '/$'
fragment DollarSlashEscape       ::= '$/'
fragment DollarDollarEscape      ::= '$$'
fragment DollarSlashDollarEscape ::= '$/$'
fragment DollarEscape            ::= '\\' '$'
fragment SlashEscape             ::= '\\' '/'
fragment Backslash               ::= '\\'
fragment Slash                   ::= '/'
fragment Dollar                  ::= '$'
fragment BOOLEAN ::= 'boolean'
fragment BYTE    ::= 'byte'
fragment CHAR    ::= 'char'
fragment DOUBLE  ::= 'double'
fragment FLOAT   ::= 'float'
fragment INT     ::= 'int'
fragment LONG    ::= 'long'
fragment SHORT   ::= 'short'
```

Operators

This chapter covers the operators of the Groovy programming language.

Arithmetic operators

Groovy supports the usual familiar arithmetic operators you find in mathematics and in other programming languages like Java. All the Java arithmetic operators are supported. Let's go through them in the following examples.

Normal arithmetic operators

The following binary arithmetic operators are available in Groovy:

Operator	Purpose	Remarks
+	addition	
-	subtraction	

Operator	Purpose	Remarks
*	multiplication	
/	division	Use `intdiv()` for integer division, and see the section about integer division for more information on the return type of the division.
%	remainder	
**	power	See the section about the power operation for more information on the return type of the operation.

Here are a few examples of usage of those operators:

```
assert  1  + 2 == 3
assert  4  - 3 == 1
assert  3  * 5 == 15
assert  3  / 2 == 1.5
assert 10  % 3 == 1
assert  2 ** 3 == 8
```

Unary operators

The + and - operators are also available as unary operators:

```
assert +3 == 3
assert -4 == 0 - 4

assert -(-1) == 1  ①
```

① Note the usage of parentheses to surround an expression to apply the unary minus to that surrounded expression.

In terms of unary arithmetics operators, the ++ (increment) and -- (decrement) operators are available, both in prefix and postfix notation:

```
def a = 2
def b = a++ * 3            ①

assert a == 3 && b == 6

def c = 3
def d = c-- * 2            ②

assert c == 2 && d == 6
```

```
def e = 1
def f = ++e + 3            ③

assert e == 2 && f == 5

def g = 4
def h = --g + 1            ④

assert g == 3 && h == 4
```

① The postfix increment will increment `a` after the expression has been evaluated and assigned into `b`

② The postfix decrement will decrement `c` after the expression has been evaluated and assigned into `d`

③ The prefix increment will increment `e` before the expression is evaluated and assigned into `f`

④ The prefix decrement will decrement `g` before the expression is evaluated and assigned into `h`

For the unary not operator on Booleans, see Conditional operators.

Assignment arithmetic operators

The binary arithmetic operators we have seen above are also available in an assignment form:

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`
- `**=`

Let's see them in action:

```
def a = 4
a += 3

assert a == 7

def b = 5
b -= 3

assert b == 2

def c = 5
c *= 3

assert c == 15
```

```
def d = 10
d /= 2

assert d == 5

def e = 10
e %= 3

assert e == 1

def f = 3
f **= 2

assert f == 9
```

Relational operators

Relational operators allow comparisons between objects, to know if two objects are the same or different, or if one is greater than, less than, or equal to the other.

The following operators are available:

Operator	Purpose
`==`	equal
`!=`	different
`<`	less than
`<=`	less than or equal
`>`	greater than
`>=`	greater than or equal
`===`	identical (Since Groovy 3.0.0)
`!==`	not identical (Since Groovy 3.0.0)

Here are some examples of simple number comparisons using these operators:

```
assert 1 + 2 == 3
assert 3 != 4

assert -2 < 3
assert 2 <= 2
assert 3 <= 4

assert 5 > 1
assert 5 >= -2
```

Both `===` and `!==` are supported which are the same as calling the `is()` method, and negating a call to the `is()` method respectively.

```
import groovy.transform.EqualsAndHashCode

@EqualsAndHashCode
class Creature { String type }

def cat = new Creature(type: 'cat')
def copyCat = cat
def lion = new Creature(type: 'cat')

assert cat.equals(lion) // Java logical equality
assert cat == lion      // Groovy shorthand operator

assert cat.is(copyCat)  // Groovy identity
assert cat === copyCat  // operator shorthand
assert cat !== lion     // negated operator shorthand
```

Logical operators

Groovy offers three logical operators for boolean expressions:

- `&&`: logical "and"
- `||`: logical "or"
- `!`: logical "not"

Let's illustrate them with the following examples:

```
assert !false           ①
assert true && true     ②
assert true || false    ③
```

① "not" false is true

② true "and" true is true

③ true "or" false is true

Precedence

The logical "not" has a higher priority than the logical "and".

```
assert (!false && false) == false   ①
```

① Here, the assertion is true (as the expression in parentheses is false), because "not" has a higher precedence than "and", so it only applies to the first "false" term; otherwise, it would have applied to the result of the "and", turned it into true, and the assertion would have failed

The logical "and" has a higher priority than the logical "or".

```
assert true || true && false        ①
```

① Here, the assertion is true, because "and" has a higher precedence than "or", therefore the "or" is executed last and returns true, having one true argument; otherwise, the "and" would have executed last and returned false, having one false argument, and the assertion would have failed

Short-circuiting

The logical `||` operator supports short-circuiting: if the left operand is true, it knows that the result will be true in any case, so it won't evaluate the right operand. The right operand will be evaluated only if the left operand is false.

Likewise for the logical `&&` operator: if the left operand is false, it knows that the result will be false in any case, so it won't evaluate the right operand. The right operand will be evaluated only if the left operand is true.

```
boolean checkIfCalled() {   ①
    called = true
}

called = false
true || checkIfCalled()
assert !called              ②

called = false
false || checkIfCalled()
assert called               ③

called = false
false && checkIfCalled()
assert !called              ④

called = false
true && checkIfCalled()
assert called               ⑤
```

① We create a function that sets the `called` flag to true whenever it's called

② In the first case, after resetting the called flag, we confirm that if the left operand to `||` is true, the function is not called, as `||` short-circuits the evaluation of the right operand

③ In the second case, the left operand is false and so the function is called, as indicated by the fact our flag is now true

④ Likewise for `&&`, we confirm that the function is not called with a false left operand

⑤ But the function is called with a true left operand

Bitwise and bit shift operators

Bitwise operators

Groovy offers four bitwise operators:

- `&`: bitwise "and"
- `|`: bitwise "or"
- `^`: bitwise "xor" (exclusive "or")
- `~`: bitwise negation

Bitwise operators can be applied on arguments which are of type `byte`, `short`, `int`, `long`, or `BigInteger`. If one of the arguments is a `BigInteger`, the result will be of type `BigInteger`; otherwise, if one of the arguments is a `long`, the result will be of type `long`; otherwise, the result will be of type `int`:

```
int a = 0b00101010
assert a == 42
int b = 0b00001000
assert b == 8
assert (a & a) == a                     ①
assert (a & b) == b                     ②
assert (a | a) == a                     ③
assert (a | b) == a                     ④

int mask = 0b11111111                   ⑤
assert ((a ^ a) & mask) == 0b00000000   ⑥
assert ((a ^ b) & mask) == 0b00100010   ⑦
assert ((~a) & mask)    == 0b11010101   ⑧
```

① bitwise and

② bitwise and returns common bits

③ bitwise or

④ bitwise or returns all '1' bits

⑤ setting a mask to check only the last 8 bits

⑥ bitwise exclusive or on self returns 0

⑦ bitwise exclusive or

⑧ bitwise negation

It's worth noting that the internal representation of primitive types follow the Java Language Specification. In particular, primitive types are signed, meaning that for a bitwise negation, it is always good to use a mask to retrieve only the necessary bits.

In Groovy, bitwise operators are overloadable, meaning that you can define the behavior of those operators for any kind of object.

Bit shift operators

Groovy offers three bit shift operators:

- `<<`: left shift
- `>>`: right shift
- `>>>`: right shift unsigned

All three operators are applicable where the left argument is of type `byte`, `short`, `int`, or `long`. The first two operators can also be applied where the left argument is of type `BigInteger`. If the left argument is a `BigInteger`, the result will be of type `BigInteger`; otherwise, if the left argument is a `long`, the result will be of type `long`; otherwise, the result will be of type `int`:

```
assert 12.equals(3 << 2)          ①
assert 24L.equals(3L << 3)        ①
assert 48G.equals(3G << 4)        ①

assert 4095 == -200 >>> 20
assert -1 == -200 >> 20
assert 2G == 5G >> 1
assert -3G == -5G >> 1
```

① `equals` method used instead of `==` to confirm result type

In Groovy, bit shift operators are overloadable, meaning that you can define the behavior of those operators for any kind of object.

Conditional operators

Not operator

The "not" operator is represented with an exclamation mark (`!`) and inverts the result of the underlying boolean expression. In particular, it is possible to combine the `not` operator with the Groovy truth:

```
assert (!true)    == false                    ①
assert (!'foo')   == false                    ②
assert (!'')      == true                     ③
```

① the negation of `true` is `false`

② 'foo' is a non-empty string, evaluating to `true`, so negation returns `false`

③ '' is an empty string, evaluating to `false`, so negation returns `true`

Ternary operator

The ternary operator is a shortcut expression that is equivalent to an if/else branch assigning some value to a variable.

Instead of:

```
if (string!=null && string.length()>0) {
    result = 'Found'
} else {
    result = 'Not found'
}
```

You can write:

```
result = (string!=null && string.length()>0) ? 'Found' : 'Not found'
```

The ternary operator is also compatible with the Groovy truth, so you can make it even simpler:

```
result = string ? 'Found' : 'Not found'
```

Elvis operator

The "Elvis operator" is a shortening of the ternary operator. One instance of where this is handy is for returning a 'sensible default' value if an expression resolves to `false`-ish (as in Groovy truth). A simple example might look like this:

```
displayName = user.name ? user.name : 'Anonymous'   ①
displayName = user.name ?: 'Anonymous'              ②
```

① with the ternary operator, you have to repeat the value you want to assign

② with the Elvis operator, the value, which is tested, is used if it is not `false`-ish

Usage of the Elvis operator reduces the verbosity of your code and reduces the risks of errors in case of refactorings, by removing the need to duplicate the expression which is tested in both the condition and the positive return value.

Elvis assignment operator

Groovy 3.0.0 introduces the Elvis operator, for example:

```
import groovy.transform.ToString

@ToString(includePackage = false)
class Element {
    String name
    int atomicNumber
}

def he = new Element(name: 'Helium')
```

```
he.with {
    name = name ?: 'Hydrogen'   // existing Elvis operator
    atomicNumber ?= 2           // new Elvis assignment shorthand
}
assert he.toString() == 'Element(Helium, 2)'
```

Object operators

Safe navigation operator

The Safe Navigation operator is used to avoid a `NullPointerException`. Typically when you have a reference to an object you might need to verify that it is not `null` before accessing methods or properties of the object. To avoid this, the safe navigation operator will simply return `null` instead of throwing an exception, like so:

```
def person = Person.find { it.id == 123 }    ①
def name = person?.name                      ②
assert name == null                          ③
```

① `find` will return a `null` instance

② use of the null-safe operator prevents from a `NullPointerException`

③ result is `null`

Direct field access operator

Normally in Groovy, when you write code like this:

```
class User {
    public final String name                 ①
    User(String name) { this.name = name}
    String getName() { "Name: $name" }       ②
}
def user = new User('Bob')
assert user.name == 'Name: Bob'              ③
```

① public field `name`

② a getter for `name` that returns a custom string

③ calls the getter

The `user.name` call triggers a call to the property of the same name, that is to say, here, to the getter for `name`. If you want to retrieve the field instead of calling the getter, you can use the direct field access operator:

```
assert user.@name == 'Bob'                   ①
```

① use of `.@` forces usage of the field instead of the getter

Method pointer operator

The method pointer operator (`.&`) can be used to store a reference to a method in a variable, in order to call it later:

```
def str = 'example of method reference'           ①
def fun = str.&toUpperCase                        ②
def upper = fun()                                 ③
assert upper == str.toUpperCase()                 ④
```

① the `str` variable contains a `String`

② we store a reference to the `toUpperCase` method on the `str` instance inside a variable named `fun`

③ `fun` can be called like a regular method

④ we can check that the result is the same as if we had called it directly on `str`

There are multiple advantages in using method pointers. First of all, the type of such a method pointer is a `groovy.lang.Closure`, so it can be used in any place a closure would be used. In particular, it is suitable to convert an existing method for the needs of the strategy pattern:

```
def transform(List elements, Closure action) {                  ①
    def result = []
    elements.each {
        result << action(it)
    }
    result
}
String describe(Person p) {                                     ②
    "$p.name is $p.age"
}
def action = this.&describe                                     ③
def list = [
    new Person(name: 'Bob',   age: 42),
    new Person(name: 'Julia', age: 35)]                         ④
assert transform(list, action) == ['Bob is 42', 'Julia is 35']  ⑤
```

① the `transform` method takes each element of the list and calls the `action` closure on them, returning a new list

② we define a function that takes a `Person` and returns a `String`

③ we create a method pointer on that function

④ we create the list of elements we want to collect the descriptors

⑤ the method pointer can be used where a `Closure` was expected

Method pointers are bound by the receiver and a method name. Arguments are resolved at runtime, meaning that if you have multiple methods with the same name, the syntax is not

different, only resolution of the appropriate method to be called will be done at runtime:

```
def doSomething(String str) { str.toUpperCase() }    ①
def doSomething(Integer x) { 2*x }                   ②
def reference = this.&doSomething                    ③
assert reference('foo') == 'FOO'                     ④
assert reference(123)   == 246                       ⑤
```

① define an overloaded `doSomething` method accepting a `String` as an argument

② define an overloaded `doSomething` method accepting an `Integer` as an argument

③ create a single method pointer on `doSomething`, without specifying argument types

④ using the method pointer with a `String` calls the `String` version of `doSomething`

⑤ using the method pointer with an `Integer` calls the `Integer` version of `doSomething`

To align with Java 8 method reference expectations, in Groovy 3 and above, you can use `new` as the method name to obtain a method pointer to the constructor:

```
def foo  = BigInteger.&new
def fortyTwo = foo('42')
assert fortyTwo == 42G
```

Also in Groovy 3 and above, you can obtain a method pointer to an instance method of a class. This method pointer takes an additional parameter being the receiver instance to invoke the method on:

```
def instanceMethod = String.&toUpperCase
assert instanceMethod('foo') == 'FOO'
```

For backwards compatibility, any static methods that happen to have the correct parameters for the call will be given precedence over instance methods for this case.

Method reference operator

The Parrot parser in Groovy 3+ supports the Java 8+ method reference operator. The method reference operator (`::`) can be used to reference a method or constructor in contexts expecting a functional interface. This overlaps somewhat with the functionality provided by Groovy's method pointer operator. Indeed, for dynamic Groovy, the method reference operator is just an alias for the method pointer operator. For static Groovy, the operator results in bytecode similar to the bytecode that Java would produce for the same context.

Some examples highlighting various supported method reference cases are shown in the following script:

```
import groovy.transform.CompileStatic
import static java.util.stream.Collectors.toList
```

```
@CompileStatic
void methodRefs() {
    assert 6G == [1G, 2G, 3G].stream().reduce(0G, BigInteger::add)
①

    assert [4G, 5G, 6G] == [1G, 2G, 3G].stream().map(3G::add).collect(toList())
②

    assert [1G, 2G, 3G] == [1L, 2L, 3L].stream().map(BigInteger::valueOf).collect
(toList())  ③

    assert [1G, 2G, 3G] == [1L, 2L, 3L].stream().map(3G::valueOf).collect(toList())
④
}

methodRefs()
```

① class instance method reference: add(BigInteger val) is an instance method in BigInteger

② object instance method reference: add(BigInteger val) is an instance method for object 3G

③ class static method reference: valueOf(long val) is a static method for class BigInteger

④ object static method reference: valueOf(long val) is a static method for object 3G (some consider this bad style in normal circumstances)

Some examples highlighting various supported constructor reference cases are shown in the following script:

```
@CompileStatic
void constructorRefs() {
    assert [1, 2, 3] == ['1', '2', '3'].stream().map(Integer::valueOf).collect(
toList())  ①

    def result = [1, 2, 3].stream().toArray(Integer[]::new)
②
    assert result instanceof Integer[]
    assert result.toString() == '[1, 2, 3]'
}

constructorRefs()
```

① class constructor reference

② array constructor reference

Regular expression operators

Pattern operator

The pattern operator (`~`) provides a simple way to create a `java.util.regex.Pattern` instance:

```
def p = ~/foo/
assert p instanceof Pattern
```

while in general, you find the pattern operator with an expression in a slashy-string, it can be used with any kind of `String` in Groovy:

```
p = ~'foo'                                  ①
p = ~"foo"                                  ②
p = ~$/dollar/slashy $ string/$             ③
p = ~"${pattern}"                           ④
```

① using single quote strings

② using double quotes strings

③ the dollar-slashy string lets you use slashes and the dollar sign without having to escape them

④ you can also use a GString!

NOTE	While you can use most String forms with the Pattern, Find and Match operators, we recommend using the slashy string most of the time to save having to remember the otherwise needed escaping requirements.

Find operator

Alternatively to building a pattern, you can use the find operator `=~` to directly create a `java.util.regex.Matcher` instance:

```
def text = "some text to match"
def m = text =~ /match/                     ①
assert m instanceof Matcher                 ②
if (!m) {                                   ③
    throw new RuntimeException("Oops, text not found!")
}
```

① `=~` creates a matcher against the `text` variable, using the pattern on the right hand side

② the return type of `=~` is a `Matcher`

③ equivalent to calling `if (!m.find(0))`

Since a `Matcher` coerces to a `boolean` by calling its `find` method, the `=~` operator is consistent with the simple use of Perl's `=~` operator, when it appears as a predicate (in `if`, `?:`, etc.). When the intent is to iterate over matches of the specified pattern (in `while`, etc.) call `find()` directly on the matcher or use the `iterator` DGM.

Match operator

The match operator (`==~`) is a slight variation of the find operator, that does not return a `Matcher` but a boolean and requires a strict match of the input string:

```
m = text ==~ /match/                                                    ①
assert m instanceof Boolean                                             ②
if (m) {                                                                ③
    throw new RuntimeException("Should not reach that point!")
}
```

① `==~` matches the subject with the regular expression, but match must be strict

② the return type of `==~` is therefore a `boolean`

③ equivalent to calling `if (text ==~ /match/)`

Comparing Find vs Match operators

Typically, the match operator is used when the pattern involves a single exact match, otherwise the find operator might be more useful.

```
assert 'two words' ==~ /\S+\s+\S+/
assert 'two words' ==~ /^\S+\s+\S+$/          ①
assert !(' leading space' ==~ /\S+\s+\S+/)    ②

def m1 = 'two words' =~ /^\S+\s+\S+$/
assert m1.size() == 1                         ③
def m2 = 'now three words' =~ /^\S+\s+\S+$/   ④
assert m2.size() == 0                         ⑤
def m3 = 'now three words' =~ /\S+\s+\S+/
assert m3.size() == 1                         ⑥
assert m3[0] == 'now three'
def m4 = ' leading space' =~ /\S+\s+\S+/
assert m4.size() == 1                         ⑦
assert m4[0] == 'leading space'
def m5 = 'and with four words' =~ /\S+\s+\S+/
assert m5.size() == 2                         ⑧
assert m5[0] == 'and with'
assert m5[1] == 'four words'
```

① equivalent, but explicit ^ and $ are discouraged since they aren't needed

② no match because of leading space

③ one match

④ ^ and $ indicate exact match required

⑤ zero matches

⑥ one match, greedily starting at first word

⑦ one match, ignores leading space

⑧ two matches

Guarding against runaway regular expressions

Java's regex engine has no native timeout. A pattern subject to *catastrophic backtracking* can take practically forever on adversarial input, hanging the evaluating thread — a Regular Expression Denial of Service (ReDoS). When either the pattern or the input comes from an untrusted source, consider using groovy.util.regex.RegexGuard, which cuts off a runaway evaluation by throwing a groovy.util.regex.RegexTimeoutException once a deadline passes. It can guard a region of code or an individual piece of data.

The `guard(timeout) { … }` form applies a per-evaluation timeout to every regex evaluation reached through Groovy's regex operators or `CharSequence` extension methods for the dynamic extent of a closure (on the same thread, including within called methods), and returns the closure's result:

```
def groups = RegexGuard.guard(200) {
    '6.0.0-beta-2'.findGroups(/(\d+)\.(\d+)\.(\d+)(?:-(.+))?/)
}
assert groups == ['6.0.0-beta-2', '6', '0', '0', 'beta-2']
```

Well-behaved evaluations are unaffected; a runaway one is cut off shortly after the deadline:

```
try {
    RegexGuard.guard(200) {
        ('1,' * 40) ==~ /(.*,){16}X/   // catastrophic backtracking
    }
    assert false, 'should have timed out'
} catch (RegexTimeoutException expected) {
}
```

Alternatively, `guard(input, timeout)` attaches a single total deadline to the data itself. Wrap untrusted input once at a trust boundary and pass it around freely; every regex evaluated against it, wherever that happens, shares the one budget:

```
def untrusted = RegexGuard.guard('1,' * 40, 200)
try {
    untrusted.findAll(~/(.*,){16}X/)
    assert false, 'should have timed out'
} catch (RegexTimeoutException expected) {
}
```

For guarding individual expressions, `RegexGuard.matches(pattern, input, timeout)` and `RegexGuard.matcher(pattern, input, timeout)` are guarded equivalents of the match and find operators, and the `@SafeRegex` AST transform rewrites the operators within an annotated scope to equivalent guarded calls automatically (via the operator-shaped `matchRegex(input, pattern, timeout)` and `findRegex(input, pattern, timeout)` entry points, which keep the operators' left-to-right operand evaluation order).

NOTE

The guard mechanism works by checking a deadline while the regex engine reads input characters, so it requires the evaluation to go through Groovy's runtime or over guarded input. Direct JDK calls bypass it: on a `String` receiver, `matches(String)`, `replaceAll(String, String)`, `replaceFirst(String, String)` and `split(String)` dispatch to `java.lang.String` itself. Their `Pattern`-argument variants (e.g. `str.replaceAll(~/regex/, 'x')`) are Groovy extension methods and are covered.

Other operators

Spread operator

The Spread-dot Operator (`*.`), often abbreviated to just Spread Operator, is used to invoke an action on all items of an aggregate object. It is equivalent to calling the action on each item and collecting the result into a list:

```
class Car {
    String make
    String model
}
def cars = [
       new Car(make: 'Peugeot', model: '508'),
       new Car(make: 'Renault', model: 'Clio')]       ①
def makes = cars*.make                                ②
assert makes == ['Peugeot', 'Renault']                ③
```

① build a list of `Car` items. The list is an aggregate of objects.

② call the spread operator on the list, accessing the `make` property of each item

③ returns a list of strings corresponding to the collection of `make` items

The expression `cars*.make` is equivalent to `cars.collect{ it.make }`. Groovy's GPath notation allows a short-cut when the referenced property isn't a property of the containing list, in that case it is automatically spread. In the previously mentioned case, the expression `cars.make` can be used, though retaining the explicit spread-dot operator is often recommended.

The spread operator is null-safe, meaning that if an element of the collection is null, it will return null instead of throwing a `NullPointerException`:

```
cars = [
   new Car(make: 'Peugeot', model: '508'),
   null,                                              ①
   new Car(make: 'Renault', model: 'Clio')]
assert cars*.make == ['Peugeot', null, 'Renault']     ②
assert null*.make == null                             ③
```

① build a list for which one of the elements is `null`

② using the spread operator will **not** throw a `NullPointerException`

③ the receiver might also be null, in which case the return value is `null`

The spread operator can be used on any class which implements the `Iterable` interface:

```
class Component {
    Integer id
    String name
}
class CompositeObject implements Iterable<Component> {
    def components = [
        new Component(id: 1, name: 'Foo'),
        new Component(id: 2, name: 'Bar')]

    @Override
    Iterator<Component> iterator() {
        components.iterator()
    }
}
def composite = new CompositeObject()
assert composite*.id == [1,2]
assert composite*.name == ['Foo','Bar']
```

Use multiple invocations of the spread-dot operator (here `cars*.models*.name`) when working with aggregates of data structures which themselves contain aggregates:

```
class Make {
    String name
    List<Model> models
}

@Canonical
class Model {
    String name
}

def cars = [
    new Make(name: 'Peugeot',
             models: [new Model('408'), new Model('508')]),
    new Make(name: 'Renault',
             models: [new Model('Clio'), new Model('Captur')])
]

def makes = cars*.name
assert makes == ['Peugeot', 'Renault']

def models = cars*.models*.name
assert models == [['408', '508'], ['Clio', 'Captur']]
assert models.sum() == ['408', '508', 'Clio', 'Captur'] // flatten one level
assert models.flatten() == ['408', '508', 'Clio', 'Captur'] // flatten all levels (one
```

```
in this case)
```

Consider using the `collectNested` DGM method instead of the spread-dot operator for collections of collections:

```
class Car {
    String make
    String model
}
def cars = [
   [
       new Car(make: 'Peugeot', model: '408'),
       new Car(make: 'Peugeot', model: '508')
   ], [
       new Car(make: 'Renault', model: 'Clio'),
       new Car(make: 'Renault', model: 'Captur')
   ]
]
def models = cars.collectNested{ it.model }
assert models == [['408', '508'], ['Clio', 'Captur']]
```

Spreading method arguments

There may be situations when the arguments of a method call can be found in a list that you need to adapt to the method arguments. In such situations, you can use the spread operator to call the method. For example, imagine you have the following method signature:

```
int function(int x, int y, int z) {
    x*y+z
}
```

then if you have the following list:

```
def args = [4,5,6]
```

you can call the method without having to define intermediate variables:

```
assert function(*args) == 26
```

It is even possible to mix normal arguments with spread ones:

```
args = [4]
assert function(*args,5,6) == 26
```

Spread list elements

When used inside a list literal, the spread operator acts as if the spread element contents were inlined into the list:

```
def items = [4,5]                       ①
def list = [1,2,3,*items,6]             ②
assert list == [1,2,3,4,5,6]            ③
```

① `items` is a list

② we want to insert the contents of the `items` list directly into `list` without having to call `addAll`

③ the contents of `items` has been inlined into `list`

Spread map elements

The spread map operator works in a similar manner as the spread list operator, but for maps. It allows you to inline the contents of a map into another map literal, like in the following example:

```
def m1 = [c:3, d:4]                     ①
def map = [a:1, b:2, *:m1]              ②
assert map == [a:1, b:2, c:3, d:4]      ③
```

① `m1` is the map that we want to inline

② we use the `*:m1` notation to spread the contents of `m1` into `map`

③ `map` contains all the elements of `m1`

The position of the spread map operator is relevant, like illustrated in the following example:

```
def m1 = [c:3, d:4]                     ①
def map = [a:1, b:2, *:m1, d: 8]        ②
assert map == [a:1, b:2, c:3, d:8]      ③
```

① `m1` is the map that we want to inline

② we use the `*:m1` notation to spread the contents of `m1` into `map`, but redefine the key `d` **after** spreading

③ `map` contains all the expected keys, but `d` was redefined

Range operator

Groovy supports the concept of ranges and provides a notation (`..`) to create ranges of objects:

```
def range = 0..5                                ①
assert (0..5).collect() == [0, 1, 2, 3, 4, 5]   ②
assert (0..<5).collect() == [0, 1, 2, 3, 4]     ③
assert (0<..5).collect() == [1, 2, 3, 4, 5]     ④
assert (0<..<5).collect() == [1, 2, 3, 4]       ⑤
```

```
assert (0..5) instanceof List                    ⑥
assert (0..5).size() == 6                        ⑦
```

① a simple range of integers, stored into a local variable

② an `IntRange`, with inclusive bounds

③ an `IntRange`, with exclusive upper bound

④ an `IntRange`, with exclusive lower bound

⑤ an `IntRange`, with exclusive lower and upper bounds

⑥ a `groovy.lang.Range` implements the `List` interface

⑦ meaning that you can call the `size` method on it

Ranges implementation is lightweight, meaning that only the lower and upper bounds are stored. You can create a range from any `Comparable` object that has `next()` and `previous()` methods to determine the next / previous item in the range. For example, you can create a range of characters this way:

```
assert ('a'..'d').collect() == ['a','b','c','d']
```

Spaceship operator

The spaceship operator (`<=>`) delegates to the `compareTo` method:

```
assert (1 <=> 1) == 0
assert (1 <=> 2) == -1
assert (2 <=> 1) == 1
assert ('a' <=> 'z') == -1
```

Subscript operator

The subscript operator is a shorthand notation for `getAt` or `putAt`, depending on whether you find it on the left hand side or the right hand side of an assignment:

```
def list = [0,1,2,3,4]
assert list[2] == 2                              ①
list[2] = 4                                      ②
assert list[0..2] == [0,1,4]                     ③
list[0..2] = [6,6,6]                             ④
assert list == [6,6,6,3,4]                       ⑤
```

① `[2]` can be used instead of `getAt(2)`

② if on left hand side of an assignment, will call `putAt`

③ `getAt` also supports ranges

④ so does `putAt`

⑤ the list is mutated

The subscript operator, in combination with a custom implementation of `getAt`/`putAt` is a convenient way for destructuring objects:

```
class User {
    Long id
    String name
    def getAt(int i) {                                          ①
        switch (i) {
            case 0: return id
            case 1: return name
        }
        throw new IllegalArgumentException("No such element $i")
    }
    void putAt(int i, def value) {                              ②
        switch (i) {
            case 0: id = value; return
            case 1: name = value; return
        }
        throw new IllegalArgumentException("No such element $i")
    }
}
def user = new User(id: 1, name: 'Alex')                        ③
assert user[0] == 1                                             ④
assert user[1] == 'Alex'                                        ⑤
user[1] = 'Bob'                                                 ⑥
assert user.name == 'Bob'                                       ⑦
```

① the `User` class defines a custom `getAt` implementation

② the `User` class defines a custom `putAt` implementation

③ create a sample user

④ using the subscript operator with index 0 allows retrieving the user id

⑤ using the subscript operator with index 1 allows retrieving the user name

⑥ we can use the subscript operator to write to a property thanks to the delegation to `putAt`

⑦ and check that it's really the property `name` which was changed

Safe index operator

Groovy 3.0.0 introduces safe indexing operator, i.e. `?[]`, which is similar to `?.`. For example:

```
String[] array = ['a', 'b']
assert 'b' == array?[1]      // get using normal array index
array?[1] = 'c'              // set using normal array index
assert 'c' == array?[1]

array = null
```

```
assert null == array?[1]     // return null for all index values
array?[1] = 'c'              // quietly ignore attempt to set value
assert null == array?[1]

def personInfo = [name: 'Daniel.Sun', location: 'Shanghai']
assert 'Daniel.Sun' == personInfo?['name']      // get using normal map index
personInfo?['name'] = 'sunlan'                  // set using normal map index
assert 'sunlan' == personInfo?['name']

personInfo = null
assert null == personInfo?['name']              // return null for all map values
personInfo?['name'] = 'sunlan'                  // quietly ignore attempt to set value
assert null == personInfo?['name']
```

`?[` is a single token: do not put a space between `?` and `[` (`a? [0]` is an incomplete ternary, not a safe index). An unclosed `?[` is reported as `Missing ']'`, including when the safe index sits inside parentheses (`(a?[0)`. `?[` cannot start an expression: `?[0]`, `foo(?[0])`, and `x = ?[0]` are reported as `'?[' requires an expression before it`.

Membership operator

The membership operator (`in`) is equivalent to calling the `isCase` method. In the context of a `List`, it is equivalent to calling `contains`, like in the following example:

```
def list = ['Grace','Rob','Emmy']
assert ('Emmy' in list)                    ①
assert ('Alex' !in list)                   ②
```

① equivalent to calling `list.contains('Emmy')` or `list.isCase('Emmy')`

② membership negation equivalent to calling `!list.contains('Emmy')` or `!list.isCase('Emmy')`

Identity operator

In Groovy, using `==` to test equality is different from using the same operator in Java. In Groovy, it is calling `equals`. If you want to compare reference equality, you should use `is` like in the following example:

```
def list1 = ['Groovy 1.8','Groovy 2.0','Groovy 2.3']       ①
def list2 = ['Groovy 1.8','Groovy 2.0','Groovy 2.3']       ②
assert list1 == list2                                      ③
assert !list1.is(list2)                                    ④
assert list1 !== list2                                     ⑤
```

① Create a list of strings

② Create another list of strings containing the same elements

③ using `==`, we test object equality, equivalent to `list1.equals(list2)` in Java

④ using `is`, we can check that references are distinct, equivalent to `list1 == list2` in Java

⑤ using `===` or `!==` (supported and recommended since Groovy 3.0.0), we can also check whether references are distinct or not, equivalent to `list1 == list2` and `list1 != list2` in Java

Coercion operator

The coercion operator (`as`) is a variant of casting. Coercion converts object from one type to another **without** them being compatible for assignment. Let's take an example:

```
String input = '42'
Integer num = (Integer) input                    ①
```

① `String` is not assignable to an `Integer`, so it will produce a `ClassCastException` at runtime

This can be fixed by using *coercion* instead:

```
String input = '42'
Integer num = input as Integer                   ①
```

① `String` is not assignable to an `Integer`, but use of `as` will *coerce* it to an `Integer`

When an object is coerced into another, unless the target type is the same as the source type, coercion will return a **new** object. The rules of coercion differ depending on the source and target types, and coercion may fail if no conversion rules are found. Custom conversion rules may be implemented thanks to the `asType` method:

```
class Identifiable {
    String name
}
class User {
    Long id
    String name
    def asType(Class target) {                                  ①
        if (target == Identifiable) {
            return new Identifiable(name: name)
        }
        throw new ClassCastException("User cannot be coerced into $target")
    }
}
def u = new User(name: 'Xavier')                                ②
def p = u as Identifiable                                       ③
assert p instanceof Identifiable                                ④
assert !(p instanceof User)                                     ⑤
```

① the `User` class defines a custom conversion rule from `User` to `Identifiable`

② we create an instance of `User`

③ we coerce the `User` instance into an `Identifiable`

④ the target is an instance of `Identifiable`

⑤ the target is not an instance of `User` anymore

Diamond operator

The diamond operator (`<>`) is a syntactic sugar only operator added to support compatibility with the operator of the same name in Java 7. It is used to indicate that generic types should be inferred from the declaration:

```
List<String> strings = new LinkedList<>()
```

As of GROOVY-12319, diamond may also be used when creating an anonymous inner class, matching Java 9 (JEP 213):

```
interface Processor<T> {
    T process(T val)
}
Processor<String> p = new Processor<>() {
    String process(String val) { val.toUpperCase() }
}
assert p.process('ok') == 'OK'
```

In dynamic Groovy, this is totally unused. In statically type checked Groovy, it is also optional since the Groovy type checker performs type inference whether this operator is present or not.

Call operator

The call operator `()` is used to call a method named `call` implicitly. For any object which defines a `call` method, you can omit the `.call` part and use the call operator instead:

```
class MyCallable {
    int call(int x) {           ①
        2*x
    }
}

def mc = new MyCallable()
assert mc.call(2) == 4          ②
assert mc(2) == 4               ③
```

① `MyCallable` defines a method named `call`. Note that it doesn't need to implement `java.util.concurrent.Callable`

② we can call the method using the classic method call syntax

③ or we can omit `.call` thanks to the call operator

Operator precedence

The table below lists all groovy operators in order of precedence.

Level	Operator(s)	Name(s)
1	`new ()`	object creation, explicit parentheses
	`() {} []`	method call, closure, literal list/map
	`. .& .@`	member access, method closure, field/attribute access
	`?. * *. *:`	safe dereferencing, spread, spread-dot, spread-map
	`~ ! (type)`	bitwise negate/pattern, not, typecast
	`[] ?[] ++ --`	list/map/array (safe) index, post inc/decrement
2	`**`	power
3	`++ -- + -`	pre inc/decrement, unary plus, unary minus
4	`* / %`	multiply, div, remainder
5	`+ -`	addition, subtraction
6	`<< >> >>> .. ..< <..< <..`	left/right (unsigned) shift, inclusive/exclusive ranges
7	`< <= > >= in !in instanceof !instanceof as`	less/greater than/or equal, in, not in, instanceof, not instanceof, type coercion
8	`== != <=> === !==`	equals, not equals, compare to, identical to, not identical to
	`=~ ==~`	regex find, regex match
9	`&`	binary/bitwise and
10	`^`	binary/bitwise xor
11	`|`	binary/bitwise or
12	`&&`	logical and
13	`||`	logical or
13.5	`==>`	logical implication
14	`? :`	ternary conditional
	`?:`	elvis operator

Level	Operator(s)	Name(s)
15	= **= *= /= %= += -= <<= >>= >>>= &= ^= |= ?=	various assignments

Operator overloading

Groovy allows you to overload the various operators so that they can be used with your own classes. Consider this simple class:

```
class Bucket {
    int size

    Bucket(int size) { this.size = size }

    Bucket plus(Bucket other) {                    ①
        return new Bucket(this.size + other.size)
    }
}
```

① `Bucket` implements a special method called `plus()`

Just by implementing the `plus()` method, the `Bucket` class can now be used with the `+` operator like so:

```
def b1 = new Bucket(4)
def b2 = new Bucket(11)
assert (b1 + b2).size == 15                        ①
```

① The two `Bucket` objects can be added together with the `+` operator

All (non-comparator) Groovy operators have a corresponding method that you can implement in your own classes. The only requirements are that your method is public, has the correct name, and has the correct number of arguments. The argument types depend on what types you want to support on the right hand side of the operator. For example, you could support the statement

```
assert (b1 + 11).size == 15
```

by implementing the `plus()` method with this signature:

```
Bucket plus(int capacity) {
    return new Bucket(this.size + capacity)
}
```

Here is a complete list of the operators and their corresponding methods:

Operator	Method	Operator	Method
`+`	a.plus(b)	`a[b]`	a.getAt(b)
`-`	a.minus(b)	`a[b] = c`	a.putAt(b, c)
`*`	a.multiply(b)	`a in b`	b.isCase(a)
`/`	a.div(b)	`<<`	a.leftShift(b)
`%`	a.mod(b)	`>>`	a.rightShift(b)
`**`	a.power(b)	`>>>`	a.rightShiftUnsigned(b)
`|`	a.or(b)	`++`	a.next()
`&`	a.and(b)	`--`	a.previous()
`^`	a.xor(b)	`+a`	a.positive()
`as`	a.asType(b)	`-a`	a.negative()
`a()`	a.call()	`~a`	a.bitwiseNegate()

Compound assignment operator overloading

Compound assignment operators (`+=`, `-=`, `*=`, ...) normally desugar to `x = x.op(y)`, which creates a new value and reassigns the variable. Since Groovy 6.0 (GEP-15), classes can define dedicated *Assign methods to override compound-assign independently of the base operator. When a `*Assign` method exists on the receiver type and the compiler can resolve it directly (for example, under `@CompileStatic` or `@TypeChecked`), it is called directly: the receiver is mutated in place, no reassignment occurs (so `final` fields and variables can use compound assignment in that case), and for property LHS the setter is *not* invoked. In purely dynamic code the dispatch happens at runtime; the in-place behaviour still applies but the `final`-reassignment check cannot be relaxed because the compiler does not yet know whether a `*Assign` method will respond at runtime.

If no `*Assign` method matches, the legacy `x = x.op(y)` desugar still applies.

Operator	Assign method	Fallback method
`+=`	`plusAssign`	`plus`
`-=`	`minusAssign`	`minus`
`*=`	`multiplyAssign`	`multiply`
`/=`	`divAssign`	`div`
`%=`	`remainderAssign`	`remainder`
`\**=`	`powerAssign`	`power`
`<⇐`	`leftShiftAssign`	`leftShift`
`>>=`	`rightShiftAssign`	`rightShift`
`>>>=`	`rightShiftUnsignedAssign`	`rightShiftUnsigned`
`&=`	`andAssign`	`and`
`|=`	`orAssign`	`or`
`^=`	`xorAssign`	`xor`

When `*Assign` is called, the expression value of `x op= y` is the (mutated) `x`, not the return value of `*Assign`. `*Assign` is also discoverable as an extension method or category, and its name can be remapped via `@OperatorRename(plusAssign="addInPlace")`.

Program structure

This chapter covers the program structure of the Groovy programming language.

Package names

Package names play exactly the same role as in Java. They allow us to separate the code base without any conflicts. Groovy classes must specify their package before the class definition, else the default package is assumed.

Defining a package is very similar to Java:

```
// defining a package named com.yoursite
package com.yoursite
```

To refer to some class `Foo` in the `com.yoursite.com` package you will need to use the fully qualified name `com.yoursite.com.Foo`, or else you can use an `import` statement as we'll see below.

Imports

In order to refer to any class you need a qualified reference to its package. Groovy follows Java's notion of allowing `import` statement to resolve class references.

For example, Groovy provides several builder classes, such as `MarkupBuilder`. `MarkupBuilder` is inside the package `groovy.xml` so in order to use this class, you need to `import` it as shown:

```
// importing the class MarkupBuilder
import groovy.xml.MarkupBuilder

// using the imported class to create an object
def xml = new MarkupBuilder()

assert xml != null
```

Default imports

Default imports are the imports that Groovy language provides by default. For example look at the following code:

```
long time = LocalDateTime.now().toEpochSecond(ZoneOffset.UTC)
Date date = new Date(time)
```

The same code in Java needs an import statement to `Date` class like this: import java.util.Date. Groovy by default imports these classes for you.

The below imports are added by groovy for you:

```
import java.lang.*
import java.util.*
import java.io.*
import java.net.*
import java.time.*
import groovy.lang.*
import groovy.util.*
import java.math.BigInteger
import java.math.BigDecimal
```

This is done because the classes from these packages are most commonly used. By importing these boilerplate code is reduced.

Simple import

A simple import is an import statement where you fully define the class name along with the package. For example the import statement `import groovy.xml.MarkupBuilder` in the code below is a simple import which directly refers to a class inside a package.

```
// importing the class MarkupBuilder
import groovy.xml.MarkupBuilder

// using the imported class to create an object
def xml = new MarkupBuilder()

assert xml != null
```

Star import

Groovy, like Java, provides a special way to import all classes from a package using *, the so-called on-demand or star import. `MarkupBuilder` is a class which is in package `groovy.xml`, alongside another class called `StreamingMarkupBuilder`. In case you need to use both classes, you can do:

```
import groovy.xml.MarkupBuilder
import groovy.xml.StreamingMarkupBuilder

def markupBuilder = new MarkupBuilder()

assert markupBuilder != null

assert new StreamingMarkupBuilder() != null
```

That's perfectly valid code. But with a * import, we can achieve the same effect with just one line. The star imports all the classes under package `groovy.xml`:

```
import groovy.xml.*

def markupBuilder = new MarkupBuilder()

assert markupBuilder != null

assert new StreamingMarkupBuilder() != null
```

One problem with * imports is that they can clutter your local namespace. But with the kinds of aliasing provided by Groovy, this can be solved easily.

Static import

Groovy's static import capability allows you to reference imported classes as if they were static methods in your own class:

```
import static java.lang.Boolean.FALSE

assert !FALSE //use directly, without Boolean prefix!
```

This is similar to Java's static import capability but is a more dynamic than Java in that it allows you to define methods with the same name as an imported method as long as you have different types:

```
import static java.lang.String.format ①

class SomeClass {

    String format(Integer i) { ②
        i.toString()
    }

    static void main(String[] args) {
        assert format('String') == 'String' ③
        assert new SomeClass().format(Integer.valueOf(1)) == '1'
    }
}
```

① static import of method

② declaration of method with same name as method statically imported above, but with a different parameter type

③ compile error in Java, but is valid Groovy code

If you have the same signature, the imported method takes precedence.

Static import aliasing

Static imports with the `as` keyword provide an elegant solution to namespace problems. Suppose you want to get a `Calendar` instance, using its `getInstance()` method. It's a static method, so we can use a static import. But instead of calling `getInstance()` every time, which can be misleading when separated from its class name, we can import it with an alias, to increase code readability:

```
import static Calendar.getInstance as now

assert now().class == Calendar.getInstance().class
```

Now, that's clean!

Static star import

A static star import is very similar to the regular star import. It will import all the static members from the given class.

For example, let's say we need to calculate sines and cosines for our application. The class `java.lang.Math` has static methods named `sin` and `cos` which fit our need. With the help of a static star import, we can do:

```
import static java.lang.Math.*

assert sin(0) == 0.0
assert cos(0) == 1.0
```

As you can see, we were able to access the methods `sin` and `cos` directly, without the `Math.` prefix.

Import aliasing

With type aliasing, we can refer to a fully qualified class name using a name of our choice. This can be done with the `as` keyword, as before.

For example we can import `java.sql.Date` as `SQLDate` and use it in the same file as `java.util.Date` without having to use the fully qualified name of either class:

```
import java.util.Date
import java.sql.Date as SQLDate

Date utilDate = new Date(1000L)
SQLDate sqlDate = new SQLDate(1000L)

assert utilDate instanceof java.util.Date
assert sqlDate instanceof java.sql.Date
```

Module import

Groovy supports Java's module import declarations (JEP 476, finalized in Java 25). A module import imports all public types from all packages exported by the named module with a single statement:

```
import module java.sql

assert Connection.name == 'java.sql.Connection'
assert DriverManager.name == 'java.sql.DriverManager'
```

This is equivalent to writing a star import (`import pkg.*`) for every package the module exports.

Module imports also include packages from any modules that the named module `requires transitive`. For example, `java.sql` transitively requires `java.xml`, `java.logging`, and `java.transaction.xa`:

```
import module java.sql

// Direct exports of java.sql
assert Connection.name == 'java.sql.Connection'

// From java.xml (requires transitive)
assert Source.name == 'javax.xml.transform.Source'

// From java.logging (requires transitive)
assert Logger.name == 'java.util.logging.Logger'
```

Module imports work with system modules (JDK modules like `java.base`, `java.sql`, `java.desktop`), JPMS modules from modular JARs on the classpath (those containing `module-info.class`), and automatic modules (JARs with an `Automatic-Module-Name` manifest entry).

Explicit single-type imports take priority over module-expanded imports, which allows you to resolve any ambiguity by adding a specific import:

```
import module java.desktop
import java.util.List

// The explicit single-type import wins over the
// module-expanded java.awt.List
assert List.name == 'java.util.List'
```

NOTE The `module` keyword is context-sensitive — it can still be used as a variable, method, or class name. Wildcard (`import module java.base.*`) and alias (`import module java.base as jb`) forms are not supported.

Namespace conflicts

Similar to Java, it is an error in Groovy to specify multiple imports with the same name but different types:

```
import java.awt.List
import java.util.List // error: name already declared
```

And to declare an import and a top-level type with the same name:

```
import java.util.List
class List { } // error: name already declared
```

However, inner types can shadow names from the unit scope:

```
import java.util.List
class Main {
    class List { } // allowed; "List" refers to this type within `Main`'s scope and
`java.util.List` elsewhere
}
```

Scripts versus classes

Groovy supports both scripts and classes. From Groovy 5, Groovy also supports JEP 445 compatible scripts.

Motivation for scripts

Take the following code for example:

Main.groovy

```
class Main {                                    ①
    static void main(String... args) {          ②
        println 'Groovy world!'                 ③
    }
}
```

① define a `Main` class, the name is arbitrary

② the `public static void main(String[])` method is usable as the main method of the class

③ the main body of the method

This is typical code that you would find coming from Java, where code **has** to be embedded into a class to be executable. Groovy makes it easier, the following code is equivalent:

Main.groovy

```
println 'Groovy world!'
```

A script can be considered as a class without needing to explicitly declare it. There are some differences which we'll cover next. First, we'll cover Groovy's main `Script` class. Then, we'll cover JEP 445 compatible classes.

`Script` class

A groovy.lang.Script is always compiled into a class. The Groovy compiler will compile the class for you, with the body of the script copied into a `run` method. The previous example is therefore compiled as if it was the following:

Main.groovy

```
import org.codehaus.groovy.runtime.InvokerHelper
class Main extends Script {                     ①
    def run() {                                 ②
        println 'Groovy world!'                 ③
    }
    static void main(String[] args) {           ④
        InvokerHelper.runScript(Main, args)     ⑤
    }
}
```

① The `Main` class extends the `groovy.lang.Script` class

② `groovy.lang.Script` requires a `run` method returning a value

③ the script body goes into the `run` method

④ the `main` method is automatically generated

⑤ and delegates the execution of the script on the `run` method

If the script is in a file, then the base name of the file is used to determine the name of the generated script class. In this example, if the name of the file is `Main.groovy`, then the script class is going to be `Main`.

Methods

It is possible to define methods into a script, as illustrated here:

```
int fib(int n) {
    n < 2 ? 1 : fib(n-1) + fib(n-2)
}
assert fib(10)==89
```

You can also mix methods and code. The generated script class will carry all methods into the script class, and assemble all script bodies into the `run` method:

```
println 'Hello'                                    ①

int power(int n) { 2**n }                          ②

println "2^6==${power(6)}"                         ③
```

① script begins

② a method is defined within the script body

③ and script continues

Statements 1 and 3 are sometimes referred to as "loose" statements. They are not contained within an explicit enclosing method or class. Loose statements are assembled sequentially into the `run` method.

So, the above code is internally converted into:

```
import org.codehaus.groovy.runtime.InvokerHelper
class Main extends Script {
    int power(int n) { 2** n}                      ①
    def run() {
        println 'Hello'                            ②
        println "2^6==${power(6)}"                 ③
    }
    static void main(String[] args) {
        InvokerHelper.runScript(Main, args)
    }
}
```

① the `power` method is copied as-is into the generated script class

② first statement is copied into the `run` method

③ second statement is copied into the `run` method

TIP	Even if Groovy creates a class from your script, it is totally transparent for the user. In particular, scripts are compiled to bytecode, and line numbers are preserved. This implies that if an exception is thrown in a script, the stack trace will show line numbers corresponding to the original script, not the generated code that we have shown.

Variables

Variables in a script do not require a type definition. This means that this script:

```
int x = 1
int y = 2
assert x+y == 3
```

will behave the same as:

```
x = 1
y = 2
assert x+y == 3
```

However, there is a semantic difference between the two:

- if the variable is declared as in the first example, it is a *local variable*. It will be declared in the `run` method that the compiler will generate and will **not** be visible outside of the script main body. In particular, such a variable will **not** be visible in other methods of the script
- if the variable is undeclared, it goes into the groovy.lang.Script#getBinding(). The binding is visible from the methods, and is especially important if you use a script to interact with an application and need to share data between the script and the application. Readers might refer to the integration guide for more information.

TIP

Another approach to making a variable visible to all methods, is to use the @Field annotation. A variable annotated this way will become a field of the generated script class and, as for local variables, access won't involve the script `Binding`. If you have a local variable or script field with the same name as a binding variable, we recommend renaming one of them to avoid potential confusion. If that's not possible, you can use `binding.varName` to access the binding variable.

Convenience variations

As mentioned previously, normally, `public static void main` and `run` methods are automatically added to your script, so it is normally illegal to add your own versions of either of those; you would see a duplicate method compiler error if you tried.

However, there are some exceptions where the above rules don't apply. If your script contains *only* a compatible main method and no other loose statements, or *only* a no-arg `run` instance method (from Groovy 5), then it is allowed. In this case, no loose statements (because there aren't any) are collected into the `run` method. The method you supplied is used instead of Groovy adding the respective method(s).

This can be useful if you need to add an annotation to the otherwise implicitly added `main` or `run` methods as this example shows:

```
@CompileStatic
static main(args) {
    println 'Groovy world!'
}
```

To be recognised as a convenience variation, as well as having no loose statements, the parameter for the `main` method should be:

- untyped as above (`Object` type),

- or of type `String[]`,
- or have no arguments (from Groovy 5).

From Groovy 5, a no-arg instance `run` variant is also supported. This also allows annotations to be added. The `run` variant follows the JEP 445 rules for field declarations (hence doesn't need to use the `@Field` annotation) as this example involving Jackson JSON serialization shows:

```
@JsonIgnoreProperties(["binding"])
def run() {
    var mapper = new ObjectMapper()
    assert mapper.writeValueAsString(this) == '{"pets":["cat","dog"]}'
}

public pets = ['cat', 'dog']
```

The `run` variant is recommended if you need your script to extend the `Script` class and have access to the script context and bindings. If you don't have that requirement, providing one of the `main` variants will create a JEP 445 compatible class which won't extend `Script`. We'll cover JEP 445 compatible scripts in more detail next.

JEP 445 compatible scripts

From Groovy 5, support has been added for JEP 445 compatible scripts containing a `main` method. Such scripts have several differences to normal Groovy `Script` classes:

- they won't have a `public static void main` method added
- they won't extend the `Script` class and hence won't have access to the script context or binding variables
- allows additional class-level *fields* and *methods* to be defined in addition to `main`
- can't have "loose" statements outside the `main` method (excluding any field definitions)

A simple example might look like:

```
void main(args) {
    println new Date()
}
```

An example with additional fields and methods might look like:

```
def main() {
    assert upper(foo) + lower(bar) == 'FOObar'
}

def upper(s) { s.toUpperCase() }

def lower = String::toLowerCase
```

```
def (foo, bar) = ['Foo', 'Bar']      ①
```

① Note that multi-assignment syntax is supported and results in separate field definitions for each component.

Differences with Java JEP 445 behavior

There are some differences with Groovy's JEP 445 support and that offered by Java:

- Java supports either a no-arg `main` method or one containing a single `String[]` parameter. Groovy also adds support for a single untyped (`Object`) parameter, e.g. `def main(args) { ⋯ }`. This addition is known by the Groovy runner but would not be known by the Java launch protocol for a JDK supporting JEP 445.
- Java supports `void` main methods. Groovy also adds support for untyped `def` (`Object`) methods, e.g. `def main(⋯)` as well as `void main(⋯)`. This addition is known by the Groovy runner but would not be known by the Java launch protocol for a JDK supporting JEP 445.
- For static `main` variants, Groovy *promotes* the no-arg or untyped variants to have the standard `public static void main(String[] args)` signature. This is for compatibility with versions of Groovy prior to Groovy 5 (where JEP 445 support was added). As a consequence, such classes are compatible with the Java launch protocol prior to JEP 445 support.
- Groovy's runner has been made aware of JEP 445 compatible classes and can run all variations for JDK11 and above and without the need for preview mode to be enabled.

Main method selection priority

When a class has multiple valid `main` method signatures, the Groovy runner selects one using a priority order aligned with JEP 512. Static methods take priority over instance methods, and methods with parameters take priority over no-arg methods:

Priority	Method signature
1	`static main(String[] args)`
2	`static main(Object args)`
3	`static main()`
4	`main(String[] args)` (instance)
5	`main(Object args)` (instance)
6	`main()` (instance)

Only one method is selected — lower-priority `main` methods are not reachable from the Groovy runner. A compile-time warning is issued for any valid `main` method that would be shadowed by a higher-priority one.

NOTE	Having multiple `main` method overloads in a single class is considered poor style. We recommend providing a single `main` method to avoid any confusion about which one will run.

Object orientation

This chapter covers the object-oriented aspects of the Groovy programming language.

Types

Primitive types

Groovy supports the same primitive types as defined by the Java Language Specification:

- integral types: `byte` (8 bit), `short` (16 bit), `int` (32 bit) and `long` (64 bit)
- floating-point types: `float` (32 bit) and `double` (64 bit)
- the `boolean` type (one of `true` or `false`)
- the `char` type (16 bit, usable as a numeric type, representing a UTF-16 code)

Also like Java, Groovy uses the respective wrapper classes when objects corresponding to any of the primitive types are required:

Table 4. primitive wrappers

Primitive type	Wrapper class
boolean	Boolean
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

Automatic boxing and unboxing occur when, for instance, calling a method requiring the wrapper class and passing it a primitive variable as the parameter, or vice-versa. This is similar to Java but Groovy takes the idea further.

In most scenarios, you can treat a primitive just like it was the full object wrapper equivalent. For instance, you can call `.toString()` or `.equals(other)` on a primitive. Groovy autowraps and unwraps between references and primitives as needed.

Here's an example using `int` which is declared as a static field in a class (discussed shortly):

```
class Foo {
    static int i
}

assert Foo.class.getDeclaredField('i').type == int.class           ①
assert Foo.i.class != int.class && Foo.i.class == Integer.class    ②
```

① Primitive type is respected in the bytecode

② Looking at the field at runtime shows it has been autowrapped

Now you may be concerned that this means every time you use a mathematical operator on a reference to a primitive that you'll incur the cost of unboxing and reboxing the primitive. But this is not the case, as Groovy will compile your operators into their method equivalents and uses those instead. Additionally, Groovy will automatically unbox to a primitive when calling a Java method that takes a primitive parameter and automatically box primitive method return values from Java. However, be aware there are some differences from Java's method resolution.

Reference Types

Apart from primitives, everything else is an object and has an associated class defining its type. We'll discuss classes, and class-related or class-like things like interfaces, traits and records shortly.

We might declare two variables, of type String and List, as follows:

```
String movie = 'The Matrix'
List actors = ['Keanu Reeves', 'Hugo Weaving']
```

Generics

Groovy carries across the same concepts with regard to generics as Java. When defining classes and methods, it is possible to use a type parameter and create a generic class, interface, method or constructor.

Usage of generic classes and methods, regardless of whether they are defined in Java or Groovy, may involve supplying a type argument.

We might declare a variable, of type *"list of string"*, as follows:

```
List<String> roles = ['Trinity', 'Morpheus']
```

Java employs type erasure for backwards compatibility with earlier versions of Java. From Groovy 6, every compilation unit — dynamic Groovy and scripts included — requires generic types to be well-formed, following the same JLS rules as Java. Dynamic Groovy still does not check generic invariance, inference, or unchecked conversions; those remain the domain of `@TypeChecked` / `@CompileStatic`.

Well-formedness (Groovy 6)

The constructs below compiled on Groovy 4 and 5 (typically by silently using the erasure). They are compile errors from Groovy 6, in both dynamic and static compilation. `new List<?>[n]` remains legal: an unbounded wildcard is reifiable.

Table 5. Generic well-formedness rejected in Groovy 6

Construct	Why	Workaround

`new T[n]` / `new T[]{ t }`	JLS 15.10.1, 4.7: array creation requires a reifiable component type. Groovy 4/5 emitted `Object[]` (or the bound) typed as `T[]`.	`(T[]) new Object[n]`, or `(T[]) new Bound[n]` when `T extends Bound`; or `Array.newInstance` with a `Class<T>` token
`class E<T> extends Exception`	JLS 8.1.2: a generic class may not extend `Throwable`	Move the type parameter onto an accessor or factory; catch the erased class
`new List<String>[n]`	JLS 15.10.1: a parameterized component type is not reifiable (`ArrayStoreException` cannot see the type argument)	`new List<?>[n]`, or an explicit `(List<String>[]) new List[n]`
`List<int>` (and nested)	JLS 4.5: a type argument must be a reference type (a primitive cannot appear in a `Signature` attribute)	`List<Integer>` (`List<int[]>` is already legal)
`new T()` / `T.class`	JLS 15.9.1 / 15.8.2: a type variable is not a class that can be instantiated or named by a class literal	Pass a `Class<T>` token
`implements X<? extends …>`	JLS 8.1.5: a superinterface may not specify a wildcard (the superclass form was already rejected)	A concrete type argument, or the raw type
`new ArrayList<?>()`	JLS 15.9: a class instance creation may not mention a wildcard	`new ArrayList<>()` or `new ArrayList<Object>()`
`<T extends A & B>` with a class type not first	JLS 4.4: only the first bound may be a class or type variable; additional bounds are interfaces	Put the class bound first: `<T extends C & I>`
`Foo<String>.class`	JLS 15.8.2: a class literal may not name a parameterized type (there is one `Class` per erasure)	`Foo.class`
`Outer.Inner<String>` with raw generic `Outer`	JLS 4.8: a non-static member of a raw type is raw	`Outer<?>.Inner<String>`

Generic array creation is not a Groovy shortcut for an unchecked cast. Write the Java idiom:

```
class Stack<T> {
    private T[] items
    private int size = 0
    Stack(int capacity) {
        items = (T[]) new Object[capacity]  // reifiable creation + unchecked cast
    }
    void push(T t) { items[size++] = t }
    T pop() { items[--size] }
}
def s = new Stack<String>(4)
s.push('a')
```

```
s.push('b')
assert s.pop() == 'b'
```

A generic class may not extend `Throwable`. Move the type parameter onto an accessor if a typed payload is needed:

```
class ValidationException extends RuntimeException {
    private final Object payload
    ValidationException(String msg, Object payload) {
        super(msg)
        this.payload = payload
    }
    def <T> T payload() { (T) payload }
}
try {
    throw new ValidationException('bad input', [field: 'name'])
} catch (ValidationException e) {
    Map m = e.payload()
    assert m.field == 'name'
}
```

Classes

Groovy classes are very similar to Java classes, and are compatible with Java ones at JVM level. They may have methods, fields and properties (think JavaBeans properties but with less boilerplate). Classes and class members can have the same modifiers (public, protected, private, static, etc.) as in Java with some minor differences at the source level which are explained shortly.

The key differences between Groovy classes and their Java counterparts are:

- Classes or methods with no visibility modifier are automatically public (a special annotation can be used to achieve package private visibility).
- Fields with no visibility modifier are turned into properties automatically, which results in less verbose code, since explicit getter and setter methods aren't needed. More on this aspect will be covered in the fields and properties section.
- Classes do not need to have the same base name as their source file definitions but it is highly recommended in most scenarios (see also the next point about scripts).
- One source file may contain one or more classes (but if a file contains any code not in a class, it is considered a script). Scripts are just classes with some special conventions and will have the same name as their source file (so don't include a class definition within a script having the same name as the script source file).

The following code presents an example class.

```
class Person {                    ①
```

```
    String name                     ②
    Integer age

    def increaseAge(Integer years) { ③
        this.age += years
    }
}
```

① class beginning, with the name `Person`

② string field and property named `name`

③ method definition

Normal class

Normal classes refer to classes which are top level and concrete. This means they can be instantiated without restrictions from any other classes or scripts. This way, they can only be public (even though the `public` keyword may be suppressed). Classes are instantiated by calling their constructors, using the `new` keyword, as in the following snippet.

```
def p = new Person()
```

Inner class

Inner classes are defined within another classes. The enclosing class can use the inner class as usual. On the other side, an inner class can access members of its enclosing class, even if they are private. Classes other than the enclosing class are not allowed to access inner classes. Here is an example:

```
class Outer {
    private String privateStr

    def callInnerMethod() {
        new Inner().methodA()       ①
    }

    class Inner {                   ②
        def methodA() {
            println "${privateStr}." ③
        }
    }
}
```

① the inner class is instantiated and its method gets called

② inner class definition, inside its enclosing class

③ even being private, a field of the enclosing class is accessed by the inner class

There are some reasons for using inner classes:

- They increase encapsulation by hiding the inner class from other classes, which do not need to know about it. This also leads to cleaner packages and workspaces.
- They provide a good organization, by grouping classes that are used by only one class.
- They lead to more maintainable codes, since inner classes are near the classes that use them.

It is common for an inner class to be an implementation of some interface whose method(s) are needed by the outer class. The code below illustrates this typical usage pattern, here being used with threads.

```
class Outer2 {
    private String privateStr = 'some string'

    def startThread() {
       new Thread(new Inner2()).start()
    }

    class Inner2 implements Runnable {
        void run() {
            println "${privateStr}."
        }
    }
}
```

Note that the class `Inner2` is defined only to provide an implementation of the method `run` to class `Outer2`. Anonymous inner classes help to eliminate verbosity in this case. That topic is covered shortly.

Groovy 3+ also supports Java syntax for non-static inner class instantiation, for example:

```
class Computer {
    class Cpu {
        int coreNumber

        Cpu(int coreNumber) {
            this.coreNumber = coreNumber
        }
    }
}

assert 4 == new Computer().new Cpu(4).coreNumber
```

Anonymous inner class

The earlier example of an inner class (`Inner2`) can be simplified with an anonymous inner class. The same functionality can be achieved with the following code:

```
class Outer3 {
```

```
    private String privateStr = 'some string'

    def startThread() {
        new Thread(new Runnable() {      ①
            void run() {
                println "${privateStr}."
            }
        }).start()                       ②
    }
}
```

① comparing with the last example of previous section, the `new Inner2()` was replaced by `new Runnable()` along with all its implementation

② the method `start` is invoked normally

Thus, there was no need to define a new class to be used just once.

Abstract class

Abstract classes represent generic concepts, thus, they cannot be instantiated, being created to be subclassed. Their members include fields/properties and abstract or concrete methods. Abstract methods do not have implementation, and must be implemented by concrete subclasses.

```
abstract class Abstract {         ①
    String name

    abstract def abstractMethod() ②

    def concreteMethod() {
        println 'concrete'
    }
}
```

① abstract classes must be declared with `abstract` keyword

② abstract methods must also be declared with `abstract` keyword

Abstract classes are commonly compared to interfaces. There are at least two important differences of choosing one or another. First, while abstract classes may contain fields/properties and concrete methods, interfaces may contain only abstract methods (method signatures). Moreover, one class can implement several interfaces, whereas it can extend just one class, abstract or not.

Inheritance

Inheritance in Groovy resembles inheritance in Java. It provides a mechanism for a child class (or subclass) to reuse code or properties from a parent (or super class). Classes related through inheritance form an inheritance hierarchy. Common behavior and members are pushed up the hierarchy to reduce duplication. Specializations occur in child classes.

Different forms of inheritance are supported:

- *implementation* inheritance where code (methods, fields or properties) from a superclass or from one or more traits is reused by a child class
- *contract* inheritance where a class promises to provide particular abstract methods defined in a superclass, or defined in one or more traits or interfaces.

Superclasses

Parent classes share visible fields, properties or methods with child classes. A child class may have at most one parent class. The `extends` keyword is used immediately prior to giving the superclass type.

Interfaces

An interface defines a contract that a class needs to conform to. Typically, an interface defines zero or more abstract method definitions, but does not define the method's implementation.

Here is a `Greeter` interface defining one `greet` method:

```
interface Greeter {                                    ①
    void greet(String name)                            ②
}
```

① an interface needs to be declared using the `interface` keyword

② the abstract method signature for the `greet` method

Such method signatures are public by default. It is an error to use `protected` or package-private methods in interfaces:

```
interface Greeter {
    protected void greet(String name)                  ①
}
```

① Using `protected` is a compile-time error

A class *implements* an interface if it defines the interface in its `implements` list or if any of its superclasses does:

```
class SystemGreeter implements Greeter {               ①
    void greet(String name) {                          ②
        println "Hello $name"
    }
}

def greeter = new SystemGreeter()
assert greeter instanceof Greeter                      ③
```

① The `SystemGreeter` declares the `Greeter` interface using the `implements` keyword

② Then implements the required `greet` method

③ Any instance of `SystemGreeter` is also an instance of the `Greeter` interface

An interface can extend another interface:

```
interface ExtendedGreeter extends Greeter {                ①
    void sayBye(String name)
}
```

① the `ExtendedGreeter` interface extends the `Greeter` interface using the `extends` keyword

It is worth noting that for a class to be an instance of an interface, it has to be explicit. For example, the following class defines the `greet` method as it is declared in the `Greeter` interface, but does not declare `Greeter` in its interfaces:

```
class DefaultGreeter {
    void greet(String name) { println "Hello" }
}

greeter = new DefaultGreeter()
assert !(greeter instanceof Greeter)
```

In other words, Groovy does not define structural typing. It is however possible to make an instance of an object implement an interface at runtime, using the `as` coercion operator:

```
greeter = new DefaultGreeter()                             ①
coerced = greeter as Greeter                               ②
assert coerced instanceof Greeter                          ③
```

① create an instance of `DefaultGreeter` that does not implement the interface

② coerce the instance into a `Greeter` at runtime

③ the coerced instance implements the `Greeter` interface

You can see that there are two distinct objects: one is the source object, a `DefaultGreeter` instance, which does not implement the interface. The other is an instance of `Greeter` that delegates to the coerced object.

TIP Groovy traits are close to interfaces, but support other important features described elsewhere in this manual. If you need more power than offered by interfaces, consider using traits.

While interfaces typically contain abstract method definitions, non-abstract methods are also possible. Variants allowed are `default`, `static`, or `private`:

- Default methods provide a mechanism to evolve API functionality. You can add new functionality to existing interfaces but maintain binary compatibility with code written for

older versions of those interfaces. Shared functionality for default methods may be placed in private methods.

- Static methods provide a mechanism to associate methods directly an interface class without it impacting the OO contracts or being overridden. You could use them for writing factory or utility methods without creating an additional utility class.

Class members

Constructors

Constructors are special methods used to initialize an object with a specific state. As with normal methods, it is possible for a class to declare more than one constructor, so long as each constructor has a unique type signature. If an object doesn't require any parameters during construction, it may use a *no-arg* constructor. If no constructors are supplied, an empty no-arg constructor will be provided by the Groovy compiler.

Groovy supports two invocation styles:

- *positional parameters* are used in a similar to how you would use Java constructors
- *named parameters* allow you to specify parameter names when invoking the constructor.

Positional parameters

To create an object by using positional parameters, the respective class needs to declare one or more constructors. In the case of multiple constructors, each must have a unique type signature. The constructors can also be added to the class using the groovy.transform.TupleConstructor annotation.

Typically, once at least one constructor is declared, the class can only be instantiated by having one of its constructors called. It is worth noting that, in this case, you can't normally create the class with named parameters. Groovy does support named parameters so long as the class contains a no-arg constructor or provides a constructor which takes a `Map` argument as the first (and potentially only) argument - see the next section for details.

There are three forms of using a declared constructor. The first one is the normal Java way, with the `new` keyword. The others rely on coercion of lists into the desired types. In this case, it is possible to coerce with the `as` keyword and by statically typing the variable.

```
class PersonConstructor {
    String name
    Integer age

    PersonConstructor(name, age) {          ①
        this.name = name
        this.age = age
    }
}

def person1 = new PersonConstructor('Marie', 1)  ②
```

```
def person2 = ['Marie', 2] as PersonConstructor  ③
PersonConstructor person3 = ['Marie', 3]         ④
```

① Constructor declaration

② Constructor invocation, classic Java way

③ Constructor usage, using coercion with `as` keyword

④ Constructor usage, using coercion in assignment

Named parameters

If no (or a no-arg) constructor is declared, it is possible to create objects by passing parameters in the form of a map (property/value pairs). This can be in handy in cases where one wants to allow several combinations of parameters. Otherwise, by using traditional positional parameters it would be necessary to declare all possible constructors. Having a constructor where the first (and perhaps only) argument is a `Map` argument is also supported - such a constructor may also be added using the groovy.transform.MapConstructor annotation.

```
class PersonWOConstructor {                                  ①
    String name
    Integer age
}

def person4 = new PersonWOConstructor()                      ②
def person5 = new PersonWOConstructor(name: 'Marie')         ③
def person6 = new PersonWOConstructor(age: 1)                ④
def person7 = new PersonWOConstructor(name: 'Marie', age: 2) ⑤
```

① No constructor declared

② No parameters given in the instantiation

③ `name` parameter given in the instantiation

④ `age` parameter given in the instantiation

⑤ `name` and `age` parameters given in the instantiation

It is important to highlight, however, that this approach gives more power to the constructor caller, while imposing an increased responsibility on the caller to get the names and value types correct. Thus, if greater control is desired, declaring constructors using positional parameters might be preferred.

Notes:

- While the example above supplied no constructor, you can also supply a no-arg constructor or a constructor where the first argument is a `Map`, most typically it's the only argument.
- When no (or a no-arg) constructor is declared, Groovy replaces the named constructor call by a call to the no-arg constructor followed by calls to the setter for each supplied named property.
- When the first argument is a Map, Groovy combines all named parameters into a Map

(regardless of ordering) and supplies the map as the first parameter. This can be a good approach if your properties are declared as `final` (since they will be set in the constructor rather than after the fact with setters).

- You can support both named and positional construction by supply both positional constructors as well as a no-arg or Map constructor.
- You can support hybrid construction by having a constructor where the first argument is a Map but there are also additional positional parameters. Use this style with caution.

Methods

Groovy methods are quite similar to other languages. Some peculiarities will be shown in the next subsections.

Method definition

A method is defined with a return type or with the `def` keyword, to make the return type untyped. A method can also receive any number of arguments, which may not have their types explicitly declared. Java modifiers can be used normally, and if no visibility modifier is provided, the method is public.

Methods in Groovy always return some value. If no `return` statement is provided, the value evaluated in the last line executed will be returned. For instance, note that none of the following methods uses the `return` keyword.

```
def someMethod() { 'method called' }                     ①
String anotherMethod() { 'another method called' }       ②
def thirdMethod(param1) { "$param1 passed" }             ③
static String fourthMethod(String param1) { "$param1 passed" } ④
```

① Method with no return type declared and no parameter

② Method with explicit return type and no parameter

③ Method with a parameter with no type defined

④ Static method with a String parameter

Named parameters

Like constructors, normal methods can also be called with named parameters. To support this notation, a convention is used where the first argument to the method is a `Map`. In the method body, the parameter values can be accessed as in normal maps (`map.key`). If the method has just a single Map argument, all supplied parameters must be named.

```
def foo(Map args) { "${args.name}: ${args.age}" }
foo(name: 'Marie', age: 1)
```

Mixing named and positional parameters

Named parameters can be mixed with positional parameters. The same convention applies, in this case, in addition to the `Map` argument as the first argument, the method in question will have additional positional arguments as needed. Supplied positional parameters when calling the method must be in order. The named parameters can be in any position. They are grouped into the map and supplied as the first parameter automatically.

```
def foo(Map args, Integer number) { "${args.name}: ${args.age}, and the number is
${number}" }
foo(name: 'Marie', age: 1, 23)  ①
foo(23, name: 'Marie', age: 1)  ②
```

① Method call with additional `number` argument of `Integer` type

② Method call with changed order of arguments

If we don't have the Map as the first argument, then a Map must be supplied for that argument instead of named parameters. Failure to do so will lead to `groovy.lang.MissingMethodException`:

```
def foo(Integer number, Map args) { "${args.name}: ${args.age}, and the number is
${number}" }
foo(name: 'Marie', age: 1, 23)  ①
```

① Method call throws `groovy.lang.MissingMethodException: No signature of method: foo() is applicable for argument types: (LinkedHashMap, Integer) values: [[name:Marie, age:1], 23]`, because the named argument `Map` parameter is not defined as the first argument

Above exception can be avoided if we replace named arguments with an explicit `Map` argument:

```
def foo(Integer number, Map args) { "${args.name}: ${args.age}, and the number is
${number}" }
foo(23, [name: 'Marie', age: 1])  ①
```

① Explicit `Map` argument in place of named arguments makes invocation valid

TIP	Although Groovy allows you to mix named and positional parameters, it can lead to unnecessary confusion. Mix named and positional arguments with caution.

Default arguments

Default arguments make parameters optional. If the argument is not supplied, the method assumes a default value.

```
def foo(String par1, Integer par2 = 1) { [name: par1, age: par2] }
assert foo('Marie').age == 1
```

Parameters are dropped from the right, however mandatory parameters are never dropped.

```
def baz(a = 'a', int b, c = 'c', boolean d, e = 'e') { "$a $b $c $d $e" }

assert baz(42, true) == 'a 42 c true e'
assert baz('A', 42, true) == 'A 42 c true e'
assert baz('A', 42, 'C', true) == 'A 42 C true e'
assert baz('A', 42, 'C', true, 'E') == 'A 42 C true E'
```

The same rule applies to constructors as well as methods. If using `@TupleConstructor`, additional configuration options apply.

Varargs

Groovy supports methods with a variable number of arguments. They are defined like this: `def foo(p1, ···, pn, T··· args)`. Here `foo` supports `n` arguments by default, but also an unspecified number of further arguments exceeding `n`.

```
def foo(Object... args) { args.length }
assert foo() == 0
assert foo(1) == 1
assert foo(1, 2) == 2
```

This example defines a method `foo`, that can take any number of arguments, including no arguments at all. `args.length` will return the number of arguments given. Groovy allows `T[]` as an alternative notation to `T···`. That means any method with an array as last parameter is seen by Groovy as a method that can take a variable number of arguments.

```
def foo(Object[] args) { args.length }
assert foo() == 0
assert foo(1) == 1
assert foo(1, 2) == 2
```

If a method with varargs is called with `null` as the vararg parameter, then the argument will be `null` and not an array of length one with `null` as the only element.

```
def foo(Object... args) { args }
assert foo(null) == null
```

If a varargs method is called with an array as an argument, then the argument will be that array instead of an array of length one containing the given array as the only element.

```
def foo(Object... args) { args }
Integer[] ints = [1, 2]
assert foo(ints) == [1, 2]
```

Another important point are varargs in combination with method overloading. In case of method overloading Groovy will select the most specific method. For example if a method `foo` takes a varargs argument of type `T` and another method `foo` also takes one argument of type `T`, the second method is preferred.

```
def foo(Object... args) { 1 }
def foo(Object x) { 2 }
assert foo() == 1
assert foo(1) == 2
assert foo(1, 2) == 1
```

Method selection algorithm

Dynamic Groovy supports multiple dispatch (aka multimethods). When calling a method, the actual method invoked is determined dynamically based on the run-time type of methods arguments. First the method name and number of arguments will be considered (including allowance for varargs), and then the type of each argument. Consider the following method definitions:

```
def method(Object o1, Object o2) { 'o/o' }
def method(Integer i, String  s) { 'i/s' }
def method(String  s, Integer i) { 's/i' }
```

Perhaps as expected, calling `method` with `String` and `Integer` parameters, invokes our third method definition.

```
assert method('foo', 42) == 's/i'
```

Of more interest here is when the types are not known at compile time. Perhaps the arguments are declared to be of type `Object` (a list of such objects in our case). Java would determine that the `method(Object, Object)` variant would be selected in all cases (unless casts were used) but as can be seen in the following example, Groovy uses the runtime type and will invoke each of our methods once (and normally, no casting is needed):

```
List<List<Object>> pairs = [['foo', 1], [2, 'bar'], [3, 4]]
assert pairs.collect { a, b -> method(a, b) } == ['s/i', 'i/s', 'o/o']
```

For each of the first two of our three method invocations an exact match of argument types was found. For the third invocation, an exact match of `method(Integer, Integer)` wasn't found but `method(Object, Object)` is still valid and will be selected.

Method selection then is about finding the *closest fit* from valid method candidates which have compatible parameter types. So, `method(Object, Object)` is also valid for the first two invocations but is not as close a match as the variants where types exactly match. To determine the closest fit, the runtime has a notion of the *distance* an actual argument type is away from the declared parameter type and tries to minimise the total distance across all parameters.

The following table illustrates some factors which affect the distance calculation.

Aspect	Example

Directly implemented interfaces match more closely than ones from further up the inheritance hierarchy.	Given these interface and method definitions:
	`interface I1 {}`
	`interface I2 extends I1 {}`
	`interface I3 {}`
	`class Clazz implements I3, I2 {}`
	`def method(I1 i1) { 'I1' }`
	`def method(I3 i3) { 'I3' }`
	The directly implemented interface will match:
	`assert method(new Clazz()) == 'I3'`

An Object array is preferred over an Object.	`def method(Object[] arg) { 'array' }`
	`def method(Object arg) { 'object' }`
	`assert method([] as Object[]) == 'array'`

Non-vararg variants are favored over vararg variants.	`def method(String s, Object... vargs) { 'vararg' }`
	`def method(String s) { 'non-vararg' }`
	`assert method('foo') == 'non-vararg'`

If two vararg variants are applicable, the one which uses the minimum number of vararg arguments is preferred.	`def method(String s, Object... vargs) { 'two vargs' }`
	`def method(String s, Integer i, Object... vargs) { 'one varg' }`
	`assert method('foo', 35, new Date()) == 'one varg'`

Aspect	Example

Interfaces are preferred over super classes.	`interface I {}`
	`class Base {}`
	`class Child extends Base implements I {}`
	
	`def method(Base b) { 'superclass' }`
	`def method(I i) { 'interface' }`
	
	`assert method(new Child()) ==`
	`'interface'`

For a primitive argument type, a declared parameter type which is the same or slightly larger is preferred.	`def method(Long l) { 'Long' }`
	`def method(Short s) { 'Short' }`
	`def method(BigInteger bi) { 'BigInteger'`
	`}`
	
	`assert method(35) == 'Long'`

In the case where two variants have exactly the same distance, this is deemed ambiguous and will cause a runtime exception:

```
def method(Date d, Object o) { 'd/o' }
def method(Object o, String s) { 'o/s' }

def ex = shouldFail {
    println method(new Date(), 'baz')
}
assert ex.message.contains('Ambiguous method overloading')
```

Casting can be used to select the desired method:

```
assert method(new Date(), (Object)'baz') == 'd/o'
assert method((Object)new Date(), 'baz') == 'o/s'
```

Exception declaration

Groovy automatically allows you to treat checked exceptions like unchecked exceptions. This means that you don't need to declare any checked exceptions that a method may throw as shown in the following example which can throw a `FileNotFoundException` if the file isn't found:

```
def badRead() {
    new File('doesNotExist.txt').text
}

shouldFail(FileNotFoundException) {
```

```
    badRead()
}
```

Nor will you be required to surround the call to the `badRead` method in the previous example within a try/catch block - though you are free to do so if you wish.

If you wish to declare any exceptions that your code might throw (checked or otherwise) you are free to do so. Adding exceptions won't change how the code is used from any other Groovy code but can be seen as documentation for the human reader of your code. The exceptions will become part of the method declaration in the bytecode, so if your code might be called from Java, it might be useful to include them. Using an explicit checked exception declaration is illustrated in the following example:

```
def badRead() throws FileNotFoundException {
    new File('doesNotExist.txt').text
}

shouldFail(FileNotFoundException) {
    badRead()
}
```

Fields and Properties

Fields

A field is a member of a class, interface or trait which stores data. A field defined in a Groovy source file has:

- a mandatory *access modifier* (`public`, `protected`, or `private`)
- one or more optional *modifiers* (`static`, `final`, `synchronized`)
- an optional *type*
- a mandatory *name*

```
class Data {
    private int id                                  ①
    protected String description                    ②
    public static final boolean DEBUG = false       ③
}
```

① a `private` field named `id`, of type `int`

② a `protected` field named `description`, of type `String`

③ a `public static final` field named *DEBUG* of type `boolean`

A field may be initialized directly at declaration:

```
class Data {
    private String id = IDGenerator.next() ①
    // ...
}
```

① the private field `id` is initialized with `IDGenerator.next()`

It is possible to omit the type declaration of a field. This is however considered a bad practice and in general it is a good idea to use strong typing for fields:

```
class BadPractice {
    private mapping                         ①
}
class GoodPractice {
    private Map<String,String> mapping      ②
}
```

① the field `mapping` doesn't declare a type

② the field `mapping` has a strong type

The difference between the two is important if you want to use optional type checking later. It is also important as a way to document the class design. However, in some cases like scripting or if you want to rely on duck typing it may be useful to omit the type.

Properties

A property is an externally visible feature of a class. Rather than just using a public field to represent such features (which provides a more limited abstraction and would restrict refactoring possibilities), the typical approach in Java is to follow the conventions outlined in the JavaBeans Specification, i.e. represent the property using a combination of a private backing field and getters/setters. Groovy follows these same conventions but provides a simpler way to define the property. You can define a property with:

- an **absent** access modifier (no `public`, `protected` or `private`)
- one or more optional *modifiers* (`static`, `final`, `synchronized`)
- an optional *type*
- a mandatory *name*

Groovy will then generate the getters/setters appropriately. For example:

```
class Person {
    String name                             ①
    int age                                 ②
}
```

① creates a backing `private String name` field, a `getName` and a `setName` method

② creates a backing `private int age` field, a `getAge` and a `setAge` method

If a property is declared `final`, no setter is generated:

```
class Person {
    final String name                   ①
    final int age                       ②
    Person(String name, int age) {
        this.name = name                ③
        this.age = age                  ④
    }
}
```

① defines a read-only property of type `String`

② defines a read-only property of type `int`

③ assigns the `name` parameter to the `name` field

④ assigns the `age` parameter to the `age` field

Properties are accessed by name and will call the getter or setter transparently, unless the code is in the class which defines the property:

```
class Person {
    String name
    void name(String name) {
        this.name = "Wonder $name"      ①
    }
    String title() {
        this.name                       ②
    }
}
def p = new Person()
p.name = 'Diana'                        ③
assert p.name == 'Diana'                ④
p.name('Woman')                         ⑤
assert p.title() == 'Wonder Woman'      ⑥
```

① `this.name` will directly access the field because the property is accessed from within the class that defines it

② similarly a read access is done directly on the `name` field

③ write access to the property is done outside of the `Person` class so it will implicitly call `setName`

④ read access to the property is done outside of the `Person` class so it will implicitly call `getName`

⑤ this will call the `name` method on `Person` which performs a direct access to the field

⑥ this will call the `title` method on `Person` which performs a direct read access to the field

It is worth noting that this behavior of accessing the backing field directly is done in order to prevent a stack overflow when using the property access syntax within a class that defines the

property.

It is possible to list the properties of a class thanks to the meta `properties` field of an instance:

```
class Person {
    String name
    int age
}
def p = new Person()
assert p.properties.keySet().containsAll(['name','age'])
```

By convention, Groovy will recognize properties even if there is no backing field provided there are getters or setters that follow the Java Beans specification. For example:

```
class PseudoProperties {
    // a pseudo property "name"
    void setName(String name) {}
    String getName() {}

    // a pseudo read-only property "age"
    int getAge() { 42 }

    // a pseudo write-only property "groovy"
    void setGroovy(boolean groovy) {  }
}
def p = new PseudoProperties()
p.name = 'Foo'                      ①
assert p.age == 42                  ②
p.groovy = true                     ③
```

① writing `p.name` is allowed because there is a pseudo-property `name`

② reading `p.age` is allowed because there is a pseudo-readonly property `age`

③ writing `p.groovy` is allowed because there is a pseudo-write-only property `groovy`

This syntactic sugar is at the core of many DSLs written in Groovy.

Property naming conventions

It is generally recommended that the first two letters of a property name are lowercase and for multi-word properties that camel case is used. In those cases, generated getters and setters will have a name formed by capitalizing the property name and adding a `get` or `set` prefix (or optionally "is" for a boolean getter). So, `getLength` would be a getter for a `length` property and `setFirstName` a setter for a `firstName` property. `isEmpty` might be the getter method name for a property named `empty`.

NOTE	Property names starting with a capital letter would have getters/setters with just the prefix added. So, the property `Foo` is allowed even though it isn't following the recommended naming conventions. For this property, the accessor methods would

be `setFoo` and `getFoo`. A consequence of this is that you aren't allowed to have both a `foo` and a `Foo` property, since they would have the same named accessor methods.

The JavaBeans specification makes a special case for properties which typically might be acronyms. If the first two letters of a property name are uppercase, no capitalization is performed (or more importantly, no decapitalization is done if generating the property name from the accessor method name). So, `getURL` would be the getter for a `URL` property.

NOTE

Because of the special "acronym handling" property naming logic in the JavaBeans specification, the conversion to and from a property name are non-symmetrical. This leads to some strange edge cases. Groovy adopts a naming convention that avoids one ambiguity that might seem a little strange but was popular at the time of Groovy's design and has remained (so far) for historical reasons. Groovy looks at the second letter of a property name. If that is a capital, the property is deemed to be one of the acronym style properties and no capitalization is done, otherwise normal capitalization is done. Although we *never* recommend it, it does allow you to have what might seem like "duplicate named" properties, e.g. you can have `aProp` and `AProp`, or `pNAME` and `PNAME`. The getters would be `getaProp` and `getAProp`, and `getpNAME` and `getPNAME` respectively.

Modifiers on a property

We have already seen that properties are defined by omitting the visibility modifier. In general, any other modifiers, e.g. `transient` would be copied across to the field. Two special cases are worth noting:

- `final`, which we saw earlier is for read-only properties, is copied onto the backing field but also causes no setter to be defined
- `static` is copied onto the backing field but also causes the accessor methods to be static

If you wish to have a modifier like `final` also carried over to the accessor methods, you can write your properties long hand or consider using a split property definition.

Annotations on a property

Annotations, including those associated with AST transforms, are copied on to the backing field for the property. This allows AST transforms which are applicable to fields to be applied to properties, e.g.:

```
class Animal {
    int lowerCount = 0
    @Lazy String name = { lower().toUpperCase() }()
    String lower() { lowerCount++; 'sloth' }
}

def a = new Animal()
assert a.lowerCount == 0  ①
assert a.name == 'SLOTH'  ②
```

```
assert a.lowerCount == 1  ③
```

① Confirms no eager initialization

② Normal property access

③ Confirms initialization upon property access

Split property definition with an explicit backing field

Groovy's property syntax is a convenient shorthand when your class design follows certain conventions which align with common JavaBean practice. If your class doesn't exactly fit these conventions, you can certainly write the getter, setter and backing field long hand like you would in Java. However, Groovy does provide a split definition capability which still provides a shortened syntax while allowing slight adjustments to the conventions. For a split definition, you write a field and a property with the same name and type. Only one of the field or property may have an initial value.

For split properties, annotations on the field remain on the backing field for the property. Annotations on the property part of the definition are copied onto the getter and setter methods.

This mechanism allows a number of common variations that property users may wish to use if the standard property definition doesn't exactly fit their needs. For example, if the backing field should be `protected` rather than `private`:

```
class HasPropertyWithProtectedField {
    protected String name  ①
    String name            ②
}
```

① Protected backing field for name property instead of normal private one

② Declare name property

Or, the same example but with a package-private backing field:

```
class HasPropertyWithPackagePrivateField {
    String name                ①
    @PackageScope String name  ②
}
```

① Declare name property

② Package-private backing field for name property instead of normal private one

As a final example, we may wish to apply method-related AST transforms, or in general, any annotation to the setters/getters, e.g. to have the accessors be synchronized:

```
class HasPropertyWithSynchronizedAccessorMethods {
    private String name        ①
    @Synchronized String name  ②
```

```
}
```

① Backing field for name property

② Declare name property with annotation for setter/getter

Explicit accessor methods

The automatic generation of accessor methods doesn't occur if there is an explicit definition of the getter or setter in the class. This allows you to modify the normal behavior of such a getter or setter if needed. Inherited accessor methods aren't normally considered but if an inherited accessor method is marked final, that will also cause no generation of an additional accessor method to honor the `final` requirement of no subclassing of such methods.

Annotations

Annotation definition

An annotation is a kind of special interface dedicated at annotating elements of the code. An annotation is a type which superinterface is the java.lang.annotation.Annotation interface. Annotations are declared in a very similar way to interfaces, using the `@interface` keyword:

```
@interface SomeAnnotation {}
```

An annotation may define members in the form of methods without bodies and an optional default value. The possible member types are limited to:

- primitive types
- java.lang.String
- java.lang.Class
- an java.lang.Enum
- another java.lang.annotation.Annotation
- or any array of the above

For example:

```
@interface SomeAnnotation {
    String value()                          ①
}
@interface SomeAnnotation {
    String value() default 'something'      ②
}
@interface SomeAnnotation {
    int step()                              ③
}
@interface SomeAnnotation {
    Class appliesTo()                       ④
```

```
}
@interface SomeAnnotation {}
@interface SomeAnnotations {
    SomeAnnotation[] value()                ⑤
}
enum DayOfWeek { mon, tue, wed, thu, fri, sat, sun }
@interface Scheduled {
    DayOfWeek dayOfWeek()                   ⑥
}
```

① an annotation defining a `value` member of type `String`

② an annotation defining a `value` member of type `String` with a default value of `something`

③ an annotation defining a `step` member of type the primitive type `int`

④ an annotation defining a `appliesTo` member of type `Class`

⑤ an annotation defining a `value` member which type is an array of another annotation type

⑥ an annotation defining a `dayOfWeek` member which type is the enumeration type `DayOfWeek`

Unlike in the Java language, in Groovy, an annotation can be used to alter the semantics of the language. It is especially true of AST transformations which will generate code based on annotations.

Annotation placement

An annotation can be applied on various elements of the code:

```
@SomeAnnotation                 ①
void someMethod() {
    // ...
}

@SomeAnnotation                 ②
class SomeClass {}

@SomeAnnotation String someVar  ③
```

① `@SomeAnnotation` applies to the `someMethod` method

② `@SomeAnnotation` applies to the `SomeClass` class

③ `@SomeAnnotation` applies to the `someVar` variable

In order to limit the scope where an annotation can be applied, it is necessary to declare it on the annotation definition, using the java.lang.annotation.Target annotation. For example, here is how you would declare that an annotation can be applied to a class or a method:

```
import java.lang.annotation.ElementType
import java.lang.annotation.Target
```

```
@Target([ElementType.METHOD, ElementType.TYPE])      ①
@interface SomeAnnotation {}                         ②
```

① the `@Target` annotation is meant to annotate an annotation with a scope.

② `@SomeAnnotation` will therefore only be allowed on `TYPE` or `METHOD`

The list of possible targets is available in the java.lang.annotation.ElementType.

WARNING	Groovy does not support the java.lang.annotation.ElementType#TYPE_PARAMETER and java.lang.annotation.ElementType#TYPE_PARAMETER element types which were introduced in Java 8.

Annotation member values

When an annotation is used, it is required to set at least all members that do not have a default value. For example:

```
@interface Page {
    int statusCode()
}

@Page(statusCode=404)
void notFound() {
    // ...
}
```

However it is possible to omit `value=` in the declaration of the value of an annotation if the member `value` is the only one being set:

```
@interface Page {
    String value()
    int statusCode() default 200
}

@Page(value='/home')                    ①
void home() {
    // ...
}

@Page('/users')                         ②
void userList() {
    // ...
}

@Page(value='error',statusCode=404)     ③
void notFound() {
    // ...
```

```
}
```

① we can omit the `statusCode` because it has a default value, but `value` needs to be set

② since `value` is the only mandatory member without a default, we can omit `value=`

③ if both `value` and `statusCode` need to be set, it is required to use `value=` for the default `value` member

Retention policy

The visibility of an annotation depends on its retention policy. The retention policy of an annotation is set using the java.lang.annotation.Retention annotation:

```
import java.lang.annotation.Retention
import java.lang.annotation.RetentionPolicy

@Retention(RetentionPolicy.SOURCE)                  ①
@interface SomeAnnotation {}                        ②
```

① the `@Retention` annotation annotates the `@SomeAnnotation` annotation

② so `@SomeAnnotation` will have a `SOURCE` retention

The list of possible retention targets and description is available in the java.lang.annotation.RetentionPolicy enumeration. The choice usually depends on whether you want an annotation to be visible at compile time or runtime.

Closure annotation parameters

An interesting feature of annotations in Groovy is that you can use a closure as an annotation value. Therefore annotations may be used with a wide variety of expressions and still have IDE support. For example, imagine a framework where you want to execute some methods based on environmental constraints like the JDK version or the OS. One could write the following code:

```
class Tasks {
    Set result = []
    void alwaysExecuted() {
        result << 1
    }
    @OnlyIf({ jdk>=6 })
    void supportedOnlyInJDK6() {
        result << 'JDK 6'
    }
    @OnlyIf({ jdk>=7 && windows })
    void requiresJDK7AndWindows() {
        result << 'JDK 7 Windows'
    }
}
```

For the `@OnlyIf` annotation to accept a `Closure` as an argument, you only have to declare the `value` as a `Class`:

```
@Retention(RetentionPolicy.RUNTIME)
@interface OnlyIf {
    Class value()                    ①
}
```

To complete the example, let's write a sample runner that would use that information:

```
class Runner {
    static <T> T run(Class<T> taskClass) {
        def tasks = taskClass.newInstance()                                    ①
        def params = [jdk: 6, windows: false]                                  ②
        tasks.class.declaredMethods.each { m ->                                ③
            if (Modifier.isPublic(m.modifiers) && m.parameterTypes.length == 0) {  ④
                def onlyIf = m.getAnnotation(OnlyIf)                           ⑤
                if (onlyIf) {
                    Closure cl = onlyIf.value().newInstance(tasks,tasks)       ⑥
                    cl.delegate = params                                       ⑦
                    if (cl()) {                                                ⑧
                        m.invoke(tasks)                                        ⑨
                    }
                } else {
                    m.invoke(tasks)                                            ⑩
                }
            }
        }
        tasks                                                                  ⑪
    }
}
```

① create a new instance of the class passed as an argument (the task class)

② emulate an environment which is JDK 6 and not Windows

③ iterate on all declared methods of the task class

④ if the method is public and takes no arguments

⑤ try to find the `@OnlyIf` annotation

⑥ if it is found get the `value` and create a new `Closure` out of it

⑦ set the `delegate` of the closure to our environment variable

⑧ call the closure, which is the annotation closure. It will return a `boolean`

⑨ if it is `true`, call the method

⑩ if the method is not annotated with `@OnlyIf`, execute the method anyway

⑪ after that, return the task object

Then the runner can be used this way:

```
def tasks = Runner.run(Tasks)
assert tasks.result == [1, 'JDK 6'] as Set
```

Meta-annotations

Declaring meta-annotations

Meta-annotations, also known as annotation aliases are annotations that are replaced at compile time by other annotations (one meta-annotation is an alias for one or more annotations). Meta-annotations can be used to reduce the size of code involving multiple annotations.

Let's start with a simple example. Imagine you have the `@Service` and `@Transactional` annotations and that you want to annotate a class with both:

```
@Service
@Transactional
class MyTransactionalService {}
```

Given the multiplication of annotations that you could add to the same class, a meta-annotation could help by reducing the two annotations with a single one having the very same semantics. For example, we might want to write this instead:

```
@TransactionalService                          ①
class MyTransactionalService {}
```

① `@TransactionalService` is a meta-annotation

A meta-annotation is declared as a regular annotation but annotated with `@AnnotationCollector` and the list of annotations it is collecting. In our case, the `@TransactionalService` annotation can be written:

```
import groovy.transform.AnnotationCollector

@Service                                       ①
@Transactional                                 ②
@AnnotationCollector                           ③
@interface TransactionalService {
}
```

① annotate the meta-annotation with `@Service`

② annotate the meta-annotation with `@Transactional`

③ annotate the meta-annotation with `@AnnotationCollector`

Behavior of meta-annotations

Groovy supports both *precompiled* and *source form* meta-annotations. This means that your meta-annotation *may* be precompiled, or you can have it in the same source tree as the one you are currently compiling.

INFO: Meta-annotations are a Groovy-only feature. There is no chance for you to annotate a Java class with a meta-annotation and hope it will do the same as in Groovy. Likewise, you cannot write a meta-annotation in Java: both the meta-annotation definition **and** usage have to be Groovy code. But you can happily collect Java annotations and Groovy annotations within your meta-annotation.

When the Groovy compiler encounters a class annotated with a meta-annotation, it **replaces** it with the collected annotations. So, in our previous example, it will replace `@TransactionalService` with `@Transactional` and `@Service`:

```
def annotations = MyTransactionalService.annotations*.annotationType()
assert (Service in annotations)
assert (Transactional in annotations)
```

The conversion from a meta-annotation to the collected annotations is performed during the *semantic analysis* compilation phase.

In addition to replacing the alias with the collected annotations, a meta-annotation is capable of processing them, including arguments.

Meta-annotation parameters

Meta-annotations can collect annotations which have parameters. To illustrate this, we will imagine two annotations, each of them accepting one argument:

```
@Timeout(after=3600)
@Dangerous(type='explosive')
```

And suppose that you want to create a meta-annotation named `@Explosive`:

```
@Timeout(after=3600)
@Dangerous(type='explosive')
@AnnotationCollector
public @interface Explosive {}
```

By default, when the annotations are replaced, they will get the annotation parameter values **as they were defined in the alias**. More interesting, the meta-annotation supports overriding specific values:

```
@Explosive(after=0)                ①
class Bomb {}
```

① the `after` value provided as a parameter to `@Explosive` overrides the one defined in the `@Timeout` annotation

If two annotations define the same parameter name, the default processor will copy the annotation value to all annotations that accept this parameter:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Foo {
   String value()                                  ①
}
@Retention(RetentionPolicy.RUNTIME)
public @interface Bar {
    String value()                                 ②
}

@Foo
@Bar
@AnnotationCollector
public @interface FooBar {}                        ③

@Foo('a')
@Bar('b')
class Bob {}                                       ④

assert Bob.getAnnotation(Foo).value() == 'a'       ⑤
println Bob.getAnnotation(Bar).value() == 'b'      ⑥

@FooBar('a')
class Joe {}                                       ⑦
assert Joe.getAnnotation(Foo).value() == 'a'       ⑧
println Joe.getAnnotation(Bar).value() == 'a'      ⑨
```

① the `@Foo` annotation defines the `value` member of type `String`

② the `@Bar` annotation also defines the `value` member of type `String`

③ the `@FooBar` meta-annotation aggregates `@Foo` and `@Bar`

④ class `Bob` is annotated with `@Foo` and `@Bar`

⑤ the value of the `@Foo` annotation on `Bob` is `a`

⑥ while the value of the `@Bar` annotation on `Bob` is `b`

⑦ class `Joe` is annotated with `@FooBar`

⑧ then the value of the `@Foo` annotation on `Joe` is `a`

⑨ and the value of the `@Bar` annotation on `Joe` is also `a`

In the second case, the meta-annotation value was copied in both `@Foo` and `@Bar` annotations.

WARNING	It is a compile time error if the collected annotations define the same members with incompatible types. For example if on the previous example `@Foo` defined

> a value of type `String` but `@Bar` defined a value of type `int`.

It is however possible to customize the behavior of meta-annotations and describe how collected annotations are expanded. We'll look at how to do that shortly but first there is an advanced processing option to cover.

Handling duplicate annotations in meta-annotations

The `@AnnotationCollector` annotation supports a `mode` parameter which can be used to alter how the default processor handles annotation replacement in the presence of duplicate annotations.

INFO: Custom processors (discussed next) may or may not support this parameter.

As an example, suppose you create a meta-annotation containing the `@ToString` annotation and then place your meta-annotation on a class that already has an explicit `@ToString` annotation. Should this be an error? Should both annotations be applied? Does one take priority over the other? There is no correct answer. In some scenarios it might be quite appropriate for any of these answers to be correct. So, rather than trying to preempt one correct way to handle the duplicate annotation issue, Groovy lets you write your own custom meta-annotation processors (covered next) and lets you write whatever checking logic you like within AST transforms - which are a frequent target for aggregating. Having said that, by simply setting the `mode`, a number of commonly expected scenarios are handled automatically for you within any extra coding. The behavior of the `mode` parameter is determined by the `AnnotationCollectorMode` enum value chosen and is summarized in the following table.

Mode	Description
DUPLICATE	Annotations from the annotation collection will always be inserted. After all transforms have been run, it will be an error if multiple annotations (excluding those with SOURCE retention) exist.
PREFER_COLLECTOR	Annotations from the collector will be added and any existing annotations with the same name will be removed.
PREFER_COLLECTOR_MERGED	Annotations from the collector will be added and any existing annotations with the same name will be removed but any new parameters found within existing annotations will be merged into the added annotation.
PREFER_EXPLICIT	Annotations from the collector will be ignored if any existing annotations with the same name are found.

PREFER_EXPLICIT_MERGED	Annotations from the collector will be ignored if any existing annotations with the same name are found but any new parameters on the collector annotation will be added to existing annotations.

Custom meta-annotation processors

A custom annotation processor will let you choose how to expand a meta-annotation into collected annotations. The behaviour of the meta-annotation is, in this case, totally up to you. To do this, you must:

- create a meta-annotation processor, extending org.codehaus.groovy.transform.AnnotationCollectorTransform
- declare the processor to be used in the meta-annotation declaration

To illustrate this, we are going to explore how the meta-annotation `@CompileDynamic` is implemented.

`@CompileDynamic` is a meta-annotation that expands itself to `@CompileStatic(TypeCheckingMode.SKIP)`. The problem is that the default meta annotation processor doesn't support enums and the annotation value `TypeCheckingMode.SKIP` is one.

The naive implementation here would not work:

```
@CompileStatic(TypeCheckingMode.SKIP)
@AnnotationCollector
public @interface CompileDynamic {}
```

Instead, we will define it like this:

```
@AnnotationCollector(processor =
"org.codehaus.groovy.transform.CompileDynamicProcessor")
public @interface CompileDynamic {
}
```

The first thing you may notice is that our interface is no longer annotated with `@CompileStatic`. The reason for this is that we rely on the `processor` parameter instead, that references a class which will **generate** the annotation.

Here is how the custom processor is implemented:

CompileDynamicProcessor.groovy

```
@CompileStatic                                                          ①
class CompileDynamicProcessor extends AnnotationCollectorTransform {    ②
    private static final ClassNode CS_NODE = ClassHelper.make(CompileStatic)    ③
    private static final ClassNode TC_NODE = ClassHelper.make(TypeCheckingMode) ④
```

```
    List<AnnotationNode> visit(AnnotationNode collector,                    ⑤
                               AnnotationNode aliasAnnotationUsage,         ⑥
                               AnnotatedNode aliasAnnotated,                ⑦
                               SourceUnit source) {                         ⑧
        def node = new AnnotationNode(CS_NODE)                              ⑨
        def enumRef = new PropertyExpression(
            new ClassExpression(TC_NODE), "SKIP")                           ⑩
        node.addMember("value", enumRef)                                    ⑪
        Collections.singletonList(node)                                     ⑫
    }
}
```

① our custom processor is written in Groovy, and for better compilation performance, we use static compilation

② the custom processor has to extend org.codehaus.groovy.transform.AnnotationCollectorTransform

③ create a class node representing the `@CompileStatic` annotation type

④ create a class node representing the `TypeCheckingMode` enum type

⑤ `collector` is the `@AnnotationCollector` node found in the meta-annotation. Usually unused.

⑥ `aliasAnnotationUsage` is the meta-annotation being expanded, here it is `@CompileDynamic`

⑦ `aliasAnnotated` is the node being annotated with the meta-annotation

⑧ `sourceUnit` is the `SourceUnit` being compiled

⑨ we create a new annotation node for `@CompileStatic`

⑩ we create an expression equivalent to `TypeCheckingMode.SKIP`

⑪ we add that expression to the annotation node, which is now `@CompileStatic(TypeCheckingMode.SKIP)`

⑫ return the generated annotation

In the example, the `visit` method is the only method which has to be overridden. It is meant to return a list of annotation nodes that will be added to the node annotated with the meta-annotation. In this example, we return a single one corresponding to `@CompileStatic(TypeCheckingMode.SKIP)`.

Traits

Traits are a structural construct of the language which allows:

- composition of behaviors
- runtime implementation of interfaces
- behavior overriding
- compatibility with static type checking/compilation

They can be seen as **interfaces** carrying both **default implementations** and **state**. A trait is defined using the `trait` keyword:

```
trait FlyingAbility {                           ①
        String fly() { "I'm flying!" }          ②
}
```

① declaration of a trait

② declaration of a method inside a trait

Then it can be used like a normal interface using the `implements` keyword:

```
class Bird implements FlyingAbility {}          ①
def b = new Bird()                              ②
assert b.fly() == "I'm flying!"                 ③
```

① Adds the trait `FlyingAbility` to the `Bird` class capabilities

② instantiate a new `Bird`

③ the `Bird` class automatically gets the behavior of the `FlyingAbility` trait

Traits allow a wide range of capabilities, from simple composition to testing, which are described thoroughly in this section.

Methods

Public methods

Declaring a method in a trait can be done like any regular method in a class:

```
trait FlyingAbility {                           ①
        String fly() { "I'm flying!" }          ②
}
```

① declaration of a trait

② declaration of a method inside a trait

Abstract methods

In addition, traits may declare *abstract* methods too, which therefore need to be implemented in the class implementing the trait:

```
trait Greetable {
    abstract String name()                              ①
    String greeting() { "Hello, ${name()}!" }           ②
}
```

① implementing class will have to declare the `name` method

② can be mixed with a concrete method

Then the trait can be used like this:

```
class Person implements Greetable {                    ①
    String name() { 'Bob' }                             ②
}

def p = new Person()
assert p.greeting() == 'Hello, Bob!'                    ③
```

① implement the trait `Greetable`

② since `name` was abstract, it is required to implement it

③ then `greeting` can be called

Private methods

Traits may also define private methods. Those methods will not appear in the trait contract interface:

```
trait Greeter {
    private String greetingMessage() {                  ①
        'Hello from a private method!'
    }
    String greet() {
        def m = greetingMessage()                       ②
        println m
        m
    }
}
class GreetingMachine implements Greeter {}             ③
def g = new GreetingMachine()
assert g.greet() == "Hello from a private method!"      ④
try {
    assert g.greetingMessage()                          ⑤
} catch (MissingMethodException e) {
    println "greetingMessage is private in trait"
}
```

① define a private method `greetingMessage` in the trait

② the public `greet` message calls `greetingMessage` by default

③ create a class implementing the trait

④ `greet` can be called

⑤ but not `greetingMessage`

WARNING	Traits only support `public` and `private` methods. Neither `protected` nor `package private` scopes are supported.

Final methods

If we have a class implementing a trait, conceptually implementations from the trait methods are "inherited" into the class. But, in reality, there is no base class containing such implementations. Rather, they are woven directly into the class. A final modifier on a method just indicates what the modifier will be for the woven method. While it would likely be considered bad style to inherit and override or multiply inherit methods with the same signature but a mix of final and non-final variants, Groovy doesn't prohibit this scenario. Normal method selection applies and the modifier used will be determined from the resulting method. You might consider creating a base class which implements the desired trait(s) if you want trait implementation methods that can't be overridden.

The meaning of this

`this` represents the implementing instance. Think of a trait as a superclass. This means that when you write:

```
trait Introspector {
    def whoAmI() { this }
}
class Foo implements Introspector {}
def foo = new Foo()
```

then calling:

```
foo.whoAmI()
```

will return the same instance:

```
assert foo.whoAmI().is(foo)
```

Interfaces

Traits may implement interfaces, in which case the interfaces are declared using the `implements` keyword:

```
interface Named {                                   ①
    String name()
}
trait Greetable implements Named {                  ②
    String greeting() { "Hello, ${name()}!" }
}
class Person implements Greetable {                 ③
    String name() { 'Bob' }                         ④
}

def p = new Person()
assert p.greeting() == 'Hello, Bob!'                ⑤
```

```
assert p instanceof Named                    ⑥
assert p instanceof Greetable                ⑦
```

① declaration of a normal interface

② add `Named` to the list of implemented interfaces

③ declare a class that implements the `Greetable` trait

④ implement the missing `name` method

⑤ the `greeting` implementation comes from the trait

⑥ make sure `Person` implements the `Named` interface

⑦ make sure `Person` implements the `Greetable` trait

Properties

A trait may define properties, like in the following example:

```
trait Named {
    String name                              ①
}
class Person implements Named {}             ②
def p = new Person(name: 'Bob')              ③
assert p.name == 'Bob'                       ④
assert p.getName() == 'Bob'                  ⑤
```

① declare a property `name` inside a trait

② declare a class which implements the trait

③ the property is automatically made visible

④ it can be accessed using the regular property accessor

⑤ or using the regular getter syntax

Fields

Private fields

Since traits allow the use of private methods, it can also be interesting to use private fields to store state. Traits will let you do that:

```
trait Counter {
    private int count = 0                    ①
    int count() { count += 1; count }        ②
}
class Foo implements Counter {}              ③
def f = new Foo()
assert f.count() == 1                        ④
assert f.count() == 2
```

① declare a private field `count` inside a trait

② declare a public method `count` that increments the counter and returns it

③ declare a class that implements the `Counter` trait

④ the `count` method can use the private field to keep state

TIP This is a major difference with Java 8 virtual extension methods. While virtual extension methods do not carry state, traits can. Moreover, traits in Groovy are supported starting with Java 6, because their implementation does not rely on virtual extension methods. This means that even if a trait can be seen from a Java class as a regular interface, that interface will **not** have default methods, only abstract ones.

Public fields

Public fields work the same way as private fields, but in order to avoid the diamond problem, field names are remapped in the implementing class:

```
trait Named {
    public String name                  ①
}
class Person implements Named {}        ②
def p = new Person()                    ③
p.Named__name = 'Bob'                   ④
```

① declare a public **field** inside the trait

② declare a class implementing the trait

③ create an instance of that class

④ the public field is available, but renamed

The name of the field depends on the fully qualified name of the trait. All dots (`.`) in package are replaced with an underscore (`_`), and the final name includes a double underscore. So if the type of the field is `String`, the name of the package is `my.package`, the name of the trait is `Foo` and the name of the field is `bar`, in the implementing class, the public field will appear as:

```
String my_package_Foo__bar
```

WARNING While traits support public fields, it is not recommended to use them and considered as a bad practice.

Composition of behaviors

Traits can be used to implement multiple inheritance in a controlled way. For example, we can have the following traits:

```
trait FlyingAbility {                   ①
```

```
        String fly() { "I'm flying!" }          ②
}
trait SpeakingAbility {
    String speak() { "I'm speaking!" }
}
```

And a class implementing both traits:

```
class Duck implements FlyingAbility, SpeakingAbility {} ①

def d = new Duck()                                      ②
assert d.fly() == "I'm flying!"                         ③
assert d.speak() == "I'm speaking!"                     ④
```

① the `Duck` class implements both `FlyingAbility` and `SpeakingAbility`

② creates a new instance of `Duck`

③ we can call the method `fly` from `FlyingAbility`

④ but also the method `speak` from `SpeakingAbility`

Traits encourage the reuse of capabilities among objects, and the creation of new classes by the composition of existing behavior.

Overriding default methods

Traits provide default implementations for methods, but it is possible to override them in the implementing class. For example, we can slightly change the example above, by having a duck which quacks:

```
class Duck implements FlyingAbility, SpeakingAbility {
    String quack() { "Quack!" }                         ①
    String speak() { quack() }                          ②
}

def d = new Duck()
assert d.fly() == "I'm flying!"                         ③
assert d.quack() == "Quack!"                            ④
assert d.speak() == "Quack!"                            ⑤
```

① define a method specific to `Duck`, named `quack`

② override the default implementation of `speak` so that we use `quack` instead

③ the duck is still flying, from the default implementation

④ `quack` comes from the `Duck` class

⑤ `speak` no longer uses the default implementation from `SpeakingAbility`

Extending traits

Simple inheritance

Traits may extend another trait, in which case you must use the `extends` keyword:

```
trait Named {
    String name                                 ①
}
trait Polite extends Named {                    ②
    String introduce() { "Hello, I am $name" }  ③
}
class Person implements Polite {}
def p = new Person(name: 'Alice')               ④
assert p.introduce() == 'Hello, I am Alice'     ⑤
```

① the `Named` trait defines a single `name` property

② the `Polite` trait **extends** the `Named` trait

③ `Polite` adds a new method which has access to the `name` property of the super-trait

④ the `name` property is visible from the `Person` class implementing `Polite`

⑤ as is the `introduce` method

Multiple inheritance

Alternatively, a trait may extend multiple traits. In that case, all super traits must be declared in the `implements` clause:

```
trait WithId {                                  ①
    Long id
}
trait WithName {                                ②
    String name
}
trait Identified implements WithId, WithName {}    ③
```

① `WithId` trait defines the `id` property

② `WithName` trait defines the `name` property

③ `Identified` is a trait which inherits both `WithId` and `WithName`

Duck typing and traits

Dynamic code

Traits can call any dynamic code, like a normal Groovy class. This means that you can, in the body of a method, call methods which are supposed to exist in an implementing class, without having to explicitly declare them in an interface. This means that traits are fully compatible with duck typing:

```
trait SpeakingDuck {
    String speak() { quack() }                          ①
}
class Duck implements SpeakingDuck {
    String methodMissing(String name, args) {
        "${name.capitalize()}!"                         ②
    }
}
def d = new Duck()
assert d.speak() == 'Quack!'                            ③
```

① the `SpeakingDuck` expects the `quack` method to be defined

② the `Duck` class does implement the method using *methodMissing*

③ calling the `speak` method triggers a call to `quack` which is handled by `methodMissing`

Dynamic methods in a trait

It is also possible for a trait to implement MOP methods like `methodMissing` or `propertyMissing`, in which case implementing classes will inherit the behavior from the trait, like in this example:

```
trait DynamicObject {                                   ①
    private Map props = [:]
    def methodMissing(String name, args) {
        name.toUpperCase()
    }
    def propertyMissing(String name) {
        props.get(name)
    }
    void setProperty(String name, Object value) {
        props.put(name, value)
    }
}

class Dynamic implements DynamicObject {
    String existingProperty = 'ok'                      ②
    String existingMethod() { 'ok' }                    ③
}
def d = new Dynamic()
assert d.existingProperty == 'ok'                       ④
assert d.foo == null                                    ⑤
d.foo = 'bar'                                           ⑥
assert d.foo == 'bar'                                   ⑦
assert d.existingMethod() == 'ok'                       ⑧
assert d.someMethod() == 'SOMEMETHOD'                   ⑨
```

① create a trait implementing several MOP methods

② the `Dynamic` class defines a property

③ the `Dynamic` class defines a method

④ calling an existing property will call the method from `Dynamic`

⑤ calling a non-existing property will call the method from the trait

⑥ will call `setProperty` defined on the trait

⑦ will call `getProperty` defined on the trait

⑧ calling an existing method on `Dynamic`

⑨ but calling a non-existing method thanks to the trait `methodMissing`

Multiple inheritance conflicts

Default conflict resolution

It is possible for a class to implement multiple traits. If some trait defines a method with the same signature as a method in another trait, we have a conflict:

```
trait A {
    String exec() { 'A' }              ①
}
trait B {
    String exec() { 'B' }              ②
}
class C implements A,B {}              ③
```

① trait `A` defines a method named `exec` returning a `String`

② trait `B` defines the very same method

③ class `C` implements both traits

In this case, the default behavior is that the method from the **last declared trait** in the `implements` clause wins. Here, `B` is declared after `A` so the method from `B` will be picked up:

```
def c = new C()
assert c.exec() == 'B'
```

User conflict resolution

In case this behavior is not the one you want, you can explicitly choose which method to call using the `Trait.super.foo` syntax. In the example above, we can ensure the method from trait A is invoked by writing this:

```
class C implements A,B {
    String exec() { A.super.exec() }    ①
}
def c = new C()
assert c.exec() == 'A'                  ②
```

① explicit call of `exec` from the trait `A`

② calls the version from `A` instead of using the default resolution, which would be the one from `B`

Runtime implementation of traits

Implementing a trait at runtime

Groovy also supports implementing traits dynamically at runtime. It allows you to "decorate" an existing object using a trait. As an example, let's start with this trait and the following class:

```
trait Extra {
    String extra() { "I'm an extra method" }           ①
}
class Something {                                      ②
    String doSomething() { 'Something' }               ③
}
```

① the `Extra` trait defines an `extra` method

② the `Something` class does **not** implement the `Extra` trait

③ `Something` only defines a method `doSomething`

Then if we do:

```
def s = new Something()
s.extra()
```

the call to extra would fail because `Something` is not implementing `Extra`. It is possible to do it at runtime with the following syntax:

```
def s = new Something() as Extra                       ①
s.extra()                                              ②
s.doSomething()                                        ③
```

① use of the **as** keyword to coerce an object to a trait **at runtime**

② then `extra` can be called on the object

③ and `doSomething` is still callable

IMPORTANT	When coercing an object to a trait, the result of the operation is not the same instance. It is guaranteed that the coerced object will implement both the trait **and** the interfaces that the original object implements, but the result will **not** be an instance of the original class.

Implementing multiple traits at once

Should you need to implement several traits at once, you can use the `withTraits` method instead of

the `as` keyword:

```
trait A { void methodFromA() {} }
trait B { void methodFromB() {} }

class C {}

def c = new C()
c.methodFromA()                    ①
c.methodFromB()                    ②
def d = c.withTraits A, B          ③
d.methodFromA()                    ④
d.methodFromB()                    ⑤
```

① call to `methodFromA` will fail because `C` doesn't implement `A`

② call to `methodFromB` will fail because `C` doesn't implement `B`

③ `withTraits` will wrap `c` into something which implements `A` and `B`

④ `methodFromA` will now pass because `d` implements `A`

⑤ `methodFromB` will now pass because `d` also implements `B`

IMPORTANT	When coercing an object to multiple traits, the result of the operation is not the same instance. It is guaranteed that the coerced object will implement both the traits **and** the interfaces that the original object implements, but the result will **not** be an instance of the original class.

Chaining behavior

Groovy supports the concept of *stackable traits*. The idea is to delegate from one trait to the other if the current trait is not capable of handling a message. To illustrate this, let's imagine a message handler interface like this:

```
interface MessageHandler {
    void on(String message, Map payload)
}
```

Then you can compose a message handler by applying small behaviors. For example, let's define a default handler in the form of a trait:

```
trait DefaultHandler implements MessageHandler {
    void on(String message, Map payload) {
        println "Received $message with payload $payload"
    }
}
```

Then any class can inherit the behavior of the default handler by implementing the trait:

```
class SimpleHandler implements DefaultHandler {}
```

Now what if you want to log all messages, in addition to the default handler? One option is to write this:

```
class SimpleHandlerWithLogging implements DefaultHandler {
    void on(String message, Map payload) {                          ①
        println "Seeing $message with payload $payload"              ②
        DefaultHandler.super.on(message, payload)                   ③
    }
}
```

① explicitly implement the `on` method

② perform logging

③ continue by delegating to the `DefaultHandler` trait

This works but this approach has drawbacks:

1. the logging logic is bound to a "concrete" handler
2. we have an explicit reference to `DefaultHandler` in the `on` method, meaning that if we happen to change the trait that our class implements, code will be broken

As an alternative, we can write another trait which responsibility is limited to logging:

```
trait LoggingHandler implements MessageHandler {                    ①
    void on(String message, Map payload) {
        println "Seeing $message with payload $payload"              ②
        super.on(message, payload)                                  ③
    }
}
```

① the logging handler is itself a handler

② prints the message it receives

③ then `super` makes it delegate the call to the next trait in the chain

Then our class can be rewritten as this:

```
class HandlerWithLogger implements DefaultHandler, LoggingHandler {}
def loggingHandler = new HandlerWithLogger()
loggingHandler.on('test logging', [:])
```

which will print:

```
Seeing test logging with payload [:]
```

```
Received test logging with payload [:]
```

As the priority rules imply that `LoggerHandler` wins because it is declared last, then a call to `on` will use the implementation from `LoggingHandler`. But the latter has a call to `super`, which means the next trait in the chain. Here, the next trait is `DefaultHandler` so **both** will be called:

The interest of this approach becomes more evident if we add a third handler, which is responsible for handling messages that start with `say`:

```
trait SayHandler implements MessageHandler {
    void on(String message, Map payload) {
        if (message.startsWith("say")) {                                    ①
            println "I say ${message - 'say'}!"
        } else {
            super.on(message, payload)                                      ②
        }
    }
}
```

① a handler specific precondition

② if the precondition is not met, pass the message to the next handler in the chain

Then our final handler looks like this:

```
class Handler implements DefaultHandler, SayHandler, LoggingHandler {}
def h = new Handler()
h.on('foo', [:])
h.on('sayHello', [:])
```

Which means:

- messages will first go through the logging handler
- the logging handler calls `super` which will delegate to the next handler, which is the `SayHandler`
- if the message starts with `say`, then the handler consumes the message
- if not, the `say` handler delegates to the next handler in the chain

This approach is very powerful because it allows you to write handlers that do not know each other and yet let you combine them in the order you want. For example, if we execute the code, it will print:

```
Seeing foo with payload [:]
Received foo with payload [:]
Seeing sayHello with payload [:]
I say Hello!
```

but if we move the logging handler to be the second one in the chain, the output is different:

```
class AlternateHandler implements DefaultHandler, LoggingHandler, SayHandler {}
h = new AlternateHandler()
h.on('foo', [:])
h.on('sayHello', [:])
```

prints:

```
Seeing foo with payload [:]
Received foo with payload [:]
I say Hello!
```

The reason is that now, since the `SayHandler` consumes the message without calling `super`, the logging handler is not called anymore.

Semantics of super inside a trait

If a class implements multiple traits and a call to an unqualified `super` is found, then:

1. if the class implements another trait, the call delegates to the next trait in the chain
2. if there isn't any trait left in the chain, `super` refers to the super class of the implementing class (*this*)

For example, it is possible to decorate final classes thanks to this behavior:

```
trait Filtering {                                       ①
    StringBuilder append(String str) {                  ②
        def subst = str.replace('o','')                 ③
        super.append(subst)                             ④
    }
    String toString() { super.toString() }              ⑤
}
def sb = new StringBuilder().withTraits Filtering       ⑥
sb.append('Groovy')
assert sb.toString() == 'Grvy'                          ⑦
```

① define a trait named `Filtering`, supposed to be applied on a `StringBuilder` at runtime

② redefine the `append` method

③ remove all 'o's from the string

④ then delegate to `super`

⑤ in case `toString` is called, delegate to `super.toString`

⑥ runtime implementation of the `Filtering` trait on a `StringBuilder` instance

⑦ the string which has been appended no longer contains the letter `o`

In this example, when `super.append` is encountered, there is no other trait implemented by the target object, so the method which is called is the original `append` method, that is to say the one from `StringBuilder`. The same trick is used for `toString`, so that the string representation of the proxy object which is generated delegates to the `toString` of the `StringBuilder` instance.

Advanced features

SAM type coercion

If a trait defines a single abstract method, it is candidate for SAM (Single Abstract Method) type coercion. For example, imagine the following trait:

```
trait Greeter {
    String greet() { "Hello $name" }        ①
    abstract String getName()               ②
}
```

① the `greet` method is not abstract and calls the abstract method `getName`

② `getName` is an abstract method

Since `getName` is the *single abstract method* in the `Greeter` trait, you can write:

```
Greeter greeter = { 'Alice' }               ①
```

① the closure "becomes" the implementation of the `getName` single abstract method

or even:

```
void greet(Greeter g) { println g.greet() } ①
greet { 'Alice' }                           ②
```

① the greet method accepts the SAM type Greeter as parameter

② we can call it directly with a closure

Differences with Java 8 default methods

In Java 8, interfaces can have default implementations of methods. If a class implements an interface and does not provide an implementation for a default method, then the implementation from the interface is chosen. Traits behave the same but with a major difference: the implementation from the trait is **always** used if the class declares the trait in its interface list **and** that it doesn't provide an implementation **even** if a super class does.

This feature can be used to compose behaviors in a very precise way, in case you want to override the behavior of an already implemented method.

To illustrate the concept, let's start with this simple example:

```
import groovy.transform.CompileStatic
import org.codehaus.groovy.control.CompilerConfiguration
import org.codehaus.groovy.control.customizers.ASTTransformationCustomizer
import org.codehaus.groovy.control.customizers.ImportCustomizer

class SomeTest {
    def config
    def shell

    void setup() {
        config = new CompilerConfiguration()
        shell = new GroovyShell(config)
    }
    void testSomething() {
        assert shell.evaluate('1+1') == 2
    }
    void otherTest() { /* ... */ }
}
```

In this example, we create a simple test case which uses two properties (*config* and *shell*) and uses those in multiple test methods. Now imagine that you want to test the same, but with another distinct compiler configuration. One option is to create a subclass of `SomeTest`:

```
class AnotherTest extends SomeTest {
    void setup() {
        config = new CompilerConfiguration()
        config.addCompilationCustomizers( ... )
        shell = new GroovyShell(config)
    }
}
```

It works, but what if you have actually multiple test classes, and that you want to test the new configuration for all those test classes? Then you would have to create a distinct subclass for each test class:

```
class YetAnotherTest extends SomeTest {
    void setup() {
        config = new CompilerConfiguration()
        config.addCompilationCustomizers( ... )
        shell = new GroovyShell(config)
    }
}
```

Then what you see is that the `setup` method of both tests is the same. The idea, then, is to create a trait:

```
trait MyTestSupport {
    void setup() {
        config = new CompilerConfiguration()
        config.addCompilationCustomizers( new ASTTransformationCustomizer
(CompileStatic) )
        shell = new GroovyShell(config)
    }
}
```

Then use it in the subclasses:

```
class AnotherTest extends SomeTest implements MyTestSupport {}
class YetAnotherTest extends SomeTest2 implements MyTestSupport {}
...
```

It would allow us to dramatically reduce the boilerplate code, and reduces the risk of forgetting to change the setup code in case we decide to change it. Even if `setup` is already implemented in the super class, since the test class declares the trait in its interface list, the behavior will be borrowed from the trait implementation!

This feature is in particular useful when you don't have access to the super class source code. It can be used to mock methods or force a particular implementation of a method in a subclass. It lets you refactor your code to keep the overridden logic in a single trait and inherit a new behavior just by implementing it. The alternative, of course, is to override the method in **every** place you would have used the new code.

IMPORTANT	It's worth noting that if you use runtime traits, the methods from the trait are **always** preferred to those of the proxied object:

```
class Person {
    String name                                          ①
}
trait Bob {
    String getName() { 'Bob' }                           ②
}

def p = new Person(name: 'Alice')
assert p.name == 'Alice'                                 ③
def p2 = p as Bob                                        ④
assert p2.name == 'Bob'                                  ⑤
```

① the `Person` class defines a `name` property which results in a `getName` method

② `Bob` is a trait which defines `getName` as returning `Bob`

③ the default object will return *Alice*

④ `p2` coerces `p` into `Bob` at runtime

⑤ `getName` returns *Bob* because `getName` is taken from the **trait**

IMPORTANT	Again, don't forget that dynamic trait coercion returns a distinct object which only implements the original interfaces, as well as the traits.

Differences with mixins

There are several conceptual differences with mixins, as they are available in Groovy. Note that we are talking about runtime mixins, not the @Mixin annotation which is deprecated in favour of traits.

First of all, methods defined in a trait are visible in bytecode:

- internally, the trait is represented as an interface (without default methods; public static methods are exposed as JVM-native interface statics) and several helper classes
- this means that an object implementing a trait effectively implements an *interface*
- those methods are visible from Java
- they are compatible with type checking and static compilation

Methods added through a mixin are, on the contrary, only visible at runtime:

```
class A { String methodFromA() { 'A' } }        ①
class B { String methodFromB() { 'B' } }        ②
A.metaClass.mixin B                             ③
def o = new A()
assert o.methodFromA() == 'A'                   ④
assert o.methodFromB() == 'B'                   ⑤
assert o instanceof A                           ⑥
assert !(o instanceof B)                        ⑦
```

① class `A` defines `methodFromA`

② class `B` defines `methodFromB`

③ mixin B into A

④ we can call `methodFromA`

⑤ we can also call `methodFromB`

⑥ the object is an instance of `A`

⑦ but it's **not** an instanceof `B`

The last point is actually a very important and illustrates a place where mixins have an advantage over traits: the instances are **not** modified, so if you mixin some class into another, there isn't a third class generated, and methods which respond to A will continue responding to A even if mixed in.

Static methods, properties and fields

NOTE	Static member support in traits is **incubating**: the behaviour described here is settled for {groovyVersion} but may still be refined. GEP-22 gives the full specification.

A trait may declare static methods, properties and fields:

- The trait is interpreted as a *template* for the implementing class: each implementing class gets its **own copy** of the trait's static members. A static member declared on a trait belongs to the implementing class, not to a single copy shared across all implementers.
- Public static methods are *declarer-bound* by default and are promoted onto the generated trait interface as JVM-native static methods, so both `Trait.m(…)` and `Impl.m(…)` resolve. A private static method stays internal to the trait and is not promoted.
- Static methods may be declared in, and called from, `@CompileStatic` and `@TypeChecked` code; they are not restricted to dynamic Groovy.
- You should typically not mix static and instance methods of the same signature. The normal trait method-selection rules apply (including multiple inheritance conflict resolution): if the selected method is static but some implemented trait has an instance variant, a compilation error occurs; if the instance variant is selected, the static variant is ignored (as with static methods in Java interfaces for this case).

Let's start with a simple example:

```
trait TestHelper {
    public static boolean CALLED = false        ①
    static void init() {                        ②
        CALLED = true                           ③
    }
}
class Foo implements TestHelper {}
Foo.init()                                      ④
assert Foo.TestHelper__CALLED                   ⑤
```

① the static field is declared in the trait

② a static method is also declared in the trait

③ the static field is updated *within* the trait

④ a static method *init* is made available to the implementing class

⑤ the static field is *remapped* to avoid the diamond issue

As usual, it is not recommended to use public fields. Anyway, should you want this, you must understand that the following code would fail:

```
Foo.CALLED = true
```

because there is *no* static field *CALLED* defined on the trait itself. Likewise, if you have two distinct implementing classes, each one gets a distinct static field:

```
class Bar implements TestHelper {}              ①
class Baz implements TestHelper {}              ②
Bar.init()                                      ③
assert Bar.TestHelper__CALLED                   ④
assert !Baz.TestHelper__CALLED                  ⑤
```

① class `Bar` implements the trait

② class `Baz` also implements the trait

③ `init` is only called on `Bar`

④ the static field `CALLED` on `Bar` is updated

⑤ but the static field `CALLED` on `Baz` is not, because it is **distinct**

Overridable static defaults with `@Virtual`

By default a trait's static method is *declarer-bound*: when other trait code calls it, it always resolves to the trait's own copy, even if an implementing class declares a static method of the same signature. That is the right choice for an internal helper or a constant that the trait owns outright.

Sometimes, though, a trait wants to publish a static **default that an implementing class can override**, and have the trait's own logic honour that override. This is the template-method pattern applied to a static member — the same override-visibility that trait *instance* methods already have. Annotate the method with `@groovy.transform.Virtual` to opt into per-implementer override dispatch:

```
import groovy.transform.CompileStatic
import groovy.transform.Virtual

@CompileStatic
trait Configurable {
    @Virtual static boolean nullableByDefault() { false }   ①
    static String describe() {                               ②
        "nullable=${nullableByDefault()}"                    ③
    }
}

@CompileStatic
class StrictEntity implements Configurable {}               ④

@CompileStatic
class LenientEntity implements Configurable {
    static boolean nullableByDefault() { true }              ⑤
}

assert StrictEntity.describe()  == 'nullable=false'          ⑥
assert LenientEntity.describe() == 'nullable=true'           ⑦
```

```
assert LenientEntity.nullableByDefault() == true          ⑧
```

① the overridable default, marked `@Virtual`

② trait logic that consults the setting

③ an unqualified call, so it dispatches through the implementing class

④ uses the trait's default

⑤ overrides the default with its own `static`

⑥ the default is seen for `StrictEntity`

⑦ the override is seen by the trait's own `describe()` for `LenientEntity`

⑧ and externally via `Impl.m()`

Reach for `@Virtual` when the overridable thing is genuinely a property of the *type* rather than of an instance — a policy or configuration default that answers a question about the class as a whole (Grails uses this for hooks such as `Validateable.defaultNullable()`), that may be consulted where no instance exists yet (class initialisation, static factory methods, building metadata), and that external callers can read statically as `Impl.setting()`. For overridable static *data* rather than behaviour, use a `@Virtual` accessor (`@Virtual static getSetting()`) rather than a static field, since a bare static-field read from trait code is not override-visible (see Inheritance of state gotchas).

A `@Virtual` static is deliberately **not** exposed on the trait interface (interface statics are declarer-bound by JVM rule, which is incompatible with override dispatch). It therefore has no qualified `Trait.m(…)` form: read a concrete class's value with `Impl.m(…)`, or use `T.super.m(…)` from within trait code for the trait's own default. A qualified `Trait.m(…)` call to a `@Virtual` static is rejected at compile time:

```
import groovy.transform.CompileStatic
import groovy.transform.Virtual

@CompileStatic
trait Configurable {
    @Virtual static boolean nullableByDefault() { false }
}

@CompileStatic
class Entity implements Configurable {
    static check() { Configurable.nullableByDefault() }    ①
}
Entity.check()
```

① rejected: a `@Virtual` static has no qualified `Trait.m(…)` form

Inheritance of state gotchas

We have seen that traits are stateful. It is possible for a trait to define fields or properties, but when a class implements a trait, it gets those fields/properties on a per-trait basis. So consider the following example:

```
trait IntCouple {
    int x = 1
    int y = 2
    int sum() { x+y }
}
```

The trait defines two properties, `x` and `y`, as well as a `sum` method. Now let's create a class which implements the trait:

```
class BaseElem implements IntCouple {
    int f() { sum() }
}
def base = new BaseElem()
assert base.f() == 3
```

The result of calling `f` is `3`, because `f` delegates to `sum` in the trait, which has state. But what if we write this instead?

```
class Elem implements IntCouple {
    int x = 3                                   ①
    int y = 4                                   ②
    int f() { sum() }                           ③
}
def elem = new Elem()
```

① Override property `x`

② Override property `y`

③ Call `sum` from trait

If you call `elem.f()`, what is the expected output? Actually it is:

```
assert elem.f() == 3
```

The reason is that the `sum` method accesses the *fields* of the trait. So it is using the `x` and `y` values defined in the trait. If you want to use the values from the implementing class, then you need to dereference fields by using getters and setters, like in this last example:

```
trait IntCouple {
    int x = 1
    int y = 2
    int sum() { getX()+getY() }
}

class Elem implements IntCouple {
    int x = 3
```

```
    int y = 4
    int f() { sum() }
}
def elem = new Elem()
assert elem.f() == 7
```

Self types

Type constraints on traits

Sometimes you will want to write a trait that can only be applied to some type. For example, you may want to apply a trait on a class that extends another class which is beyond your control, and still be able to call those methods. To illustrate this, let's start with this example:

```
class CommunicationService {
    static void sendMessage(String from, String to, String message) {       ①
        println "$from sent [$message] to $to"
    }
}

class Device { String id }                                                   ②

trait Communicating {
    void sendMessage(Device to, String message) {
        CommunicationService.sendMessage(id, to.id, message)                 ③
    }
}

class MyDevice extends Device implements Communicating {}                    ④

def bob = new MyDevice(id:'Bob')
def alice = new MyDevice(id:'Alice')
bob.sendMessage(alice,'secret')                                              ⑤
```

① A `Service` class, beyond your control (in a library, ...) defines a `sendMessage` method

② A `Device` class, beyond your control (in a library, ...)

③ Defines a communicating trait for devices that can call the service

④ Defines `MyDevice` as a communicating device

⑤ The method from the trait is called, and `id` is resolved

It is clear, here, that the `Communicating` trait can only apply to `Device`. However, there's no explicit contract to indicate that, because traits cannot extend classes. However, the code compiles and runs perfectly fine, because `id` in the trait method will be resolved dynamically. The problem is that there is nothing that prevents the trait from being applied to any class which is **not** a `Device`. Any class which has an `id` would work, while any class that does not have an `id` property would cause a runtime error.

The problem is even more complex if you want to enable type checking or apply `@CompileStatic` on the trait: because the trait knows nothing about itself being a `Device`, the type checker will complain saying that it does not find the `id` property.

One possibility is to explicitly add a `getId` method in the trait, but it would not solve all issues. What if a method requires `this` as a parameter, and actually requires it to be a `Device`?

```
class SecurityService {
    static void check(Device d) { if (d.id==null) throw new SecurityException() }
}
```

If you want to be able to call `this` in the trait, then you will explicitly need to cast `this` into a `Device`. This can quickly become unreadable with explicit casts to `this` everywhere.

The @SelfType annotation

In order to make this contract explicit, and to make the type checker aware of the *type of itself*, Groovy provides a `@SelfType` annotation that will:

- let you declare the types that a class that implements this trait must inherit or implement
- throw a compile-time error if those type constraints are not satisfied

So in our previous example, we can fix the trait using the `@groovy.transform.SelfType` annotation:

```
@SelfType(Device)
@CompileStatic
trait Communicating {
    void sendMessage(Device to, String message) {
        SecurityService.check(this)
        CommunicationService.sendMessage(id, to.id, message)
    }
}
```

Now if you try to implement this trait on a class that is **not** a device, a compile-time error will occur:

```
class MyDevice implements Communicating {} // forgot to extend Device
```

The error will be:

```
class 'MyDevice' implements trait 'Communicating' but does not extend self type class
'Device'
```

In conclusion, self types are a powerful way of declaring constraints on traits without having to declare the contract directly in the trait or having to use casts everywhere, maintaining separation of concerns as tight as it should be.

Differences with Sealed annotation (incubating)

Both `@Sealed` and `@SelfType` restrict classes which use a trait but in orthogonal ways. Consider the following example:

```
interface HasHeight { double getHeight() }
interface HasArea { double getArea() }

@SelfType([HasHeight, HasArea])                        ①
@Sealed(permittedSubclasses=[UnitCylinder,UnitCube])   ②
trait HasVolume {
    double getVolume() { height * area }
}

final class UnitCube implements HasVolume, HasHeight, HasArea {
    // for the purposes of this example: h=1, w=1, l=1
    double height = 1d
    double area = 1d
}

final class UnitCylinder implements HasVolume, HasHeight, HasArea {
    // for the purposes of this example: h=1, diameter=1
    // radius=diameter/2, area=PI * r^2
    double height = 1d
    double area = Math.PI * 0.5d**2
}

assert new UnitCube().volume == 1d
assert new UnitCylinder().volume == 0.7853981633974483d
```

① All usages of the `HasVolume` trait must implement or extend both `HasHeight` and `HasArea`

② Only `UnitCube` or `UnitCylinder` can use the trait

For the degenerate case where a single class implements a trait, e.g.:

```
final class Foo implements FooTrait {}
```

Then, either:

```
@SelfType(Foo)
trait FooTrait {}
```

or:

```
@Sealed(permittedSubclasses='Foo') ①
trait FooTrait {}
```

① Or just `@Sealed` if `Foo` and `FooTrait` are in the same source file

could express this constraint. Generally, the former of these is preferred.

Limitations

Compatibility with AST transformations

CAUTION	Traits are not officially compatible with AST transformations. Some of them, like `@CompileStatic` will be applied on the trait itself (not on implementing classes), while others will apply on both the implementing class and the trait. There is absolutely no guarantee that an AST transformation will run on a trait as it does on a regular class, so use it at your own risk!

Prefix and postfix operations

Within traits, prefix and postfix operations are not allowed if they update a field of the trait:

```
trait Counting {
    int x
    void inc() {
        x++                                 ①
    }
    void dec() {
        --x                                 ②
    }
}
class Counter implements Counting {}
def c = new Counter()
c.inc()
```

① `x` is defined within the trait, postfix increment is not allowed

② `x` is defined within the trait, prefix decrement is not allowed

A workaround is to use the `+=` operator instead.

Record classes (incubating)

Record classes, or *records* for short, are a special kind of class useful for modelling plain data aggregates. They provide a compact syntax with less ceremony than normal classes. Groovy already has AST transforms such as `@Immutable` and `@Canonical` which already dramatically reduce ceremony but records have been introduced in Java and record classes in Groovy are designed to align with Java record classes.

For example, suppose we want to create a `Message` record representing an email message. For the purposes of this example, let's simplify such a message to contain just a *from* email address, a *to* email address, and a message *body*. We can define such a record as follows:

```
record Message(String from, String to, String body) { }
```

We'd use the record class in the same way as a normal class, as shown below:

```
def msg = new Message('me@myhost.com', 'you@yourhost.net', 'Hello!')
assert msg.toString() == 'Message[from=me@myhost.com, to=you@yourhost.net,
body=Hello!]'
```

The reduced ceremony saves us from defining explicit fields, getters and `toString`, `equals` and `hashCode` methods. In fact, it's a shorthand for the following rough equivalent:

```
final class Message extends Record {
    private final String from
    private final String to
    private final String body
    private static final long serialVersionUID = 0

    /* constructor(s) */

    final String toString() { /*...*/ }

    final boolean equals(Object other) { /*...*/ }

    final int hashCode() { /*...*/ }

    String from() { from }
    // other getters ...
}
```

Note the special naming convention for record getters. They are the same name as the field (rather than the often common JavaBean convention of capitalized with a "get" prefix). Rather than referring to a record's fields or properties, the term *component* is typically used for records. So our `Message` record has `from`, `to`, and `body` components.

Like in Java, you can override the normally implicitly supplied methods by writing your own:

```
record Point3D(int x, int y, int z) {
    String toString() {
        "Point3D[coords=$x,$y,$z]"
    }
}

assert new Point3D(10, 20, 30).toString() == 'Point3D[coords=10,20,30]'
```

You can also use generics with records in the normal way. For example, consider the following `Coord` record definition:

```
record Coord<T extends Number>(T v1, T v2){
    double distFromOrigin() { Math.sqrt(v1()**2 + v2()**2 as double) }
}
```

It can be used as follows:

```
def r1 = new Coord<Integer>(3, 4)
assert r1.distFromOrigin() == 5
def r2 = new Coord<Double>(6d, 2.5d)
assert r2.distFromOrigin() == 6.5d
```

Special record features

Compact constructor

Records have an implicit constructor. This can be overridden in the normal way by providing your own constructor - you need to make sure you set all the fields if you do this. However, for succinctness, a compact constructor syntax can be used where the parameter declaration part of a normal constructor is elided. For this special case, the normal implicit constructor is still provided but is augmented by the supplied statements in the compact constructor definition:

```
public record Warning(String message) {
    public Warning {
        Objects.requireNonNull(message)
        message = message.toUpperCase()
    }
}

def w = new Warning('Help')
assert w.message() == 'HELP'
```

Serializability

Groovy *native* records follow the special conventions for serializability which apply to Java records. Groovy *record-like* classes (discussed below) follow normal Java class serializability conventions.

Groovy enhancements

Argument defaults

Groovy supports default values for constructor arguments. This capability is also available for records as shown in the following record definition which has default values for `y` and `color`:

```
record ColoredPoint(int x, int y = 0, String color = 'white') {}
```

Arguments when left off (dropping one or more arguments from the right) are replaced with their

defaults values as shown in the following example:

```
assert new ColoredPoint(5, 5, 'black').toString() == 'ColoredPoint[x=5, y=5,
color=black]'
assert new ColoredPoint(5, 5).toString() == 'ColoredPoint[x=5, y=5, color=white]'
assert new ColoredPoint(5).toString() == 'ColoredPoint[x=5, y=0, color=white]'
```

This processing follows normal Groovy conventions for default arguments for constructors, essentially automatically providing the constructors with the following signatures:

```
ColoredPoint(int, int, String)
ColoredPoint(int, int)
ColoredPoint(int)
```

Named arguments may also be used (default values also apply here):

```
assert new ColoredPoint(x: 5).toString() == 'ColoredPoint[x=5, y=0, color=white]'
assert new ColoredPoint(x: 0, y: 5).toString() == 'ColoredPoint[x=0, y=5,
color=white]'
```

You can disable default argument processing as shown here:

```
@TupleConstructor(defaultsMode=DefaultsMode.OFF)
record ColoredPoint2(int x, int y, String color) {}
assert new ColoredPoint2(4, 5, 'red').toString() == 'ColoredPoint2[x=4, y=5,
color=red]'
```

This will produce a single constructor as per the default with Java. It will be an error if you drop off arguments in this scenario.

You can force all properties to have a default value as shown here:

```
@TupleConstructor(defaultsMode=DefaultsMode.ON)
record ColoredPoint3(int x, int y = 0, String color = 'white') {}
assert new ColoredPoint3(y: 5).toString() == 'ColoredPoint3[x=0, y=5, color=white]'
```

Any property/field without an explicit initial value will be given the default value for the argument's type (null, or zero/false for primitives).

Diving deeper

We previously described a `Message` record and displayed it's rough equivalent. Groovy in fact steps through an intermediate stage where the `record` keyword is replaced by the `class` keyword and an accompanying `@RecordType` annotation:

```
@RecordType
class Message {
    String from
    String to
    String body
}
```

Then `@RecordType` itself is processed as a *meta-annotation* (annotation collector) and expanded into its constituent sub-annotations such as `@TupleConstructor`, `@POJO`, `@RecordBase`, and others. This is in some sense an implementation detail which can often be ignored. However, if you wish to customise or configure the record implementation, you may wish to drop back to the `@RecordType` style or augment your record class with one of the constituent sub-annotations.

Declarative `toString` customization

As per Java, you can customize a record's `toString` method by writing your own. If you prefer a more declarative style, you can alternatively use Groovy's `@ToString` transform to override the default record `toString`. As an example, you can a three-dimensional point record as follows:

```
package threed

import groovy.transform.ToString

@ToString(ignoreNulls=true, cache=true, includeNames=true,
          leftDelimiter='[', rightDelimiter=']', nameValueSeparator='=')
record Point(Integer x, Integer y, Integer z=null) { }

assert new Point(10, 20).toString() == 'threed.Point[x=10, y=20]'
```

We customise the `toString` by including the package name (excluded by default for records) and by caching the `toString` value since it won't change for this immutable record. We are also ignoring null values (the default value for `z` in our definition).

We can have a similar definition for a two-dimensional point:

```
package twod

import groovy.transform.ToString

@ToString(ignoreNulls=true, cache=true, includeNames=true,
          leftDelimiter='[', rightDelimiter=']', nameValueSeparator='=')
record Point(Integer x, Integer y) { }

assert new Point(10, 20).toString() == 'twod.Point[x=10, y=20]'
```

We can see here that without the package name it would have the same toString as our previous

example.

Obtaining a list of the record component values

We can obtain the component values from a record as a list like so:

```
record Point(int x, int y, String color) { }

def p = new Point(100, 200, 'green')
def (x, y, c) = p.toList()
assert x == 100
assert y == 200
assert c == 'green'
```

You can use `@RecordOptions(toList=false)` to disable this feature.

Obtaining a map of the record component values

We can obtain the component values from a record as a map like so:

```
record Point(int x, int y, String color) { }

def p = new Point(100, 200, 'green')
assert p.toMap() == [x: 100, y: 200, color: 'green']
```

You can use `@RecordOptions(toMap=false)` to disable this feature.

Obtaining the number of components in a record

We can obtain the number of components in a record like so:

```
record Point(int x, int y, String color) { }

def p = new Point(100, 200, 'green')
assert p.size() == 3
```

You can use `@RecordOptions(size=false)` to disable this feature.

Obtaining the nth component from a record

We can use Groovy's normal positional indexing to obtain a particular component in a record like so:

```
record Point(int x, int y, String color) { }

def p = new Point(100, 200, 'green')
assert p[1] == 200
```

You can use `@RecordOptions(getAt=false)` to disable this feature.

Optional Groovy features

Copying

It can be useful to make a copy of a record with some components changed. This can be done using an optional `copyWith` method which takes named arguments. Record components are set from the supplied arguments. For components not mentioned, a (shallow) copy of the original record component is used. Here is how you might use `copyWith` for the `Fruit` record:

```
@RecordOptions(copyWith=true)
record Fruit(String name, double price) {}
def apple = new Fruit('Apple', 11.6)
assert 'Apple' == apple.name()
assert 11.6 == apple.price()

def orange = apple.copyWith(name: 'Orange')
assert orange.toString() == 'Fruit[name=Orange, price=11.6]'
```

The `copyWith` functionality can be disabled by setting the `RecordOptions#copyWith` annotation attribute to `false`.

Deep immutability

As with Java, records by default offer shallow immutability. Groovy's `@Immutable` transform performs defensive copying for a range of mutable data types. Records can make use of this defensive copying to gain deep immutability as follows:

```
@ImmutableProperties
record Shopping(List items) {}

def items = ['bread', 'milk']
def shop = new Shopping(items)
items << 'chocolate'
assert shop.items() == ['bread', 'milk']
```

These examples illustrate the principal behind Groovy's record feature offering three levels of convenience:

- Using the `record` keyword for maximum succinctness
- Supporting low-ceremony customization using declarative annotations
- Allowing normal method implementations when full control is required

Obtaining the components of a record as a typed tuple

You can obtain the components of a record as a typed tuple:

```
import groovy.transform.*

@RecordOptions(components=true)
record Point(int x, int y, String color) { }

@CompileStatic
def method() {
    def p1 = new Point(100, 200, 'green')
    def (int x1, int y1, String c1) = p1.components()
    assert x1 == 100
    assert y1 == 200
    assert c1 == 'green'

    def p2 = new Point(10, 20, 'blue')
    def (x2, y2, c2) = p2.components()
    assert x2 * 10 == 100
    assert y2 ** 2 == 400
    assert c2.toUpperCase() == 'BLUE'

    def p3 = new Point(1, 2, 'red')
    assert p3.components() instanceof Tuple3
}

method()
```

Groovy has a limited number of `TupleN` classes. If you have a large number of components in your record, you might not be able to use this feature.

Other differences to Java

Groovy supports creating *record-like* classes as well as native records. Record-like classes don't extend Java's `Record` class and such classes won't be seen by Java as records but will otherwise have similar properties.

The `@RecordOptions` annotation (part of `@RecordType`) supports a `mode` annotation attribute which can take one of three values (with `AUTO` being the default):

NATIVE

Produces a class similar to what Java would do.

EMULATE

Produces a record-like class.

AUTO

Equivalent to `NATIVE` at Groovy 6's JDK17+ baseline (default).

Whether you use the `record` keyword or the `@RecordType` annotation is independent of the mode.

Sealed hierarchies

Sealed classes, interfaces and traits restrict which subclasses can extend/implement them. Prior to sealed classes, class hierarchy designers had two main options:

- Make a class final to allow no extension.
- Make the class public and non-final to allow extension by anyone.

Sealed classes provide a middle-ground compared to these all or nothing choices.

Sealed classes are also more flexible than other tricks previously used to try to achieve a middle-ground. For example, for class hierarchies, access modifiers like protected and package-private give some ability to restrict inheritance hierarchies but often at the expense of flexible use of those hierarchies.

Sealed hierarchies provide full inheritance within a known hierarchy of classes, interfaces and traits but disable or only provide controlled inheritance outside the hierarchy.

As an example, suppose we want to create a shape hierarchy containing only circles and squares. We also want a shape interface to be able to refer to instances in our hierarchy. We can create the hierarchy as follows:

```
sealed interface ShapeI permits Circle,Square { }
final class Circle implements ShapeI { }
final class Square implements ShapeI { }
```

Groovy also supports an alternative annotation syntax. We think the keyword style is nicer but you might choose the annotation style if your editor doesn't yet have Groovy 4 support.

```
@Sealed(permittedSubclasses=[Circle,Square]) interface ShapeI { }
final class Circle implements ShapeI { }
final class Square implements ShapeI { }
```

We can have a reference of type `ShapeI` which, thanks to the `permits` clause, can point to either a `Circle` or `Square` and, since our classes are `final`, we know no additional classes will be added to our hierarchy in the future. At least not without changing the `permits` clause and recompiling.

In general, we might want to have some parts of our class hierarchy immediately locked down like we have here, where we marked the subclasses as `final` but other times we might want to allow further controlled inheritance.

```
sealed class Shape permits Circle,Polygon,Rectangle { }

final class Circle extends Shape { }

class Polygon extends Shape { }
non-sealed class RegularPolygon extends Polygon { }
```

```
final class Hexagon extends Polygon { }

sealed class Rectangle extends Shape permits Square{ }
final class Square extends Rectangle { }
```

▼ *<Click to see the alternate annotations syntax>*

```
@Sealed(permittedSubclasses=[Circle,Polygon,Rectangle]) class Shape { }

final class Circle extends Shape { }

class Polygon extends Shape { }
@NonSealed class RegularPolygon extends Polygon { }
final class Hexagon extends Polygon { }

@Sealed(permittedSubclasses=Square) class Rectangle extends Shape { }
final class Square extends Rectangle { }
```

In this example, our permitted subclasses for `Shape` are `Circle`, `Polygon`, and `Rectangle`. `Circle` is `final` and hence that part of the hierarchy cannot be extended. `Polygon` is implicitly non-sealed and `RegularPolygon` is explicitly marked as `non-sealed`. That means our hierarchy is open to any further extension by subclassing, as seen with `Polygon → RegularPolygon` and `RegularPolygon → Hexagon`. `Rectangle` is itself sealed which means that part of the hierarchy can be extended but only in a controlled way (only `Square` is permitted).

Sealed classes are useful for creating enum-like related classes which need to contain instance specific data. For instance, we might have the following enum:

```
enum Weather { Rainy, Cloudy, Sunny }
def forecast = [Weather.Rainy, Weather.Sunny, Weather.Cloudy]
assert forecast.toString() == '[Rainy, Sunny, Cloudy]'
```

but we now wish to also add weather specific instance data to weather forecasts. We can alter our abstraction as follows:

```
sealed abstract class Weather { }
@Immutable(includeNames=true) class Rainy extends Weather { Integer expectedRainfall }
@Immutable(includeNames=true) class Sunny extends Weather { Integer expectedTemp }
@Immutable(includeNames=true) class Cloudy extends Weather { Integer expectedUV }
def forecast = [new Rainy(12), new Sunny(35), new Cloudy(6)]
assert forecast.toString() == '[Rainy(expectedRainfall:12), Sunny(expectedTemp:35),
Cloudy(expectedUV:6)]'
```

Sealed hierarchies are also useful when specifying Algebraic or Abstract Data Types (ADTs) as shown in the following example:

```
import groovy.transform.*

sealed interface Tree<T> {}
@Singleton final class Empty implements Tree {
    String toString() { 'Empty' }
}
@Canonical final class Node<T> implements Tree<T> {
    T value
    Tree<T> left, right
}

Tree<Integer> tree = new Node<>(42, new Node<>(0, Empty.instance, Empty.instance),
Empty.instance)
assert tree.toString() == 'Node(42, Node(0, Empty, Empty), Empty)'
```

Sealed hierarchies work well with records as shown in the following example:

```
sealed interface Expr {}
record ConstExpr(int i) implements Expr {}
record PlusExpr(Expr e1, Expr e2) implements Expr {}
record MinusExpr(Expr e1, Expr e2) implements Expr {}
record NegExpr(Expr e) implements Expr {}

def threePlusNegOne = new PlusExpr(new ConstExpr(3), new NegExpr(new ConstExpr(1)))
assert threePlusNegOne.toString() == 'PlusExpr[e1=ConstExpr[i=3],
e2=NegExpr[e=ConstExpr[i=1]]]'
```

Differences to Java

- Java provides no default modifier for subclasses of sealed classes and requires that one of `final`, `sealed` or `non-sealed` be specified. Groovy defaults to *non-sealed* but you can still use `non-sealed/@NonSealed` if you wish. We anticipate the style checking tool CodeNarc will eventually have a rule that looks for the presence of `non-sealed` so developers wanting that stricter style will be able to use CodeNarc and that rule if they want.
- Currently, Groovy doesn't check that all classes mentioned in `permittedSubclasses` are available at compile-time and compiled along with the base sealed class. This may change in a future version of Groovy.

Groovy supports annotating classes as sealed as well as "native" sealed classes.

The `@SealedOptions` annotation supports a `mode` annotation attribute which can take one of three values (with `AUTO` being the default):

NATIVE

Produces a class similar to what Java would do. Produces an error when compiling on JDKs earlier than JDK17.

EMULATE

Indicates the class is sealed using the `@Sealed` annotation. This mechanism works with the Groovy compiler for JDK8+ but is not recognised by the Java compiler.

AUTO

Produces a native sealed class for JDK17+ and emulates the sealed nature otherwise.

Whether you use the `sealed` keyword or the `@Sealed` annotation is independent of the mode.

Closures

This chapter covers Groovy Closures. A closure in Groovy is an open, anonymous, block of code that can take arguments, return a value and be assigned to a variable. A closure may reference variables declared in its surrounding scope. In opposition to the formal definition of a closure, `Closure` in the Groovy language can also contain free variables which are defined outside of its surrounding scope. While breaking the formal concept of a closure, it offers a variety of advantages which are described in this chapter.

Syntax

Defining a closure

A closure definition follows this syntax:

```
{ [closureParameters -> ] statements }
```

Where `[closureParameters->]` is an optional comma-delimited list of parameters, and statements are 0 or more Groovy statements. The parameters look similar to a method parameter list, and these parameters may be typed or untyped.

When a parameter list is specified, the `->` character is required and serves to separate the arguments from the closure body. The *statements* portion consists of 0, 1, or many Groovy statements.

Some examples of valid closure definitions:

```
{ item++ }                                          ①

{ -> item++ }                                       ②

{ println it }                                      ③

{ it -> println it }                                ④

{ name -> println name }                            ⑤

{ String x, int y ->                                ⑥
    println "hey ${x} the value is ${y}"
```

```
}

{ reader ->                                                    ⑦
    def line = reader.readLine()
    line.trim()
}
```

① A closure referencing a variable named `item`

② It is possible to explicitly separate closure parameters from code by adding an arrow (`->`)

③ A closure using an implicit parameter (`it`)

④ An alternative version where `it` is an explicit parameter

⑤ In that case it is often better to use an explicit name for the parameter

⑥ A closure accepting two typed parameters

⑦ A closure can contain multiple statements

Closures as an object

A closure is an instance of the `groovy.lang.Closure` class, making it assignable to a variable or a field as any other variable, despite being a block of code:

```
def listener = { e -> println "Clicked on $e.source" }      ①
assert listener instanceof Closure
Closure callback = { println 'Done!' }                      ②
Closure<Boolean> isTextFile = {
    File it -> it.name.endsWith('.txt')                     ③
}
```

① You can assign a closure to a variable, and it is an instance of `groovy.lang.Closure`

② If not using `def` or `var`, use `groovy.lang.Closure` as the type

③ Optionally, you can specify the return type of the closure by using the generic type of `groovy.lang.Closure`

Calling a closure

A closure, as an anonymous block of code, can be called like any other method. If you define a closure which takes no argument like this:

```
def code = { 123 }
```

Then the code inside the closure will only be executed when you *call* the closure, which can be done by using the variable as if it was a regular method:

```
assert code() == 123
```

Alternatively, you can be explicit and use the `call` method:

```
assert code.call() == 123
```

The principle is the same if the closure accepts arguments:

```
def isOdd = { int i -> i%2 != 0 }                    ①
assert isOdd(3) == true                              ②
assert isOdd.call(2) == false                        ③

def isEven = { it%2 == 0 }                           ④
assert isEven(3) == false                            ⑤
assert isEven.call(2) == true                        ⑥
```

① define a closure which accepts an `int` as a parameter

② it can be called directly

③ or using the `call` method

④ same goes for a closure with an implicit argument (`it`)

⑤ which can be called directly using `(arg)`

⑥ or using `call`

Unlike a method, a closure **always** returns a value when called. The next section discusses how to declare closure arguments, when to use them and what is the implicit "it" parameter.

Parameters

Normal parameters

Parameters of closures follow the same principle as parameters of regular methods:

- an optional type
- a name
- an optional default value

Parameters are separated with commas:

```
def closureWithOneArg = { str -> str.toUpperCase() }
assert closureWithOneArg('groovy') == 'GROOVY'

def closureWithOneArgAndExplicitType = { String str -> str.toUpperCase() }
assert closureWithOneArgAndExplicitType('groovy') == 'GROOVY'

def closureWithTwoArgs = { a,b -> a+b }
assert closureWithTwoArgs(1,2) == 3
```

```
def closureWithTwoArgsAndExplicitTypes = { int a, int b -> a+b }
assert closureWithTwoArgsAndExplicitTypes(1,2) == 3

def closureWithTwoArgsAndOptionalTypes = { a, int b -> a+b }
assert closureWithTwoArgsAndOptionalTypes(1,2) == 3

def closureWithTwoArgAndDefaultValue = { int a, int b=2 -> a+b }
assert closureWithTwoArgAndDefaultValue(1) == 3
```

Implicit parameter

When a closure does not explicitly define a parameter list (using `->`), a closure **always** defines an implicit parameter, named `it`. This means that this code:

```
def greeting = { "Hello, $it!" }
assert greeting('Patrick') == 'Hello, Patrick!'
```

is strictly equivalent to this one:

```
def greeting = { it -> "Hello, $it!" }
assert greeting('Patrick') == 'Hello, Patrick!'
```

If you want to declare a closure which accepts no argument and must be restricted to calls without arguments, then you **must** declare it with an explicit empty argument list:

```
def magicNumber = { -> 42 }

// this call will fail because the closure doesn't accept any argument
magicNumber(11)
```

Varargs

It is possible for a closure to declare variable arguments like any other method. *Vargs* methods are methods that can accept a variable number of arguments if the last parameter is of variable length (or an array) like in the next examples:

```
def concat1 = { String... args -> args.join('') }           ①
assert concat1('abc','def') == 'abcdef'                     ②
def concat2 = { String[] args -> args.join('') }            ③
assert concat2('abc', 'def') == 'abcdef'

def multiConcat = { int n, String... args ->                ④
    args.join('')*n
}
assert multiConcat(2, 'abc','def') == 'abcdefabcdef'
```

① A closure accepting a variable number of strings as first parameter

② It may be called using any number of arguments **without** having to explicitly wrap them into an array

③ The same behavior is directly available if the *args* parameter is declared as an array

④ As long as the **last** parameter is an array or an explicit vargs type

Delegation strategy

Groovy closures vs lambda expressions

Groovy defines closures as instances of the Closure class. It makes it very different from lambda expressions in Java 8. Delegation is a key concept in Groovy closures which has no equivalent in lambdas. The ability to *change the delegate* or *change the delegation strategy* of closures make it possible to design beautiful domain specific languages (DSLs) in Groovy.

Owner, delegate and this

To understand the concept of delegate, we must first explain the meaning of `this` inside a closure. A closure actually defines 3 distinct things:

- `this` corresponds to the *enclosing class* where the closure is defined
- `owner` corresponds to the *enclosing object* where the closure is defined, which may be either a class or a closure
- `delegate` corresponds to a third party object where methods calls or properties are resolved whenever the receiver of the message is not defined

The meaning of this

In a closure, calling `getThisObject` will return the enclosing class where the closure is defined. It is equivalent to using an explicit `this`:

```
class Enclosing {
    void run() {
        def whatIsThisObject = { getThisObject() }          ①
        assert whatIsThisObject() == this                   ②
        def whatIsThis = { this }                           ③
        assert whatIsThis() == this                         ④
    }
}
class EnclosedInInnerClass {
    class Inner {
        Closure cl = { this }                               ⑤
    }
    void run() {
        def inner = new Inner()
        assert inner.cl() == inner                          ⑥
    }
}
```

```
class NestedClosures {
    void run() {
        def nestedClosures = {
            def cl = { this }                                   ⑦
            cl()
        }
        assert nestedClosures() == this                         ⑧
    }
}
```

① a closure is defined inside the `Enclosing` class, and returns `getThisObject`

② calling the closure will return the instance of `Enclosing` where the closure is defined

③ in general, you will just want to use the shortcut `this` notation

④ and it returns **exactly** the same object

⑤ if the closure is defined in an inner class

⑥ `this` in the closure **will** return the inner class, not the top-level one

⑦ in case of nested closures, like here `cl` being defined inside the scope of `nestedClosures`

⑧ then `this` corresponds to the closest outer class, not the enclosing closure!

It is of course possible to call methods from the enclosing class this way:

```
class Person {
    String name
    int age
    String toString() { "$name is $age years old" }

    String dump() {
        def cl = {
            String msg = this.toString()                ①
            println msg
            msg
        }
        cl()
    }
}
def p = new Person(name:'Janice', age:74)
assert p.dump() == 'Janice is 74 years old'
```

① the closure calls `toString` on `this`, which will actually call the `toString` method on the enclosing object, that is to say the `Person` instance

Owner of a closure

The owner of a closure is very similar to the definition of this in a closure with a subtle difference: it will return the direct enclosing object, be it a closure or a class:

```
class Enclosing {
    void run() {
        def whatIsOwnerMethod = { getOwner() }                ①
        assert whatIsOwnerMethod() == this                    ②
        def whatIsOwner = { owner }                           ③
        assert whatIsOwner() == this                          ④
    }
}
class EnclosedInInnerClass {
    class Inner {
        Closure cl = { owner }                                ⑤
    }
    void run() {
        def inner = new Inner()
        assert inner.cl() == inner                            ⑥
    }
}
class NestedClosures {
    void run() {
        def nestedClosures = {
            def cl = { owner }                                ⑦
            cl()
        }
        assert nestedClosures() == nestedClosures             ⑧
    }
}
```

① a closure is defined inside the `Enclosing` class, and returns `getOwner`

② calling the closure will return the instance of `Enclosing` where the closure is defined

③ in general, you will just want to use the shortcut `owner` notation

④ and it returns **exactly** the same object

⑤ if the closure is defined in an inner class

⑥ `owner` in the closure **will** return the inner class, not the top-level one

⑦ but in case of nested closures, like here `cl` being defined inside the scope of `nestedClosures`

⑧ then `owner` corresponds to the enclosing closure, hence a different object from `this`!

Delegate of a closure

The delegate of a closure can be accessed by using the `delegate` property or calling the `getDelegate` method. It is a powerful concept for building domain specific languages in Groovy. While `this` and `owner` refer to the lexical scope of a closure, the delegate is a user defined object that a closure will use. By default, the delegate is set to `owner`:

```
class Enclosing {
    void run() {
        def cl = { getDelegate() }                            ①
```

```
        def cl2 = { delegate }                          ②
        assert cl() == cl2()                            ③
        assert cl() == this                             ④
        def enclosed = {
            { -> delegate }.call()                      ⑤
        }
        assert enclosed() == enclosed                   ⑥
    }
}
```

① you can get the delegate of a closure calling the `getDelegate` method

② or using the `delegate` property

③ both return the same object

④ which is the enclosing class or closure

⑤ in particular in case of nested closures

⑥ `delegate` will correspond to the `owner`

The delegate of a closure can be changed to **any object**. Let's illustrate this by creating two classes which are not subclasses of each other but both define a property called `name`:

```
class Person {
    String name
}
class Thing {
    String name
}

def p = new Person(name: 'Norman')
def t = new Thing(name: 'Teapot')
```

Then let's define a closure which fetches the `name` property on the delegate:

```
def upperCasedName = { delegate.name.toUpperCase() }
```

Then by changing the delegate of the closure, you can see that the target object will change:

```
upperCasedName.delegate = p
assert upperCasedName() == 'NORMAN'
upperCasedName.delegate = t
assert upperCasedName() == 'TEAPOT'
```

At this point, the behavior is not different from having a `target` variable defined in the lexical scope of the closure:

```
def target = p
def upperCasedNameUsingVar = { target.name.toUpperCase() }
assert upperCasedNameUsingVar() == 'NORMAN'
```

However, there are major differences:

- in the last example, *target* is a local variable referenced from within the closure
- the delegate can be used transparently, that is to say without prefixing method calls with `delegate.` as explained in the next paragraph.

Delegation strategy

Whenever, in a closure, a property is accessed without explicitly setting a receiver object, then a delegation strategy is involved:

```
class Person {
    String name
}
def p = new Person(name:'Igor')
def cl = { name.toUpperCase() }                ①
cl.delegate = p                                ②
assert cl() == 'IGOR'                          ③
```

① `name` is not referencing a variable in the lexical scope of the closure

② we can change the delegate of the closure to be an instance of `Person`

③ and the method call will succeed

The reason this code works is that the `name` property will be resolved transparently on the `delegate` object! This is a very powerful way to resolve properties or method calls inside closures. There's no need to set an explicit `delegate.` receiver: the call will be made because the default delegation strategy of the closure makes it so. A closure actually defines multiple resolution strategies that you can choose:

- `Closure.OWNER_FIRST` is the **default strategy**. If a property/method exists on the **owner**, then it will be called on the owner. If not, then the **delegate** is used.
- `Closure.DELEGATE_FIRST` reverses the logic: the **delegate** is used first, then the **owner**
- `Closure.OWNER_ONLY` will only resolve the property/method lookup on the owner: the delegate will be ignored.
- `Closure.DELEGATE_ONLY` will only resolve the property/method lookup on the delegate: the owner will be ignored.
- `Closure.TO_SELF` can be used by developers who need advanced meta-programming techniques and wish to implement a custom resolution strategy: the resolution will not be made on the owner or the delegate but only on the closure class itself. It makes only sense to use this if you implement your own subclass of `Closure`.

Let's illustrate the default "owner first" strategy with this code:

```
class Person {
    String name
    def pretty = { "My name is $name" }             ①
    String toString() {
        pretty()
    }
}
class Thing {
    String name                                     ②
}

def p = new Person(name: 'Sarah')
def t = new Thing(name: 'Teapot')

assert p.toString() == 'My name is Sarah'           ③
p.pretty.delegate = t                               ④
assert p.toString() == 'My name is Sarah'           ⑤
```

① for the illustration, we define a closure member which references "name"

② both the `Person` and the `Thing` class define a `name` property

③ Using the default strategy, the `name` property is resolved on the owner first

④ so if we change the `delegate` to `t` which is an instance of `Thing`

⑤ there is no change in the result: `name` is first resolved on the `owner` of the closure

However, it is possible to change the resolution strategy of the closure:

```
p.pretty.resolveStrategy = Closure.DELEGATE_FIRST
assert p.toString() == 'My name is Teapot'
```

By changing the `resolveStrategy`, we are modifying the way Groovy will resolve the "implicit this" references: in this case, `name` will first be looked in the delegate, then if not found, on the owner. Since `name` is defined in the delegate, an instance of `Thing`, then this value is used.

The difference between "delegate first" and "delegate only" or "owner first" and "owner only" can be illustrated if one of the delegate (resp. owner) does **not** have such a method or property:

```
class Person {
    String name
    int age
    def fetchAge = { age }
}
class Thing {
    String name
}
```

```
def p = new Person(name:'Jessica', age:42)
def t = new Thing(name:'Printer')
def cl = p.fetchAge
cl.delegate = p
assert cl() == 42          ①
cl.delegate = t
assert cl() == 42          ①

cl.resolveStrategy = Closure.DELEGATE_ONLY
cl.delegate = p
assert cl() == 42          ②
cl.delegate = t
try {
    cl()                   ③
    assert false
} catch (MissingPropertyException ex) {
    // "age" is not defined on the delegate
}
```

① for "owner first" it doesn't matter what the delegate is

② for "delegate only" having `p` as the delegate succeeds

③ for "delegate only" having `t` as the delegate fails

In this example, we define two classes which both have a `name` property but only the `Person` class declares an `age`. The `Person` class also declares a closure which references `age`. We can change the default resolution strategy from "owner first" to "delegate only". Since the owner of the closure is the `Person` class, then we can check that if the delegate is an instance of `Person`, calling the closure is successful, but if we call it with a delegate being an instance of `Thing`, it fails with a `groovy.lang.MissingPropertyException`. Despite the closure being defined inside the `Person` class, the owner is not used.

NOTE	A comprehensive explanation about how to use this feature to develop DSLs can be found in a dedicated section of the manual.

Delegation strategy in the presence of metaprogramming

When describing the "owner first" delegation strategy we spoke about using a property/method from the owner if it "*existed*" otherwise using the respective property/method from the delegate. And a similar story for "delegate first" but in reverse. Instead of using the word "*existed*", it would have been more accurate to use the wording "*handled*". That means that for "owner first", if the property/method exists in the owner, or it has a propertyMissing/methodMissing hook, then the owner will handle the member access.

We can see this in action with a slightly altered version of our previous example:

```
class Person {
    String name
```

```
    int age
    def fetchAge = { age }
}
class Thing {
    String name
    def propertyMissing(String name) { -1 }
}

def p = new Person(name:'Jessica', age:42)
def t = new Thing(name:'Printer')
def cl = p.fetchAge
cl.resolveStrategy = Closure.DELEGATE_FIRST
cl.delegate = p
assert cl() == 42
cl.delegate = t
assert cl() == -1
```

In this example, even though our instance of the `Thing` class (our delegate for the last use of `cl`) has no `age` property, the fact that it handles the missing property via its `propertyMissing` hook, means that `age` will be `-1`.

Closures in GStrings

Take the following code:

```
def x = 1
def gs = "x = ${x}"
assert gs == 'x = 1'
```

The code behaves as you would expect, but what happens if you add:

```
x = 2
assert gs == 'x = 2'
```

You will see that the assert fails! There are two reasons for this:

- a GString only evaluates lazily the `toString` representation of values
- the syntax `${x}` in a GString does **not** represent a closure but an **expression** to `$x`, evaluated when the GString is created.

In our example, the `GString` is created with an expression referencing `x`. When the `GString` is created, the **value** of `x` is 1, so the `GString` is created with a value of 1. When the assert is triggered, the `GString` is evaluated and 1 is converted to a `String` using `toString`. When we change `x` to 2, we did change the value of `x`, but it is a different object, and the `GString` still references the old one.

TIP	A `GString` will only change its `toString` representation if the values it references are mutating. If the references change, nothing will happen.

If you need a real closure in a GString and for example enforce lazy evaluation of variables, you need to use the alternate syntax `${→ x}` like in the fixed example:

```
def x = 1
def gs = "x = ${-> x}"
assert gs == 'x = 1'

x = 2
assert gs == 'x = 2'
```

And let's illustrate how it differs from mutation with this code:

```
class Person {
    String name
    String toString() { name }          ①
}
def sam = new Person(name:'Sam')        ②
def lucy = new Person(name:'Lucy')      ③
def p = sam                             ④
def gs = "Name: ${p}"                   ⑤
assert gs == 'Name: Sam'                ⑥
p = lucy                                ⑦
assert gs == 'Name: Sam'                ⑧
sam.name = 'Lucy'                       ⑨
assert gs == 'Name: Lucy'               ⑩
```

① the `Person` class has a `toString` method returning the `name` property

② we create a first `Person` named *Sam*

③ we create another `Person` named *Lucy*

④ the `p` variable is set to `Sam`

⑤ and a closure is created, referencing the value of `p`, that is to say *Sam*

⑥ so when we evaluate the string, it returns *Sam*

⑦ if we change `p` to *Lucy*

⑧ the string still evaluates to *Sam* because it was the **value** of `p` when the `GString` was created

⑨ so if we mutate *Sam* to change the name to *Lucy*

⑩ this time the `GString` is correctly mutated

So if you don't want to rely on mutating objects or wrapping objects, you **must** use closures in `GString` by explicitly declaring an empty argument list:

```
class Person {
    String name
    String toString() { name }
}
```

```
def sam = new Person(name:'Sam')
def lucy = new Person(name:'Lucy')
def p = sam
// Create a GString with lazy evaluation of "p"
def gs = "Name: ${-> p}"
assert gs == 'Name: Sam'
p = lucy
assert gs == 'Name: Lucy'
```

Closure coercion

Closures can be converted into interfaces or single-abstract method types. Please refer to this section of the manual for a complete description.

Functional programming

Closures, like lambda expressions in Java 8 are at the core of the functional programming paradigm in Groovy. Some functional programming operations on functions are available directly on the `Closure` class, like illustrated in this section.

Currying

In Groovy, currying refers to the concept of partial application. It does **not** correspond to the real concept of currying in functional programming because of the different scoping rules that Groovy applies on closures. Currying in Groovy will let you set the value of one parameter of a closure, and it will return a new closure accepting one less argument.

Left currying

Left currying is the fact of setting the left-most parameter of a closure, like in this example:

```
def nCopies = { int n, String str -> str*n }    ①
def twice = nCopies.curry(2)                    ②
assert twice('bla') == 'blabla'                 ③
assert twice('bla') == nCopies(2, 'bla')        ④
```

① the `nCopies` closure defines two parameters

② `curry` will set the first parameter to `2`, creating a new closure (function) which accepts a single `String`

③ so the new function call be called with only a `String`

④ and it is equivalent to calling `nCopies` with two parameters

Right currying

Similarly to left currying, it is possible to set the right-most parameter of a closure:

```
def nCopies = { int n, String str -> str*n }    ①
```

```
def blah = nCopies.rcurry('bla')                ②
assert blah(2) == 'blabla'                      ③
assert blah(2) == nCopies(2, 'bla')             ④
```

① the `nCopies` closure defines two parameters

② `rcurry` will set the last parameter to `bla`, creating a new closure (function) which accepts a single `int`

③ so the new function call be called with only an `int`

④ and it is equivalent to calling `nCopies` with two parameters

Index based currying

In case a closure accepts more than 2 parameters, it is possible to set an arbitrary parameter using `ncurry`:

```
def volume = { double l, double w, double h -> l*w*h }      ①
def fixedWidthVolume = volume.ncurry(1, 2d)                 ②
assert volume(3d, 2d, 4d) == fixedWidthVolume(3d, 4d)       ③
def fixedWidthAndHeight = volume.ncurry(1, 2d, 4d)          ④
assert volume(3d, 2d, 4d) == fixedWidthAndHeight(3d)        ⑤
```

① the `volume` function defines 3 parameters

② `ncurry` will set the second parameter (index = 1) to `2d`, creating a new volume function which accepts length and height

③ that function is equivalent to calling `volume` omitting the width

④ it is also possible to set multiple parameters, starting from the specified index

⑤ the resulting function accepts as many parameters as the initial one minus the number of parameters set by `ncurry`

Currying functional interfaces

When working with `java.util.function` types — for example `BiPredicate`, `BiFunction` or `BiConsumer` — `Closure.rcurry` is not applicable. Two sibling utility classes provide right-partial helpers, differing only in the type returned:

- `org.apache.groovy.util.Lambdas#curryWith` returns a bare `Predicate`/`Function`/`Consumer`. Use when the result will flow into JDK APIs (e.g. `Stream.filter`, `Optional.map`) or into DGM methods that already accept SAM types (e.g. `partitionPoint`, `takeIf`, `groupConsecutive`).
- `org.apache.groovy.util.Closures#curryWith` returns a hybrid value that is both the SAM type **and** a `Closure`. Use when the result should also be usable with DGM methods that accept a `Closure` (e.g. `findAll`, `collect`, `each`, `find`, `any`, `every`).

```
import static org.apache.groovy.util.Lambdas.curryWith as curryWithSam
import static org.apache.groovy.util.Closures.curryWith as curryWithClosure
import java.util.function.BiPredicate
```

```
BiPredicate<Integer, Integer> divisibleBy = (n, d) -> n % d == 0

// Lambdas.curryWith: flows into JDK streams / SAM-accepting DGM
assert [1, 2, 3, 4, 5, 6].stream().filter(curryWithSam(divisibleBy, 2)).toList() == [
2, 4, 6]

// Closures.curryWith: flows into Closure-accepting DGM as well
assert [1, 2, 3, 4, 5, 6].findAll(curryWithClosure(divisibleBy, 2)) == [2, 4, 6]
```

In both forms a single allocation is paid per `curryWith` call rather than per element of the traversal.

`Closures.from(Predicate)` (and the `Function`/`Consumer` overloads) lifts an existing single-argument SAM into the same hybrid form, useful when an external API hands you a `Predicate` you want to feed into a `Closure`-accepting DGM method:

```
import static org.apache.groovy.util.Closures.from
import java.util.function.Predicate

Predicate<Integer> isEven = n -> n % 2 == 0
assert [1, 2, 3, 4].findAll(from(isEven)) == [2, 4]
```

Three overloads are provided:

- `curryWith(BiPredicate<T,P>, P) → Predicate<T>`
- `curryWith(BiFunction<T,P,R>, P) → Function<T,R>`
- `curryWith(BiConsumer<T,P>, P) → Consumer<T>`

Memoization

Memoization allows the result of the call of a closure to be cached. It is interesting if the computation done by a function (closure) is slow, but you know that this function is going to be called often with the same arguments. A typical example is the Fibonacci suite. A naive implementation may look like this:

```
def fib
fib = { long n -> n<2?n:fib(n-1)+fib(n-2) }
assert fib(15) == 610 // slow!
```

It is a naive implementation because 'fib' is often called recursively with the same arguments, leading to an exponential algorithm:

- computing `fib(15)` requires the result of `fib(14)` and `fib(13)`
- computing `fib(14)` requires the result of `fib(13)` and `fib(12)`

Since calls are recursive, you can already see that we will compute the same values again and again, although they could be cached. This naive implementation can be "fixed" by caching the

result of calls using `memoize`:

```
fib = { long n -> n<2?n:fib(n-1)+fib(n-2) }.memoize()
assert fib(25) == 75025 // fast!
```

WARNING	The cache works **using the actual values of the arguments**. This means that you should be very careful if you use memoization with something else than primitive or boxed primitive types.

The behavior of the cache can be tweaked using alternate methods:

- `memoizeAtMost` will generate a new closure which caches **at most** *n* values
- `memoizeAtLeast` will generate a new closure which caches **at least** *n* values
- `memoizeBetween` will generate a new closure which caches **at least** *n* values and **at most** *n* values

The cache used in all memoize variants is an LRU cache.

Composition

Closure composition corresponds to the concept of function composition, that is to say creating a new function by composing two or more functions (chaining calls), as illustrated in this example:

```
def plus2  = { it + 2 }
def times3 = { it * 3 }

def times3plus2 = plus2 << times3
assert times3plus2(3) == 11
assert times3plus2(4) == plus2(times3(4))

def plus2times3 = times3 << plus2
assert plus2times3(3) == 15
assert plus2times3(5) == times3(plus2(5))

// reverse composition
assert times3plus2(3) == (times3 >> plus2)(3)
```

Trampoline

Recursive algorithms are often restricted by a physical limit: the maximum stack height. For example, if you call a method that recursively calls itself too deep, you will eventually receive a `StackOverflowException`.

An approach that helps in those situations is by using `Closure` and its trampoline capability.

Closures are wrapped in a `TrampolineClosure`. Upon calling, a trampolined `Closure` will call the original `Closure` waiting for its result. If the outcome of the call is another instance of a `TrampolineClosure`, created perhaps as a result to a call to the `trampoline()` method, the `Closure` will

again be invoked. This repetitive invocation of returned trampolined Closures instances will continue until a value other than a trampolined `Closure` is returned. That value will become the final result of the trampoline. That way, calls are made serially, rather than filling the stack.

Here's an example of the use of `trampoline()` to implement the factorial function:

```
def factorial
factorial = { int n, def accu = 1G ->
    if (n < 2) return accu
    factorial.trampoline(n - 1, n * accu)
}
factorial = factorial.trampoline()

assert factorial(1)    == 1
assert factorial(3)    == 1 * 2 * 3
assert factorial(1000) // == 402387260.. plus another 2560 digits
```

Method pointers

It is often practical to be able to use a regular method as a closure. For example, you might want to use the currying abilities of a closure, but those are not available to normal methods. In Groovy, you can obtain a closure from any method with the method pointer operator.

Semantics

This chapter covers the semantics of the Groovy programming language.

Statements

Variable definition

Variables can be defined using either their type (like `String`) or by using the keyword `def` followed by a variable name:

```
String x
def y
```

`def` acts as a type placeholder, i.e. a replacement for the type name, when you do not want to give an explicit type. It could be that you don't care about the type at compile time or are relying on type inference (with Groovy's static nature). It is mandatory for variable definitions to have a type or placeholder. If left out, the type name will be deemed to refer to an existing variable (presumably declared earlier). For scripts, undeclared variables are assumed to come from the Script binding. In other cases, you will get a missing property (dynamic Groovy) or compile time error (static Groovy). If you think of `def` as an alias of `Object`, you will understand in an instant.

Variable definitions can provide an initial value, in which case it's like having a declaration and assignment (which we cover next) all in one.

NOTE

Variable definition types can be refined by using generics, like in `List<String> names`. To learn more about the generics support, please read the generics section.

NOTE

Java introduced the `var` reserved type from Java 10. It also acts like a type placeholder for variable definitions, similar to `def` above. So, for compatibility with Java, Groovy also lets you define variables using `var` as follows:

```
var z
```

In the context of variable definitions, you can think of `var` as an alias for `def`. You might use `var` if you have cut-n-pasted some Java code into your codebase, or if the audience (readers or maintainers) of your codebase are primarily Java-aware folks, and you want to make the code look familiar to them.

Variable assignment

You can assign values to variables for later use. Try the following:

```
x = 1
println x

x = new java.util.Date()
println x

x = -3.1499392
println x

x = false
println x

x = "Hi"
println x
```

Multiple assignment

Groovy supports multiple assignment, i.e. where multiple variables can be assigned at once, e.g.:

```
def (a, b, c) = [10, 20, 'foo']
assert a == 10 && b == 20 && c == 'foo'
```

You can provide types as part of the declaration if you wish:

```
def (int i, String j) = [10, 'foo']
assert i == 10 && j == 'foo'
```

As well as used when declaring variables it also applies to existing variables:

```
def nums = [1, 3, 5]
def a, b, c
(a, b, c) = nums
assert a == 1 && b == 3 && c == 5
```

The syntax works for arrays as well as lists, as well as methods that return either of these:

```
def (_, month, year) = "18th June 2009".split()
assert "In $month of $year" == 'In June of 2009'
```

Overflow and Underflow

If the left hand side has too many variables, excess ones are filled with null's:

```
def (a, b, c) = [1, 2]
assert a == 1 && b == 2 && c == null
```

If the right hand side has too many variables, the extra ones are ignored:

```
def (a, b) = [1, 2, 3]
assert a == 1 && b == 2
```

Object destructuring with multiple assignment

In the section describing Groovy's operators, the case of the subscript operator has been covered, explaining how you can override the `getAt()`/`putAt()` method.

With this technique, we can combine multiple assignments and the subscript operator methods to implement *object destructuring*.

Consider the following immutable `Coordinates` class, containing a pair of longitude and latitude doubles, and notice our implementation of the `getAt()` method:

```
@Immutable
class Coordinates {
    double latitude
    double longitude

    double getAt(int idx) {
        if (idx == 0) latitude
        else if (idx == 1) longitude
        else throw new Exception('Wrong coordinate index, use 0 or 1')
    }
```

```
}
```

Now let's instantiate this class and destructure its longitude and latitude:

```
def coordinates = new Coordinates(latitude: 43.23, longitude: 3.67) ①

def (la, lo) = coordinates                                           ②

assert la == 43.23                                                   ③
assert lo == 3.67
```

① we create an instance of the `Coordinates` class

② then, we use a multiple assignment to get the individual longitude and latitude values

③ and we can finally assert their values.

Control structures

Conditional structures

if / else

Groovy supports the usual if - else syntax from Java

```
def x = false
def y = false

if ( !x ) {
    x = true
}

assert x == true

if ( x ) {
    x = false
} else {
    y = true
}

assert x == y
```

Groovy also supports the normal Java "nested" if then else if syntax:

```
if ( ... ) {
    ...
} else if (...) {
    ...
} else {
```

```
    ...
}
```

switch / case

The switch statement in Groovy is backwards compatible with Java code; so you can fall through cases sharing the same code for multiple matches.

One difference though is that the Groovy switch statement can handle any kind of switch value and different kinds of matching can be performed.

```
def x = 1.23
def result = ""

switch (x) {
  case "foo":
    result = "found foo"
    // lets fall through

  case "bar":
    result += "bar"

  case [4, 5, 6, 'inList']:
    result = "list"
    break

  case 12..30:
    result = "range"
    break

  case Integer:
    result = "integer"
    break

  case Number:
    result = "number"
    break

  case ~/fo*/: // toString() representation of x matches the pattern?
    result = "foo regex"
    break

  case { it < 0 }: // or { x < 0 }
    result = "negative"
    break

  default:
    result = "default"
}
```

```
assert result == "number"
```

Switch supports the following kinds of comparisons:

- Class case values match if the switch value is an instance of the class
- Regular expression case values match if the `toString()` representation of the switch value matches the regex
- Collection case values match if the switch value is contained in the collection. This also includes ranges (since they are Lists)
- Closure case values match if the calling the closure returns a result which is true according to the Groovy truth
- If none of the above are used then the case value matches if the case value equals the switch value

NOTE	When using a closure case value, the default `it` parameter is actually the switch value (in our example, variable `x`).

Groovy also supports switch expressions as specified by JEP 361: Switch Expressions (GROOVY-12255). A switch expression produces a value. Arms may use arrow labels (`case L →`), which do not fall through, or traditional colon labels (`case L:`) with an explicit `yield`. A switch expression must complete by yielding a value or by throwing; unlike a switch statement, an unmatched selector throws `IllegalStateException`. Under `@TypeChecked` / `@CompileStatic` a non-exhaustive switch expression (no `default`, and not a complete enum) is a compile-time error.

As in Java, the arrow only decides fall-through; whether a `switch` is a statement or an expression is decided by its position (GROOVY-12399). A `switch` whose value is not used is a statement, whatever its arm shape, and simply does nothing when unmatched. That includes a `switch` that is the last statement of a closure, a script or a method: the implicit return applies to the statement, exactly as it does to an `if` or a colon-form `switch`, so an unmatched selector leaves the body's value at the return type's default. Use `return switch (…) {…}`, or assign the result, for the expression rules (GROOVY-12408).

```
def partner = switch(person) {
    case 'Romeo'  -> 'Juliet'
    case 'Adam'   -> 'Eve'
    case 'Antony' -> 'Cleopatra'
    case 'Bonnie' -> 'Clyde'
}
```

Arrow arms that need more than one statement use a block and `yield`:

```
int n = 2
int value = switch (n) {
    case 1 -> 10
    case 2 -> {
```

```
        int doubled = n * 10
        yield doubled
    }
    default -> throw new IllegalArgumentException(String.valueOf(n))
}
```

Multiple constants may share an arm, and Groovy's usual `isCase` matching (Class, regex, Collection, Closure) is available in both the statement and the expression forms. Under static compilation, when every case label is an `int`, `String` or enum constant, the compiler emits `tableswitch` / `lookupswitch` bytecode in the same style as `javac`.

Looping structures

Classic for loop

Groovy supports the standard Java / C for loop:

```
String message = ''
for (int i = 0; i < 5; i += 1) {
    message += 'Hi '
}
assert message == 'Hi Hi Hi Hi Hi '
```

Enhanced classic Java-style for loop

The more elaborate form of Java's classic for loop with comma-separate expressions is now supported. Example:

```
def facts = []
def count = 5
for (int fact = 1, i = 1; i <= count; i++, fact *= i) {
    facts << fact
}
assert facts == [1, 2, 6, 24, 120]
```

Multi-assignment in combination with for loop

Groovy has supported multi-assignment statements since Groovy 1.6:

```
// multi-assignment with types
def (String x, int y) = ['foo', 42]
assert "$x $y" == 'foo 42'
```

These can now appear in for loops:

```
// multi-assignment goes loopy
```

```
def baNums = []
for (def (String u, int v) = ['bar', 42]; v < 45; u++, v++) {
    baNums << "$u $v"
}
assert baNums == ['bar 42', 'bas 43', 'bat 44']
```

for in loop

The for loop in Groovy is much simpler and works with any kind of array, collection, Map, etc.

```
// iterate over a range
def x = 0
for ( i in 0..9 ) {
    x += i
}
assert x == 45

// iterate over a list
x = 0
for ( i in [0, 1, 2, 3, 4] ) {
    x += i
}
assert x == 10

// iterate over an array
x = 0
for ( i in new int[]{0, 1, 2, 3, 4} ) {
    x += i
}
assert x == 10

// iterate over a map
def map = [a:1, b:2, c:3]
x = 0
for ( e in map ) {
    x += e.value
}
assert x == 6

// iterate over values in a map
x = 0
for ( v in map.values() ) {
    x += v
}
assert x == 6

// iterate over the characters in a string
def list = []
for ( c in 'abc' ) {
    list.add(c)
```

```
}
assert list == ['a', 'b', 'c']

// iterate with index
for ( int i, k in map.keySet() ) {
    assert map.get(k) == i + 1
}
```

NOTE	Groovy also supports the Java colon variation with colons: `for (char c : text) {}`

NOTE	**Closure, lambda, and AIC capture of for-in variables (GROOVY-11792).** When a for-in loop variable is shared with a deferred closure, lambda, or anonymous inner class, each iteration uses a fresh binding so the capture observes the value from the iteration that created it (similar in spirit to Java enhanced-for / lambda capture for deferred use). Assignments to the loop variable within the same iteration remain visible to captures created in that iteration. Classic `for` / `while` loops are unchanged. To restore the historical single shared binding (every capture sees the final loop value), set system property `groovy.forin.per.iteration.capture=false` or call `CompilerConfiguration.setForInPerIterationCaptureEnabled(false)` (language-compatibility switch, independent of optimization options including `"all" → false`).

while loop

Groovy supports the usual while {...} loops like Java:

```
def x = 0
def y = 5

while ( y-- > 0 ) {
    x++
}

assert x == 5
```

do/while loop

Java's class do/while loop is now supported. Example:

```
// classic Java-style do..while loop
def count = 5
def fact = 1
do {
    fact *= count--
} while(count > 1)
assert fact == 120
```

Exception handling

Exception handling is the same as Java.

try / catch / finally

You can specify a complete `try-catch-finally`, a `try-catch`, or a `try-finally` set of blocks.

NOTE	Braces are required around each block's body.

```
try {
    'moo'.toLong()   // this will generate an exception
    assert false     // asserting that this point should never be reached
} catch ( e ) {
    assert e in NumberFormatException
}
```

We can put code within a 'finally' clause following a matching 'try' clause, so that regardless of whether the code in the 'try' clause throws an exception, the code in the finally clause will always execute:

```
def z
try {
    def i = 7, j = 0
    try {
        def k = i / j
        assert false        //never reached due to Exception in previous line
    } finally {
        z = 'reached here'  //always executed even if Exception thrown
    }
} catch ( e ) {
    assert e in ArithmeticException
    assert z == 'reached here'
}
```

Multi-catch

With the multi catch block (since Groovy 2.0), we're able to define several exceptions to be catch and treated by the same catch block:

```
try {
    /* ... */
} catch ( IOException | NullPointerException e ) {
    /* one block to handle 2 exceptions */
}
```

ARM Try with resources

Groovy often provides better alternatives to Java 7's `try`-with-resources statement for Automatic Resource Management (ARM). That syntax is now supported for Java programmers migrating to Groovy and still wanting to use the old style:

```
class FromResource extends ByteArrayInputStream {
    @Override
    void close() throws IOException {
        super.close()
        println "FromResource closing"
    }

    FromResource(String input) {
        super(input.toLowerCase().bytes)
    }
}

class ToResource extends ByteArrayOutputStream {
    @Override
    void close() throws IOException {
        super.close()
        println "ToResource closing"
    }
}

def wrestle(s) {
    try (
            FromResource from = new FromResource(s)
            ToResource to = new ToResource()
    ) {
        to << from
        return to.toString()
    }
}

def wrestle2(s) {
    FromResource from = new FromResource(s)
    try (from; ToResource to = new ToResource()) { // Enhanced try-with-resources in
Java 9+
        to << from
        return to.toString()
    }
}

assert wrestle("ARM was here!").contains('arm')
assert wrestle2("ARM was here!").contains('arm')
```

Which yields the following output:

```
ToResource closing
FromResource closing
ToResource closing
FromResource closing
```

Power assertion

Unlike Java with which Groovy shares the `assert` keyword, the latter in Groovy behaves very differently. First of all, an assertion in Groovy is always executed, independently of the `-ea` flag of the JVM. It makes this a first class choice for unit tests. The notion of "power asserts" is directly related to how the Groovy `assert` behaves.

A power assertion is decomposed into 3 parts:

```
assert [left expression] == [right expression] : (optional message)
```

The result of the assertion is very different from what you would get in Java. If the assertion is true, then nothing happens. If the assertion is false, then it provides a visual representation of the value of each sub-expressions of the expression being asserted. For example:

```
assert 1+1 == 3
```

Will yield:

```
Caught: Assertion failed:

assert 1+1 == 3
        |  |
        2  false
```

Power asserts become very interesting when the expressions are more complex, like in the next example:

```
def x = 2
def y = 7
def z = 5
def calc = { a,b -> a*b+1 }
assert calc(x,y) == [x,z].sum()
```

Which will print the value for each sub-expression:

```
assert calc(x,y) == [x,z].sum()
       |    | |  |   | |  |
       15   2 7  |   2 5  7
```

```
                false
```

In case you don't want a pretty printed error message like above, you can fall back to a custom error message by changing the optional message part of the assertion, like in this example:

```
def x = 2
def y = 7
def z = 5
def calc = { a,b -> a*b+1 }
assert calc(x,y) == z*z : 'Incorrect computation result'
```

Which will print the following error message:

```
Incorrect computation result. Expression: (calc.call(x, y) == (z * z)). Values: z = 5,
z = 5
```

Labeled statements

Any statement can be associated with a label. Labels do not impact the semantics of the code and can be used to make the code easier to read like in the following example:

```
given:
    def x = 1
    def y = 2
when:
    def z = x+y
then:
    assert z == 3
```

It is important to understand that by default labels have no impact on the semantics of the code, however they belong to the abstract syntax tree (AST) so it is possible for an AST transformation to use that information to perform transformations over the code, hence leading to different semantics. This is in particular what the Spock Framework does to make testing easier.

It is possible to use labels in the `break` instruction as a target for jump, as in the next example:

```
out:for (int i=0; i<9; i++) {
    for (int j=0; j<i; j++) {
        println "j=$j"
        if (j == 5) {
            break out
        }
    }
    println "i=$i"
}
```

`instanceof` / `!instanceof` pattern variables (JEP 394)

Since Groovy 6, a type pattern on `instanceof` or `!instanceof` introduces a **pattern variable** whose scope follows Java's flow rules (JLS §6.3 / JEP 394) for the common shapes used in application code:

```
Object o = 'hi'
if (o instanceof String s) {
    assert s.toUpperCase() == 'HI'   // s is a local in the then-block
} else {
    // s is *not* in scope here (dynamic: MissingPropertyException; static: error)
}
// s is not in scope after a fall-through if either

if (o !instanceof Integer i) {
    // i is *not* in scope here (condition true means o is not an Integer)
} else {
    assert i instanceof Integer      // i is bound when !instanceof is false
}
```

The form `e !instanceof T t` is equivalent to `!(e instanceof T t)` for both binding and scoping. Short-circuit boolean operators and ternary arms respect the same "definitely bound on this path" rules (`&&` exposes true-path bindings on the right-hand side; `||` does not). Under `@TypeChecked` / `@CompileStatic` the type checker rejects out-of-scope uses at compile time; in dynamic Groovy the same uses become dynamic property lookups and typically throw `groovy.lang.MissingPropertyException` at runtime.

NOTE	**While loops (intentional partial support).** A `while` condition's true-path pattern variables are in scope in the loop body, and short-circuit rules apply inside the condition. Groovy does **not** introduce false-path bindings **after** the loop when the body cannot complete normally (unlike the full JLS while rule). Pattern names never leak past the loop. Full after-loop introduction may be considered later; it is not part of the 6.0 contract.

This is a **breaking change** from earlier Groovy releases that left pattern locals visible more widely in dynamic mode. See GROOVY-12242 and `COMPATIBILITY.md`.

Expressions

Expressions are the building blocks of Groovy programs that are used to reference existing values and execute code to create new ones.

Groovy supports many of the same kinds of expressions as Java, including:

Table 6. Expressions like Java

Example expression(s)	Description
`foo`	the name of a variable, field, parameter, …
`this`, `super`, `it`	special names

`true`, `10`, `"bar"`	literals
`String.class`	Class literal
`( expression )`	parenthesised expressions
`foo++`, `~bar`	Unary operator expressions
`foo + bar`, `bar * baz`	Binary operator expressions
`foo ? bar : baz`	Ternary operator expressions
`(Integer x, Integer y) -> x + y`	Lambda expressions

`assert 'bar' == switch('foo') {`
`  case 'foo' -> 'bar'`
`  default  -> 'other'`
`}`
	switch expressions

Groovy also has some of its own special expressions:

Table 7. Special expressions

Example expression(s)	Description
`String`	Abbreviated class literal (when not ambiguous)
`{ x, y -> x + y }`	Closure expressions
`[1, 3, 5]`	literal list expressions
`[a:2, b:4, c:6]`	literal map expressions

Groovy also expands on the normal dot-notation used in Java for member access. Groovy provides special support for accessing hierarchical data structures by specifying the path in the hierarchy of some data of interest. These *Groovy path* expressions are known as GPath expressions.

GPath expressions

`GPath` is a path expression language integrated into Groovy which allows parts of nested structured data to be identified. In this sense, it has similar aims and scope as XPath does for XML. GPath is often used in the context of processing XML, but it really applies to any object graph. Where XPath uses a filesystem-like path notation, a tree hierarchy with parts separated by a slash `/`, GPath **use a dot-object notation** to perform object navigation.

As an example, you can specify a path to an object or element of interest:

- `a.b.c` → for XML, yields all the `c` elements inside `b` inside `a`
- `a.b.c` → for POJOs, yields the `c` properties for all the `b` properties of `a` (sort of like `a.getB().getC()` in JavaBeans)

In both cases, the GPath expression can be viewed as a query on an object graph. For POJOs, the object graph is most often built by the program being written through object instantiation and composition; for XML processing, the object graph is the result of `parsing` the XML text, most often with classes like XmlParser or XmlSlurper. See Processing XML for more in-depth details on

consuming XML in Groovy.

TIP

When querying the object graph generated from XmlParser or XmlSlurper, a GPath expression can refer to attributes defined on elements with the `@` notation:

- `a["@href"]` → map-like notation : the href attribute of all the a elements
- `a.'@href'` → property notation : an alternative way of expressing this
- `a.@href` → direct notation : yet another alternative way of expressing this

Object navigation

Let's see an example of a GPath expression on a simple *object graph*, the one obtained using java reflection. Suppose you are in a non-static method of a class having another method named `aMethodFoo`

```
void aMethodFoo() { println "This is aMethodFoo." } ⑳
```

the following GPath expression will get the name of that method:

```
assert ['aMethodFoo'] == this.class.methods.name.grep(~/.*Foo/)
```

More precisely, the above GPath expression produces a list of String, each being the name of an existing method on `this` where that name ends with `Foo`.

Now, given the following methods also defined in that class:

```
void aMethodBar() { println "This is aMethodBar." } ①
void anotherFooMethod() { println "This is anotherFooMethod." } ②
void aSecondMethodBar() { println "This is aSecondMethodBar." } ③
```

then the following GPath expression will get the names of **(1)** and **(3)**, but not **(2)** or **(0)**:

```
assert ['aMethodBar', 'aSecondMethodBar'] as Set == this.class.methods.name.grep
(~/.*Bar/) as Set
```

Expression Deconstruction

We can decompose the expression `this.class.methods.name.grep(~/.*Bar/)` to get an idea of how a GPath is evaluated:

`this.class`
: property accessor, equivalent to `this.getClass()` in Java, yields a `Class` object.

`this.class.methods`
: property accessor, equivalent to `this.getClass().getMethods()`, yields an array of `Method` objects.

`this.class.methods.name`

apply a property accessor on each element of an array and produce a list of the results.

`this.class.methods.name.grep(…)`

call method `grep` on each element of the list yielded by `this.class.methods.name` and produce a list of the results.

WARNING	A sub-expression like `this.class.methods` yields an array because this is what calling `this.getClass().getMethods()` in Java would produce. `GPath` expressions do not have a convention where a `s` means a list or anything like that.

One powerful feature of GPath expression is that property access on a collection is converted to a *property access on each element of the collection* with the results collected into a collection. Therefore, the expression `this.class.methods.name` could be expressed as follows in Java:

```
List<String> methodNames = new ArrayList<String>();
for (Method method : this.getClass().getMethods()) {
   methodNames.add(method.getName());
}
return methodNames;
```

Array access notation can also be used in a GPath expression where a collection is present :

```
assert 'aSecondMethodBar' == this.class.methods.name.grep(~/.*Bar/).sort()[1]
```

NOTE	array access are zero-based in GPath expressions

GPath for XML navigation

Here is an example with an XML document and various form of GPath expressions:

```
def xmlText = """
              | <root>
              |   <level>
              |      <sublevel id='1'>
              |        <keyVal>
              |          <key>mykey</key>
              |          <value>value 123</value>
              |        </keyVal>
              |      </sublevel>
              |      <sublevel id='2'>
              |        <keyVal>
              |          <key>anotherKey</key>
              |          <value>42</value>
              |        </keyVal>
              |        <keyVal>
              |          <key>mykey</key>
```

```
                |          <value>fizzbuzz</value>
                |        </keyVal>
                |      </sublevel>
                |   </level>
                | </root>
                """
def root = new XmlSlurper().parseText(xmlText.stripMargin())
assert root.level.size() == 1 ①
assert root.level.sublevel.size() == 2 ②
assert root.level.sublevel.findAll { it.@id == 1 }.size() == 1 ③
assert root.level.sublevel[1].keyVal[0].key.text() == 'anotherKey' ④
```

① There is one `level` node under `root`

② There are two `sublevel` nodes under `root/level`

③ There is one element `sublevel` having an attribute `id` with value `1`

④ Text value of `key` element of first `keyVal` element of second `sublevel` element under `root/level` is 'anotherKey'

Further details about GPath expressions for XML are in the XML User Guide.

Promotion and coercion

Number promotion

The rules of number promotion are specified in the section on math operations.

Closure to type coercion

Assigning a closure to a SAM type

A SAM type is a type which defines a single abstract method. This includes:

Functional interfaces

```
interface Predicate<T> {
    boolean accept(T obj)
}
```

Abstract classes with single abstract method

```
abstract class Greeter {
    abstract String getName()
    void greet() {
        println "Hello, $name"
    }
}
```

Any closure can be converted into a SAM type using the `as` operator:

```
Predicate filter = { it.contains 'G' } as Predicate
assert filter.accept('Groovy') == true

Greeter greeter = { 'Groovy' } as Greeter
greeter.greet()
```

However, the `as Type` expression is optional since Groovy 2.2.0. You can omit it and simply write:

```
Predicate filter = { it.contains 'G' }
assert filter.accept('Groovy') == true

Greeter greeter = { 'Groovy' }
greeter.greet()
```

which means you are also allowed to use method pointers, as shown in the following example:

```
boolean doFilter(String s) { s.contains('G') }

Predicate filter = this.&doFilter
assert filter.accept('Groovy') == true

Greeter greeter = GroovySystem.&getVersion
greeter.greet()
```

Calling a method accepting a SAM type with a closure

The second and probably more important use case for closure to SAM type coercion is calling a method which accepts a SAM type. Imagine the following method:

```
public <T> List<T> filter(List<T> source, Predicate<T> predicate) {
    source.findAll { predicate.accept(it) }
}
```

Then you can call it with a closure, without having to create an explicit implementation of the interface:

```
assert filter(['Java','Groovy'], { it.contains 'G'} as Predicate) == ['Groovy']
```

But since Groovy 2.2.0, you are also able to omit the explicit coercion and call the method as if it used a closure:

```
assert filter(['Java','Groovy']) { it.contains 'G'} == ['Groovy']
```

As you can see, this has the advantage of letting you use the closure syntax for method calls, that is

to say put the closure outside the parenthesis, improving the readability of your code.

Closure to arbitrary type coercion

In addition to SAM types, a closure can be coerced to any type and in particular interfaces. Let's define the following interface:

```
interface FooBar {
    int foo()
    void bar()
}
```

You can coerce a closure into the interface using the `as` keyword:

```
def impl = { println 'ok'; 123 } as FooBar
```

This produces a class for which all methods are implemented using the closure:

```
assert impl.foo() == 123
impl.bar()
```

But it is also possible to coerce a closure to any class. For example, we can replace the `interface` that we defined with `class` without changing the assertions:

```
class FooBar {
    int foo() { 1 }
    void bar() { println 'bar' }
}

def impl = { println 'ok'; 123 } as FooBar

assert impl.foo() == 123
impl.bar()
```

Map to type coercion

Usually using a single closure to implement an interface or a class with multiple methods is not the way to go. As an alternative, Groovy allows you to coerce a map into an interface or a class. In that case, keys of the map are interpreted as method names, while the values are the method implementation. The following example illustrates the coercion of a map into an `Iterator`:

```
def map
map = [
  i: 10,
  hasNext: { map.i > 0 },
```

```
  next: { map.i-- },
]
def iter = map as Iterator
```

Of course this is a rather contrived example, but illustrates the concept. You only need to implement those methods that are actually called, but if a method is called that doesn't exist in the map a `MissingMethodException` or an `UnsupportedOperationException` is thrown, depending on the arguments passed to the call, as in the following example:

```
interface X {
    void f()
    void g(int n)
    void h(String s, int n)
}

x = [ f: {println "f called"} ] as X
x.f() // method exists
x.g() // MissingMethodException here
x.g(5) // UnsupportedOperationException here
```

The type of the exception depends on the call itself:

- `MissingMethodException` if the arguments of the call do not match those from the interface/class
- `UnsupportedOperationException` if the arguments of the call match one of the overloaded methods of the interface/class

Intersection-type cast and coercion

Since Groovy 6.0, the cast and `as` operators accept *intersection types* — a class component (at most one) and any number of interface components joined by `&`, mirroring Java's JLS §15.16 intersection cast. The most common use is to opt a lambda or method reference into `Serializable`:

```
// A lambda that is also Serializable
Runnable r = (Runnable & java.io.Serializable) () -> println("hi")
assert r instanceof Runnable
assert r instanceof java.io.Serializable
r.run()
```

For statically compiled lambdas and method references, the additional interfaces are threaded through `LambdaMetafactory.altMetafactory` via `FLAG_MARKERS` / `FLAG_SERIALIZABLE` — there is no runtime proxy and the result implements every component natively, so it can be serialised and restored:

```
@CompileStatic
class T {
    static Runnable make() {
        return (Runnable & java.io.Serializable) () -> println("hi")
```

```
    }
}

// The lambda factory uses LambdaMetafactory.altMetafactory with
// FLAG_SERIALIZABLE so the result can be safely written to and read
// from an ObjectStream.
def r = T.make()
def baos = new ByteArrayOutputStream()
new ObjectOutputStream(baos).withCloseable { it.writeObject(r) }
def restored = null
new ObjectInputStream(new ByteArrayInputStream(baos.toByteArray())).withCloseable {
    restored = it.readObject()
}
assert restored instanceof Runnable
```

The same syntax works in the `as` form, with parentheses around the intersection:

```
// Closures already implement Runnable, Serializable and Cloneable, so
// intersections of those is an identity coercion (no proxy is built).
def cl = { -> 1 }
def coerced = cl as (Runnable & java.io.Serializable)
assert coerced.is(cl) // same instance
```

For closure literals and maps, `as` builds a `groovy.util.ProxyGenerator` aggregate that implements every interface component:

```
interface Action { void perform() }

def calls = 0
def proxy = ([perform: { -> calls++ }] as (Action & java.io.Serializable))
assert proxy instanceof Action
assert proxy instanceof java.io.Serializable
proxy.perform()
proxy.perform()
assert calls == 2
```

Method references support the same intersection-cast forms:

```
@CompileStatic
class T {
    static Function<String, Integer> make() {
        return (Function<String, Integer> & java.io.Serializable) String::length
    }
}

Function<String, Integer> f = T.make()
assert f instanceof java.io.Serializable
```

```
assert f.apply("hello") == 5
```

The well-formedness rules follow the JLS:

- at most one component may be a class — and if present, it must come first;
- the class component (if any) must not be `final`;
- primitive components are not allowed;
- when the operand is a lambda, method reference or closure literal, exactly one component may have an abstract method (the SAM-bearing functional interface). The remainder must be marker-compatible interfaces (no abstract methods, or abstract methods identical to the SAM).

String to enum coercion

Groovy allows transparent `String` (or `GString`) to enum values coercion. Imagine you define the following enum:

```
enum State {
    up,
    down
}
```

then you can assign a string to the enum without having to use an explicit `as` coercion:

```
State st = 'up'
assert st == State.up
```

It is also possible to use a `GString` as the value:

```
def val = "up"
State st = "${val}"
assert st == State.up
```

However, this would throw a runtime error (`IllegalArgumentException`):

```
State st = 'not an enum value'
```

Note that it is also possible to use implicit coercion in switch statements:

```
State switchState(State st) {
    switch (st) {
        case 'up':
            return State.down // explicit constant
        case 'down':
            return 'up' // implicit coercion for return types
```

```
    }
}
```

in particular, see how the `case` use string constants. But if you call a method that uses an enum with a `String` argument, you still have to use an explicit `as` coercion:

```
assert switchState('up' as State) == State.down
assert switchState(State.down) == State.up
```

Custom type coercion

It is possible for a class to define custom coercion strategies by implementing the `asType` method. Custom coercion is invoked using the `as` operator and is never implicit. As an example, imagine you defined two classes, `Polar` and `Cartesian`, like in the following example:

```
class Polar {
    double r
    double phi
}
class Cartesian {
   double x
   double y
}
```

And that you want to convert from polar coordinates to cartesian coordinates. One way of doing this is to define the `asType` method in the `Polar` class:

```
def asType(Class target) {
    if (Cartesian==target) {
        return new Cartesian(x: r*cos(phi), y: r*sin(phi))
    }
}
```

which allows you to use the `as` coercion operator:

```
def sigma = 1E-16
def polar = new Polar(r:1.0,phi:PI/2)
def cartesian = polar as Cartesian
assert abs(cartesian.x-sigma) < sigma
```

Putting it all together, the `Polar` class looks like this:

```
class Polar {
    double r
    double phi
```

```
    def asType(Class target) {
        if (Cartesian==target) {
            return new Cartesian(x: r*cos(phi), y: r*sin(phi))
        }
    }
}
```

but it is also possible to define `asType` outside of the `Polar` class, which can be practical if you want to define custom coercion strategies for "closed" classes or classes for which you don't own the source code, for example using a metaclass:

```
Polar.metaClass.asType = { Class target ->
    if (Cartesian==target) {
        return new Cartesian(x: r*cos(phi), y: r*sin(phi))
    }
}
```

Class literals vs variables and the as operator

Using the `as` keyword is only possible if you have a static reference to a class, like in the following code:

```
interface Greeter {
    void greet()
}
def greeter = { println 'Hello, Groovy!' } as Greeter // Greeter is known statically
greeter.greet()
```

But what if you get the class by reflection, for example by calling `Class.forName`?

```
Class clazz = Class.forName('Greeter')
```

Trying to use the reference to the class with the `as` keyword would fail:

```
greeter = { println 'Hello, Groovy!' } as clazz
// throws:
// unable to resolve class clazz
// @ line 9, column 40.
//   greeter = { println 'Hello, Groovy!' } as clazz
```

It is failing because the `as` keyword only works with class literals. Instead, you need to call the `asType` method:

```
greeter = { println 'Hello, Groovy!' }.asType(clazz)
```

```
greeter.greet()
```

Optionality

Optional parentheses

Method calls can omit the parentheses if there is at least one parameter and there is no ambiguity:

```
println 'Hello World'
def maximum = Math.max 5, 10
```

Parentheses are required for method calls without parameters or ambiguous method calls:

```
println()
println(Math.max(5, 10))
```

Optional semicolons

In Groovy semicolons at the end of the line can be omitted, if the line contains only a single statement.

This means that:

```
assert true;
```

can be more idiomatically written as:

```
assert true
```

Multiple statements in a line require semicolons to separate them:

```
boolean a = true; assert a
```

Optional return keyword

In Groovy, the last expression evaluated in the body of a method or a closure is returned. This means that the `return` keyword is optional.

```
int add(int a, int b) {
    return a+b
}
assert add(1, 2) == 3
```

Can be shortened to:

```
int add(int a, int b) {
    a+b
}
assert add(1, 2) == 3
```

Optional public keyword

By default, Groovy classes and methods are `public`. Therefore this class:

```
public class Server {
    public String toString() { "a server" }
}
```

is identical to this class:

```
class Server {
    String toString() { "a server" }
}
```

The Groovy Truth

Groovy decides whether an expression is true or false by applying the rules given below.

Boolean expressions

True if the corresponding Boolean value is `true`.

```
assert true
assert !false
```

Collections and Arrays

Non-empty Collections and arrays are true.

```
assert [1, 2, 3]
assert ![]
```

Matchers

True if the Matcher has at least one match.

```
assert ('a' =~ /a/)
```

```
assert !('a' =~ /b/)
```

Iterators and Enumerations

Iterators and Enumerations with further elements are coerced to true.

```
assert [0].iterator()
assert ![].iterator()
Vector v = [0] as Vector
Enumeration enumeration = v.elements()
assert enumeration
enumeration.nextElement()
assert !enumeration
```

Maps

Non-empty Maps are evaluated to true.

```
assert ['one' : 1]
assert ![:]
```

Strings

Non-empty Strings, GStrings and CharSequences are coerced to true.

```
assert 'a'
assert !''
def nonEmpty = 'a'
assert "$nonEmpty"
def empty = ''
assert !"$empty"
```

Numbers

Non-zero numbers are true.

```
assert 1
assert 3.5
assert !0
```

Object References

Non-null object references are coerced to true.

```
assert new Object()
```

```
assert !null
```

Customizing the truth with asBoolean() methods

In order to customize whether groovy evaluates your object to `true` or `false` implement the `asBoolean()` method:

```
class Color {
    String name

    boolean asBoolean(){
        name == 'green' ? true : false
    }
}
```

Groovy will call this method to coerce your object to a boolean value, e.g.:

```
assert new Color(name: 'green')
assert !new Color(name: 'red')
```

Typing

Optional typing

Optional typing is the idea that a program can work even if you don't put an explicit type on a variable. Being a dynamic language, Groovy naturally implements that feature, for example when you declare a variable:

```
String aString = 'foo'                  ①
assert aString.toUpperCase()            ②
```

① `foo` is declared using an explicit type, `String`

② we can call the `toUpperCase` method on a `String`

Groovy will let you write this instead:

```
def aString = 'foo'                     ①
assert aString.toUpperCase()            ②
```

① `foo` is declared using `def`

② we can still call the `toUpperCase` method, because the type of `aString` is resolved at runtime

So it doesn't matter that you use an explicit type here. It is in particular interesting when you combine this feature with static type checking, because the type checker performs type inference.

Likewise, Groovy doesn't make it mandatory to declare the types of a parameter in a method:

```
String concat(String a, String b) {
    a+b
}
assert concat('foo','bar') == 'foobar'
```

can be rewritten using `def` as both return type and parameter types, in order to take advantage of duck typing, as illustrated in this example:

```
def concat(def a, def b) {                          ①
    a+b
}
assert concat('foo','bar') == 'foobar'              ②
assert concat(1,2) == 3                             ③
```

① both the return type and the parameter types use `def`

② it makes it possible to use the method with `String`

③ but also with `int` since the `plus` method is defined

TIP Using the `def` keyword here is recommended to describe the intent of a method which is supposed to work on any type, but technically, we could use `Object` instead and the result would be the same: `def` is, in Groovy, strictly equivalent to using `Object`.

Eventually, the type can be removed altogether from both the return type and the descriptor. But if you want to remove it from the return type, you then need to add an explicit modifier for the method, so that the compiler can make a difference between a method declaration and a method call, like illustrated in this example:

```
private concat(a,b) {                               ①
    a+b
}
assert concat('foo','bar') == 'foobar'              ②
assert concat(1,2) == 3                             ③
```

① if we want to omit the return type, an explicit modifier has to be set.

② it is still possible to use the method with `String`

③ and also with `int`

TIP Omitting types is in general considered a bad practice in method parameters or method return types for public APIs. While using `def` in a local variable is not really a problem because the visibility of the variable is limited to the method itself, while set on a method parameter, `def` will be converted to `Object` in the method signature, making it difficult for users to know which is the expected type of the arguments. This means that you should limit this to cases where you are explicitly relying on duck

typing.

Static type checking

By default, Groovy performs minimal type checking at compile time. Since it is primarily a dynamic language, most checks that a static compiler would normally do aren't possible at compile time. A method added via runtime metaprogramming might alter a class or object's runtime behavior. Let's illustrate why in the following example:

```
class Person {                                                  ①
    String firstName
    String lastName
}
def p = new Person(firstName: 'Raymond', lastName: 'Devos')     ②
assert p.formattedName == 'Raymond Devos'                       ③
```

① the `Person` class only defines two properties, `firstName` and `lastName`

② we can create an instance of Person

③ and call a method named `formattedName`

It is quite common in dynamic languages for code such as the above example not to throw any error. How can this be? In Java, this would typically fail at compile time. However, in Groovy, it will not fail at compile time, and if coded correctly, will also not fail at runtime. In fact, to make this work at runtime, **one** possibility is to rely on runtime metaprogramming. So just adding this line after the declaration of the `Person` class is enough:

```
Person.metaClass.getFormattedName = { "$delegate.firstName $delegate.lastName" }
```

This means that in general, in Groovy, you can't make any assumption about the type of an object beyond its declaration type, and even if you know it, you can't determine at compile time what method will be called, or which property will be retrieved. It has a lot of interest, going from writing DSLs to testing, which is discussed in other sections of this manual.

However, if your program doesn't rely on dynamic features and that you come from the static world (in particular, from a Java mindset), not catching such "errors" at compile time can be surprising. As we have seen in the previous example, the compiler cannot be sure this is an error. To make it aware that it is, you have to explicitly instruct the compiler that you are switching to a type checked mode. This can be done by annotating a class or a method with `@groovy.transform.TypeChecked`.

When type checking is activated, the compiler performs much more work:

- type inference is activated, meaning that even if you use `def` on a local variable for example, the type checker will be able to infer the type of the variable from the assignments
- method calls are resolved at compile time, meaning that if a method is not declared on a class, the compiler will throw an error

- in general, all the compile time errors that you are used to find in a static language will appear: method not found, property not found, incompatible types for method calls, number precision errors, ...

In this section, we will describe the behavior of the type checker in various situations and explain the limits of using `@TypeChecked` on your code.

The `@TypeChecked` annotation

Activating type checking at compile time

The `groovy.transform.TypeChecked` annotation enables type checking. It can be placed on a class:

```
@groovy.transform.TypeChecked
class Calculator {
    int sum(int x, int y) { x+y }
}
```

Or on a method:

```
class Calculator {
    @groovy.transform.TypeChecked
    int sum(int x, int y) { x+y }
}
```

In the first case, all methods, properties, fields, inner classes, ... of the annotated class will be type checked, whereas in the second case, only the method and potential closures or anonymous inner classes that it contains will be type checked.

Skipping sections

The scope of type checking can be restricted. For example, if a class is type checked, you can instruct the type checker to skip a method by annotating it with `@TypeChecked(TypeCheckingMode.SKIP)`:

```
import groovy.transform.TypeChecked
import groovy.transform.TypeCheckingMode

@TypeChecked                                        ①
class GreetingService {
    String greeting() {                             ②
        doGreet()
    }

    @TypeChecked(TypeCheckingMode.SKIP)             ③
    private String doGreet() {
        def b = new SentenceBuilder()
        b.Hello.my.name.is.John                     ④
```

```
        b
    }
}
def s = new GreetingService()
assert s.greeting() == 'Hello my name is John'
```

① the `GreetingService` class is marked as type checked

② so the `greeting` method is automatically type checked

③ but `doGreet` is marked with `SKIP`

④ the type checker doesn't complain about missing properties here

In the previous example, `SentenceBuilder` relies on dynamic code. There's no real `Hello` method or property, so the type checker would normally complain and compilation would fail. Since the method that uses the builder is marked with `TypeCheckingMode.SKIP`, type checking is *skipped* for this method, so the code will compile, even if the rest of the class is type checked.

The most specific annotation wins: a method or nested class whose own `@TypeChecked` or `@CompileStatic` annotation has the default (non-`SKIP`) mode is checked even when an enclosing class is marked with `SKIP` mode (or with `@CompileDynamic`). Conversely, a class-level `SKIP` acts as the default for members that carry no type-checking annotation of their own. Note that the two annotations opt out of different things: `@CompileStatic(TypeCheckingMode.SKIP)` (or `@CompileDynamic`) on a method disables static *compilation* but does not exempt that method from an enclosing class's `@TypeChecked` checking; only `@TypeChecked(TypeCheckingMode.SKIP)` does that.

The following sections describe the semantics of type checking in Groovy.

Type checking assignments

An object `o` of type `A` can be assigned to a variable of type `T` if and only if:

- `T` equals `A`

```
Date now = new Date()
```

- *or* `T` is one of `String`, `boolean`, `Boolean` or `Class`

```
String s = new Date() // implicit call to toString
Boolean boxed = 'some string'       // Groovy truth
boolean prim = 'some string'        // Groovy truth
Class clazz = 'java.lang.String'    // class coercion
```

- *or* `o` is null and `T` is not a primitive type

```
String s = null         // passes
int i = null            // fails
```

- *or* `T` is an array and `A` is an array and the component type of `A` is assignable to the component type of `T`

```
int[] i = new int[4]        // passes
int[] i = new String[4]     // fails
```

- *or* `T` is an array and `A` is a collection or stream and the component type of `A` is assignable to the component type of `T`

```
int[] i = [1,2,3]               // passes
int[] i = [1,2, new Date()]     // fails
Set set = [1,2,3]
Number[] na = set               // passes
def stream = Arrays.stream(1,2,3)
int[] i = stream                // passes
```

- *or* `T` is a superclass of `A`

```
AbstractList list = new ArrayList()     // passes
LinkedList list = new ArrayList()       // fails
```

- *or* `T` is an interface implemented by `A`

```
List list = new ArrayList()             // passes
RandomAccess list = new LinkedList()    // fails
```

- *or* `T` or `A` are a primitive type and their boxed types are assignable

```
int i = 0
Integer bi = 1
int x = Integer.valueOf(123)
```

```
double d = Float.valueOf(5f)
```

- *or* `T` extends `groovy.lang.Closure` and `A` is a SAM-type (single abstract method type)

```
Runnable r = { println 'Hello' }
interface SAMType {
    int doSomething()
}
SAMType sam = { 123 }
assert sam.doSomething() == 123
abstract class AbstractSAM {
    int calc() { 2* value() }
    abstract int value()
}
AbstractSAM c = { 123 }
assert c.calc() == 246
```

- *or* `T` and `A` derive from `java.lang.Number` and conform to the following table

Table 8. Number types (java.lang.XXX)

T	A	Examples

Double	Any but BigDecimal or BigInteger	`Double d1 = 4d`
		`Double d2 = 4f`
		`Double d3 = 4l`
		`Double d4 = 4i`
		`Double d5 = (short) 4`
		`Double d6 = (byte) 4`

Float	Any type but BigDecimal, BigInteger or Double	`Float f1 = 4f`
		`Float f2 = 4l`
		`Float f3 = 4i`
		`Float f4 = (short) 4`
		`Float f5 = (byte) 4`

Long	Any type but BigDecimal, BigInteger, Double or Float	`Long l1 = 4l`
		`Long l2 = 4i`
		`Long l3 = (short) 4`
		`Long l4 = (byte) 4`

T	A	Examples

Integer	Any type but BigDecimal, BigInteger, Double, Float or Long	`Integer i1 = 4i`
		`Integer i2 = (short) 4`
		`Integer i3 = (byte) 4`

Short	Any type but BigDecimal, BigInteger, Double, Float, Long or Integer	`Short s1 = (short) 4` `Short s2 = (byte) 4`
Byte	Byte	`Byte b1 = (byte) 4`

List and map constructors

In addition to the assignment rules above, if an assignment is deemed invalid, in type checked mode, a *list* literal or a *map* literal `A` can be assigned to a variable of type `T` if:

- the assignment is a variable declaration and `A` is a list literal and `T` has a constructor whose parameters match the types of the elements in the list literal
- the assignment is a variable declaration and `A` is a map literal and `T` has a no-arg constructor and a property for each of the map keys

For example, instead of writing:

```
@groovy.transform.TupleConstructor
class Person {
    String firstName
    String lastName
}
Person classic = new Person('Ada','Lovelace')
```

You can use a "list constructor":

```
Person list = ['Ada','Lovelace']
```

or a "map constructor":

```
Person map = [firstName:'Ada', lastName:'Lovelace']
```

If you use a map constructor, additional checks are done on the keys of the map to check if a property of the same name is defined. For example, the following will fail at compile time:

```
@groovy.transform.TupleConstructor
class Person {
```

```
    String firstName
    String lastName
}
Person map = [firstName:'Ada', lastName:'Lovelace', age: 24]     ①
```

① The type checker will throw an error `No such property: age for class: Person` at compile time

Method resolution

In type checked mode, methods are resolved at compile time. Resolution works by name and arguments. The return type is irrelevant to method selection. Types of arguments are matched against the types of the parameters following those rules:

An argument `o` of type `A` can be used for a parameter of type `T` if and only if:

- `T` equals `A`

```
int sum(int x, int y) {
    x+y
}
assert sum(3,4) == 7
```

- *or* `T` is a `String` and `A` is a `GString`

```
String format(String str) {
    "Result: $str"
}
assert format("${3+4}") == "Result: 7"
```

- *or* `o` is null and `T` is not a primitive type

```
String format(int value) {
    "Result: $value"
}
assert format(7) == "Result: 7"
format(null)           // fails
```

- *or* `T` is an array and `A` is an array and the component type of `A` is assignable to the component type of `T`

```
String format(String[] values) {
```

```
    "Result: ${values.join(' ')}"
}
assert format(['a','b'] as String[]) == "Result: a b"
format([1,2] as int[])              // fails
```

- *or* `T` is a superclass of `A`

```
String format(AbstractList list) {
    list.join(',')
}
format(new ArrayList())             // passes
String format(LinkedList list) {
    list.join(',')
}
format(new ArrayList())             // fails
```

- *or* `T` is an interface implemented by `A`

```
String format(List list) {
    list.join(',')
}
format(new ArrayList())                 // passes
String format(RandomAccess list) {
    'foo'
}
format(new LinkedList())                // fails
```

- *or* `T` or `A` are a primitive type and their boxed types are assignable

```
int sum(int x, Integer y) {
    x+y
}
assert sum(3, new Integer(4)) == 7
assert sum(new Integer(3), 4) == 7
assert sum(new Integer(3), new Integer(4)) == 7
assert sum(new Integer(3), 4) == 7
```

- *or* `T` extends `groovy.lang.Closure` and `A` is a SAM-type (single abstract method type)

```
interface SAMType {
```

```
    int doSomething()
}
int twice(SAMType sam) { 2*sam.doSomething() }
assert twice { 123 } == 246
abstract class AbstractSAM {
    int calc() { 2* value() }
    abstract int value()
}
int eightTimes(AbstractSAM sam) { 4*sam.calc() }
assert eightTimes { 123 } == 984
```

- *or* `T` and `A` derive from `java.lang.Number` and conform to the same rules as assignment of numbers

If a method with the appropriate name and arguments is not found at compile time, an error is thrown. The difference with "normal" Groovy is illustrated in the following example:

```
class MyService {
    void doSomething() {
        printLine 'Do something'           ①
    }
}
```

① `printLine` is an error, but since we're in a dynamic mode, the error is not caught at compile time

The example above shows a class that Groovy will be able to compile. However, if you try to create an instance of `MyService` and call the `doSomething` method, then it will fail **at runtime**, because `printLine` doesn't exist. Of course, we already showed how Groovy could make this a perfectly valid call, for example by catching `MethodMissingException` or implementing a custom metaclass, but if you know you're not in such a case, `@TypeChecked` comes handy:

```
@groovy.transform.TypeChecked
class MyService {
    void doSomething() {
        printLine 'Do something'           ①
    }
}
```

① `printLine` is this time a compile-time error

Just adding `@TypeChecked` will trigger compile time method resolution. The type checker will try to find a method `printLine` accepting a `String` on the `MyService` class, but cannot find one. It will fail compilation with the following message:

```
Cannot find matching method MyService#printLine(java.lang.String)
```

IMPORTANT	It is important to understand the logic behind the type checker: it is a

compile-time check, so by definition, the type checker is not aware of any kind of **runtime** metaprogramming that you do. This means that code which is perfectly valid without `@TypeChecked` will **not** compile anymore if you activate type checking. This is in particular true if you think of duck typing:

```
class Duck {
    void quack() {              ①
        println 'Quack!'
    }
}
class QuackingBird {
    void quack() {              ②
        println 'Quack!'
    }
}
@groovy.transform.TypeChecked
void accept(quacker) {
    quacker.quack()             ③
}
accept(new Duck())              ④
```

① we define a `Duck` class which defines a `quack` method

② we define another `QuackingBird` class which also defines a `quack` method

③ `quacker` is loosely typed, so since the method is `@TypeChecked`, we will obtain a compile-time error

④ even if in non type-checked Groovy, this would have passed

There are possible workarounds, like introducing an interface, but basically, by activating type checking, you gain type safety but you loose some features of the language. Hopefully, Groovy introduces some features like flow typing to reduce the gap between type-checked and non type-checked Groovy.

Type inference

Principles

When code is annotated with `@TypeChecked`, the compiler performs type inference. It doesn't simply rely on static types, but also uses various techniques to infer the types of variables, return types, literals, … so that the code remains as clean as possible even if you activate the type checker.

The simplest example is inferring the type of a variable:

```
def message = 'Welcome to Groovy!'              ①
println message.toUpperCase()                   ②
println message.upper() // compile time error   ③
```

① a variable is declared using the `def` keyword

② calling `toUpperCase` is allowed by the type checker

③ calling `upper` will fail at compile time

The reason the call to `toUpperCase` works is because the type of `message` was *inferred* as being a `String`.

Variables vs fields in type inference

It is worth noting that although the compiler performs type inference on local variables, it does **not** perform any kind of type inference on fields, always falling back to the **declared type** of a field. To illustrate this, let's take a look at this example:

```
class SomeClass {
    def someUntypedField
①
    String someTypedField
②

    void someMethod() {
        someUntypedField = '123'
③
        someUntypedField = someUntypedField.toUpperCase()  // compile-time error
④
    }

    void someSafeMethod() {
        someTypedField = '123'
⑤
        someTypedField = someTypedField.toUpperCase()
⑥
    }

    void someMethodUsingLocalVariable() {
        def localVariable = '123'
⑦
        someUntypedField = localVariable.toUpperCase()
⑧
    }
}
```

① `someUntypedField` uses `def` as a declaration type

② `someTypedField` uses `String` as a declaration type

③ we can assign **anything** to `someUntypedField`

④ yet calling `toUpperCase` fails at compile time because the field is not typed properly

⑤ we can assign a `String` to a field of type `String`

⑥ and this time `toUpperCase` is allowed

⑦ if we assign a `String` to a local variable

⑧ then calling `toUpperCase` is allowed on the local variable

Why such a difference? The reason is *thread safety*. At compile time, we can't make **any** guarantee about the type of a field. Any thread can access any field at any time and between the moment a field is assigned a variable of some type in a method and the time is used the line after, another thread may have changed the contents of the field. This is not the case for local variables: we know if they "escape" or not, so we can make sure that the type of a variable is constant (or not) over time. Note that even if a field is final, the JVM makes no guarantee about it, so the type checker doesn't behave differently if a field is final or not.

TIP	This is one of the reasons why we recommend to use **typed** fields. While using `def` for local variables is perfectly fine thanks to type inference, this is not the case for fields, which also belong to the public API of a class, hence the type is important.

Collection literal type inference

Groovy provides a syntax for various type literals. There are three native collection literals in Groovy:

- lists, using the `[]` literal
- maps, using the `[:]` literal
- ranges, using `from..to` (inclusive), `from..<to` (right exclusive),`from<..to` (left exclusive) and `from<..<to` (full exclusive)

The inferred type of a literal depends on the elements of the literal, as illustrated in the following table:

Literal	Inferred type
`def list = []`	`java.util.List`
`def list = ['foo','bar']`	`java.util.List<String>`
`def list = ["${foo}","${bar}"]`	`java.util.List<GString>` be careful, a `GString` is **not** a `String`!
`def map = [:]`	`java.util.LinkedHashMap`
`def map1 = [someKey: 'someValue']` `def map2 = ['someKey': 'someValue']`	`java.util.LinkedHashMap<String,String>`

Literal	Inferred type
`def map = ["${someKey}": 'someValue']`	`java.util.LinkedHashMap<GString,String>` be careful, the key is a `GString`!
`def intRange = (0..10)`	`groovy.lang.IntRange`
`def charRange = ('a'..'z')`	`groovy.lang.Range<String>` : uses the type of the bounds to infer the component type of the range

As you can see, with the noticeable exception of the `IntRange`, the inferred type makes use of generics types to describe the contents of a collection. In case the collection contains elements of different types, the type checker still performs type inference of the components, but uses the notion of least upper bound.

Least upper bound

In Groovy, the *least upper bound* of two types `A` and `B` is defined as a type which:

- superclass corresponds to the common super class of `A` and `B`
- interfaces correspond to the interfaces implemented by both `A` and `B`
- if `A` or `B` is a primitive type and that `A` isn't equal to `B`, the least upper bound of `A` and `B` is the least upper bound of their wrapper types

If `A` and `B` only have one (1) interface in common and that their common superclass is `Object`, then the LUB of both is the common interface.

The least upper bound represents the minimal type to which both `A` and `B` can be assigned. So for example, if `A` and `B` are both `String`, then the LUB (least upper bound) of both is also `String`.

```
class Top {}
class Bottom1 extends Top {}
class Bottom2 extends Top {}

assert leastUpperBound(String, String) == String                    ①
assert leastUpperBound(ArrayList, LinkedList) == AbstractList       ②
assert leastUpperBound(ArrayList, List) == List                     ③
assert leastUpperBound(List, List) == List                          ④
assert leastUpperBound(Bottom1, Bottom2) == Top                     ⑤
assert leastUpperBound(List, Serializable) == Object                ⑥
```

① the LUB of `String` and `String` is `String`

② the LUB of `ArrayList` and `LinkedList` is their common super type, `AbstractList`

③ the LUB of `ArrayList` and `List` is their only common interface, `List`

④ the LUB of two identical interfaces is the interface itself

⑤ the LUB of `Bottom1` and `Bottom2` is their superclass `Top`

⑥ the LUB of two types which have nothing in common is `Object`

In those examples, the LUB is always representable as a normal, JVM supported, type. But Groovy internally represents the LUB as a type which can be more complex, and that you wouldn't be able to use to define a variable for example. To illustrate this, let's continue with this example:

```
interface Foo {}
class Top {}
class Bottom extends Top implements Serializable, Foo {}
class SerializableFooImpl implements Serializable, Foo {}
```

What is the least upper bound of `Bottom` and `SerializableFooImpl`? They don't have a common super class (apart from `Object`), but they do share 2 interfaces (`Serializable` and `Foo`), so their least upper bound is a type which represents the union of two interfaces (`Serializable` and `Foo`). This type cannot be defined in the source code, yet Groovy knows about it.

In the context of collection type inference (and generic type inference in general), this becomes handy, because the type of the components is inferred as the least upper bound. We can illustrate why this is important in the following example:

```
interface Greeter { void greet() }                  ①
interface Salute { void salute() }                  ②

class A implements Greeter, Salute {                ③
    void greet() { println "Hello, I'm A!" }
    void salute() { println "Bye from A!" }
}
class B implements Greeter, Salute {                ④
    void greet() { println "Hello, I'm B!" }
    void salute() { println "Bye from B!" }
    void exit() { println 'No way!' }               ⑤
}
def list = [new A(), new B()]                       ⑥
list.each {
    it.greet()                                      ⑦
    it.salute()                                     ⑧
    it.exit()                                       ⑨
}
```

① the `Greeter` interface defines a single method, `greet`

② the `Salute` interface defines a single method, `salute`

③ class `A` implements both `Greeter` and `Salute` but there's no explicit interface extending both

④ same for `B`

⑤ but `B` defines an additional `exit` method

⑥ the type of `list` is inferred as "list of the LUB of `A` and `B`"

⑦ so it is possible to call `greet` which is defined on both `A` and `B` through the `Greeter` interface

⑧ and it is possible to call `salute` which is defined on both `A` and `B` through the `Salute` interface

⑨ yet calling `exit` is a compile time error because it doesn't belong to the LUB of `A` and `B` (only defined in `B`)

The error message will look like:

```
[Static type checking] - Cannot find matching method Greeter or Salute#exit()
```

which indicates that the `exit` method is neither defines on `Greeter` nor `Salute`, which are the two interfaces defined in the least upper bound of `A` and `B`.

instanceof inference

In normal, non type checked, Groovy, you can write things like:

```
class Greeter {
    String greeting() { 'Hello' }
}

void doSomething(def o) {
    if (o instanceof Greeter) {     ①
        println o.greeting()        ②
    }
}

doSomething(new Greeter())
```

① guard the method call with an `instanceof` check

② make the call

The method call works because of dynamic dispatch (the method is selected at runtime). The equivalent code in Java would require to cast `o` to a `Greeter` before calling the `greeting` method, because methods are selected at compile time:

```
if (o instanceof Greeter) {
    System.out.println(((Greeter)o).greeting());
}
```

However, in Groovy, even if you add `@TypeChecked` (and thus activate type checking) on the `doSomething` method, the cast is **not** necessary. The compiler embeds *instanceof* inference that makes the cast optional.

Flow typing

Flow typing is an important concept of Groovy in type checked mode and an extension of type inference. The idea is that the compiler is capable of inferring the type of variables in the flow of the code, not just at initialization:

```
@groovy.transform.TypeChecked
void flowTyping() {
    def o = 'foo'                       ①
    o = o.toUpperCase()                 ②
    o = 9d                              ③
    o = Math.sqrt(o)                    ④
}
```

① first, `o` is declared using `def` and assigned a `String`

② the compiler inferred that `o` is a `String`, so calling `toUpperCase` is allowed

③ `o` is reassigned with a `double`

④ calling `Math.sqrt` passes compilation because the compiler knows that at this point, `o` is a `double`

So the type checker is *aware* of the fact that the concrete type of a variable is different over time. In particular, if you replace the last assignment with:

```
o = 9d
o = o.toUpperCase()
```

The type checker will now fail at compile time, because it knows that `o` is a `double` when `toUpperCase` is called, so it's a type error.

It is important to understand that it is not the fact of declaring a variable with `def` that triggers type inference. Flow typing works for **any** variable of any type. Declaring a variable with an explicit type only constrains what you can assign to the variable:

```
@groovy.transform.TypeChecked
void flowTypingWithExplicitType() {
    List list = ['a','b','c']          ①
    list = list*.toUpperCase()         ②
    list = 'foo'                       ③
}
```

① `list` is declared as an unchecked `List` and assigned a list literal of strings

② this line passes compilation because of flow typing: the type checker knows that `list` is at this point an `ArrayList<String>`

③ but you can't assign a `String` to a `List` so this is a type checking error

You can also note that even if the variable is declared **without** generics information, the type checker knows what is the component type. Therefore, such code would fail compilation:

```
@groovy.transform.TypeChecked
void flowTypingTypeConstraints1() {
    def list = ['a','b','c']            ①
    list.add(1)                         ②
}
@groovy.transform.TypeChecked
void flowTypingTypeConstraints2() {
    List<?> list = []                   ③
    list.addAll(['a','b','c'])          ④
    list.add(1)                         ⑤
}
```

① `list` is inferred as `ArrayList<String>`

② so adding an `int` to a `ArrayList<String>` is a compile-time error

③ `list` is declared as `List<?>`

④ the inferred type of `list` here is `List<capture-of ?>`, so calling `addAll` with a list of anything is a compile-time error

⑤ and calling `add` with an `int` is also a compile-time error for the same reason; only `add(null)` is allowed

Fixing this requires adding an explicit, non-wildcard type argument:

```
@groovy.transform.TypeChecked
void flowTypingTypeConstraints3() {
    List<Serializable> list = []        ①
    list.addAll(['a','b','c'])          ②
    list.add(1)                         ③
}
```

① `list` is declared as `List<Serializable>` and initialized with an empty list

② elements added to the list conform to the declaration type of the list

③ and adding an integer is allowed

Flow typing has been introduced to reduce the difference in semantics between classic and static Groovy. In particular, consider the behavior of this code in Java:

```
public Integer compute(String str) {
    return str.length();
}
public String compute(Object o) {
    return "Nope";
}
// ...
Object string = "Some string";          ①
Object result = compute(string);        ②
```

```
System.out.println(result);             ③
```

① `o` is declared as an `Object` and assigned a `String`

② we call the `compute` method with `o`

③ and print the result

In Java, this code will output `Nope`, because method selection is done at compile time and based on the **declared** types. So even if `o` is a `String` at runtime, it is still the `Object` version which is called, because `o` has been declared as an `Object`. To be short, in Java, declared types are most important, be it variable types, parameter types or return types.

In Groovy, we could write:

```
int compute(String string) { string.length() }
String compute(Object o) { "Nope" }
Object o = 'string'
def result = compute(o)
println result
```

But this time, it will return `6`, because the method which is chosen **at runtime**, based on the *actual* argument types. So at runtime, `o` is a `String` so the `String` variant is used. Note that this behavior has nothing to do with type checking, it's the way Groovy works in general: dynamic dispatch.

In type checked Groovy, we want to make sure the type checker selects the same method **at compile time**, that the runtime would choose. It is not possible in general, due to the semantics of the language, but we can make things better with flow typing. With flow typing, `o` is *inferred* as a `String` when the `compute` method is called, so the version which takes a `String` and returns an `int` is chosen. This means that we can infer the return type of the method to be an `int`, and not a `String`. This is important for subsequent calls and type safety.

So in type checked Groovy, flow typing is a very important concept, which also implies that if `@TypeChecked` is applied, methods are selected based on the *inferred types* of the arguments, not on the declared types. This doesn't ensure 100% type safety, because the type checker *may* select a wrong method, but it ensures the closest semantics to dynamic Groovy.

Advanced type inference

A combination of flow typing and least upper bound inference is used to perform advanced type inference and ensure type safety in multiple situations. In particular, program control structures are likely to alter the inferred type of a variable:

```
class Top {
   void methodFromTop() {}
}
class Bottom extends Top {
   void methodFromBottom() {}
}
def o
```

```
if (someCondition) {
    o = new Top()                                ①
} else {
    o = new Bottom()                             ②
}
o.methodFromTop()                                ③
o.methodFromBottom()  // compilation error       ④
```

① if `someCondition` is true, `o` is assigned a `Top`

② if `someCondition` is false, `o` is assigned a `Bottom`

③ calling `methodFromTop` is safe

④ but calling `methodFromBottom` is not, so it's a compile time error

When the type checker visits an `if/else` control structure, it checks all variables which are assigned in `if/else` branches and computes the least upper bound of all assignments. This type is the type of the inferred variable after the `if/else` block, so in this example, `o` is assigned a `Top` in the `if` branch and a `Bottom` in the `else` branch. The LUB of those is a `Top`, so after the conditional branches, the compiler infers `o` as being a `Top`. Calling `methodFromTop` will therefore be allowed, but not `methodFromBottom`.

The same reasoning exists with closures and in particular closure shared variables. A closure shared variable is a variable which is defined outside of a closure, but used inside a closure, as in this example:

```
def text = 'Hello, world!'                       ①
def closure = {
    println text                                 ②
}
```

① a variable named `text` is declared

② `text` is used from inside a closure. It is a *closure shared variable*.

Groovy allows developers to use those variables without requiring them to be final. This means that a closure shared variable can be reassigned inside a closure:

```
String result
doSomething { String it ->
    result = "Result: $it"
}
result = result?.toUpperCase()
```

The problem is that a closure is an independent block of code that can be executed (or not) at **any** time. In particular, `doSomething` may be asynchronous, for example. This means that the body of a closure doesn't belong to the main control flow. For that reason, the type checker also computes, for each closure shared variable, the LUB of all assignments of the variable, and will use that `LUB` as the inferred type outside of the scope of the closure, like in this example:

```
class Top {
   void methodFromTop() {}
}
class Bottom extends Top {
   void methodFromBottom() {}
}
def o = new Top()                                   ①
Thread.start {
    o = new Bottom()                                ②
}
o.methodFromTop()                                   ③
o.methodFromBottom()  // compilation error          ④
```

① a closure-shared variable is first assigned a `Top`

② inside the closure, it is assigned a `Bottom`

③ `methodFromTop` is allowed

④ `methodFromBottom` is a compilation error

Here, it is clear that when `methodFromBottom` is called, there's no guarantee, at compile-time or runtime that the type of `o` will *effectively* be a `Bottom`. There are chances that it will be, but we can't make sure, because it's asynchronous. So the type checker will only allow calls on the least upper bound, which is here a `Top`.

Closures and type inference

The type checker performs special inference on closures, resulting on additional checks on one side and improved fluency on the other side.

Return type inference

The first thing that the type checker is capable of doing is inferring the *return type* of a closure. This is simply illustrated in the following example:

```
@groovy.transform.TypeChecked
int testClosureReturnTypeInference(String arg) {
    def cl = { "Arg: $arg" }                        ①
    def val = cl()                                  ②

    val.length()                                    ③
}
```

① a closure is defined, and it returns a string (more precisely a `GString`)

② we call the closure and assign the result to a variable

③ the type checker inferred that the closure would return a string, so calling `length()` is allowed

As you can see, unlike a method which declares its return type explicitly, there's no need to declare the return type of a closure: its type is inferred from the body of the closure.

Closures vs methods

It's worth noting that return type inference is only applicable to closures. While the type checker could do the same on a method, it is in practice not desirable: *in general*, methods can be overridden and it is not statically possible to make sure that the method which is called is not an overridden version. So flow typing would actually think that a method returns something, while in reality, it could return something else, like illustrated in the following example:

```
@TypeChecked
class A {
    def compute() { 'some string' }             ①
    def computeFully() {
        compute().toUpperCase()                 ②
    }
}
@TypeChecked
class B extends A {
    def compute() { 123 }                       ③
}
```

① class `A` defines a method `compute` which effectively returns a `String`

② this will fail compilation because the return type of `compute` is `def`(aka `Object`)

③ class `B` extends `A` and redefines `compute`, this type returning an `int`

As you can see, if the type checker relied on the inferred return type of a method, with flow typing, the type checker could determine that it is ok to call `toUpperCase`. It is in fact an **error**, because a subclass can override `compute` and return a different object. Here, `B#compute` returns an `int`, so someone calling `computeFully` on an instance of `B` would see a runtime error. The compiler prevents this from happening by using the declared return type of methods instead of the inferred return type.

For consistency, this behavior is the same for **every** method, even if they are static or final.

Parameter type inference

In addition to the return type, it is possible for a closure to infer its parameter types from the context. There are two ways for the compiler to infer the parameter types:

- through *implicit SAM type coercion*
- through API metadata

To illustrate this, lets start with an example that will fail compilation due to the inability for the type checker to infer the parameter types:

```
class Person {
```

```
    String name
    int age
}

void inviteIf(Person p, Closure<Boolean> predicate) {           ①
    if (predicate.call(p)) {
        // send invite
        // ...
    }
}

@groovy.transform.TypeChecked
void failCompilation() {
    Person p = new Person(name: 'Gerard', age: 55)
    inviteIf(p) {                                               ②
        it.age >= 18 // No such property: age                   ③
    }
}
```

① the `inviteIf` method accepts a `Person` and a `Closure`

② we call it with a `Person` and a `Closure`

③ yet `it` is not statically known as being a `Person` and compilation fails

In this example, the closure body contains `it.age`. With dynamic, not type checked code, this would work, because the type of `it` would be a `Person` at runtime. Unfortunately, at compile-time, there's no way to know what is the type of `it`, just by reading the signature of `inviteIf`.

Explicit closure parameters

To be short, the type checker doesn't have enough contextual information on the `inviteIf` method to determine statically the type of `it`. This means that the method call needs to be rewritten like this:

```
inviteIf(p) { Person it ->                                      ①
    it.age >= 18
}
```

① the type of `it` needs to be declared explicitly

By explicitly declaring the type of the `it` variable, you can work around the problem and make this code statically checked.

Parameters inferred from single-abstract method types

For an API or framework designer, there are two ways to make this more elegant for users, so that they don't have to declare an explicit type for the closure parameters. The first one, and easiest, is to replace the closure with a SAM type:

```
interface Predicate<On> { boolean apply(On e) }                    ①

void inviteIf(Person p, Predicate<Person> predicate) {             ②
    if (predicate.apply(p)) {
        // send invite
        // ...
    }
}

@groovy.transform.TypeChecked
void passesCompilation() {
    Person p = new Person(name: 'Gerard', age: 55)

    inviteIf(p) {                                                  ③
        it.age >= 18                                               ④
    }
}
```

① declare a `SAM` interface with an `apply` method

② `inviteIf` now uses a `Predicate<Person>` instead of a `Closure<Boolean>`

③ there's no need to declare the type of the `it` variable anymore

④ `it.age` compiles properly, the type of `it` is inferred from the `Predicate#apply` method signature

TIP

By using this technique, we leverage the *automatic coercion of closures to SAM types* feature of Groovy. Whether you should use a *SAM type* or a *Closure* really depends on what you need to do. In a lot of cases, using a SAM interface is enough, especially if you consider functional interfaces as they are found in Java 8. However, closures provide features that are not accessible to functional interfaces. In particular, closures can have a delegate, and owner and can be manipulated as objects (for example, cloned, serialized, curried, ...) before being called. They can also support multiple signatures (polymorphism). So if you need that kind of manipulation, it is preferable to switch to the most advanced type inference annotations which are described below.

The original issue that needs to be solved when it comes to closure parameter type inference, that is to say, statically determining the types of the arguments of a closure *without* having to have them explicitly declared, is that the Groovy type system inherits the Java type system, which is insufficient to describe the types of the arguments.

The `@ClosureParams` annotation

Groovy provides an annotation, `@ClosureParams` which is aimed at completing type information. This annotation is primarily aimed at framework and API developers who want to extend the capabilities of the type checker by providing type inference metadata. This is important if your library makes use of closures and that you want the maximum level of tooling support too.

Let's illustrate this by fixing the original example, introducing the `@ClosureParams` annotation:

```
import groovy.transform.stc.ClosureParams
import groovy.transform.stc.FirstParam
void inviteIf(Person p, @ClosureParams(FirstParam) Closure<Boolean> predicate) {
①
    if (predicate.call(p)) {
        // send invite
        // ...
    }
}
inviteIf(p) {                                                                    ②
    it.age >= 18
}
```

① the closure parameter is annotated with `@ClosureParams`

② it's not necessary to use an explicit type for `it`, which is inferred

The `@ClosureParams` annotation minimally accepts one argument, which is named a *type hint*. A type hint is a class which is responsible for completing type information at compile time for the closure. In this example, the type hint being used is `groovy.transform.stc.FirstParam` which indicated to the type checker that the closure will accept one parameter whose type is the type of the first parameter of the method. In this case, the first parameter of the method is `Person`, so it indicates to the type checker that the first parameter of the closure is in fact a `Person`.

A second optional argument is named *options*. Its semantics depend on the *type hint* class. Groovy comes with various bundled type hints, illustrated in the table below:

Table 9. Predefined type hints

Type hint	Polymorphic?	Description and examples

`FirstParam`
`SecondParam`
`ThirdParam`
	No	The first (resp. second, third) parameter type of the method
		
		`import groovy.transform.stc.FirstParam`
		`void doSomething(String str, @ClosureParams(FirstParam)`
		`Closure c) {`
		`  c(str)`
		`}`
		`doSomething('foo') { println it.toUpperCase() }`
		
		`import groovy.transform.stc.SecondParam`
		`void withHash(String str, int seed, @ClosureParams`
		`(SecondParam) Closure c) {`
		`  c(31*str.hashCode()+seed)`
		`}`
		`withHash('foo', (int)System.currentTimeMillis()) {`
		`  int mod = it%2`
		`}`
		
		`import groovy.transform.stc.ThirdParam`
		`String format(String prefix, String postfix, String o,`
		`@ClosureParams(ThirdParam) Closure c) {`
		`  "$prefix${c(o)}$postfix"`
		`}`
		`assert format('foo', 'bar', 'baz') {`
		`  it.toUpperCase()`
		`} == 'fooBAZbar'`

`FirstParam.FirstGenericType`
`SecondParam.FirstGenericType`
`ThirdParam.FirstGenericType`
	No	The first generic type of the first (resp. second, third) parameter of the method
		
		`import groovy.transform.stc.FirstParam`
		`public <T> void doSomething(List<T> strings,`
		`@ClosureParams(FirstParam.FirstGenericType) Closure c)`
		`{`
		`  strings.each {`
		`  c(it)`
		`  }`
		`}`
		`doSomething(['foo','bar']) { println it.toUpperCase() }`
		`doSomething([1,2,3]) { println(2*it) }`
		
		Variants for `SecondGenericType` and `ThirdGenericType` exist for all `FirstParam`, `SecondParam` and `ThirdParam` type hints.

Type hint	Polymorphic?	Description and examples

`SimpleType`	No	A type hint for which the type of closure parameters comes from the options string.
		
		`import groovy.transform.stc.SimpleType`
		`public void doSomething(@ClosureParams(value=`
		`SimpleType,options=['java.lang.String','int']) Closure`
		`c) {`
		`  c('foo',3)`
		`}`
		`doSomething { str, len ->`
		`  assert str.length() == len`
		`}`
		
		This type hint supports a **single** signature and each of the parameter is specified as a value of the *options* array using a fully-qualified type name or a primitive type.

`MapEntryOrKeyValue`	Yes	A dedicated type hint for closures that either work on a `Map.Entry` single parameter, or two parameters corresponding to the key and the value.
		
		`import groovy.transform.stc.MapEntryOrKeyValue`
		`public <K,V> void doSomething(Map<K,V> map,`
		`@ClosureParams(MapEntryOrKeyValue) Closure c) {`
		`  // ...`
		`}`
		`doSomething([a: 'A']) { k,v ->`
		`  assert k.toUpperCase() == v.toUpperCase()`
		`}`
		`doSomething([abc: 3]) { e ->`
		`  assert e.key.length() == e.value`
		`}`
		
		This type hint **requires** that the first argument is a `Map` type, and infers the closure parameter types from the map actual key/value types.

Type hint	Polymorphic?	Description and examples

`FromAbstractTypeMethods`	Yes	Infers closure parameter types from the abstract method of some type. A signature is inferred for **each** abstract method.
		
		`import groovy.transform.stc.FromAbstractTypeMethods`
		`abstract class Foo {`
		`  abstract void firstSignature(int x, int y)`
		`  abstract void secondSignature(String str)`
		`}`
		`void doSomething(@ClosureParams(value`
		`=FromAbstractTypeMethods, options=["Foo"]) Closure cl)`
		`{`
		`  // ...`
		`}`
		`doSomething { a, b -> a+b }`
		`doSomething { s -> s.toUpperCase() }`
		
		If there are multiple signatures like in the example above, the type checker will **only** be able to infer the types of the arguments if the arity of each method is different. In the example above, `firstSignature` takes 2 arguments and `secondSignature` takes 1 argument, so the type checker can infer the argument types based on the number of arguments. But see the optional resolver class attribute discussed next.

Type hint	Polymorphic?	Description and examples

`FromString`	Yes	Infers the closure parameter types from the `options` argument. The `options` argument consists of an array of comma-separated non-primitive types. Each element of the array corresponds to a single signature, and each comma in an element separate parameters of the signature. In short, this is the most generic type hint, and each string of the `options` map is **parsed** as if it was a signature literal. While being very powerful, this type hint must be avoided if you can because it increases the compilation times due to the necessity of parsing the type signatures.
		
		A single signature for a closure accepting a `String`:
		
		`import groovy.transform.stc.FromString`
		`void doSomething(@ClosureParams(value=FromString,`
		`options=["String","String,Integer"]) Closure cl) {`
		`    // ...`
		`}`
		`doSomething { s -> s.toUpperCase() }`
		`doSomething { s,i -> s.toUpperCase()*i }`
		
		A polymorphic closure, accepting either a `String` or a `String,Integer`:
		
		`import groovy.transform.stc.FromString`
		`void doSomething(@ClosureParams(value=FromString,`
		`options=["String","String,Integer"]) Closure cl) {`
		`    // ...`
		`}`
		`doSomething { s -> s.toUpperCase() }`
		`doSomething { s,i -> s.toUpperCase()*i }`
		
		A polymorphic closure, accepting either a `T` or a pair `T,T`:
		
		`import groovy.transform.stc.FromString`
		`public <T> void doSomething(T e, @ClosureParams(value`
		`=FromString, options=["T","T,T"]) Closure cl) {`
		`    // ...`
		`}`
		`doSomething('foo') { s -> s.toUpperCase() }`
		`doSomething('foo') { s1,s2 -> assert s1.toUpperCase()`
		`== s2.toUpperCase() }`

TIP Even though you use `FirstParam`, `SecondParam` or `ThirdParam` as a type hint, it doesn't strictly mean that the argument which will be passed to the closure **will** be the first

> (resp. second, third) argument of the method call. It only means that the **type** of the parameter of the closure will be the **same** as the type of the first (resp. second, third) argument of the method call.

In short, the lack of the `@ClosureParams` annotation on a method accepting a `Closure` will **not** fail compilation. If present (and it can be present in Java sources as well as Groovy sources), then the type checker has **more** information and can perform additional type inference. This makes this feature particularly interesting for framework developers.

A third optional argument is named *conflictResolutionStrategy*. It can reference a class (extending from `ClosureSignatureConflictResolver`) that can perform additional resolution of parameter types if more than one are found after initial inference calculations are complete. Groovy comes with a default type resolver which does nothing, and another which selects the first signature if multiple are found. The resolver is only invoked if more than one signature is found and is by design a post processor. Any statements which need injected typing information must pass one of the parameter signatures determined through type hints. The resolver then picks among the returned candidate signatures.

`@DelegatesTo`

The `@DelegatesTo` annotation is used by the type checker to infer the type of the delegate. It allows the API designer to instruct the compiler what is the type of the delegate and the delegation strategy. The `@DelegatesTo` annotation is discussed in a specific section.

The `@ClassTag` annotation

On the JVM, generic type arguments are erased at runtime, so a method that needs the runtime class of a type argument must take an explicit `Class` token, even though that token repeats a type the compiler already knows. The `groovy.transform.stc.ClassTag` parameter annotation (incubating since Groovy 6) marks such a token as *compiler-supplied*: under `@TypeChecked` or `@CompileStatic`, a call that omits the token can select the tagged overload, and the type checker synthesises the `X.class` argument from the receiver's type argument. The `asChecked` methods from the GDK declare their token this way:

```
List<String> names = []
List<String> checked = names.asChecked()        ①
checked << 'Alice'                              ②
```

① there is no token-less `asChecked()`: the type checker selects `asChecked(List<T>, Class<T>)` and injects `String.class`, reified from the receiver's type argument

② the checked view rejects ill-typed elements at runtime, exactly as if `asChecked(String)` had been written

Any API can declare tagged parameters; each `@ClassTag` parameter is filled at its declared position and the supplied arguments map onto the remaining parameters in order:

```
import groovy.transform.stc.ClassTag
```

```
class Registry<T> {
    Map<String, T> byName = [:]
    T create(String name, @ClassTag Class<T> type) {    ①
        T instance = type.getDeclaredConstructor().newInstance()
        byName[name] = instance
        instance
    }
}

Registry<StringBuilder> registry = new Registry<>()
registry.create('first')                                ②
```

① the token parameter repeats the receiver's type argument `T`, so it is marked compiler-supplied

② equivalent to `registry.create('first', StringBuilder)`: the token is reified from `Registry<StringBuilder>`

Selection is normally *additive*: the tagged overload is only chosen when the call as written matches no method, so the meaning of existing code never changes. An API author may go further and declare a tagged overload with `@ClassTag(preempt=true)`, allowing it to *preempt* a token-less overload declared by the same class: existing calls are transparently upgraded to the checked variant. Since Groovy 6, the checked `withDefault` variants declare this (their lenient sibling could silently corrupt a typed map's key set, see GROOVY-11807):

```
Map<String, Integer> counts = [:]
def wordCounts = counts.withDefault { 0 }       ①
assert wordCounts['alpha'] == 0                 ②
```

① under static compilation the call binds `withDefault(Map<K,V>, Class<K>, Class<V>, Closure)` with `String.class` and `Integer.class` injected; the returned map rejects ill-typed keys and values

② reads behave as before for well-typed keys

Preemption is deliberately constrained by three independent controls:

- *intent*: only an overload explicitly marked `@ClassTag(preempt=true)` may displace a token-less one; an unmarked tag is strictly additive;
- *containment*: a tagged overload can only preempt a token-less overload declared by the same class (or, for instance methods, the same class hierarchy) — adding a jar to the compile classpath can never re-route calls to methods it does not own;
- *consent*: the consuming build may veto all preemption:

```
def config = new CompilerConfiguration()
config.classTagPreemptionDisabled = true            ①
```

① also available without a configuration script via the `groovy.classtag.preemption.disable` system property; code compiled with the veto behaves exactly as it did before any library declared preemption

Some boundaries worth knowing:

- injection is a static-checking feature: dynamic Groovy ignores the annotation, so dynamic code passes the `Class` token explicitly;
- the receiver's type argument must be statically known — `def`, raw and wildcard receivers synthesise no token — and only the erased class is reified (`List<List<String>>` yields `List.class`);
- when every token would erase to `Object` (for example a `List<Object>` receiver), no token is synthesised: a checked `Object` view could never reject anything;
- passing the `Class` explicitly always works, in every mode.

Static compilation

Dynamic vs static

In the type checking section, we have seen that Groovy provides optional type checking thanks to the `@TypeChecked` annotation. The type checker runs at compile time and performs a static analysis of dynamic code. The program will behave exactly the same whether type checking has been enabled or not. This means that the `@TypeChecked` annotation is neutral in regard to the semantics of a program. Even though it may be necessary to add type information in the sources so that the program is considered type safe, in the end, the semantics of the program are the same.

While this may sound fine, there is actually one issue with this: type checking of dynamic code, done at compile time, is by definition only correct if no runtime specific behavior occurs. For example, the following program passes type checking:

```
class Computer {
    int compute(String str) {
        str.length()
    }
    String compute(int x) {
        String.valueOf(x)
    }
}

@groovy.transform.TypeChecked
void test() {
    def computer = new Computer()
    computer.with {
        assert compute(compute('foobar')) =='6'
    }
}
```

There are two `compute` methods. One accepts a `String` and returns an `int`, the other accepts an `int` and returns a `String`. If you compile this, it is considered type safe: the inner `compute('foobar')` call will return an `int`, and calling `compute` on this `int` will in turn return a `String`.

Now, before calling `test()`, consider adding the following line:

```
Computer.metaClass.compute = { String str -> new Date() }
```

Using runtime metaprogramming, we're actually modifying the behavior of the `compute(String)` method, so that instead of returning the length of the provided argument, it will return a `Date`. If you execute the program, it will fail at runtime. Since this line can be added from anywhere, in any thread, there's absolutely no way for the type checker to statically make sure that no such thing happens. In short, the type checker is vulnerable to monkey patching. This is just one example, but this illustrates the concept that doing static analysis of a dynamic program is inherently wrong.

The Groovy language provides an alternative annotation to `@TypeChecked` which will actually make sure that the methods which are inferred as being called **will** effectively be called at runtime. This annotation turns the Groovy compiler into a **static compiler**, where all method calls are resolved at compile time **and** the generated bytecode makes sure that this happens: the annotation is `@groovy.transform.CompileStatic`.

The `@CompileStatic` annotation

The `@CompileStatic` annotation can be added anywhere the `@TypeChecked` annotation can be used, that is to say on a class or a method. It is not necessary to add both `@TypeChecked` and `@CompileStatic`, as `@CompileStatic` performs everything `@TypeChecked` does, but in addition triggers static compilation.

Let's take the example which failed, but this time let's replace the `@TypeChecked` annotation with `@CompileStatic`:

```
class Computer {
    int compute(String str) {
        str.length()
    }
    String compute(int x) {
        String.valueOf(x)
    }
}

@groovy.transform.CompileStatic
void test() {
    def computer = new Computer()
    computer.with {
        assert compute(compute('foobar')) =='6'
    }
}
Computer.metaClass.compute = { String str -> new Date() }
test()
```

This is the **only** difference. If we execute this program, this time, there is no runtime error. The `test` method became immune to monkey patching, because the `compute` methods which are called in its body are linked at compile time, so even if the metaclass of `Computer` changes, the program still behaves **as expected by the type checker**.

Key benefits

There are several benefits of using `@CompileStatic` on your code:

- type safety
- immunity to monkey patching
- performance improvements

The performance improvements depend on the kind of program you are executing. If it is I/O bound, the difference between statically compiled code and dynamic code is barely noticeable. On highly CPU intensive code, since the bytecode which is generated is very close, if not equal, to the one that Java would produce for an equivalent program, the performance is greatly improved.

TIP

Using the *invokedynamic* version of Groovy, which is accessible to people using JDK 7 and above, the performance of the dynamic code should be very close to the performance of statically compiled code. Sometimes, it can even be faster! There is only one way to determine which version you should choose: measuring. The reason is that depending on your program **and** the JVM that you use, the performance can be significantly different. In particular, the *invokedynamic* version of Groovy is very sensitive to the JVM version in use.

Type checking extensions

TIP

This chapter is the SDK reference for **writing** a type checking extension. For the extensions Groovy already ships with (regex checking, format-string checking, null safety, frame conditions, purity, parallel-reduction associativity, and monadic comprehensions), see the Built-in auxiliary type checkers chapter.

Writing a type checking extension

Towards a smarter type checker

Despite being a dynamic language, Groovy can be used with a static type checker at compile time, enabled using the `@TypeChecked` annotation. In this mode, the compiler becomes more verbose and throws errors for, example, typos, non-existent methods, etc. This comes with a few limitations though, most of them coming from the fact that Groovy remains inherently a dynamic language. For example, you wouldn't be able to use type checking on code that uses the markup builder:

```
def builder = new MarkupBuilder(out)
builder.html {
    head {
        // ...
    }
    body {
        p 'Hello, world!'
    }
}
```

In the previous example, none of the `html`, `head`, `body` or `p` methods exist. However if you execute the code, it works because Groovy uses dynamic dispatch and converts those method calls at runtime. In this builder, there's no limitation about the number of tags that you can use, nor the attributes, which means there is no chance for a type checker to know about all the possible methods (tags) at compile time, unless you create a builder dedicated to HTML for example.

Groovy is a platform of choice when it comes to implement internal DSLs. The flexible syntax, combined with runtime and compile-time metaprogramming capabilities make Groovy an interesting choice because it allows the programmer to focus on the DSL rather than on tooling or implementation. Since Groovy DSLs are Groovy code, it's easy to have IDE support without having to write a dedicated plugin for example.

In a lot of cases, DSL engines are written in Groovy (or Java) then user code is executed as scripts, meaning that you have some kind of wrapper on top of user logic. The wrapper may consist, for example, in a `GroovyShell` or `GroovyScriptEngine` that performs some tasks transparently before running the script (adding imports, applying AST transforms, extending a base script,...). Often, user written scripts come to production without testing because the DSL logic comes to a point where **any** user may write code using the DSL syntax. In the end, a user may just ignore that what they write is actually **code**. This adds some challenges for the DSL implementer, such as securing execution of user code or, in this case, early reporting of errors.

For example, imagine a DSL which goal is to drive a rover on Mars remotely. Sending a message to the rover takes around 15 minutes. If the rover executes the script and fails with an error (say a typo), you have two problems:

- first, feedback comes only after 30 minutes (the time needed for the rover to get the script and the time needed to receive the error)
- second, some portion of the script has been executed and you may have to change the fixed script significantly (implying that you need to know the current state of the rover...)

Type checking extensions is a mechanism that will allow the developer of a DSL engine to make those scripts safer by applying the same kind of checks that static type checking allows on regular groovy classes.

The principle, here, is to fail early, that is to say fail compilation of scripts as soon as possible, and if possible provide feedback to the user (including nice error messages).

In short, the idea behind type checking extensions is to make the compiler aware of all the runtime metaprogramming tricks that the DSL uses, so that scripts can benefit the same level of compile-time checks as a verbose statically compiled code would have. We will see that you can go even further by performing checks that a normal type checker wouldn't do, delivering powerful compile-time checks for your users.

The extensions attribute

The `@TypeChecked` annotation supports an attribute named `extensions`. This parameter takes an array of strings corresponding to a list of *type checking extensions scripts*. Those scripts are found at **compile time** on classpath. For example, you would write:

```
@TypeChecked(extensions='/path/to/myextension.groovy')
void foo() { ...}
```

In that case, the *foo* methods would be type checked with the rules of the normal type checker completed by those found in the *myextension.groovy* script. Note that while internally the type checker supports multiple mechanisms to implement type checking extensions (including plain old java code), the recommended way is to use those type checking extension scripts.

Parameterized extensions

Extensions can accept named parameters using Groovy's named-argument style within the extension string. The example below uses `NullChecker`, one of the bundled checkers; the same syntax applies to any extension that reads the `options` map.

```
@TypeChecked(extensions = 'groovy.typecheckers.NullChecker(strict: true)')
void foo() { ... }
```

Parameters are passed as `key: value` pairs inside parentheses appended to the extension name. Supported value types include booleans (`true`/`false`), integers, strings (single or double quoted), and `null`. Multiple parameters are separated by commas. Note that string values cannot contain commas or colons since these are used as delimiters:

```
@TypeChecked(extensions = 'com.example.MyChecker(strict: true, threshold: 5, mode:
"lenient")')
void foo() { ... }
```

Within an extension, the parameters are available via the `options` property, which returns an unmodifiable `Map<String, Object>`:

```
// in a TypeCheckingDSL extension:
Object run() {
    boolean strict = options?.strict ?: false
    // ...
}
```

For precompiled Java extensions (subclasses of `TypeCheckingExtension` or `AbstractTypeCheckingExtension`), options are accessible via the `getOptions()` method.

The parameterized syntax also works when configuring extensions programmatically:

```
new ASTTransformationCustomizer(TypeChecked, extensions: ['my.Extension(debug:
true)'])
```

A DSL for type checking

The idea behind type checking extensions is to use a DSL to extend the type checker capabilities. This DSL allows you to hook into the compilation process, more specifically the type checking phase, using an "event-driven" API. For example, when the type checker enters a method body, it throws a *beforeVisitMethod* event that the extension can react to:

```
beforeVisitMethod { methodNode ->
 println "Entering ${methodNode.name}"
}
```

Imagine that you have this rover DSL at hand. A user would write:

```
robot.move 100
```

If you have a class defined as such:

```
class Robot {
    Robot move(int qt) { this }
}
```

The script can be type checked before being executed using the following script:

```
def config = new CompilerConfiguration()
config.addCompilationCustomizers(
    new ASTTransformationCustomizer(TypeChecked)            ①
)
def shell = new GroovyShell(config)                         ②
def robot = new Robot()
shell.setVariable('robot', robot)
shell.evaluate(script)                                      ③
```

① a compiler configuration adds the `@TypeChecked` annotation to all classes

② use the configuration in a `GroovyShell`

③ so that scripts compiled using the shell are compiled with `@TypeChecked` without the user having to add it explicitly

Using the compiler configuration above, we can apply *@TypeChecked* transparently to the script. In that case, it will fail at compile time:

```
[Static type checking] - The variable [robot] is undeclared.
```

Now, we will slightly update the configuration to include the ``extensions'' parameter:

```
config.addCompilationCustomizers(
    new ASTTransformationCustomizer(
        TypeChecked,
        extensions:['robotextension.groovy'])
)
```

Then add the following to your classpath:

robotextension.groovy

```
unresolvedVariable { var ->
    if ('robot'==var.name) {
        storeType(var, classNodeFor(Robot))
        handled = true
    }
}
```

Here, we're telling the compiler that if an *unresolved variable* is found and that the name of the variable is *robot*, then we can make sure that the type of this variable is `Robot`.

Type checking extensions API

AST

The type checking API is a low level API, dealing with the Abstract Syntax Tree. You will have to know your AST well to develop extensions, even if the DSL makes it much easier than just dealing with AST code from plain Java or Groovy.

Events

The type checker sends the following events, to which an extension script can react:

Event name	setup
Called When	Called after the type checker finished initialization
Arguments	none
Usage	`setup { // this is called before anything else }` Can be used to perform setup of your extension

Event name	finish
Called When	Called after the type checker completed type checking
Arguments	none

Usage	`finish { // this is after completion // of all type checking }` Can be used to perform additional checks after the type checker has finished its job.

Event name	unresolvedVariable
Called When	Called when the type checker finds an unresolved variable
Arguments	VariableExpression vexp
Usage	`unresolvedVariable { VariableExpression vexp -> if (vexp.name == 'people') { storeType(vexp, LIST_TYPE) handled = true } }` Allows the developer to help the type checker with user-injected variables.

Event name	unresolvedProperty
Called When	Called when the type checker cannot find a property on the receiver
Arguments	PropertyExpression pexp
Usage	`unresolvedProperty { PropertyExpression pexp -> if (pexp.propertyAsString == 'longueur' && getType(pexp.objectExpression) == STRING_TYPE) { storeType(pexp, int_TYPE) handled = true } }` Allows the developer to handle "dynamic" properties

Event name	unresolvedAttribute
Called When	Called when the type checker cannot find an attribute on the receiver
Arguments	AttributeExpression aexp

Usage	
	``
	unresolvedAttribute { AttributeExpression aexp ->
	    if (getType(aexp.objectExpression) == STRING_TYPE) {
	        storeType(aexp, STRING_TYPE)
	        handled = true
	    }
	}
	``
	
	Allows the developer to handle missing attributes

Event name	beforeMethodCall
Called When	Called before the type checker starts type checking a method call
Arguments	MethodCall call

Usage	
	``
	beforeMethodCall { call ->
	    if (isMethodCallExpression(call)
	            && call.methodAsString=='toUpperCase') {
	        addStaticTypeError('Not allowed',call)
	        handled = true
	    }
	}
	``
	
	Allows you to intercept method calls before the type checker performs its own checks. This is useful if you want to replace the default type checking with a custom one for a limited scope. In that case, you must set the handled flag to true, so that the type checker skips its own checks.

Event name	afterMethodCall
Called When	Called once the type checker has finished type checking a method call
Arguments	MethodCall call

Usage	
	``
	afterMethodCall { call ->
	    if (getTargetMethod(call).name=='toUpperCase') {
	        addStaticTypeError('Not allowed',call)
	        handled = true
	    }
	}
	``
	
	Allow you to perform additional checks after the type checker has done its own checks. This is in particular useful if you want to perform the standard type checking tests but also want to ensure additional type safety, for example checking the arguments against each other.Note that `afterMethodCall` is called even if you did `beforeMethodCall` and set the handled flag to true.

Event name	onMethodSelection
Called When	Called by the type checker when it finds a method appropriate for a method call
Arguments	Expression expr, MethodNode node

Usage	
	``
	onMethodSelection { expr, node ->
	    if (node.declaringClass.name == 'java.lang.String') {
	        // calling a method on 'String'
	        // let's perform additional checks!
	        if (++count>2) {
	            addStaticTypeError("You can use only 2 calls on String in your source code",expr)
	        }
	    }
	}
	``
	
	The type checker works by inferring argument types of a method call, then chooses a target method. If it finds one that corresponds, then it triggers this event. It is for example interesting if you want to react on a specific method call, such as entering the scope of a method that takes a closure as argument (as in builders).Please note that this event may be thrown for various types of expressions, not only method calls (binary expressions for example).

Event name	methodNotFound
Called When	Called by the type checker when it fails to find an appropriate method for a method call
Arguments	ClassNode receiver, String name, ArgumentListExpression argList, ClassNode[] argTypes,MethodCall call

Usage	```
	methodNotFound { receiver, name, argList, argTypes, call ->
	    // receiver is the inferred type of the receiver
	    // name is the name of the called method
	    // argList is the list of arguments the method was called with
	    // argTypes is the array of inferred types for each argument
	    // call is the method call for which we couldn't find a target method
	    if (receiver==classNodeFor(String)
	            && name=='longueur'
	            && argList.size()==0) {
	        handled = true
	        return newMethod('longueur', classNodeFor(String))
	    }
	}
	``
	
	Unlike `onMethodSelection`, this event is sent when the type checker cannot find a target method for a method call (instance or static). It gives you the chance to intercept the error before it is sent to the user, but also set the target method.For this, you need to return a list of `MethodNode`. In most situations, you would either return: an empty list, meaning that you didn't find a corresponding method, a list with exactly one element, saying that there's no doubt about the target methodIf you return more than one MethodNode, then the compiler would throw an error to the user stating that the method call is ambiguous, listing the possible methods.For convenience, if you want to return only one method, you are allowed to return it directly instead of wrapping it into a list.

Event name	beforeVisitMethod
Called When	Called by the type checker before type checking a method body
Arguments	MethodNode node

Usage	```
	beforeVisitMethod { methodNode -> // tell the type checker we will handle the body by ourselves handled = methodNode.name.startsWith('skip')
	}
	``
	
	The type checker will call this method before starting to type check a method body. If you want, for example, to perform type checking by yourself instead of letting the type checker do it, you have to set the handled flag to true. This event can also be used to help define the scope of your extension (for example, applying it only if you are inside method foo).

Event name	afterVisitMethod
Called When	Called by the type checker after type checking a method body
Arguments	MethodNode node

Usage	
	``
	afterVisitMethod { methodNode ->
	    scopeExit {
	        if (methods>2) {
	            addStaticTypeError("Method ${methodNode.name}
	contains more than 2 method calls", methodNode)
	        }
	    }
	}
	``
	
	Gives you the opportunity to perform additional checks after a method body is visited by the type checker. This is useful if you collect information, for example, and want to perform additional checks once everything has been collected.

Event name	beforeVisitClass
Called When	Called by the type checker before type checking a class
Arguments	ClassNode node

Usage	
	``
	beforeVisitClass { ClassNode classNode ->
	    def name = classNode.nameWithoutPackage
	    if (!(name[0] in 'A'..'Z')) {
	        addStaticTypeError("Class '${name}' doesn't start with
	an uppercase letter",classNode)
	    }
	}
	``
	
	If a class is type checked, then before visiting the class, this event will be sent. It is also the case for inner classes defined inside a class annotated with `@TypeChecked`. It can help you define the scope of your extension, or you can even totally replace the visit of the type checker with a custom type checking implementation. For that, you would have to set the `handled` flag to `true`.

Event name	afterVisitClass
Called When	Called by the type checker after having finished the visit of a type checked class
Arguments	ClassNode node

Usage	
	``
	afterVisitClass { ClassNode classNode ->
	    def name = classNode.nameWithoutPackage
	    if (!(name[0] in 'A'..'Z')) {
	        addStaticTypeError("Class '${name}' doesn't start with an uppercase letter",classNode)
	    }
	}
	``
	
	Called for every class being type checked after the type checker finished its work. This includes classes annotated with `@TypeChecked` and any inner/anonymous class defined in the same class with is not skipped.

Event name	incompatibleAssignment
Called When	Called when the type checker thinks that an assignment is incorrect, meaning that the right-hand side of an assignment is incompatible with the left-hand side
Arguments	ClassNode lhsType, ClassNode rhsType, Expression assignment

Usage	
	``
	incompatibleAssignment { lhsType, rhsType, expr ->
	    if (isBinaryExpression(expr) && isAssignment(expr.operation.type)) {
	        if (lhsType==classNodeFor(int) && rhsType==classNodeFor(Closure)) {
	            handled = true
	        }
	    }
	}
	``
	
	Gives the developer the ability to handle incorrect assignments. This is for example useful if a class overrides `setProperty`, because in that case it is possible that assigning a variable of one type to a property of another type is handled through that runtime mechanism. In that case, you can help the type checker just by telling it that the assignment is valid (using `handled` set to `true`).

Event name	incompatibleReturnType
Called When	Called when the type checker thinks that a return value is incompatibe with the return type of the enclosing closure or method
Arguments	ReturnStatement statement, ClassNode valueType

Usage	
	``
	incompatibleReturnType { stmt, type ->
	    if (type == STRING_TYPE) {
	        handled = true
	    }
	}
	``
	
	Gives the developer the ability to handle incorrect return values. This is for example useful when the return value will undergo implicit conversion or the enclosing closure's target type is difficult to infer properly. In that case, you can help the type checker just by telling it that the assignment is valid (by setting the `handled` property).

Event name	ambiguousMethods
Called When	Called when the type checker cannot choose between several candidate methods
Arguments	List<MethodNode> methods, Expression origin

Usage	
	``
	ambiguousMethods { methods, origin ->
	    // choose the method which has an Integer as parameter type
	    methods.find { it.parameters.any { it.type == classNodeFor(Integer) } }
	}
	``
	
	Gives the developer the ability to handle incorrect assignments. This is for example useful if a class overrides `setProperty`, because in that case it is possible that assigning a variable of one type to a property of another type is handled through that runtime mechanism. In that case, you can help the type checker just by telling it that the assignment is valid (using `handled` set to `true`).

Of course, an extension script may consist of several blocks, and you can have multiple blocks responding to the same event. This makes the DSL look nicer and easier to write. However, reacting to events is far from sufficient. If you know you can react to events, you also need to deal with the errors, which implies several *helper* methods that will make things easier.

Working with extensions

Support classes

The DSL relies on a support class called org.codehaus.groovy.transform.stc.GroovyTypeCheckingExtensionSupport . This class itself extends org.codehaus.groovy.transform.stc.TypeCheckingExtension . Those two classes define a number of *helper* methods that will make working with the AST easier, especially regarding type checking. One interesting thing to know is that you **have access to the type checker**. This means that you can programmatically call methods of the type checker, including those that allow you to

throw compilation errors.

The extension script delegates to the org.codehaus.groovy.transform.stc.GroovyTypeCheckingExtensionSupport class, meaning that you have direct access to the following variables:

- *context*: the type checker context, of type org.codehaus.groovy.transform.stc.TypeCheckingContext
- *typeCheckingVisitor*: the type checker itself, a org.codehaus.groovy.transform.stc.StaticTypeCheckingVisitor instance
- *generatedMethods*: a list of "generated methods", which is in fact the list of "dummy" methods that you can create inside a type checking extension using the `newMethod` calls

The type checking context contains a lot of information that is useful in context for the type checker. For example, the current stack of enclosing method calls, binary expressions, closures, ... This information is in particular important if you have to know *where* you are when an error occurs and that you want to handle it.

In addition to facilities provided by `GroovyTypeCheckingExtensionSupport` and `StaticTypeCheckingVisitor`, a type-checking DSL script imports static members from org.codehaus.groovy.ast.ClassHelper and org.codehaus.groovy.transform.stc.StaticTypeCheckingSupport granting access to common types via `OBJECT_TYPE`, `STRING_TYPE`, `THROWABLE_TYPE`, etc. and checks like `missesGenericsTypes(ClassNode)`, `isClassClassNodeWrappingConcreteType(ClassNode)` and so on.

Class nodes

Handling class nodes is something that needs particular attention when you work with a type checking extension. Compilation works with an abstract syntax tree (AST) and the tree may not be complete when you are type checking a class. This also means that when you refer to types, you must not use class literals such as `String` or `HashSet`, but to class nodes representing those types. This requires a certain level of abstraction and understanding how Groovy deals with class nodes. To make things easier, Groovy supplies several helper methods to deal with class nodes. For example, if you want to say "the type for String", you can write:

```
assert classNodeFor(String) instanceof ClassNode
```

You would also note that there is a variant of *classNodeFor* that takes a `String` as an argument, instead of a `Class`. In general, you should **not** use that one, because it would create a class node for which the name is `String`, but without any method, any property, ... defined on it. The first version returns a class node that is *resolved* but the second one returns one that is *not*. So the latter should be reserved for very special cases.

The second problem that you might encounter is referencing a type which is not yet compiled. This may happen more often than you think. For example, when you compile a set of files together. In that case, if you want to say "that variable is of type Foo" but `Foo` is not yet compiled, you can still refer to the `Foo` class node using `lookupClassNodeFor`:

```
assert lookupClassNodeFor('Foo') instanceof ClassNode
```

Helping the type checker

Say that you know that variable `foo` is of type `Foo` and you want to tell the type checker about it. Then you can use the `storeType` method, which takes two arguments: the first one is the node for which you want to store the type and the second one is the type of the node. If you look at the implementation of `storeType`, you would see that it delegates to the type checker equivalent method, which itself does a lot of work to store node metadata. You would also see that storing the type is not limited to variables: you can set the type of any expression.

Likewise, getting the type of an AST node is just a matter of calling `getType` on that node. This would in general be what you want, but there's something that you must understand:

- `getType` returns the **inferred type** of an expression. This means that it will not return, for a variable declared of type `Object` the class node for `Object`, but the inferred type of this variable **at this point of the code** (flow typing)
- if you want to access the origin type of a variable (or field/parameter), then you must call the appropriate method on the AST node

Throwing an error

To throw a type checking error, you only have to call the `addStaticTypeError` method which takes two arguments:

- a *message* which is a string that will be displayed to the end user
- an *AST node* responsible for the error. It's better to provide the best suiting AST node because it will be used to retrieve the line and column numbers

isXXXExpression

It is often required to know the type of an AST node. For readability, the DSL provides a special isXXXExpression method that will delegate to `x instance of XXXExpression`. For example, instead of writing:

```
if (node instanceof BinaryExpression) {
   ...
}
```

you can just write:

```
if (isBinaryExpression(node)) {
   ...
}
```

Virtual methods

When you perform type checking of dynamic code, you may often face the case when you know that a method call is valid but there is no "real" method behind it. As an example, take the Grails dynamic finders. You can have a method call consisting of a method named *findByName(...)*. As there's no *findByName* method defined in the bean, the type checker would complain. Yet, you would know that this method wouldn't fail at runtime, and you can even tell what is the return type of this method. For this case, the DSL supports two special constructs that consist of *phantom methods*. This means that you will return a method node that doesn't really exist but is defined in the context of type checking. Three methods exist:

- `newMethod(String name, Class returnType)`
- `newMethod(String name, ClassNode returnType)`
- `newMethod(String name, Callable<ClassNode> return Type)`

All three variants do the same: they create a new method node which name is the supplied name and define the return type of this method. Moreover, the type checker would add those methods in the `generatedMethods` list (see `isGenerated` below). The reason why we only set a name and a return type is that it is only what you need in 90% of the cases. For example, in the `findByName` example upper, the only thing you need to know is that `findByName` wouldn't fail at runtime, and that it returns a domain class. The `Callable` version of return type is interesting because it defers the computation of the return type when the type checker actually needs it. This is interesting because in some circumstances, you may not know the actual return type when the type checker demands it, so you can use a closure that will be called each time `getReturnType` is called by the type checker on this method node. If you combine this with deferred checks, you can achieve pretty complex type checking including handling of forward references.

```
newMethod(name) {
    // each time getReturnType on this method node will be called, this closure will
be called!
    println 'Type checker called me!'
    lookupClassNodeFor(Foo) // return type
}
```

Should you need more than the name and return type, you can always create a new `MethodNode` by yourself.

Scoping

Scoping is very important in DSL type checking and is one of the reasons why we couldn't use a *pointcut* based approach to DSL type checking. Basically, you must be able to define very precisely when your extension applies and when it does not. Moreover, you must be able to handle situations that a regular type checker would not be able to handle, such as forward references:

```
point a(1,1)
line a,b // b is referenced afterwards!
point b(5,2)
```

Say for example that you want to handle a builder:

```
builder.foo {
   bar
   baz(bar)
}
```

Your extension, then, should only be active once you've entered the `foo` method, and inactive outside this scope. But you could have complex situations like multiple builders in the same file or embedded builders (builders in builders). While you should not try to fix all this from start (you must accept limitations to type checking), the type checker does offer a nice mechanism to handle this: a scoping stack, using the `newScope` and `scopeExit` methods.

- `newScope` creates a new scope and puts it on top of the stack
- `scopeExits` pops a scope from the stack

A scope consists of:

- a parent scope
- a map of custom data

If you want to look at the implementation, it's simply a `LinkedHashMap` (org.codehaus.groovy.transform.stc.GroovyTypeCheckingExtensionSupport.TypeCheckingScope), but it's quite powerful. For example, you can use such a scope to store a list of closures to be executed when you exit the scope. This is how you would handle forward references:

```
def scope = newScope()
scope.secondPassChecks = []
//...
scope.secondPassChecks << { println 'executed later' }
// ...
scopeExit {
    secondPassChecks*.run() // execute deferred checks
}
```

That is to say, that if at some point you are not able to determine the type of an expression, or that you are not able to check at this point that an assignment is valid or not, you can still make the check later… This is a very powerful feature. Now, `newScope` and `scopeExit` provide some interesting syntactic sugar:

```
newScope {
    secondPassChecks = []
}
```

At anytime in the DSL, you can access the current scope using `getCurrentScope()` or more simply `currentScope`:

```
//...
currentScope.secondPassChecks << { println 'executed later' }
// ...
```

The general schema would then be:

- determine a *pointcut* where you push a new scope on stack and initialize custom variables within this scope
- using the various events, you can use the information stored in your custom scope to perform checks, defer checks,...
- determine a *pointcut* where you exit the scope, call `scopeExit` and eventually perform additional checks

Other useful methods

For the complete list of helper methods, please refer to the org.codehaus.groovy.transform.stc.GroovyTypeCheckingExtensionSupport and org.codehaus.groovy.transform.stc.TypeCheckingExtension classes. However, take special attention to those methods:

- `isDynamic`: takes a VariableExpression as argument and returns true if the variable is a DynamicExpression, which means, in a script, that it wasn't defined using a type or `def`.
- `isGenerated`: takes a MethodNode as an argument and tells if the method is one that was generated by the type checker extension using the `newMethod` method
- `isAnnotatedBy`: takes an AST node and a Class (or ClassNode), and tells if the node is annotated with this class. For example: `isAnnotatedBy(node, NotNull)`
- `getTargetMethod`: takes a method call as argument and returns the `MethodNode` that the type checker has determined for it
- `delegatesTo`: emulates the behaviour of the `@DelegatesTo` annotation. It allows you to tell that the argument will delegate to a specific type (you can also specify the delegation strategy)

Advanced type checking extensions

Precompiled type checking extensions

All the examples above use type checking scripts. They are found in source form in classpath, meaning that:

- a Groovy source file, corresponding to the type checking extension, is available on compilation classpath
- this file is compiled by the Groovy compiler for each source unit being compiled (often, a source unit corresponds to a single file)

It is a very convenient way to develop type checking extensions, however it implies a slower compilation phase, because of the compilation of the extension itself for each file being compiled. For those reasons, it can be practical to rely on a precompiled extension. You have two options to do

this:

- write the extension in Groovy, compile it, then use a reference to the extension class instead of the source
- write the extension in Java, compile it, then use a reference to the extension class

Writing a type checking extension in Groovy is the easiest path. Basically, the idea is that the type checking extension script becomes the body of the main method of a type checking extension class, as illustrated here:

```
import org.codehaus.groovy.transform.stc.GroovyTypeCheckingExtensionSupport

class PrecompiledExtension extends GroovyTypeCheckingExtensionSupport.TypeCheckingDSL
{     ①
    @Override
    Object run() {
②
        unresolvedVariable { var ->
            if ('robot'==var.name) {
                storeType(var, classNodeFor(Robot))
③
                handled = true
            }
        }
    }
}
```

① extending the `TypeCheckingDSL` class is the easiest

② then the extension code needs to go inside the `run` method

③ and you can use the very same events as an extension written in source form

Setting up the extension is very similar to using a source form extension:

```
config.addCompilationCustomizers(
    new ASTTransformationCustomizer(
        TypeChecked,
        extensions:['typing.PrecompiledExtension'])
)
```

The difference is that instead of using a path in classpath, you just specify the fully qualified class name of the precompiled extension.

In case you really want to write an extension in Java, then you will not benefit from the type checking extension DSL. The extension above can be rewritten in Java this way:

```
import org.codehaus.groovy.ast.ClassHelper;
import org.codehaus.groovy.ast.expr.VariableExpression;
```

```
import org.codehaus.groovy.transform.stc.AbstractTypeCheckingExtension;
import org.codehaus.groovy.transform.stc.StaticTypeCheckingVisitor;

public class PrecompiledJavaExtension extends AbstractTypeCheckingExtension {
①

    public PrecompiledJavaExtension(final StaticTypeCheckingVisitor
typeCheckingVisitor) {
        super(typeCheckingVisitor);
    }

    @Override
    public boolean handleUnresolvedVariableExpression(final VariableExpression vexp) {
②
        if ("robot".equals(vexp.getName())) {
            storeType(vexp, ClassHelper.make(Robot.class));
            setHandled(true);
            return true;
        }
        return false;
    }
}
```

① extend the `AbstractTypeCheckingExtension` class

② then override the `handleXXX` methods as required

Using @Grab in a type checking extension

It is totally possible to use the `@Grab` annotation in a type checking extension. This means you can include libraries that would only be available at compile time. In that case, you must understand that you would increase the time of compilation significantly (at least, the first time it grabs the dependencies).

Sharing or packaging type checking extensions

A type checking extension is just a script that need to be on classpath. As such, you can share it as is, or bundle it in a jar file that would be added to classpath.

Global type checking extensions

While you can configure the compiler to transparently add type checking extensions to your script, there is currently no way to apply an extension transparently just by having it on classpath.

Type checking extensions and @CompileStatic

Type checking extensions are used with `@TypeChecked` but can also be used with `@CompileStatic`. However, you must be aware that:

- a type checking extension used with `@CompileStatic` will in general not be sufficient to let the compiler know how to generate statically compilable code from "unsafe" code

- it is possible to use a type checking extension with `@CompileStatic` just to enhance type checking, that is to say introduce **more** compilation errors, without actually dealing with dynamic code

Let's explain the first point, which is that even if you use an extension, the compiler will not know how to compile your code statically: technically, even if you tell the type checker what is the type of a dynamic variable, for example, it would not know how to compile it. Is it `getBinding('foo')`, `getProperty('foo')`, `delegate.getFoo()`,...? There's absolutely no direct way to tell the static compiler how to compile such code even if you use a type checking extension (that would, again, only give hints about the type).

One possible solution for this particular example is to instruct the compiler to use mixed mode compilation. The more advanced one is to use AST transformations during type checking but it is far more complex.

Type checking extensions allow you to help the type checker where it fails, but it also allows you to fail where it doesn't. In that context, it makes sense to support extensions for `@CompileStatic` too. Imagine an extension that is capable of type checking SQL queries. In that case, the extension would be valid in both dynamic and static context, because without the extension, the code would still pass.

Mixed mode compilation

In the previous section, we highlighted the fact that you can activate type checking extensions with `@CompileStatic`. In that context, the type checker would not complain anymore about some unresolved variables or unknown method calls, but it would still wouldn't know how to compile them statically.

Mixed mode compilation offers a third way, which is to instruct the compiler that whenever an unresolved variable or method call is found, then it should fall back to a dynamic mode. This is possible thanks to type checking extensions and a special `makeDynamic` call.

To illustrate this, let's come back to the `Robot` example:

```
robot.move 100
```

And let's try to activate our type checking extension using `@CompileStatic` instead of `@TypeChecked`:

```
def config = new CompilerConfiguration()
config.addCompilationCustomizers(
    new ASTTransformationCustomizer(
        CompileStatic,                                  ①
        extensions:['robotextension.groovy'])           ②
)
def shell = new GroovyShell(config)
def robot = new Robot()
shell.setVariable('robot', robot)
shell.evaluate(script)
```

① Apply `@CompileStatic` transparently

② Activate the type checking extension

The script will run fine because the static compiler is told about the type of the `robot` variable, so it is capable of making a direct call to `move`. But before that, how did the compiler know how to get the `robot` variable? In fact by default, in a type checking extension, setting `handled=true` on an unresolved variable will automatically trigger a dynamic resolution, so in this case you don't have anything special to make the compiler use a mixed mode. However, let's slightly update our example, starting from the robot script:

```
move 100
```

Here you can notice that there is no reference to `robot` anymore. Our extension will not help then because we will not be able to instruct the compiler that `move` is done on a `Robot` instance. This example of code can be executed in a totally dynamic way thanks to the help of a groovy.util.DelegatingScript:

```
def config = new CompilerConfiguration()
config.scriptBaseClass = 'groovy.util.DelegatingScript'     ①
def shell = new GroovyShell(config)
def runner = shell.parse(script)                            ②
runner.setDelegate(new Robot())                             ③
runner.run()                                                ④
```

① we configure the compiler to use a `DelegatingScript` as the base class

② the script source needs to be parsed and will return an instance of `DelegatingScript`

③ we can then call `setDelegate` to use a `Robot` as the delegate of the script

④ then execute the script. `move` will be directly executed on the delegate

If we want this to pass with `@CompileStatic`, we have to use a type checking extension, so let's update our configuration:

```
config.addCompilationCustomizers(
    new ASTTransformationCustomizer(
        CompileStatic,                                      ①
        extensions:['robotextension2.groovy'])              ②
)
```

① apply `@CompileStatic` transparently

② use an alternate type checking extension meant to recognize the call to `move`

Then in the previous section we have learnt how to deal with unrecognized method calls, so we are able to write this extension:

robotextension2.groovy

```
methodNotFound { receiver, name, argList, argTypes, call ->
    if (isMethodCallExpression(call)                          ①
        && call.implicitThis                                  ②
        && 'move'==name                                       ③
        && argTypes.length==1                                 ④
        && argTypes[0] == classNodeFor(int)                   ⑤
    ) {
        handled = true                                        ⑥
        newMethod('move', classNodeFor(Robot))                ⑦
    }
}
```

① if the call is a method call (not a static method call)

② that this call is made on "implicit this" (no explicit `this.`)

③ that the method being called is `move`

④ and that the call is done with a single argument

⑤ and that argument is of type `int`

⑥ then tell the type checker that the call is valid

⑦ and that the return type of the call is `Robot`

If you try to execute this code, then you could be surprised that it actually fails at runtime:

```
java.lang.NoSuchMethodError: java.lang.Object.move()Ltyping/Robot;
```

The reason is very simple: while the type checking extension is sufficient for `@TypeChecked`, which does not involve static compilation, it is not enough for `@CompileStatic` which requires additional information. In this case, you told the compiler that the method existed, but you didn't explain to it **what** method it is in reality, and what is the receiver of the message (the delegate).

Fixing this is very easy and just implies replacing the `newMethod` call with something else:

robotextension3.groovy

```
methodNotFound { receiver, name, argList, argTypes, call ->
    if (isMethodCallExpression(call)
        && call.implicitThis
        && 'move'==name
        && argTypes.length==1
        && argTypes[0] == classNodeFor(int)
    ) {
        makeDynamic(call, classNodeFor(Robot))                ①
    }
}
```

① tell the compiler that the call should be made dynamic

The `makeDynamic` call does 3 things:

- it returns a virtual method just like `newMethod`
- automatically sets the `handled` flag to `true` for you
- but also marks the `call` to be done dynamically

So when it comes time for the compiler to generate bytecode for the call to `move`, since it is now marked as a dynamic call, it falls back to the dynamic compiler and lets it handle the call. And since the extension tells us that the return type of the dynamic call is a `Robot`, subsequent calls will be done statically!

Some might wonder why the static compiler doesn't do this by default without an extension. It is a design decision. If the code is statically compiled, we normally want compile-time type safety and the best performance. If unrecognized variables/method calls were automatically made dynamic, you'd partially lose compile-time type safety. Typos and some scenarios of incorrect typing would no longer be caught at compile time!

In short, if you want to have mixed mode compilation, it **has** to be explicit, through a type checking extension, so that the compiler, and the designer of the DSL, are totally aware of exactly when typing rules are relaxed.

`makeDynamic` can be used on 3 kinds of AST nodes:

- a method node (`MethodNode`)
- a variable (`VariableExpression`)
- a property expression (`PropertyExpression`)

If that is not enough, then it means that static compilation cannot be done directly and that you have to rely on AST transformations.

Transforming the AST in an extension

Type checking extensions look very attractive from an AST transformation design point of view: extensions have access to context like inferred types, which is often nice to have. And an extension has a direct access to the abstract syntax tree. Since you have access to the AST, there is nothing in theory that prevents you from modifying the AST. However, we do not recommend you to do so, unless you are an advanced AST transformation designer and well aware of the compiler internals:

- First of all, you would explicitly break the contract of type checking, which is to annotate, and only annotate the AST. Type checking should **not** modify the AST tree because you wouldn't be able to guarantee anymore that code without the *@TypeChecked* annotation behaves the same without the annotation.
- If your extension is meant to work with *@CompileStatic*, then you **can** modify the AST because this is indeed what *@CompileStatic* will eventually do. Static compilation doesn't guarantee the same semantics at dynamic Groovy so there is effectively a difference between code compiled with *@CompileStatic* and code compiled with *@TypeChecked*. It's up to you to choose whatever

strategy you want to update the AST, but probably using an AST transformation that runs before type checking is easier.

- if you cannot rely on a transformation that kicks in before the type checker, then you must be **very** careful

WARNING	The type checking phase is the last phase running in the compiler before bytecode generation. All other AST transformations run before that and the compiler does a very good job at "fixing" incorrect AST generated before the type checking phase. As soon as you perform a transformation during type checking, for example directly in a type checking extension, then you have to do all this work of generating a 100% compiler compliant abstract syntax tree by yourself, which can easily become complex. That's why we do not recommend to go that way if you are beginning with type checking extensions and AST transformations.

Examples

Examples of real life type checking extensions are easy to find. You can download the source code for Groovy and take a look at the TypeCheckingExtensionsTest class which is linked to various extension scripts.

An example of a complex type checking extension can be found in the Markup Template Engine source code: this template engine relies on a type checking extension and AST transformations to transform templates into fully statically compiled code. Sources for this can be found here.

Async/Await (Incubating)

Introduction

Groovy provides native `async`/`await` support, enabling developers to write concurrent code in a sequential, readable style. Rather than dealing with callbacks, `CompletableFuture` chains, or manual thread management, you express concurrency with two constructs:

- `async { ⋯ }` — start a closure on a background thread, returning an `Awaitable`
- `await expr` — block until an asynchronous result is available

On JDK 21+, async tasks automatically leverage virtual threads for optimal scalability. On JDK 17–20, a cached thread pool is used as a fallback.

The examples throughout this guide use an online multiplayer card game as a running theme — dealing hands, racing for the fastest play, streaming cards, and managing tournament rounds.

Getting Started

Your first async/await

`async { ⋯ }` starts work immediately on a background thread and returns an `Awaitable`. Use `await` to collect the result:

```
def deck = ['2♠', '3♠', 'K♠', 'A♠']
def card = async { deck.shuffled()[0] }
println "You drew: ${await card}"
```

Exception handling

`await` unwraps `CompletionException` and `ExecutionException` automatically. Standard `try`/`catch` works exactly as with synchronous code:

```
def drawFromEmpty = async {
    throw new IllegalStateException('deck is empty')
}
try {
    await drawFromEmpty
} catch (IllegalStateException e) {
    // Original exception — no CompletionException wrapper
    assert e.message == 'deck is empty'
}
```

CompletableFuture interop

`await` works directly with `CompletableFuture`, `CompletionStage`, and `Future` from Java libraries:

```
// await works with CompletableFuture from Java libraries
def future = CompletableFuture.supplyAsync { 'A♠' }
assert await(future) == 'A♠'
```

Running Tasks in Parallel

The real power of async/await appears when you need to run several tasks concurrently and coordinate their results.

Waiting for all: `Awaitable.all()`

Deal cards to multiple players at the same time and wait for all hands:

```
// Deal cards to three players concurrently
def deck = [*1..9,'J','Q','K','A'].collectMany { rank ->
    ['♠','♥','♦','♣'].collect { suit -> "$rank$suit" }
}.shuffled()
def index = new AtomicInteger()
def draw5 = { -> def i = index.getAndAdd(5); deck[i..i+4] }

def alice = async { draw5() }
def bob   = async { draw5() }
def carol = async { draw5() }
```

```
def (a, b, c) = await Awaitable.all(alice, bob, carol)
assert a.size() == 5 && b.size() == 5 && c.size() == 5
```

Multi-argument `await` is syntactic sugar for `Awaitable.all()`:

```
def a = async { 1 }
def b = async { 2 }
def c = async { 3 }

// Parenthesized multi-arg await (sugar for Awaitable.all):
def results = await(a, b, c)
assert results == [1, 2, 3]
```

Racing: `Awaitable.any()`

Like JavaScript's `Promise.race()` — settles as soon as the first task settles, returning its result *or* propagating its error. A task that fails first therefore fails the whole `any`, even if a later task would have succeeded (use `first` if you want the first *success* instead). Useful for fallback patterns or latency-sensitive code:

```
// Race two servers — use whichever responds first
def primary  = async { Thread.sleep(200); 'primary-response' }
def fallback = async { 'fallback-response' }

def response = await Awaitable.any(primary, fallback)
assert response == 'fallback-response'
```

First success: `Awaitable.first()`

Like JavaScript's `Promise.any()` — returns the first *successful* result, silently ignoring individual failures. Only fails when every task fails:

```
// Try multiple sources — use the first that succeeds
def failing    = async { throw new RuntimeException('server down') }
def succeeding = async { 'card-data-from-cache' }

def result = await Awaitable.first(failing, succeeding)
assert result == 'card-data-from-cache'
```

Inspecting all outcomes: `Awaitable.allSettled()`

Waits for all tasks to finish (succeed or fail) without throwing. Returns an `AwaitResult` list where each entry has `success`, `value`, and `error` fields:

```
def save1 = async { 42 }
```

```
def save2 = async { throw new RuntimeException('db error') }

def results = await Awaitable.allSettled(save1, save2)
assert results[0].success && results[0].value == 42
assert !results[1].success && results[1].error.message == 'db error'
```

Combinator summary

Combinator	Completes when	On failure	Use case
`Awaitable.all`	All succeed	Fails immediately on first failure (fail-fast)	Gather results from independent tasks
`Awaitable.allSettled`	All complete (success or fail)	Never throws; failures captured in `AwaitResult` list	Inspect every outcome, e.g. partial-success reporting
`Awaitable.any`	First task settles (success or failure)	Propagates the first settlement's result or error (JS `Promise.race`)	Latency-sensitive races, fastest-response wins
`Awaitable.first`	First task succeeds, or all fail	Throws only when every source fails (aggregate error)	Hedged requests, graceful degradation with fallbacks

NOTE `all` and `allSettled` accept an empty source list and complete with an empty result. `any` and `first` require at least one source (and reject `null` elements), throwing `IllegalArgumentException` otherwise — a "race" or "first success" with no participants has no meaningful result.

Generators and Streaming

Producing values with `yield return`

An `async` closure containing `yield return` becomes a *generator* — it lazily produces a sequence of values. The generator runs on a background thread and blocks on each `yield return` until the consumer is ready, providing natural back-pressure:

```
def dealCards = async {
    def suits = ['♠', '♥', '♦', '♣']
    def ranks = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']
    for (suit in suits) {
        for (rank in ranks) {
            yield return "$rank$suit"
        }
    }
}
def cards = dealCards.collect()
assert cards.size() == 52
assert cards.first() == 'A♠'
```

```
assert cards.last() == 'K♠'
```

Generators return a standard `Iterable`, so regular `for` loops and Groovy collection methods (`collect`, `findAll`, `take`, etc.) work out of the box:

```
def scores = async {
    for (s in [100, 250, 75]) { yield return s }
}
// Generators return Iterable — regular for and collect work
assert scores.collect { it * 2 } == [200, 500, 150]
```

Channels

Channels provide Go-style inter-task communication. A producer sends values into a channel; a consumer receives them as they arrive. The channel handles synchronization and optional buffering — no shared mutable state needed:

```
def cardStream = AsyncChannel.create(3)

// Dealer — sends cards concurrently
async {
    for (card in ['A♠', 'K♠', 'Q♠', 'J♠']) {
        await cardStream.send(card)
    }
    cardStream.close()
}

// Player — receives cards as they arrive
def hand = []
for await (card in cardStream) {
    hand << card
}
assert hand == ['A♠', 'K♠', 'Q♠', 'J♠']
```

Channels support unbuffered (rendezvous, `create()`) and buffered (`create(n)`) modes. Sending blocks when the buffer is full; receiving blocks when empty. With virtual threads, this blocking is essentially free.

Since channels implement `Iterable`, they also work with regular `for` loops and Groovy collection methods.

Channel Composition

`AsyncChannel` supports composable pipeline operations. Each composition method returns a new channel connected to the source by a background task.

filter

```
def source = AsyncChannel.create(10)
def evens = source.filter { it % 2 == 0 }

async {
    (1..10).each { source.send(it) }
    source.close()
}

for await (val in evens) { println val }  // 2, 4, 6, 8, 10
```

map

```
def doubled = source.map { it * 2 }
```

Chaining

Operations compose naturally:

```
def pipeline = source
    .filter { it > 50 }
    .map { it * 2 }

for await (val in pipeline) { process(val) }
```

merge

Combines two channels into one, interleaving values as they arrive:

```
def merged = channelA.merge(channelB)
for await (event in merged) { handle(event) }
```

split

Partitions a channel into two based on a predicate:

```
def (highPriority, normal) = orders.split { it.priority > 5 }
```

tap

Sends a copy of each value to a monitoring channel while passing through unchanged:

```
def monitor = AsyncChannel.create(100)
```

```
def output = source.tap(monitor)
// output gets all values; monitor gets copies for logging
```

Channel Select

`ChannelSelect` waits for the first available value from multiple channels — like Go's `select` statement:

```
import groovy.concurrent.ChannelSelect

def prices = AsyncChannel.create(10)
def alerts = AsyncChannel.create(10)

def sel = ChannelSelect.from(prices, alerts)
def result = await sel.select()
println "Channel ${result.index}: ${result.value}"
```

A select takes exactly one value from exactly one channel; the other channels are left untouched. When several channels already have a value waiting, the one listed first is taken. If every channel is closed and drained, `select()` fails with `ChannelClosedException`.

That priority by list order means a channel that always has a value waiting starves the channels listed after it. `fair()` returns a select that rotates instead — each call starts at the channel after the one that last won, like JCSP's `fairSelect` — so keep the instance across calls:

```
def sel = ChannelSelect.from(requests, control).fair()
while (true) {
    def result = await sel.select()
    // control is served even while requests never runs dry
}
```

`random()` chooses uniformly at random among the ready channels instead, as Go's `select` does. It keeps no state between calls, so it is equally fair from any number of threads, but it gives no bound on how long a ready channel may be passed over; `fair()` guarantees every ready channel is taken within *n* calls.

Mixed offers

A select may also *offer to send* — the mixed choice of the CSP literature: "I will send my opener, or take my peer's if it sends first". Offers are built with the static factories `send` and `receive` and combined with `offers`, mirroring the send cases of Go's `select`:

```
import static groovy.concurrent.ChannelSelect.*

def result = await offers(send(ping, 1), receive(pong)).select()
if (result.send) {
    // my opener committed: continue the ping branch
```

```
} else {
    // my peer opened first: result.value is the pong opener
}
```

Exactly one offer commits; a committed send behaves exactly like `ping.send(1)`, while a retired one leaves no trace — no buffered residue, no waiting sender — just as a losing receive leaves its channel untouched. `fair()` and `random()` order the scan over offers exactly as they do over channels, and send offers require channels created by `AsyncChannel.create`.

A send offer commits as soon as its channel can accept the value: when a receiver is waiting, or when buffer space holds it. Two peers racing a mixed choice against each other therefore agree on a single opener only over rendezvous (capacity-0) channels, where a send cannot complete unilaterally; over buffered channels with space, each peer's send offer commits at once under its own select and the peers proceed down different branches of the same conversation.

Preconditions

Each branch may also carry a *precondition* — the guarded choice of CSP. `when` writes the guard onto the offer itself, as occam, Ada, Erlang and Kotlin do:

```
def sel = offers(receive(input).when { held.size() < capacity },   ①
                 receive(request).when { !held.isEmpty() })        ②
while (true) {
    def result = await sel.select()
    if (result.index == 0) {
        held.addLast(result.value)
    } else {
        await reply.send(held.removeFirst())
    }
}
```

① take input only while there is room

② answer a request only while there is something to hand over

A guarded-off offer is not registered on its channel, so nothing is taken from it and nothing is sent to it, but it keeps its position, so `result.index` denotes the same branch either way. Dropping an offer from the argument list instead would renumber every branch after it, silently changing what an index denotes.

The guard is a closure rather than a boolean because it is consulted afresh on every `select` call — that is what lets one select be built once and reused as the state it guards on changes. `when` returns a guarded copy, leaving the original offer alone, and guards conjoin: over a second `when`, both must hold.

`select` also accepts the same preconditions positionally, one flag per offer, which suits a mask computed as a whole:

```
def result = await sel.select(held.size() < capacity, !held.isEmpty())
```

A flag is conjoined with whatever guard its offer carries — both must hold for the offer to be registered — and passing a number of flags other than the number of offers is an `IllegalArgumentException`. Enabling no branch at all is a deadlock rather than a wait, so `select` fails with `IllegalStateException`.

A branch can also be recognised by identity instead of by position: `result.channel` is the channel the winning offer transferred over.

Timeouts

A deadline can be a branch of the select, as it is in Go, Rust, Clojure and Kotlin, instead of an exception thrown around the whole select by `orTimeout`. There are two forms, one for a timeout that re-arms on every round and one for a fixed deadline.

The *timer offer* `after` arms a fresh timer each time `select` is called and commits, with the `Instant` it fired, if no channel offer could commit first. It is the "wait for work, but not forever" branch of a select that is built once and reused, and because the timer lives inside the call, the held instance keeps its `fair()` rotation from round to round:

```
import static groovy.concurrent.ChannelSelect.*

def sel = offers(receive(work), after(100)).fair()
while (true) {
    def result = await sel.select()
    if (result.timeout) {
        // nothing arrived within 100 ms of this call: result.value is the Instant
    } else {
        handle(result.value)
    }
}
```

A timer offer that loses its round is cancelled with the other losing branches, so it leaves nothing armed. It reports `result.timeout` rather than a channel: `result.channel` is `null` for a timeout, and `result.index` still names the branch. So a select that carries a timer offer and dispatches on `result.channel` must test `result.timeout` first, as the loop above does. Like any offer it can be guarded with `when`, and a guarded-off timer offer arms nothing. Several timer offers give per-branch deadlines.

A timer offer with a positive delay is never ready at the moment the select registers its offers, so the choice policy considers only the channel offers that are ready at once; the timer commits only when none was. A delay that is not positive has already elapsed, so that offer is ready at registration and the policy treats it like any other ready branch: priority takes whichever is listed first, and `fair()` rotates over it. Listed last, `after(0)` is the `default` clause of Go's `select` — take a transfer if one can complete at once, otherwise proceed without waiting:

```
def result = await offers(receive(work), after(0)).select()
if (result.timeout) {
    // nothing was ready; do something else and come back
}
```

The *timer channel* `AsyncChannel.after` is the fixed-deadline form: a capacity-1 channel that delivers one value, the `Instant` it fired, once the given delay has elapsed from its creation, and then closes, as Go's `time.After` does. Received from within a select, it is a deadline shared by every round that selects on it:

```
def deadline = AsyncChannel.after(5000)
def sel = ChannelSelect.from(work, deadline)
while (true) {
    def result = await sel.select()
    if (result.index == 1) break    // five seconds since the deadline was set
    handle(result.value)
}
```

Because it is an ordinary channel, a timer channel composes with everything a channel does: it can be guarded with `when`, and once fired `fair()` and `random()` treat it like any other ready channel. A timer channel that loses a select keeps its value: it fires all the same and holds the instant until received. Once that value has been taken, further receives fail with `ChannelClosedException` rather than waiting forever, and `isClosed()` reports that the timer has fired. Closing a timer channel before it fires cancels it.

Both forms accept a `java.time.Duration` in place of milliseconds, and in both a delay that is not positive has already elapsed: the timer channel is created already holding its instant, and the timer offer is ready at registration. That suits a deadline computed from an absolute time that has already passed.

BroadcastChannel

A `BroadcastChannel` delivers every value to all subscribers (one-to-many), unlike `AsyncChannel` which is point-to-point:

```
import groovy.concurrent.BroadcastChannel

def broadcast = BroadcastChannel.create()
def sub1 = broadcast.subscribe()
def sub2 = broadcast.subscribe()

async {
    broadcast.send('hello')
    broadcast.send('world')
    broadcast.close()
}
```

```
// Both subscribers receive both messages
for await (msg in sub1) { println "Sub1: $msg" }
for await (msg in sub2) { println "Sub2: $msg" }
```

Late subscribers only receive messages sent after they subscribe.

Flow.Publisher interop

A `BroadcastChannel` can also be exposed as a `java.util.concurrent.Flow.Publisher` via `asPublisher()`. Each call to `subscribe(Flow.Subscriber)` on the returned publisher creates a new per-subscriber `AsyncChannel` under the hood and respects standard Reactive Streams demand (`request(n)`):

```
def broadcast = BroadcastChannel.create()
def publisher = broadcast.asPublisher()

// Compose with any Reactive Streams operator, or consume with `for await`:
for await (msg in publisher) {
    println "Got: $msg"
}
```

WARNING

Backpressure policy. The publisher bridge uses *lossless, sender-gated* backpressure: `send()` awaits delivery to every live subscriber, and each per-subscriber channel has a bounded buffer (default 16). A subscriber that never calls `request(n)` — or requests slowly — will fill its buffer, after which `broadcast.send(⋯)` stalls for **all** subscribers until that subscriber catches up. In other words, the slowest subscriber controls producer throughput.

This matches the point-to-point semantics of `subscribe()` (values are neither dropped nor reordered). If you need decoupled per-subscriber policies (drop-newest, drop-oldest, latest-only, or unbounded buffering), wrap the publisher with a Reactive Streams operator of your choice, or use a subscriber that drains promptly with `request(Long.MAX_VALUE)`.

NOTE

Terminal signals. Per Reactive Streams §1.7, terminal completion is not gated by demand: once the underlying `BroadcastChannel` is closed and the per-subscriber buffer is drained, the subscriber receives `onComplete` even if it has no outstanding `request(n)`. Likewise, `onError` for a spec-violating `request(n ⇐ 0)` (§3.9) is signalled through the same single-threaded delivery path as `onNext` / `onComplete`, preserving §1.3 signal serialization; subscribers do not see those signals racing on the caller's thread.

Deferred Cleanup with `defer`

The `defer` keyword schedules a cleanup action to run when the enclosing `async` closure completes, regardless of success or failure. Multiple deferred actions execute in LIFO order — last registered, first to run — making it natural to pair resource acquisition with cleanup:

```
def log = []
def task = async {
    log << 'open connection'
    defer { log << 'close connection' }
    log << 'open transaction'
    defer { log << 'close transaction' }
    log << 'save game state'
    'saved'
}
assert await(task) == 'saved'
// Deferred actions run in LIFO order — last registered, first to run
assert log == ['open connection', 'open transaction', 'save game state',
               'close transaction', 'close connection']
```

Deferred actions always run, even when an exception occurs:

```
def cleaned = false
def task = async {
    defer { cleaned = true }
    throw new RuntimeException('save failed')
}
try {
    await task
} catch (RuntimeException e) {
    assert e.message == 'save failed'
}
// Deferred actions run even when an exception occurs
assert cleaned
```

If a deferred action returns an `Awaitable` or `Future`, the result is awaited before the next deferred action runs, ensuring orderly cleanup of asynchronous resources.

Structured Concurrency

Structured concurrency ensures that concurrent tasks have clear ownership and bounded lifetimes — no orphaned background work, no silent failures leaking across your application. This idea is gaining momentum across the industry (Java's JEP 453, Go's `errgroup`, Kotlin's coroutine scopes) because it makes concurrent code easier to reason about, test, and debug. Groovy's `AsyncScope` provides these guarantees today, even on JDK versions before Project Loom's `StructuredTaskScope` ships as a final API.

`AsyncScope` binds the lifetime of child tasks to a scope. When the scope exits, all children are guaranteed complete (or cancelled). This prevents orphaned tasks and silent failures:

```
// Run a tournament round — all tables play concurrently
def results = AsyncScope.withScope { scope ->
    def table1 = scope.async { [winner: 'Alice',  score: 320] }
```

```
    def table2 = scope.async { [winner: 'Bob',    score: 280] }
    def table3 = scope.async { [winner: 'Carol',  score: 410] }
    [await(table1), await(table2), await(table3)]
}
// All tables guaranteed complete when withScope returns
assert results.size() == 3
assert results.max { it.score }.winner == 'Carol'
```

By default, the scope uses **fail-fast** semantics — if any child fails, all siblings are cancelled immediately:

```
try {
    AsyncScope.withScope { scope ->
        scope.async { Thread.sleep(5000); 'still playing' }
        scope.async { throw new RuntimeException('player disconnected') }
    }
} catch (RuntimeException e) {
    // First failure cancels all siblings and propagates
    assert e.message == 'player disconnected'
}
```

The scope waits for every child to finish, even without explicit `await` calls:

```
def completed = new AtomicInteger(0)
AsyncScope.withScope { scope ->
    3.times { scope.async { Thread.sleep(50); completed.incrementAndGet() } }
}
// All children have completed — even without explicit await
assert completed.get() == 3
```

On JDK 25+, scope tracking uses `ScopedValue` for optimal virtual thread performance (no per-thread storage, automatic inheritance). On JDK 17–24, a `ThreadLocal` fallback is used transparently.

Advanced Topics

Consuming with `for await`

`for await` iterates over any async source. For generators and plain collections, it works identically to a regular `for` loop:

```
def topCards = async {
    for (card in ['A□', 'K□', 'Q□']) {
        yield return card
    }
}
def hand = []
for await (card in topCards) {
```

```
    hand << card
}
assert hand == ['A♠', 'K♠', 'Q♠']
```

The key value of `for await` is with **reactive types** (Reactor `Flux`, RxJava `Observable`, `java.util.concurrent.Flow.Publisher`) where it automatically converts the source to a blocking iterable via the adapter SPI. Without `for await`, you would need to call the conversion manually (e.g., `flux.toIterable()`). For generators and plain collections, a regular `for` loop works identically.

The JDK's `Flow.Publisher` is supported out of the box — no adapter module needed. This means streams exposed by built-in sources such as `Agent.changes()` (see Agent) and `BroadcastChannel.asPublisher()` (see BroadcastChannel), or any third-party publisher using the JDK interfaces, compose directly with `for await` and `await`:

```
import groovy.concurrent.Agent

def agent = Agent.create(0)
try {
    async {
        3.times { agent.send { it + 1 } }
        agent.shutdown()
    }
    for await (v in agent.changes()) {
        println "value is now $v"
    }
} finally {
    agent.shutdown()
}
```

Timeouts and Delays

Apply a deadline to any task. If it doesn't complete in time, a `TimeoutException` is thrown:

```
def slowPlayer = async { Thread.sleep(5000); 'finally played' }
try {
    await Awaitable.orTimeoutMillis(slowPlayer, 100)
} catch (TimeoutException e) {
    // Player took too long — turn forfeited
    assert true
}
```

Or use a fallback value instead of throwing:

```
def slowPlayer = async { Thread.sleep(5000); 'deliberate move' }
def move = await Awaitable.completeOnTimeoutMillis(slowPlayer, 'auto-pass', 100)
assert move == 'auto-pass'
```

`Awaitable.delay()` pauses without blocking a thread:

```
long start = System.currentTimeMillis()
await Awaitable.delay(100)   // pause without blocking a thread
assert System.currentTimeMillis() - start >= 90
```

Framework Adapters

`await` natively understands `Awaitable`, `CompletableFuture`, `CompletionStage`, and `Future`. `for await` natively understands `java.util.concurrent.Flow.Publisher` in addition to regular `Iterable` sources. Third-party async types can be supported by implementing the `AwaitableAdapter` SPI and registering it via `META-INF/services/groovy.concurrent.AwaitableAdapter`.

Drop-in adapter modules are provided for:

- **`groovy-reactor`** — `await` on `Mono`, `for await` over `Flux`
- **`groovy-rxjava`** — `await` on `Single`/`Maybe`/`Completable`, `for await` over `Observable`/`Flowable`

For example, without the adapter you must manually convert RxJava types:

```
// Without groovy-rxjava — manual conversion
def result = Single.just('hello').toCompletionStage().toCompletableFuture().join()
```

With `groovy-rxjava` on the classpath, the conversion is transparent:

```
// With groovy-rxjava — adapter handles the plumbing
def result = await Awaitable.from(Single.just('hello'))
```

Executor Configuration

By default, `async` uses:

- **JDK 21+**: a virtual-thread-per-task executor
- **JDK 17–20**: a cached daemon thread pool (max 256 threads, configurable via `groovy.async.parallelism` system property)

You can override the executor:

```
import org.apache.groovy.runtime.async.AsyncSupport
import java.util.concurrent.Executors

AsyncSupport.setExecutor(Executors.newFixedThreadPool(4))
AsyncSupport.setExecutor(null)  // restore platform default
// or equivalently:
AsyncSupport.resetExecutor()
```

Integration with JDK Classes

`await` works with any JDK API that returns a `CompletableFuture`, `CompletionStage`, or `Future`. This means you can combine process execution, asynchronous file I/O, and HTTP calls in a single `async` block — all running concurrently:

```
// Task 1: Run a process and await its completion
def proc = async {
    def p = echoCmd.execute() // echo Hello from Groovy
    await p.onExit()
    p.text.trim()
}

// Task 2: Read a file asynchronously
def fileContent = async {
    await etcPassword.textAsync
}

// Task 3: Fetch a web page using Groovy's HttpBuilder
def webContent = async {
    def http = HttpBuilder.http { baseUri bankUrl }
    http.get('/withdraw/100').body
}

// All three tasks run concurrently — collect results
var (echo, file, html) = await proc, fileContent, webContent
assert echo == 'Hello from Groovy'
assert file =~ /sEcrEt/
assert html == '<html>SUCCESS</html>'
```

Task 3 uses Groovy's `HttpBuilder` (from `groovy-http-builder`), which wraps JDK `HttpClient` with a concise DSL. For lower-level control, you can also use `HttpClient` directly:

```
import java.net.http.HttpClient
import java.net.http.HttpRequest
import java.net.http.HttpResponse

def webContent = async {
    def client = HttpClient.newHttpClient()
    def req = HttpRequest.newBuilder()
        .uri(URI.create('https://api.example.com/data'))
        .build()
    def resp = await client.sendAsync(req, HttpResponse.BodyHandlers.ofString())
    resp.body()
}
```

Key JDK classes that work with `await` out of the box:

JDK class	Method returning async result
`Process`	`onExit()` → `CompletableFuture<Process>`
`HttpClient`	`sendAsync(…)` → `CompletableFuture<HttpResponse<T>>`
`Path` (via `groovy-nio`)	`getTextAsync()`, `getBytesAsync()`, `writeAsync(…)` → `CompletableFuture`
`AsynchronousFileChannel`	`read(…)`, `write(…)`, `lock(…)` → `Future<Integer>` / `Future<FileLock>`
`CompletableFuture`	`supplyAsync(…)`, `allOf(…)`, `anyOf(…)` → `CompletableFuture<T>`

Async Closure Factories

Because `async { … }` is just an expression that returns an `Awaitable`, you can wrap it in an ordinary method or closure to create a reusable factory:

```
// A method that returns an async task — a simple factory
def dealCard(List deck) {
    async { deck.shuffled()[0] }
}

def numPlayers = 3
// each player gets a card from their own deck
def cards = (1..numPlayers).collect { dealCard(['A♠', 'K♠', 'Q♠', 'J♠']) }

def hands = await Awaitable.all(*cards)
assert hands.size() == 3
assert hands.every { it in ['A♠', 'K♠', 'Q♠', 'J♠'] }
```

This is a natural way to build reusable async building blocks without any special framework support.

Best Practices

Prefer returning values over mutating shared state

Async closures run on separate threads. Mutating shared variables from multiple closures is a race condition:

```
// UNSAFE — shared mutation without synchronization
var count = 0
def tasks = (1..100).collect { async { count++ } }
tasks.each { await it }
// count may not be 100!
```

Instead, return values and collect results:

```
// SAFE — no shared mutation
def tasks = (1..100).collect { n -> async { n } }
def results = await Awaitable.all(*tasks)
assert results.sum() == 5050
```

When shared mutable state is unavoidable, use the appropriate concurrency-aware type, e.g. `AtomicInteger` for a shared counter, or thread-safe types from `java.util.concurrent` for players concurrently drawing cards from a shared deck.

Choosing the right tool

Feature	Use when…
`async`/`await`	You have sequential steps involving I/O or blocking work and want code that reads top-to-bottom.
`Awaitable.all` / `any` / `first`	You need to launch independent tasks and collect all results, race them, or take the first success.
`yield return` / `for await`	You're producing or consuming a stream of values — paginated APIs, card dealing, log tailing.
`defer`	You acquire resources at different points and want guaranteed cleanup without nested `try`/`finally`.
`AsyncChannel`	Two or more tasks need to communicate — producer/consumer, fan-out/fan-in, or hand-off.
`AsyncScope`	You want child task lifetimes tied to a scope with automatic cancellation on failure.
Framework adapters	You're already using Reactor or RxJava and want `await` / `for await` to work with their types.

Quick Reference

Construct	Description
`async { … }`	Start a closure on a background thread. Returns `Awaitable` (or `Iterable` for generators).
`await expr`	Block until the result is available. Rethrows the original exception.
`await(a, b, c)`	Wait for all — syntactic sugar for `await Awaitable.all(a, b, c)`.
`yield return expr`	Produce a value from an async generator. Consumer blocks until ready.
`for await (x in src)`	Iterate over an async source (generator, channel, Flux, Observable, etc.).
`defer expr`	Schedule a cleanup action (LIFO order) inside an `async` closure.
`AsyncChannel.create(n)`	Create a buffered (or unbuffered) channel for inter-task communication.
`ChannelSelect.after(ms)`	Timer offer: a `ChannelSelect` branch that re-arms each round and commits with the `Instant` it fired.

Construct	Description
`AsyncChannel.after(ms)`	Timer channel: delivers the `Instant` it fired, then closes. A fixed deadline as a `ChannelSelect` branch.
`AsyncScope.withScope { … }`	Structured concurrency — all children complete (or are cancelled) on scope exit.
`Awaitable.orTimeoutMillis`	Apply a deadline. Throws `TimeoutException` if the task exceeds it.
`Awaitable.completeOnTimeoutMillis`	Apply a deadline with a fallback value instead of throwing.
`Awaitable.delay(ms)`	Non-blocking pause.

All keywords (`async`, `await`, `defer`) are contextual — they can still be used as variable or method names in existing code.

Monadic comprehensions (Incubating)

The `groovy-macro-library` module provides `DO`, a comprehension macro that rewrites a sequence of `name in source` generators followed by a body into a chain of bind operations on the **carrier** type. It gives Scala-style for-comprehension / Haskell-style do-notation ergonomics to any type with monadic shape — `Optional`, `Stream`, `CompletableFuture`, Groovy's `Awaitable` and `DataflowVariable`, and user-defined types that opt in.

NOTE `DO` is incubating (since Groovy 6.0) and may change. The macro is compile-time only; the generated code calls the `Comprehensions` runtime support in core, so a program using `DO` needs only the core `groovy` jar at runtime — `groovy-macro-library` is a compile-time dependency.

```
def result = DO(a in Optional.of(2),
                b in Optional.of(3)) {
    Optional.of(a + b)
}
assert result.get() == 5
```

Desugaring

Every generator becomes a bind; the body is the innermost closure body and must itself yield a value of the carrier type (the do-notation rule — there is no implicit lifting in this version). The example above expands to:

```
Comprehensions.bind(Optional.of(2)) { a ->
    Comprehensions.bind(Optional.of(3)) { b ->
        Optional.of(a + b)
    }
}
```

Because the macro expands before type information is available, it does not emit a carrier-specific method name directly. `Comprehensions.bind` resolves the right operation at runtime; under `@CompileStatic` the `groovy.typecheckers.MonadicChecker` extension supplies the static types (see Static type checking).

IMPORTANT	For all but the most trivial uses, `DO` under `@CompileStatic` or `@TypeChecked` requires the `MonadicChecker` extension (`@CompileStatic(extensions = 'groovy.typecheckers.MonadicChecker')`). The runtime dispatcher signature `Comprehensions.bind(Object, Closure):Object` erases the carrier's element type and the comprehension's result type, so without the extension each generator's bound name and the whole `DO` expression both fall through to `Object` — which the static type checker will reject as soon as the body or downstream code does anything type-specific with either. See Static type checking.

A name bound by an earlier generator is in scope in the source expression of every later generator and in the body:

```
def result = DO(a in Optional.of(10),
                b in Optional.of(a * 2)) {   // b's source depends on a
    Optional.of(a + b)
}
assert result.get() == 30
```

Short-circuiting is delivered by the carrier, not by the macro: an empty or failed carrier simply propagates and the body is never evaluated.

```
def result = DO(a in Optional.empty(),
                b in Optional.of(3)) {
    Optional.of(b)            // never reached
}
assert result.isEmpty()
```

Participating carriers

A type participates as a carrier when, in order, the first match winning:

1. it is on the standard allow-list — `java.util.Optional`, `java.util.stream.Stream`, `java.util.concurrent.CompletionStage` (covering `CompletableFuture`), and `groovy.concurrent.Awaitable` (covering `DataflowVariable`);
2. it is *structural* — it has a single-argument `flatMap` (and, for the map role, `map`);
3. it is annotated `@groovy.transform.Monadic`, optionally declaring non-conventional method names.

The allow-list also recognises common Functional Java carriers by name — `fj.data.Option`, `fj.data.List`, `fj.data.Stream`, `fj.data.Validation` and `fj.P1` — using that library's `bind`/`map`

convention. Groovy takes no dependency on Functional Java; the names are matched reflectively, and the generator closure is coerced to `fj.F` automatically. `fj.data.Either` is not directly monadic in Functional Java (bind lives on its `.right()`/`.left()` projections) and is not a carrier; use a projection explicitly.

NOTE	`Awaitable` and `DataflowVariable` bind via `thenCompose`; their `then` method is the *map* operation, not bind. `DO` over `Awaitable` therefore composes asynchronous values without an imperative `await` at each step:

```
import groovy.concurrent.Awaitable
import static org.apache.groovy.runtime.async.AsyncSupport.await

def total = DO(a in Awaitable.of(2),
               b in Awaitable.of(40)) {
    Awaitable.of(a + b)
}
assert await(total) == 42
```

`Stream` yields the usual cartesian composition:

```
import java.util.stream.Stream

def pairs = DO(x in Stream.of(1, 2),
               y in Stream.of('a', 'b')) {
    Stream.of("$x$y".toString())
}
assert pairs.toList() == ['1a', '1b', '2a', '2b']
```

A user type opts in with `@Monadic`, which may name a non-conventional bind and map method:

```
import groovy.transform.Monadic
import java.util.function.Function

@Monadic(bind = 'chain', map = 'transform', unit = 'of')
class Result {
    final Object value
    Result(Object value) { this.value = value }
    static Result of(value) { new Result(value) }     // unit: not used by DO,
declared for law tooling
    Result chain(Function f) { (Result) f.apply(value) }
    Result transform(Function f) { new Result(f.apply(value)) }
    boolean equals(o) { o instanceof Result && value == o.value }   // results compare
by value...
    int hashCode() { value == null ? 0 : value.hashCode() }
}

def r = DO(a in Result.of(3),
```

```
            b in Result.of(4)) {
    Result.of(a * b)
}
assert r == new Result(12)                          // ...so == means "same wrapped
value"
```

The monad laws (left identity, right identity, associativity) are not enforced by the compiler; as with `@Reducer`/`@Associative`, lawful behaviour is the participating type's responsibility. Because a comprehension composes monadic *values*, a data-style carrier should provide structural `equals`/`hashCode` (Groovy's `==` dispatches to `equals`) — otherwise results compare by identity and laws such as right identity and functor identity silently fail to hold. This does not apply to effectful carriers like `Stream`, `CompletableFuture` and `Awaitable`, whose laws hold only up to observational equivalence. A carrier may optionally name its unit (`of`/`pure`) factory via `@Monadic(unit = '…')`; `DO` does not use it — the body lifts explicitly — but it lets law-deriving tooling synthesize the unit-dependent laws (left and right identity).

Static type checking

Activate the `MonadicChecker` type-checking extension to use `DO` under `@CompileStatic` or `@TypeChecked`. It types each generator's bound name as the carrier's element type (so the body type-checks), restores the comprehension's result type, and rejects a non-participating carrier with a compile error naming the type and the missing shape:

```
import groovy.transform.CompileStatic

@CompileStatic(extensions = 'groovy.typecheckers.MonadicChecker')
class Calc {
    static int sum() {
        DO(a in Optional.of(2),
           b in Optional.of(3)) {
            Optional.of(a + b)
        }.get()
    }
}
assert Calc.sum() == 5
```

TIP The typical symptom of forgetting the extension is a static type checking error reporting that some operation cannot be found on `Object` — either inside the body (a generator's bound name has erased to `Object`) or on the `DO` expression itself (the result has erased to `Object`). Adding `extensions = 'groovy.typecheckers.MonadicChecker'` to the `@CompileStatic`/`@TypeChecked` annotation resolves it.

Lint for hand-written chains: MonadicShapeChecker

If you write `flatMap` / `map` chains by hand (without `DO`) over the same participating carriers, the sister `MonadicShapeChecker` extension provides a compile-time lint:

```
@TypeChecked(extensions = 'groovy.typecheckers.MonadicShapeChecker')
```

It catches the two classic shape mistakes — a `bind` (i.e. `flatMap`) closure that returns a non-carrier value, and a `map` closure that returns the same carrier where `flatMap` was meant. A `strict` option tightens what is flagged; the default lenient mode reports only high-confidence violations. Both checkers are documented from the type-checker side in the Built-in auxiliary type checkers chapter.

When to use `DO`

`DO` is a value-composition notation. It complements, rather than competes with, the concurrency constructs: use imperative `async`/`await` when code reads as a sequence of dependent steps you intend to run now (see Prefer returning values over mutating shared state and Choosing the right tool); reach for `DO` when the composed value is the deliverable — an `Awaitable` to combine further, an `Optional`/validation result, a parser result, or a custom `Result` type — and you want one uniform notation across carriers.

CAUTION	This version is deliberately narrow. The body must yield a carrier value (no implicit `pure`/`unit` lifting); there are no `if` guard clauses; and each `DO` works over a single carrier (nest `DO` blocks for more). `break`/`continue` are not valid in the body, and `return` follows the standard closure rule. These are the same closure-body constraints the `@Parallel` for-loop transform documents.

Parallel Collections (Incubating)

Introduction

Groovy adds parallel collection methods directly onto `Collection`, enabling data parallelism with familiar Groovy idioms. These methods use Java parallel streams under the hood, with pool isolation via `ForkJoinPool` to prevent interference with the common pool.

All methods use `java.util.function` types (`Consumer`, `Function`, `Predicate`, etc.) and Groovy closures are automatically converted via SAM coercion.

Getting Started

Parallel methods are available on any `Collection` without imports:

```
def result = (1..1000).toList().collectParallel { it * 2 }
assert result.sort() == (2..2000).step(2).toList()
```

By default, operations run on `ForkJoinPool.commonPool()`. For pool isolation, use `ParallelScope.withPool`:

```
import groovy.concurrent.ParallelScope
import groovy.concurrent.Pool
```

```
ParallelScope.withPool(Pool.cpu()) { scope ->
    def bigList = (1..100_000).toList()
    def sum = bigList.injectParallel(0) { a, b -> a + b }
    println "Sum: $sum"
}
```

Pool and ParallelScope

Pool

`Pool` is a managed thread pool extending `Executor` and `AutoCloseable`:

```
import groovy.concurrent.Pool

def pool = Pool.cpu()       // Sized to available processors (ForkJoinPool)
def pool2 = Pool.fixed(8)   // Fixed-size ForkJoinPool
def pool3 = Pool.io()       // Virtual threads on JDK 21+, sized pool otherwise
def pool4 = Pool.virtual()  // Virtual-thread-per-task (JDK 21+)
```

`Pool.cpu()` and `Pool.fixed()` create `ForkJoinPool`-backed pools, ideal for CPU-bound parallel work. `Pool.io()` and `Pool.virtual()` create virtual thread pools for I/O-bound tasks.

ParallelScope

`ParallelScope.withPool` binds a pool for the duration of a block. Parallel collection methods inside the block automatically use it:

```
import groovy.concurrent.ParallelScope
import groovy.concurrent.Pool

ParallelScope.withPool(4) { scope ->
    // All *Parallel methods use a 4-thread ForkJoinPool
    def results = bigList.collectParallel { transform(it) }
    def filtered = results.findAllParallel { it > threshold }
    filtered.eachParallel { process(it) }
}
// Pool is shut down automatically
```

ConcurrentConfig

Global defaults are managed by `ConcurrentConfig`:

```
import groovy.concurrent.ConcurrentConfig

// Default parallelism: processors + 1 (GPars convention)
println ConcurrentConfig.defaultParallelism
```

```
// Override via system property: -Dgroovy.concurrent.poolsize=16
// Or programmatically:
ConcurrentConfig.defaultParallelism = 16
```

Parallel Collection Methods

Transformation

Method	Description
`collectParallel { }`	Transforms each element (parallel map)
`collectManyParallel { }`	Transforms and flattens (parallel flatMap)

```
def names = people.collectParallel { it.name }
def allTags = articles.collectManyParallel { it.tags }
```

Filtering

Method	Description
`findAllParallel { }`	Elements matching predicate
`findParallel { }`	First element matching predicate
`findAnyParallel { }`	Any element matching (may be faster, no order guarantee)
`grepParallel(filter)`	Pattern match via `isCase` (Class, regex, Range, etc.)
`splitParallel { }`	Partition into [matching, non-matching]

```
def adults = people.findAllParallel { it.age >= 18 }
def firstMatch = data.findParallel { it.isValid() }
def strings = mixed.grepParallel(String)
def (evens, odds) = numbers.splitParallel { it % 2 == 0 }
```

Iteration

Method	Description
`eachParallel { }`	Side-effect iteration
`eachWithIndexParallel { item, idx -> }`	Iteration with index

```
files.eachParallel { process(it) }
items.eachWithIndexParallel { item, idx -> println "$idx: $item" }
```

Predicates

Method	Description
`anyParallel { }`	True if any element matches
`everyParallel { }`	True if all elements match
`countParallel { }`	Count of matching elements

```
if (orders.anyParallel { it.isPriority() }) { ... }
assert positiveNumbers.everyParallel { it > 0 }
def errorCount = logs.countParallel { it.level == 'ERROR' }
```

Aggregation

Method	Description
`sumParallel { a, b → }`	Reduce with binary operator
`injectParallel(seed) { a, b → }`	Reduce with seed (accumulator must be associative)
`minParallel { a, b → }`	Minimum by comparator
`maxParallel { a, b → }`	Maximum by comparator
`groupByParallel { }`	Group into map by classifier

```
def total = amounts.sumParallel { a, b -> a + b }
def product = numbers.injectParallel(1) { a, b -> a * b }
def youngest = people.minParallel { a, b -> a.age <=> b.age }
def byDept = employees.groupByParallel { it.department }
```

NOTE `injectParallel` requires an associative accumulator for correct parallel results. Non-associative accumulators produce undefined results.

@Parallel for loops

The `@Parallel` annotation converts a `for-in` loop into a parallel operation with structured completion — all iterations finish before the next statement executes:

```
import groovy.transform.Parallel
import java.util.concurrent.CopyOnWriteArrayList

def results = new CopyOnWriteArrayList()

@Parallel
for (item in [1, 2, 3, 4, 5]) {
    results << item * 10
}
```

```
// All iterations complete before this line
assert results.sort() == [10, 20, 30, 40, 50]
```

Inside a `ParallelScope.withPool` block, `@Parallel` uses the bound pool:

```
ParallelScope.withPool(Pool.cpu()) { scope ->
    @Parallel
    for (n in data) {
        process(n)
    }
}
```

Under the hood, `@Parallel` rewrites the for loop body into a closure passed to `eachParallel`. This means the loop body is no longer a traditional loop — it is a closure invocation. The loop variable is also internally renamed (e.g., `item` becomes `$parallel_item`) to avoid scope conflicts in the generated code.

Because the body becomes a closure:

- `break` and `continue` are **not supported** — they are not valid inside closures and will cause a compile error.
- `return` inside the body returns from the closure (skipping the current iteration), not from the enclosing method.

NOTE Shared mutable state inside `@Parallel` loops must be thread-safe (e.g., `AtomicInteger`, `CopyOnWriteArrayList`, `ConcurrentHashMap`).

TIP If you need the best debugging experience (e.g., stepping through loop bodies or inspecting variable names), consider using the `*Parallel` collection methods directly (such as `eachParallel`, `collectParallel`) instead of `@Parallel`, or be aware of the previously mentioned internal variable renaming.

Best Practices

Choosing between parallel collections and async/await

Parallel collections and `@Parallel` run on a `ForkJoinPool` with a small number of worker threads (typically matching your CPU core count). This is ideal for **CPU-bound** work — computation, transformation, aggregation — where each task keeps the CPU busy.

For **I/O-bound** or **latency-bound** work (network calls, database queries, sleeps, delays), use `async`/`await` with virtual threads instead. Virtual threads can scale to millions of concurrent tasks because blocking one doesn't consume an OS thread.

Workload	Right tool	Why
CPU-bound (computation, parsing, math)	`@Parallel`, `collectParallel`, etc.	ForkJoinPool work-stealing maximises CPU utilisation
I/O-bound (HTTP calls, DB queries, file I/O)	`async`/`await` with virtual threads	Millions of concurrent waits without blocking OS threads
Mixed	`async`/`await` with `Pool.io()`	Virtual threads handle both compute and I/O
Value composition over a monadic carrier (`Optional`, `Awaitable`, `Stream`, …)	`DO` comprehensions (`groovy-macro-library`)	Flattens nested `flatMap`/`thenCompose` chains into uniform `name in source` notation

To illustrate, consider processing 100 items where each involves a short delay (simulating I/O or latency):

```
// CPU-bound work — @Parallel is the right tool
@Parallel
for (item in bigList) {
    results << expensiveComputation(item)  // keeps CPU busy, no blocking
}
```

```
// I/O-bound work — async/await is the right tool
AsyncScope.withScope { scope ->
    urls.each { url ->
        scope.async { fetchUrl(url) }  // virtual thread blocks on I/O (free)
    }
}
// All fetches complete here — structured concurrency
```

```
// Anti-pattern — blocking inside parallel collections
@Parallel
for (item in 1..100) {
    Thread.sleep(50)    // blocks a ForkJoinPool worker!
    process(item)
}
// Only ~8 items run concurrently (pool size), rest queue behind.
// Total time: ~625ms instead of ~50ms with virtual threads.
```

The anti-pattern looks parallel but the `ForkJoinPool` has only ~8 workers. Each `Thread.sleep` ties up a worker, so at most ~8 items sleep concurrently. The remaining 92 wait in the queue.

NOTE `await Awaitable.delay(ms)` inside a parallel collection body has the same problem — `await` blocks the calling ForkJoinPool worker thread while waiting for the delay to complete. Use `async`/`await` outside parallel collections for delay-based work.

Use thread-safe collections for shared state

The parallel body runs on multiple threads simultaneously. Use concurrent collections or atomics for any shared mutable state:

```
// Bad — ArrayList is not thread-safe
def results = []
items.eachParallel { results << transform(it) }  // race condition!

// Good — use CopyOnWriteArrayList or collectParallel
def results = items.collectParallel { transform(it) }
```

Concurrent Dataflow (Incubating)

Introduction

Dataflow programming lets you express dependencies between concurrent computations declaratively. Values are bound once and read many times — the runtime handles ordering automatically. You never need to think about which thread runs first.

Groovy's dataflow support integrates natively with async/await and the `Awaitable` API.

DataflowVariable

A `DataflowVariable` is a single-assignment variable. It starts unbound, can be written to exactly once, and any thread that reads it before binding will block until a value is available.

```
import groovy.concurrent.DataflowVariable

def x = new DataflowVariable()
def y = new DataflowVariable()
def z = new DataflowVariable()

async {
    z << await(x) + await(y)    // blocks until x and y are bound
}

async { x << 10 }              // bind x — order doesn't matter
async { y << 5 }               // bind y

println "Result: ${await(z)}"  // 15
```

Key properties:

- **Write-once**: calling `bind()` or `<<` a second time throws `IllegalStateException`
- **Implements `Awaitable`**: works with `await`, combinators (`Awaitable.all`), and `then` chaining
- **Thread-safe**: safe to bind from one thread and read from any number of others

Error binding

```
def v = new DataflowVariable()
v.bindError(new RuntimeException('computation failed'))
// Any thread that awaits v will receive the exception
```

Chaining with `then`

```
def v = new DataflowVariable()
def doubled = v.then { it * 2 }

async { v << 21 }
assert await(doubled) == 42
```

Combining multiple variables

```
def a = new DataflowVariable()
def b = new DataflowVariable()
def c = new DataflowVariable()

async { a << 1 }
async { b << 2 }
async { c << 3 }

def results = await(a, b, c)
assert results == [1, 2, 3]
```

Dataflows

`Dataflows` is a convenience class that auto-creates `DataflowVariable` instances on demand. Property reads block until bound; property writes bind the variable.

```
import groovy.concurrent.Dataflows

def df = new Dataflows()

async { df.z = df.x + df.y }   // blocks on x and y, binds z
async { df.x = 10 }
async { df.y = 5 }

println "Result: ${df.z}"      // 15
```

This is the most concise way to write dataflow programs in Groovy. Variables are created lazily — no declarations needed.

```
def df = new Dataflows()

async { df.fullName = "${df.first} ${df.last}" }
async { df.first = 'John' }
async { df.last = 'Doe' }

assert df.fullName == 'John Doe'
```

Accessing the underlying variable

Use `getVariable()` when you need the `DataflowVariable` itself (e.g., to pass to `Awaitable.all` or check `isBound`):

```
def df = new Dataflows()
def v = df.getVariable('x')   // DataflowVariable, does not block
assert !v.isBound()
async { df.x = 42 }
assert await(v) == 42
```

Actors and Agents (Incubating)

Introduction

Actors and agents provide thread-safe concurrent state management through message passing. Instead of protecting shared state with locks, you send messages to an actor or agent which processes them one at a time on a dedicated thread. This eliminates data races by design.

Groovy provides two levels of abstraction:

- **Agent** — a thread-safe mutable value updated via functions
- **Actor** — a message-processing entity with flexible dispatch

Both use virtual threads on JDK 21+ for efficient scaling.

Agent

An `Agent` wraps a value that can be read by any thread but modified only through serialized update functions:

```
import groovy.concurrent.Agent

def counter = Agent.create(0)
counter.send { it + 1 }
counter.send { it + 1 }
counter.send { it + 1 }
```

```
assert await(counter.getAsync()) == 3
```

Reading

```
// Snapshot read — non-blocking, returns current value
def current = counter.get()

// Consistent read — waits for pending updates to complete
def consistent = await(counter.getAsync())
```

Updating

```
// Fire-and-forget update
counter.send { it + 1 }

// Update and get new value
def newValue = await(counter.sendAndGet { it * 2 })
```

Complex state

Agents work with any value type:

```
def inventory = Agent.create([:])

inventory.send { state -> state + [apples: 10] }
inventory.send { state -> state + [bananas: 5] }
inventory.send { state ->
    state.collectEntries { k, v -> [k, v * 2] }
}

assert await(inventory.getAsync()) == [apples: 20, bananas: 10]
```

Observing state changes

An agent exposes a `Flow.Publisher<T>` of state transitions via `changes()`. Each successful update emits the new value to every subscriber that is already subscribed at the time of the update. The stream is hot (no replay of prior state), per-subscriber buffered, and closes with `onComplete` when `shutdown()` is called:

```
def agent = Agent.create(0)
try {
    async {
        3.times { agent.send { it + 1 } }
        Thread.sleep(50)
        agent.shutdown()
    }
```

```
    def seen = []
    for await (v in agent.changes()) {
        seen << v
    }
    assert seen == [1, 2, 3]
} finally {
    agent.shutdown()
}
```

Slow subscribers drop newly-offered values rather than backpressuring the agent's update loop — buffered values are still delivered in order, but the most recent update may be skipped if a subscriber's buffer is full (default size 256). If `changes()` is first called after `shutdown()`, the returned publisher is already closed and subscribers receive `onComplete` immediately.

Actor

An `Actor` processes messages from a queue on a dedicated thread. Two factory methods cover the common patterns:

Reactor (stateless)

A reactor applies a function to each message. The return value becomes the reply for `sendAndGet` callers:

```
import groovy.concurrent.Actor

def doubler = Actor.reactor { it * 2 }
assert await(doubler.sendAndGet(5)) == 10
assert await(doubler.sendAndGet(21)) == 42
doubler.stop()
```

Reactors are ideal for pure-function message processing — validators, transformers, calculators:

```
def validator = Actor.reactor { msg ->
    if (msg instanceof String && msg.length() > 0) 'valid'
    else 'invalid'
}
```

Stateful

A stateful actor maintains state across messages. The handler receives `(state, message)` and returns the new state:

```
def counter = Actor.stateful(0) { state, msg ->
    switch (msg) {
        case 'increment': return state + 1
        case 'decrement': return state - 1
```

```
        case 'reset':     return 0
        default:          return state
    }
}

counter.send('increment')
counter.send('increment')
counter.send('decrement')
assert await(counter.sendAndGet('increment')) == 2
counter.stop()
```

For `sendAndGet`, the new state is the reply. This makes it easy to query the current state:

```
// Send a no-op message to read the state
def currentState = await(counter.sendAndGet('query'))
```

Typed message dispatch

Use pattern matching for rich message protocols:

```
def account = Actor.stateful(0.0) { balance, msg ->
    switch (msg) {
        case { it instanceof Map && it.deposit }:
            return balance + msg.deposit
        case { it instanceof Map && it.withdraw }:
            if (msg.withdraw > balance)
                throw new RuntimeException('Insufficient funds')
            return balance - msg.withdraw
        default:
            return balance
    }
}

account.send([deposit: 100])
account.send([withdraw: 30])
def balance = await(account.sendAndGet([deposit: 0]))
assert balance == 70.0
account.stop()
```

Construction options

Actors take an optional `ActorOptions` configuring the mailbox, the executor, and a handful of opt-in behaviours. The builder is value-based; each `with*` method returns a new configuration.

```
import groovy.concurrent.Actor
import groovy.concurrent.ActorOptions
```

```
def options = ActorOptions.DEFAULTS
    .withBoundedMailbox(1000, ActorOptions.Overflow.BLOCK)

def actor = Actor.reactor(handler, options)
```

Bounded mailbox

By default the mailbox is unbounded. For backpressure — or to cap memory when producers can outrun the actor — configure a capacity and an overflow policy:

```
import groovy.concurrent.ActorOptions.Overflow

// BLOCK — sender blocks until capacity is free
ActorOptions.DEFAULTS.withBoundedMailbox(100, Overflow.BLOCK)

// FAIL — send throws IllegalStateException when the mailbox is full
ActorOptions.DEFAULTS.withBoundedMailbox(100, Overflow.FAIL)

// DROP_NEWEST — the incoming message is silently dropped; for
// sendAndGet, the returned Awaitable binds to IllegalStateException
ActorOptions.DEFAULTS.withBoundedMailbox(100, Overflow.DROP_NEWEST)
```

A handler that calls `ctx.self().send(…)` on a full `BLOCK` mailbox would deadlock (the handler is the actor's only consumer). The actor detects this and fails fast with `IllegalStateException` rather than parking the worker thread.

Per-actor executor

Actors default to a shared async executor (virtual threads on JDK 21+). For workload isolation, hand the actor its own executor:

```
import java.util.concurrent.Executors

def pool = Executors.newSingleThreadExecutor()
def actor = Actor.reactor(handler,
    ActorOptions.DEFAULTS.withExecutor(pool))
```

Other options — `withStashBound` and `withCurrentSelf` — are covered in the FSM section where they're directly used.

Lifecycle

Both actors and agents support lifecycle management. An actor has a three-state lifecycle:

`isActive()`	`isTerminated()`	**Meaning**
`true`	`false`	Accepting new sends and processing them.

isActive()	isTerminated()	Meaning
false	false	Draining — stop() was called, no new sends are accepted, but already-queued messages are still being processed.
false	true	Terminated — the worker thread has exited; all queued messages are processed and any stashed sendAndGet replies have been rejected.

```
def actor = Actor.reactor { it }
assert actor.isActive()
assert !actor.isTerminated()

actor.stop()                                    // flips isActive() to false immediately
assert !actor.isActive()                         // new sends fail from here (a send
racing with stop() may still land)

// The worker may still be draining queued messages. Poll for terminated
// when you actually need to be sure the actor has finished shutting down.
while (!actor.isTerminated()) Thread.sleep(10)
assert actor.isTerminated()
```

Actors implement AutoCloseable, so they work with try-with-resources (Groovy or Java):

```
Actor.reactor { it * 2 }.withCloseable { actor ->
    assert await(actor.sendAndGet(5)) == 10
}
// actor is stopped
```

Error Handling

Exceptions thrown by a handler are delivered to sendAndGet callers through the awaited reply:

```
def risky = Actor.reactor { throw new RuntimeException('oops') }
try {
    await(risky.sendAndGet('anything'))
} catch (RuntimeException e) {
    assert e.message == 'oops'
}
risky.stop()
```

Fire-and-forget send calls have no reply to carry the exception. To observe handler failures from those messages — for logging, metrics, or shutting the actor down — register an onError callback:

```
def actor = Actor.reactor { msg -> process(msg) }
    .onError { Throwable t, msg ->
        log.warn("handler failed on {}: {}", msg, t.message)
```

```
    }
```

The context-aware overload receives an `ActorContext` first, useful when the callback needs to react beyond logging — for example, stopping the actor:

```
actor.onError { ctx, t, msg -> ctx.self().stop() }
```

Exceptions thrown from inside the `onError` callback itself are caught and discarded so they cannot destabilise the dispatch loop.

FSM-style actors

A stateful actor models a single behaviour. For multi-phase protocols — where the actor responds differently depending on which "state" it is in — three additional capabilities on `ActorContext` cover most of what you'd reach for from a dedicated FSM library:

- `ctx.become(newHandler)` — swap the active handler at runtime; state is preserved across the swap.
- `ctx.stash()` / `ctx.unstashAll()` — defer the current message for later replay (typically because it arrived in the wrong phase).
- `ctx.scheduleOnce(msg, delay)` / `ctx.scheduleAtFixedRate(msg, initialDelay, interval)` — self-send a message after a delay, with a cancellable handle.

These capabilities are only available via the context-aware factory overloads, which give the handler an `ActorContext` parameter.

Swapping behaviour with become

```
import groovy.concurrent.Actor
import groovy.concurrent.ActorContext
import groovy.concurrent.StatefulHandler

// Phase A increments by 1; phase B increments by 10.
StatefulHandler phaseB = { ctx, s, m -> s + 10 } as StatefulHandler

def actor = Actor.stateful(0, { ActorContext ctx, int s, m ->
    if (m == 'swap') { ctx.become(phaseB); return s }
    s + 1
} as StatefulHandler)

assert await(actor.sendAndGet('inc'))  == 1
assert await(actor.sendAndGet('swap')) == 1    // state preserved
assert await(actor.sendAndGet('inc'))  == 11   // now phase B
```

The swap takes effect on the next message: the current handler invocation completes normally (including binding any reply). State is preserved verbatim; the new handler receives the same state

value the current one would have returned.

Deferring messages with stash

When a message arrives while the actor is in the wrong phase, defer it with `ctx.stash()`. When the phase transitions to one that can handle it, call `ctx.unstashAll()` to replay all deferred messages in FIFO order at the head of the mailbox:

```
StatefulHandler connected = { ctx, s, m -> "got($m)" } as StatefulHandler

def actor = Actor.stateful('init', { ActorContext ctx, s, m ->
    if (m == 'ready') {
        ctx.become(connected)
        ctx.unstashAll()      // replay anything stashed while not ready
        return s
    }
    ctx.stash()               // any other message is deferred for now
    s
} as StatefulHandler)

def r1 = actor.sendAndGet('A')    // stashed
def r2 = actor.sendAndGet('B')    // stashed
actor.send('ready')               // → connected, then replays A and B
assert await(r1) == 'got(A)'
assert await(r2) == 'got(B)'
```

The stash is unbounded by default.

WARNING	An actor that stashes messages from a source whose volume you do not control (network input, external clients, untrusted callers) can grow the stash without limit and exhaust the JVM heap if the phase transition that would call `unstashAll()` never arrives. For any such actor, set a bound via `withStashBound` at construction time.

```
def options = ActorOptions.DEFAULTS
    .withStashBound(64, ActorOptions.StashOverflow.REJECT)
```

Policies are `FAIL` (throw from `stash()`), `DROP_OLDEST` (evict the oldest stashed message, binding its `sendAndGet` reply to error), and `REJECT` (bind the current message's reply to error without stashing it).

Scheduled self-sends

For state timeouts, heartbeats, retries, and periodic work, the context exposes one-shot and recurring schedules over the shared async scheduler:

```
import java.time.Duration
```

```
def actor = Actor.stateful(0, { ActorContext ctx, s, m ->
    if (m == 'start') {
        ctx.scheduleAtFixedRate('tick',
            Duration.ofSeconds(1), Duration.ofSeconds(5))
        return s
    }
    if (m == 'tick') return s + 1
    s
} as StatefulHandler)
```

`scheduleOnce` returns a `Cancellable` so a phase transition can call off a pending timer:

```
def timer = ctx.scheduleOnce('auth-timeout', Duration.ofSeconds(30))
// ... later, when AuthResult arrives:
timer.cancel()
```

`Actor.stop()` cancels every outstanding timer the actor scheduled — no zombies after shutdown.

End-to-end: a connection handshake

A common FSM shape — `disconnected → authenticating → connected` — combining all three capabilities:

```
import groovy.concurrent.Actor
import groovy.concurrent.ActorContext
import groovy.concurrent.StatefulHandler

StatefulHandler disconnected, authenticating, connected

disconnected = { ctx, s, m ->
    if (m == 'connect') { ctx.become(authenticating); return s }
    s
} as StatefulHandler

authenticating = { ctx, s, m ->
    if (m == 'auth-ok') {
        ctx.become(connected)
        ctx.unstashAll()
        return s
    }
    ctx.stash()                // commands arriving during auth are deferred
    s
} as StatefulHandler

connected = { ctx, s, m ->
    // Real connected handlers use s — here we bump an "ops processed"
    // counter to demonstrate that state survives every transition.
    [ops: s.ops + 1]
```

```
} as StatefulHandler

def actor = Actor.stateful([ops: 0], disconnected)
actor.send('connect')
def r1 = actor.sendAndGet([cmd: 'read'])    // stashed in authenticating
actor.send('auth-ok')                       // → connected; r1 now processes
assert await(r1) == [ops: 1]                // state preserved across phases
actor.stop()
```

Choosing Between Agent and Actor

Aspect	Agent	Actor
State	Single value, updated via function	Arbitrary, managed by handler
Messages	Update functions only	Any message type with dispatch
Reply	`getAsync()` / `sendAndGet(fn)` returns new value	`sendAndGet(msg)` returns handler result
Multi-phase behaviour	—	`become` / `stash` / scheduled self-sends
Use case	Thread-safe counters, caches, accumulators	State machines, services, typed protocols

Both guarantee sequential message processing — no locks needed.

@ActiveObject / @ActiveMethod

For a more OOP approach, annotate a class with `@ActiveObject` and its methods with `@ActiveMethod`. The compiler automatically routes annotated method calls through an internal actor — callers just see a normal class:

```
import groovy.transform.ActiveObject
import groovy.transform.ActiveMethod

@ActiveObject
class Account {
    private double balance = 0

    @ActiveMethod
    void deposit(double amount) { balance += amount }

    @ActiveMethod
    void withdraw(double amount) {
        if (amount > balance) throw new RuntimeException('Insufficient funds')
        balance -= amount
    }
```

```
    @ActiveMethod
    double getBalance() { balance }
}

def account = new Account()
account.deposit(100)         // async, serialized, blocks until done
account.deposit(50)
account.withdraw(30)
assert account.getBalance() == 120.0
```

Methods without `@ActiveMethod` run on the caller's thread as normal.

Blocking vs non-blocking

By default, `@ActiveMethod` calls block until the actor processes them. For non-blocking calls, set `blocking = false`:

```
@ActiveObject
class Service {
    @ActiveMethod(blocking = false)
    def compute(int x) { x * x }
}

def svc = new Service()
def result = svc.compute(7)  // returns Awaitable immediately
assert await(result) == 49
```

Thread safety by annotation

The key benefit: you write normal-looking classes with normal methods. Thread safety is guaranteed by the annotation — all `@ActiveMethod` calls are serialized through the actor. No locks, no concurrent collections, no race conditions.

This also makes the code highly readable for AI tools — the `@ActiveObject` annotation explicitly declares the concurrency model, and each `@ActiveMethod` contains pure business logic without messaging boilerplate.

Groovy Concurrent API for Java (Incubating)

Introduction

The `groovy-concurrent-java` module provides Groovy's concurrent and parallel API as a standalone Java library — no Groovy runtime required.

Java developers get access to:

- **Structured concurrency** — `AsyncScope`, `Awaitable`
- **Pools** — `Pool`, `ParallelScope`, `ConcurrentConfig`

- **Actors** — `Actor.reactor()`, `Actor.stateful()`
- **Agents** — `Agent` (thread-safe mutable state)
- **Dataflow** — `DataflowVariable`, `AsyncChannel`, `BroadcastChannel`, `ChannelSelect`

This module is published as `org.apache.groovy:groovy-concurrent-java` and contains a filtered subset of the classes from Groovy core. It must not be used alongside the full Groovy runtime — Gradle enforces mutual exclusion via a shared capability, and a runtime warning is logged if both jars are detected on the classpath.

Setup

Gradle

```
dependencies {
    implementation 'org.apache.groovy:groovy-concurrent-java:6.0.0'
}
```

Maven

```
<dependency>
    <groupId>org.apache.groovy</groupId>
    <artifactId>groovy-concurrent-java</artifactId>
    <version>6.0.0</version>
</dependency>
```

IMPORTANT	Do not add `groovy-concurrent-java` if you already depend on `groovy` (the full runtime). The concurrent classes are included in core.

Structured Concurrency

```
import groovy.concurrent.AsyncScope;
import groovy.concurrent.Pool;
import org.apache.groovy.runtime.async.AsyncSupport;

// Basic scope — waits for all children before returning
var result = AsyncScope.withScope(scope -> {
    var a = scope.async(() -> fetchUser(id));
    var b = scope.async(() -> fetchOrders(id));
    return Map.of(
        "user", AsyncSupport.await(a),
        "orders", AsyncSupport.await(b)
    );
});

// With a pool for executor isolation
try (var pool = Pool.cpu()) {
```

```
    AsyncScope.withScope(pool, scope -> {
        var task = scope.async(() -> computeResult());
        return AsyncSupport.await(task);
    });
}
```

Actors

```
import groovy.concurrent.Actor;
import org.apache.groovy.runtime.async.AsyncSupport;

// Reactor — stateless, each message produces a reply
var doubler = Actor.<Integer, Integer>reactor(n -> n * 2);
int result = AsyncSupport.await(doubler.sendAndGet(21)); // 42
doubler.stop();

// Stateful — maintains state across messages
var counter = Actor.<String, Integer>stateful(0, (state, msg) -> {
    if ("increment".equals(msg)) return state + 1;
    return state;
});
counter.send("increment");
counter.send("increment");
int count = AsyncSupport.await(counter.sendAndGet("increment")); // 3
counter.stop();
```

Agents

```
import groovy.concurrent.Agent;
import org.apache.groovy.runtime.async.AsyncSupport;

var counter = Agent.create(0);
counter.send(n -> n + 1);
counter.send(n -> n + 1);
counter.send(n -> n + 1);

int result = AsyncSupport.await(counter.getAsync()); // 3
counter.shutdown();
```

Dataflow Variables

```
import groovy.concurrent.Awaitable;
import groovy.concurrent.DataflowVariable;
import org.apache.groovy.runtime.async.AsyncSupport;

var x = new DataflowVariable<Integer>();
```

```
var y = new DataflowVariable<Integer>();

// z depends on x and y — blocks until both are bound
var z = Awaitable.<Integer>go(() ->
    AsyncSupport.await(x) + AsyncSupport.await(y)
);

// Bind in any order
AsyncSupport.getExecutor().execute(() -> x.bind(10));
AsyncSupport.getExecutor().execute(() -> y.bind(5));

int result = AsyncSupport.await(z); // 15
```

Pools

```
import groovy.concurrent.ParallelScope;
import groovy.concurrent.Pool;
import org.apache.groovy.runtime.async.AsyncSupport;

var result = ParallelScope.withPool(4, scope -> {
    var tasks = new java.util.ArrayList<groovy.concurrent.Awaitable<Integer>>();
    for (int i = 1; i <= 4; i++) {
        int n = i;
        tasks.add(scope.async(() -> n * 10));
    }
    int sum = 0;
    for (var task : tasks) {
        sum += (int) AsyncSupport.await(task);
    }
    return sum;
}); // 100
```

Features not available in the Java module

The following features require the Groovy runtime and are only available when using the full `groovy` dependency:

Feature	Reason
`Dataflows` class	Uses Groovy's `propertyMissing` for dynamic property access
`async { }` / `await` keywords	Groovy language syntax, transformed by the Groovy compiler
`defer { }` / `yield return`	Groovy language syntax
`for await (x in channel)`	Groovy compiler-generated iteration
`@ActiveObject` / `@ActiveMethod`	Groovy AST transformation
Parallel collection methods (`collectParallel`, etc.)	Registered as Groovy extension methods on `Collection`

Feature	Reason
`AsyncClosureUtils` (`wrapAsync`, `wrapAsyncGenerator`)	Return `Closure` subclasses
Closure-based APIs (e.g., `scope.async { }`)	Groovy SAM coercion from Closure to `Supplier`/`Function`

Java users have full access to the underlying APIs using `java.util.function` types (`Supplier`, `Function`, `Consumer`, `Predicate`, etc.) and explicit `AsyncSupport.await()` calls instead of the `await` keyword.

Tools

Running Groovy from the commandline

groovy, the Groovy command

`groovy` invokes the Groovy command line processor. It allows you to run inline Groovy expressions, and scripts, tests or application within groovy files. It plays a similar role to `java` in the Java world but handles inline scripts and rather than invoking class files, it is normally called with scripts and will automatically call the Groovy compiler as needed.

The easiest way to run a Groovy script, test or application is to run the following command at your shell prompt:

```
> groovy MyScript.groovy
```

The `.groovy` part is optional. The `groovy` command supports a number of command line switches:

Short version	Long version	Description	Example
-a	--autosplit <splitPattern>	split lines using splitPattern (default '\s') using implicit 'split' variable	
-b	--basescript <class>	Base class name for scripts (must derive from Script)	
-c	--encoding <charset>	specify the encoding of the files	
-cp <path>	-classpath <path> --classpath <path>	Specify the compilation classpath. Must be the first argument.	groovy -cp lib/dep.jar MyScript
	--configscript <path>	Advanced compiler configuration script	groovy --configscript config/config.groovy src/Person.groovy
-D	--define <name=value>	define a system property	
-d	--debug	debug mode will print out full stack traces	

Short version	Long version	Description	Example
	--disableopt <optlist>	disables one or all optimization elements. optlist can be a comma separated list with the elements: all (disables all optimizations), int (disable any int based optimizations)	
-e <script>		specify an inline command line script	groovy -e "println new Date()"
-h	--help	Displays usage information for the command line groovy command	groovy --help
-i <extension>		modify files in place; create backup if extension is given (e.g. '.bak')	
-l <[host:]port>		listen on a port and process inbound lines (default: 1960); binds the loopback address unless a host is given	groovy -l 0.0.0.0:1960 -e "line.reverse()"
-n		process files line by line using implicit 'line' variable	
-p		process files line by line and print result (see also -n)	
-v	--version	display the Groovy and JVM versions	groovy -v
-pa	--parameters	Generates metadata for reflection on method parameter names on JDK 8 and above. Defaults to false.	groovy --parameters Person.groovy
-pr	--enable-preview	Enable preview Java features (jdk12+ only).	groovy --enable -preview Person.groovy

NOTE When no classpath is specified, `groovy` uses the current directory, the same as the `java` interpreter. When you supply a classpath with `-cp`/`-classpath` or the `CLASSPATH`

> environment variable, it is used as-is and the current directory is not added for you; add `.` to your classpath explicitly if you also need it. This applies to the command-line tools generally (`groovyc`, `groovysh`, and so on).

The `-l` option is a developer convenience for experimenting with a line-driven script, in the same spirit as `groovysh` but reading from a socket instead of a terminal. It is not an application server.

It runs the given script once for every line a client sends, with the line available as the implicit `line` variable and the connection as `out` and `socket`. The listening socket is bound to the loopback address, so only clients on the same machine can connect. Supply a host to listen more widely — `0.0.0.0:1960` for every interface, or a single address to pick one. The script is read once, when the listener starts, and recompiled for each connection so that one connection cannot leave state behind for the next; editing the file has no effect until the listener is restarted.

WARNING	A listener bound beyond loopback accepts connections from any peer that can reach the port. Groovy performs no authentication and the connection is not encrypted, so the script itself owns whatever checking is needed.

Configuring logging

Groovy's command-line tools use JDK Platform Logging (`System.Logger`) for diagnostic messages. By default, this is backed by `java.util.logging` (JUL).

Logging configuration file

JUL reads its configuration from exactly one source. The source is resolved in the following order:

1. The file specified by `-Djava.util.logging.config.file=⋯` on the command line (via `JAVA_OPTS`).
2. The file `~/.groovy/logging.properties`, if it exists. Groovy automatically loads this at startup.
3. The JDK default at `$JAVA_HOME/conf/logging.properties`.

The first source found is used and the rest are ignored.

Example configuration

```
# ~/.groovy/logging.properties
.level=INFO
handlers=java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level=ALL
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter
java.util.logging.SimpleFormatter.format=%1$tT %4$s [%3$s] %5$s%6$s%n

# Show Grape provider discovery details (or comment out for quieter operation)
groovy.grape.Grape.level=FINE
```

Logger reference

The following table lists all classes that emit log messages, grouped by module. The *JUL level*

column shows the `java.util.logging` level to set in `logging.properties` (the Platform Logging level is shown in parentheses).

Core (`groovy`)

Logger name	JUL level	What is logged
`groovy.grape.Grape`	FINE (DEBUG), WARNING, SEVERE (ERROR)	GrapeEngine provider discovery, selection, and instantiation failures
`groovy.ui.GroovyMain`	SEVERE (ERROR)	Script execution errors and uncaught exceptions
`org.apache.groovy.antlr.LexerFrame`	SEVERE (ERROR)	Parser errors in the lexer debugging tool
`org.codehaus.groovy.classgen.asm.OperandStack`	SEVERE (ERROR)	Stack operation errors during bytecode generation
`org.codehaus.groovy.classgen.asm.util.LoggableTextifier`	FINE (DEBUG)	ASM bytecode disassembly output
`org.codehaus.groovy.control.XStreamUtils`	FINE (DEBUG), WARNING	XStream serialization availability and failures
`org.codehaus.groovy.tools.DgmConverter`	INFO	DGM method stub generation progress
`org.codehaus.groovy.tools.FileSystemCompiler`	WARNING, SEVERE (ERROR)	Compilation errors and warnings
`org.codehaus.groovy.tools.GroovyStarter`	SEVERE (ERROR)	Startup configuration errors
`org.codehaus.groovy.tools.javac.JavacJavaCompiler`	INFO	Joint compilation progress

Grape Ivy (`groovy-grape-ivy`)

Logger name	JUL level	What is logged
`groovy.grape.ivy`	INFO, FINE (DEBUG), FINEST (TRACE)	Ivy dependency resolution, downloads, and cache operations

Ant tasks (`groovy-ant`)

Logger name	JUL level	What is logged
`org.codehaus.groovy.ant.Groovyc`	WARNING	Ant task compilation warnings

Logger name	JUL level	What is logged
`org.codehaus.groovy.ant.Groovydoc`	WARNING	Ant task groovydoc warnings

Console (`groovy-console`)

Logger name	JUL level	What is logged
`groovy.console.ui.ConsoleTextEditor`	WARNING	Text editor component warnings

GroovyDoc (`groovy-groovydoc`)

Logger name	JUL level	What is logged
`org.codehaus.groovy.tools.groovydoc.FileOutputTool`	WARNING	File output warnings
`org.codehaus.groovy.tools.groovydoc.antlr4.GroovyDocParser`	WARNING	Doc parsing warnings
`org.codehaus.groovy.tools.groovydoc.GroovyDocTemplateEngine`	WARNING, SEVERE (ERROR)	Template processing errors
`org.codehaus.groovy.tools.groovydoc.GroovyRootDocBuilder`	WARNING	Doc building warnings

JSON (`groovy-json`)

Logger name	JUL level	What is logged
`org.apache.groovy.json.internal.Exceptions`	SEVERE (ERROR)	JSON parsing exception details
`org.apache.groovy.json.internal.Sys`	WARNING	System property access warnings

Servlet (`groovy-servlet`)

Logger name	JUL level	What is logged
`groovy.servlet.GroovyServlet`	SEVERE (ERROR)	Servlet script execution errors

Swing (`groovy-swing`)

Logger name	JUL level	What is logged
`groovy.swing.table.TableSorter`	WARNING	Table sorting comparison warnings

Testing (`groovy-test`, `groovy-test-junit5`)

Logger name	JUL level	What is logged
`groovy.test.GroovyTestSuite`	INFO	Test suite class loading info

Logger name	JUL level	What is logged
`groovy.junit5.plugin.JUnit5Runner`	WARNING	JUnit5 runner warnings

TIP To enable all diagnostic logging, set `.level=FINE` in your configuration file. To focus on a specific area, set the logger name to `FINE` (or `FINER`/`FINEST` for trace-level detail) and leave the root level at `INFO` or `WARNING`.

NOTE If a different `System.LoggerFinder` is on the classpath (e.g., via SLF4J/Logback or Log4j2), platform logging calls are routed to that backend instead, and `logging.properties` has no effect. Configure the alternative backend using its own mechanism (e.g., `logback.xml` or `log4j2.xml`).

Compiling Groovy

groovyc, the Groovy compiler

`groovyc` is the Groovy compiler command line tool. It allows you to compile Groovy sources into bytecode. It plays the same role as `javac` in the Java world. The easiest way to compile a Groovy script or class is to run the following command:

```
groovyc MyClass.groovy
```

This will produce a `MyClass.class` file (as well as other .class files depending on the contents of the source). `groovyc` supports a number of command line switches:

Short version	Long version	Description	Example
-cp	-classpath, --classpath	Specify the compilation classpath. Must be the first argument.	groovyc -cp lib/dep.jar MyClass.groovy
	--sourcepath	Directory where to find source files. Not used anymore. Specifying this parameter will have no effect.	
	--temp	Temporary directory for the compiler	
	--encoding	Encoding of the source files	groovyc --encoding utf-8 script.groovy
	--help	Displays help for the command line groovyc tool	groovyc --help
-d		Specify where to place generated class files.	groovyc -d target Person.groovy

Short version	Long version	Description	Example
	--check	Check sources for errors without writing class files. Every check runs, including class verification and bytecode generation.	groovyc --check Person.groovy
-v	--version	Displays the compiler version	groovyc -v
-e	--exception	Displays the stack trace in case of compilation error	groovyc -e script.groovy
-w	--warningLevel	Sets how many warnings are reported after a successful compile: `0` (none), `1` (likely errors, the default), `2` (possible errors) or `3` (all). See Compiler warnings.	groovyc -w 2 MyClass.groovy
-j	--jointCompilation*	Enables joint compilation	groovyc -j A.groovy B.java
-b	--basescript	Base class name for scripts (must derive from Script)	
	--configscript	Advanced compiler configuration script	groovyc --configscript config/config.groovy src/Person.groovy
	--error-format	How errors and warnings are rendered: `full` (the default, with the source line and a caret) or `short` (one `file:line:column: severity: message` line per diagnostic, for editors and CI log parsers).	groovyc --error -format=short Person.groovy
-t	--tolerance	The number of non-fatal errors to collect before compilation is abandoned. Defaults to `10`; use `0` for unlimited.	groovyc -t 0 Person.groovy

Short version	Long version	Description	Example
-Jproperty=value		Properties to be passed to `javac` if joint compilation is enabled	groovyc -j -Jtarget=1.6 -Jsource=1.6 A.groovy B.java
-Fflag		Flags to be passed to `javac` if joint compilation is enabled	groovyc -j -Fnowarn A.groovy B.java
-pa	--parameters	Generates metadata for reflection on method parameter names. Requires Java 8+.	groovyc --parameters Person.groovy
-pr	--enable-preview	Enable preview Java features (jdk12+ only).	groovy --enable -preview Person.groovy
@argfile		Read options and source files from specified file.	groovyc @conf/args

Notes: * for a full description of joint compilation, see the joint compilation section.

Compiler warnings

While compiling, Groovy may report *warnings*: diagnostics that don't stop compilation but flag code that is suspicious or unlikely to behave as intended — for example, an unreachable `main` method or a property that can't override a `final` accessor. On a successful compile, `groovyc` prints any collected warnings to `stderr`, using the same format (message, source line and caret) as compilation errors, followed by a summary count.

Every warning carries an *importance*, and the `-w` (`--warningLevel`) option controls how many are reported. Only warnings whose importance is at or below the configured level are kept, so a lower level is quieter:

Level	Name	Reports
`0`	none	No warnings at all.
`1`	likely errors	**(default)** Diagnostics that very probably indicate a mistake — for example an unreachable `main` method, a property that can't override a `final` accessor, or a duplicate constructor generated from default argument values or `@TupleConstructor`/`@MapConstructor`.

Level	Name	Reports
2	possible errors	The above, plus diagnostics that often have benign causes — for example a field hidden by a dynamic property under `@TypeChecked`, or problems found while scanning global AST transformation descriptors.
3	paranoia	The above, plus any remaining lower-priority warnings.

For example, to also see the level 2 warnings:

```
groovyc -w 2 MyClass.groovy
```

NOTE The warning *level* can also be set programmatically via `CompilerConfiguration#setWarningLevel`, or for the `groovy` command through the `groovy.warnings` system property. Displaying the collected warnings on a successful compile is done by the `groovyc` command line; other embeddings (for instance, code driving a `CompilationUnit` directly) can instead read them from the unit's `ErrorCollector`.

Ant task

See the groovyc Ant task documentation. It allows the Groovy compiler to be invoked from Apache Ant.

Gant

Gant is a tool for scripting Ant tasks using Groovy instead of XML to specify the logic. As such, it has exactly the same features as the Groovyc Ant task.

Gradle

Gradle is a build tool that allows you to leverage the flexibility of Ant, while keeping the simplicity of convention over configuration that tools like Maven offer. Builds are specified using a Groovy DSL, which offers great flexibility and succinctness.

Maven integration

There are several approaches to compiling Groovy code in your Maven projects. GMavenPlus is the most flexible and feature rich, but like most Groovy compiler tools, it can have difficulties with joint Java-Groovy projects (for the same reason GMaven and Gradle can have issues). The Groovy-Eclipse compiler plugin for Maven sidesteps the joint compilation issues. Read this for a deeper discussion of the benefits and disadvantages of the two approaches.

A third approach is to use Maven's Ant plugin to compile a groovy project. Note that the Ant plugin is bound to the compile and test-compile phases of the build in the example below. It will be invoked during these phases and the contained tasks will be carried out which runs the Groovy compiler over the source and test directories. The resulting Java classes will coexist with and be treated like any standard Java classes compiled from Java source and will appear no different to the JRE, or the JUnit runtime.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycomp.MyGroovy</groupId>
    <artifactId>MyGroovy</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>Maven Example building a Groovy project</name>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.codehaus.groovy</groupId>
            <artifactId>groovy-all</artifactId>
            <version>2.5.0</version>
            <type>pom</type> <!-- required JUST since Groovy 2.5.0 -->
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <artifactId>maven-antrun-plugin</artifactId>
                <executions>
                    <execution>
                        <id>compile</id>
                        <phase>compile</phase>
                        <configuration>
                            <tasks>
                                <mkdir dir="${basedir}/src/main/groovy"/>
                                <taskdef name="groovyc"
                                    classname="org.codehaus.groovy.ant.Groovyc">
                                    <classpath refid="maven.compile.classpath"/>
                                </taskdef>
                                <mkdir dir="${project.build.outputDirectory}"/>
                                <groovyc destdir="${project.build.outputDirectory}"
                                    srcdir="${basedir}/src/main/groovy/"
listfiles="true">
```

```
                                <classpath refid="maven.compile.classpath"/>
                            </groovyc>
                        </tasks>
                    </configuration>
                    <goals>
                        <goal>run</goal>
                    </goals>
                </execution>
                <execution>
                    <id>test-compile</id>
                    <phase>test-compile</phase>
                    <configuration>
                        <tasks>
                            <mkdir dir="${basedir}/src/test/groovy"/>
                            <taskdef name="groovyc"
                                classname="org.codehaus.groovy.ant.Groovyc">
                                <classpath refid="maven.test.classpath"/>
                            </taskdef>
                            <mkdir dir="${project.build.testOutputDirectory}"/>
                            <groovyc
destdir="${project.build.testOutputDirectory}"
                                srcdir="${basedir}/src/test/groovy/"
listfiles="true">
                                <classpath refid="maven.test.classpath"/>
                            </groovyc>
                        </tasks>
                    </configuration>
                    <goals>
                        <goal>run</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
  </build>
</project>
```

This assumes you have a Maven project setup with `groovy` subfolders as peers to the java src and test subfolders. You can use the `java/jar` archetype to set this up then rename the java folders to groovy or keep the java folders and just create groovy peer folders. There exists, also a groovy plugin which has not been tested or used in production. After defining the build section as in the above example, you can invoke the typical Maven build phases normally. For example, `mvn test` will execute the test phase, compiling Groovy source and Groovy test source and finally executing the unit tests. If you run `mvn jar` it will execute the jar phase bundling up all of your compiled production classes into a jar after all the unit tests pass. For more detail on Maven build phases consult the Maven2 documentation.

GMaven and GMavenPlus

GMaven

GMaven is the original Maven plugin for Groovy, supporting both compiling and scripting Groovy.

Important:

You should be aware that GMaven is **not supported anymore** and can have difficulties with joint compilation. GMavenPlus can be a good replacement, but if you are having problems with joint compilation, you might consider the Groovy Eclipse maven plugin.

GMavenPlus

GMavenPlus is a rewrite of GMaven and is in active development. It supports most of the features of GMaven (a couple notable exceptions being mojo Javadoc tags and support for older Groovy versions). Its joint compilation uses stubs (which means it has the same potential issues as GMaven and Gradle). The main advantages over its predecessor are that it supports recent Groovy versions, InvokeDynamic, Groovy on Android, GroovyDoc, and configuration scripts.

GMaven 2

Unlike the name might seem to suggest, GMaven 2 is not aimed at replacing GMaven. In fact, it removes the non-scripting features of the GMaven plugin. It has not yet had any release and appears to be inactive currently.

The Groovy Eclipse Maven plugin

Groovy-Eclipse provides a compiler plugin for Maven. Using the compiler plugin, it is possible to compile your maven projects using the Groovy-Eclipse compiler. One feature unavailable elsewhere is stubless joint compilation.

Joint compilation

Joint compilation means that the Groovy compiler will parse the Groovy source files, create stubs for all of them, invoke the Java compiler to compile the stubs along with Java sources, and then continue compilation in the normal Groovy compiler way. This allows mixing of Java and Groovy files without constraint.

Joint compilation can be enabled using the `-j` flag with the command-line compiler, or using a nested tag and all the attributes and further nested tags as required for the Ant task.

It is important to know that if you don't enable joint compilation and try to compile Java source files with the Groovy compiler, the Java source files will be compiled as if they were Groovy sources. In some situations, this might work since most of the Java syntax is compatible with Groovy, but there are a number of places where semantics could be different.

Android support

It is possible to write an Android application in Groovy. However this requires a special version of the compiler, meaning that you cannot use the regular groovyc tool to target Android bytecode. In particular, Groovy provides specific JAR files for Android, which have a classifier of `grooid`. In order to make things easier, a Gradle plugin adds support for the Groovy language in the Android Gradle

toolchain.

The plugin can be applied like this:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.1.2'
        classpath 'org.codehaus.groovy:groovy-android-gradle-plugin:1.0.0'
    }
}

apply plugin: 'groovyx.android'
```

Then you will need to add a dependency on the `grooid` version of the Groovy compiler:

```
dependencies {
    compile 'org.codehaus.groovy:groovy:2.4.7:grooid'
}
```

Note that if a Groovy jar does not provide a `grooid` classifier alternative, then it means that the jar is directly compatible with Android. In that case, you can add the dependency directly like this:

```
dependencies {
    compile 'org.codehaus.groovy:groovy:2.4.7:grooid'       // requires the grooid
classifier
    compile ('org.codehaus.groovy:groovy-json:2.4.7') {     // no grooid version
available
        transitive = false                                  // so do not depend on
non-grooid version
    }
}
```

Note that the `transitive=false` parameter for `groovy-json` will let Gradle download the JSON support jar without adding a dependency onto the normal jar of Groovy.

Please make sure to go to the plugin homepage in order to find the latest documentation and version.

Groovysh, the Groovy shell

Groovy : Groovy Shell

The Groovy Shell, aka. `groovysh` is a command-line application which allows easy access to evaluate

Groovy expressions, define classes and run simple experiments.

Features

- Rich cross-platform line editing, history and completion thanks to JLine3.
- ANSI colors (prompt, exception traces, etc).
- Simple, yet robust, command system with online help, user alias support and more.

Command-line Options and Arguments

The shell supports several options to control various features.

```
> groovysh --help

Usage: groovysh [options] [...]
The Groovy Shell, aka groovysh, is a command-line application which allows easy
access to evaluate Groovy expressions, define classes and run simple experiments.
  -c, --encoding=<CHARSET>   The encoding to use
  -C, --color[=<FLAG>]       Enable or disable use of ANSI colors
      -cp, -classpath, --classpath
                             Specify where to find the class files - must be
                               first argument
  -d, --debug                Enable debug output
  -D, --define=<name=value>  Define a system property
  -e, --evaluate=<CODE>      Evaluate the code first when starting interactive
                               session
  -h, --help                 Display this help message
      -pa, --parameters      Generate metadata for reflection on method
                               parameter names
      -pr, --enable-preview  Enable preview Java features (jdk12+ only) - must
                               be after classpath but before other arguments
  -q, --quiet                Suppress superfluous output
  -T, --terminal=<TYPE>      Specify the terminal TYPE to use
  -v, --verbose              Enable verbose output
  -V, --version              Display the version
```

Here is an example showing the `-q|--quiet` and `-e|--evaluate` options:

```
➜  install bin/groovysh -q -e "println new Date()"
groovysh 5.0.0-SNAPSHOT
--------------------------------------------------------------------------------
Wed Aug 13 10:14:20 AEST 2025
groovy> /q
➜  install bin/groovysh -q -e "/cat -n answers.groovy ||| /load answers.groovy"
groovysh 5.0.0-SNAPSHOT
--------------------------------------------------------------------------------
     1  def answers = [universe: 42]
     2  println "The answer to the ultimate question is: ${answers.universe}"
The answer to the ultimate question is: 42
Loaded: /Users/paulk/working/install/answers.groovy
groovy> /vars
def answers = [universe: 42]
groovy>
```

Repl model

The Groovy Shell is a Read-Eval-Print Loop (REPL) which allows you to interactively evaluate Groovy expressions and statements, define classes and other types, invoke commands, and run simple experiments.

When you input a line, the shell will try to determine if the input you have given is a complete valid expression, statement, or definition. If not complete, it will prompt you for more input. If it is complete, it will execute the input, and print the result, if any, to the console. Each input you enter is executed in isolation.

There are some exceptions to this conceptual model. Any import statements entered will be remembered and used for all subsequent evaluations. Similarly, with some caveats we'll discuss next, any previously defined classes, methods, and potentially variables will be available.

The shell has the concept of shared variables. Given that subsequent statements are run in isolation, you should store any results needed for later use in shared variables.

Many Groovy tutorials and examples use the `def` keyword or a type to define variables. Script examples might distinguish between local variables with a type and script binding variables where no type, nor the `def` or `var` type placeholders, are given. The script binding is the exact equivalent to the shell's shared variables.

Because such statements are so common, the shell has a special mode called *interpreter mode* which allows you to use typed variables. The following table summarizes the differences between the two modes:

interpreterMode	**off**	**on**
imports	remembered	
types	available	
methods	converted to closure shared variables	remembered
shared variables	available	
local variables	forgotten	remembered

Conceptually, for things that are *remembered* in the above table, it is as if you included the related code at the start of each of your inputs.

The shell also has the concept of the "current buffer". This is the collection of all type, method, imports, and variable definitions. The ordering of these snippets is retained, which is important for the shell, since it is only executing one snippet at a time, it typically requires everything needed for a snippet to be pre-defined. This is different to a compiler which might compile multiple source files at once, and resolve references between types as needed.

Evaluating Expressions

Simple Expressions

```
groovy> println "Hello"
Hello
```

Evaluation Result

When a complete expression is found, it is compiled and evaluated. The result of the evaluation is stored into the _ variable.

Multi-line Expressions

Multi-line/complex expressions (like closure or class definitions) may be defined over several lines. When the shell detects that it has a complete expression it will compile and evaluate it.

Defining types

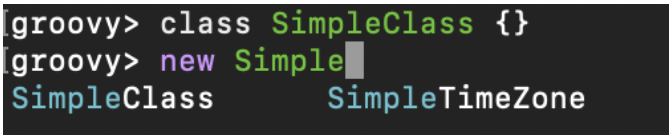
You can define all the normal types, e.g. a class:

```
groovy> class Foo {
add: } >   def bar() {
add: }}>     println "baz"
add: }}>   }
add: } > }
```

And use it in the normal way.

```
groovy> foo = new Foo()
groovy> foo.bar()
baz
```

Defined classes are known to the shell and can be used in completion:



Variables

Shell variables are **all** untyped (i.e. no `def` or other type information).

This **will** set a shell variable:

```
foo = "bar"
```

But, this will evaluate a local variable and will **not** be saved to the shell's environment:

```
def foo = "bar"
```

This behavior can be changed by activating interpreter mode.

Variables definitions having the same name as existing definitions will replace the old definition with the new one.

Methods

Methods can be defined in the shell, and will be saved for later use.

Defining a method is easy:

```
groovy> def hello(name) {
add: }>   println("Hello $name")
add: }> }
```

And then using it is as one might expect:

```
groovy> hello "Jason"
Hello Jason
```

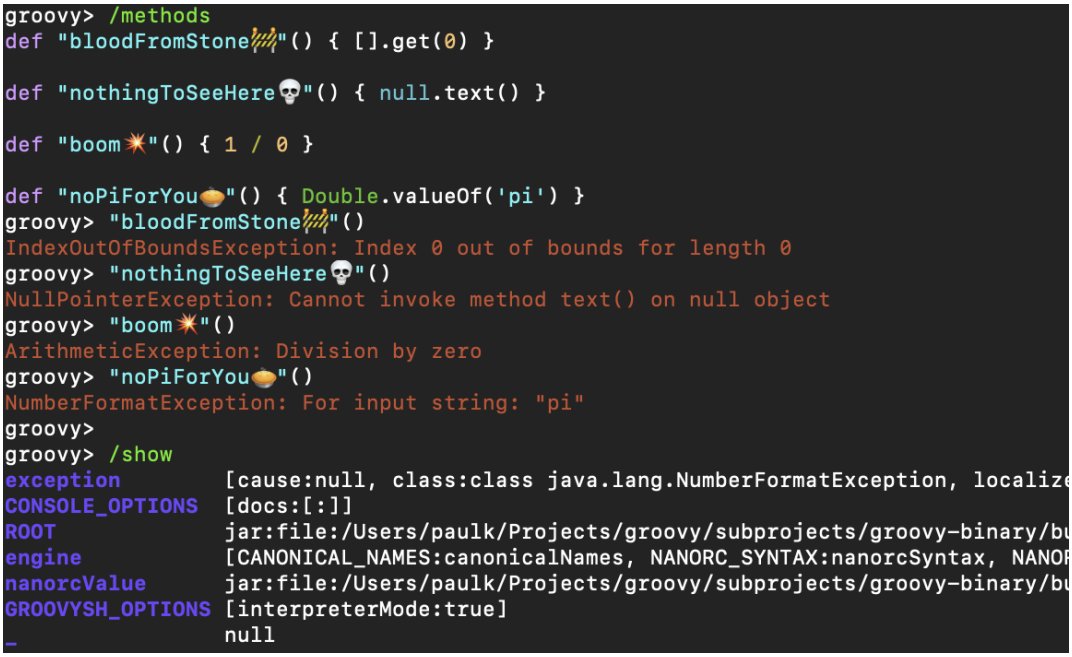
If a method definition has the same signature as an existing definition, the old definition will be replaced with the new one.

When in *interpreterMode*, methods are remembered and given as extra statements when executing the next input.

When not in *interpreterMode*, the shell internally creates a closure to encapsulate the method and stores it in the shared variables. In this case, variables and methods share the same namespace.

Exceptions

If an exception is thrown, the shell will print the exception message and set a special `exception` shared variable.



Commands

The shell has a number of different commands, which provide rich access to the shell's environment.

Commands all have a *name*, e.g. `/help` and `/prnt`. Commands may also have some predefined system *aliases*, e.g. `/h`. Users may also create their own aliases. This section will list commands in alphabetical order, but you can also use the `/help` command to list the available commands:

```
groovy> /help
    /!           execute shell command
    /alias       create command alias
    /cat         concatenate and print FILES
    /cd          change directory
    /classloader display/manage Groovy classLoader data
    /clear       clear terminal
    /colors      view 256-color table and ANSI-styles
    /console     launch Groovy console
    /date        display date
    /del         delete console variables, methods, classes and imports
    /doc         open document on browser
    /echo        echos a value
    /exit        exit from app/script
    /grab        add maven repository dependencies to classpath
    /grep        search for PATTERN in each FILE or standard input.
    /head        display first lines of files
    /help        command help
    /highlighter manage nanorc theme system
    /history     list history of commands
    /imports     show/delete import statements
    /inspect     display/browse object info on terminal/object browser
    /keymap      manipulate keymaps
    /less        file pager
```

```
    /load        load state/a file into the buffer
    /ls          list files
    /methods     show/delete methods
    /nano        edit files
    /pipe        create/delete pipe operator
    /prnt        print object
    /pwd         print working directory
    /reset       clear the buffer
    /save        save state/the buffer to a file
    /setopt      set options
    /setvar      set lineReader variable value
    /show        list console variables
    /slurp       slurp file or string variable context to object
    /sort        writes sorted standard input to standard output.
    /tail        display last lines of files
    /ttop        display and update sorted information about threads
    /types       show/delete types
    /unalias     remove command alias
    /unsetopt    unset options
    /vars        show/delete variable declarations
    /wc          word, line, character, and byte count
    /widget      manipulate widgets
```

While in the interactive shell, you can ask for help for any command to get more details about its syntax or function. You can use `/help <command>` or `<command> --help`. Here is an example of what happens when you ask for help for the `/help` command:

```
groovy> /help /help
help -  command help
Usage: help [TOPIC...]
  -? --help                     Displays command help
     --groups                   Commands are grouped by registries
  -i --info                     List commands with a short command info
```

`/alias`

Create an alias for a commandline fragment. The fragment could be Groovy code or a shell command. When evaluating a commandline, the alias will be replaced with the fragment:

```
groovy> /alias now println new Date()
groovy> now
Sun Aug 03 13:29:47 AEST 2025
groovy> /alias /cat "/less -F"
groovy> /cat answers.json
{
    "answer": {
        "universe":
        42
    }
}
groovy> /alias
[/cat:/less -F, now:println new Date(), /x:exit, /h:help]
```

The `/cat` command is already available. As can be seen here, aliases are replaced before command selection, so with the above alias in place, we actually invoke `/less -F` command, which has syntax highlighting, instead of the builtin `/cat` which doesn't.

The fragment is expected to be at the start of a line but other text may follow:

```
groovy> now + 1
Mon Aug 04 13:37:06 AEST 2025
groovy> /cat -N answers.json
     1 {
     2     "answer": {
     3         "universe":
     4         42
     5     }
     6 }
```

Aliases are persisted in a `.groovysh/aliases.json` file in the user home directory.

See also the `/unalias` command, which allows aliases to be removed.

`/cat`

Concatenate and print files:

```
groovy> /cat answers.json answers.toml
{
    "answer": {
        "universe":
        42
    }
}
[answer]
universe = 42
```

See also the `/less` command, which can be used to display the contents of a file a page at a time with syntax highlighting.

`/cd`

Change the current working directory. `groovysh` keeps track of a *current working directory*. You can

view it with `/pwd` and change it with `/cd`.

The `groovysh` file commands (`/cat`, `/less`, `/head`, `/slurp`, `/load`, `/ls`, etc.) will use the current working directory.

NOTE	The JVM running `groovysh` also has its own concept of the current working directory, which is the directory in which `groovysh` was started, so `/!ls` and `/ls` may not show the same files. Using `/!cd` only changes the working directory within the context of that one shell command.

There is a predefined shared variable `PWD` which can assist with invoking shell commands. So, on unix platforms, `/ls` will list the same files as `/!ls $PWD`.

See also the `/pwd` command, which prints the current working directory.

```
groovy> /pwd
/Users/paulk/Projects/groovy
groovy> /cd subprojects
groovy> /pwd
/Users/paulk/Projects/groovy/subprojects
groovy> /!pwd
/Users/paulk/Projects/groovy
groovy> /ls -C
                    groovy-dateutil      groovy-swing
..                  groovy-docgenerator  groovy-templates
binary-compatibility groovy-ginq         groovy-test
groovy-all          groovy-groovydoc     groovy-test-junit5
groovy-ant          groovy-groovysh      groovy-testng
groovy-astbuilder   groovy-jmx           groovy-toml
groovy-binary       groovy-json          groovy-typecheckers
groovy-bom          groovy-jsr223        groovy-xml
groovy-cli-commons  groovy-macro         groovy-yaml
groovy-cli-picocli  groovy-macro-library performance
groovy-console      groovy-nio           stress
groovy-contracts    groovy-servlet       tests-preview
groovy-datetime     groovy-sql
groovy> /!ls -C sub*
binary-compatibility  groovy-docgenerator   groovy-swing
groovy-all            groovy-ginq           groovy-templates
groovy-ant            groovy-groovydoc      groovy-test
groovy-astbuilder     groovy-groovysh       groovy-test-junit5
groovy-binary         groovy-jmx            groovy-testng
groovy-bom            groovy-json           groovy-toml
groovy-cli-commons    groovy-jsr223         groovy-typecheckers
groovy-cli-picocli    groovy-macro          groovy-xml
groovy-console        groovy-macro-library  groovy-yaml
groovy-contracts      groovy-nio            performance
groovy-datetime       groovy-servlet        stress
groovy-dateutil       groovy-sql            tests-preview
groovy> /!ls -C $PWD
binary-compatibility  groovy-docgenerator   groovy-swing
groovy-all            groovy-ginq           groovy-templates
groovy-ant            groovy-groovydoc      groovy-test
groovy-astbuilder     groovy-groovysh       groovy-test-junit5
groovy-binary         groovy-jmx            groovy-testng
groovy-bom            groovy-json           groovy-toml
groovy-cli-commons    groovy-jsr223         groovy-typecheckers
groovy-cli-picocli    groovy-macro          groovy-xml
groovy-console        groovy-macro-library  groovy-yaml
groovy-contracts      groovy-nio            performance
groovy-datetime       groovy-servlet        stress
groovy-dateutil       groovy-sql            tests-preview
```

`/classloader`

Display and manage the Groovy classloader data.

Let's `/grab` a dependency, define a class using it, and then use the `/classloader` command to see the classloader data:

```
groovy> /grab org.apache.commons:commons-collections4:4.5.0
groovy> import org.apache.commons.collections4.bag.HashBag
groovy> class SizeTrackingList extends ArrayList<String> {
add: } >   private HashBag sizes = new HashBag()
add: } >
add: } >   boolean add(String s) {
add: }}>     super.add(s)
add: }}>     sizes.add(s.size())
add: }}>   }
add: } >
add: } >   def getSizes() { sizes.toString() }
add: } > }
groovy> fruit = new SizeTrackingList()
groovy> fruit << 'apple'
[apple]
groovy> fruit << 'orange'
[apple, orange]
groovy> fruit << 'banana'
[apple, orange, banana]
groovy> fruit.sizes
[1:5,2:6]
groovy> /classloader
definedPackages
    package org.apache.commons.collections4.set, Apache Commons Collections, version 4.5
    package org.apache.commons.collections4, Apache Commons Collections, version 4.5
    package org.apache.commons.collections4.bag, Apache Commons Collections, version 4.5
    package org.apache.commons.collections4.collection, Apache Commons Collections, version 4.5
    package org.apache.commons.collections4.iterators, Apache Commons Collections, version 4.5
loadedClasses
    SizeTrackingList
classPath
    /Users/paulk/.groovy/grapes/org.apache.commons/commons-collections4/jars/commons-collections4-4.5.0.jar
groovy>
```

`/clear`

Clears the screen.

`/colors`

Displays the available colors.

```
groovy> /colors
256-color table, fg:<name> / 38;5;<n>
                 bg:<name> / 48;5;<n>

  black    red      green    yellow   blue     magenta  cyan     white
  !black   !red     !green   !yellow  !blue    !magenta !cyan    !white

  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33
  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69
  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87
  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105
 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123
 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141
 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177
 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195
 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213
 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231

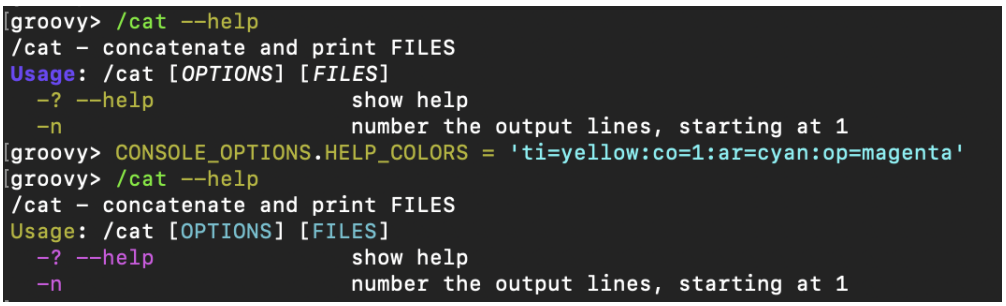
  232 233 234 235 236 237 238 239 240 241 242 243
  244 245 246 247 248 249 250 251 252 253 254 255
```

This can be useful when configuring the console options, for example the `PRNT_COLORS`:

```
groovy> csv=/slurp whiskey.csv
groovy> /prnt $csv --maxrows=5 -r
   RowID Spicy Sweetness Smoky Winey Floral Nutty Honey Tobacco Fruity Distillery   Malty Medicinal Body
 0    82     0         1     1     0      2     1     1       0      2 Tobermory        2         0    1
 1    83     2         3     2     1      1     1     2       0      0 Tomatin         2         0    2
 2    84     2         3     1     1      2     1     2       0      1 Tomintoul       2         0    0
 3    85     0         2     1     1      0     2     1       0      0 Tomore          1         0    2
 4    86     2         3     0     1      1     1     0       1      2 Tullibardine    2         0    2
Truncated output: 5/86
groovy> CONSOLE_OPTIONS.PRNT_COLORS = 'th=!magenta:rn=NUMBER:rs=black,75'
groovy> /prnt $csv --maxrows=5 -r -hodd
   RowID Spicy Sweetness Smoky Winey Floral Nutty Honey Tobacco Fruity Distillery   Malty Medicinal Body
 0    82     0         1     1     0      2     1     1       0      2 Tobermory        2         0    1
 1    83     2         3     2     1      1     1     2       0      0 Tomatin         2         0    2
 2    84     2         3     1     1      2     1     2       0      1 Tomintoul       2         0    0
 3    85     0         2     1     1      0     2     1       0      0 Tomore          1         0    2
 4    86     2         3     0     1      1     1     0       1      2 Tullibardine    2         0    2
Truncated output: 5/86
```

As this example shows, you can mix color names (like *black* and *magenta*), theme colors (like *NUMBER*), and color numbers (like *75*).

Here is an example for `HELP_COLORS`:



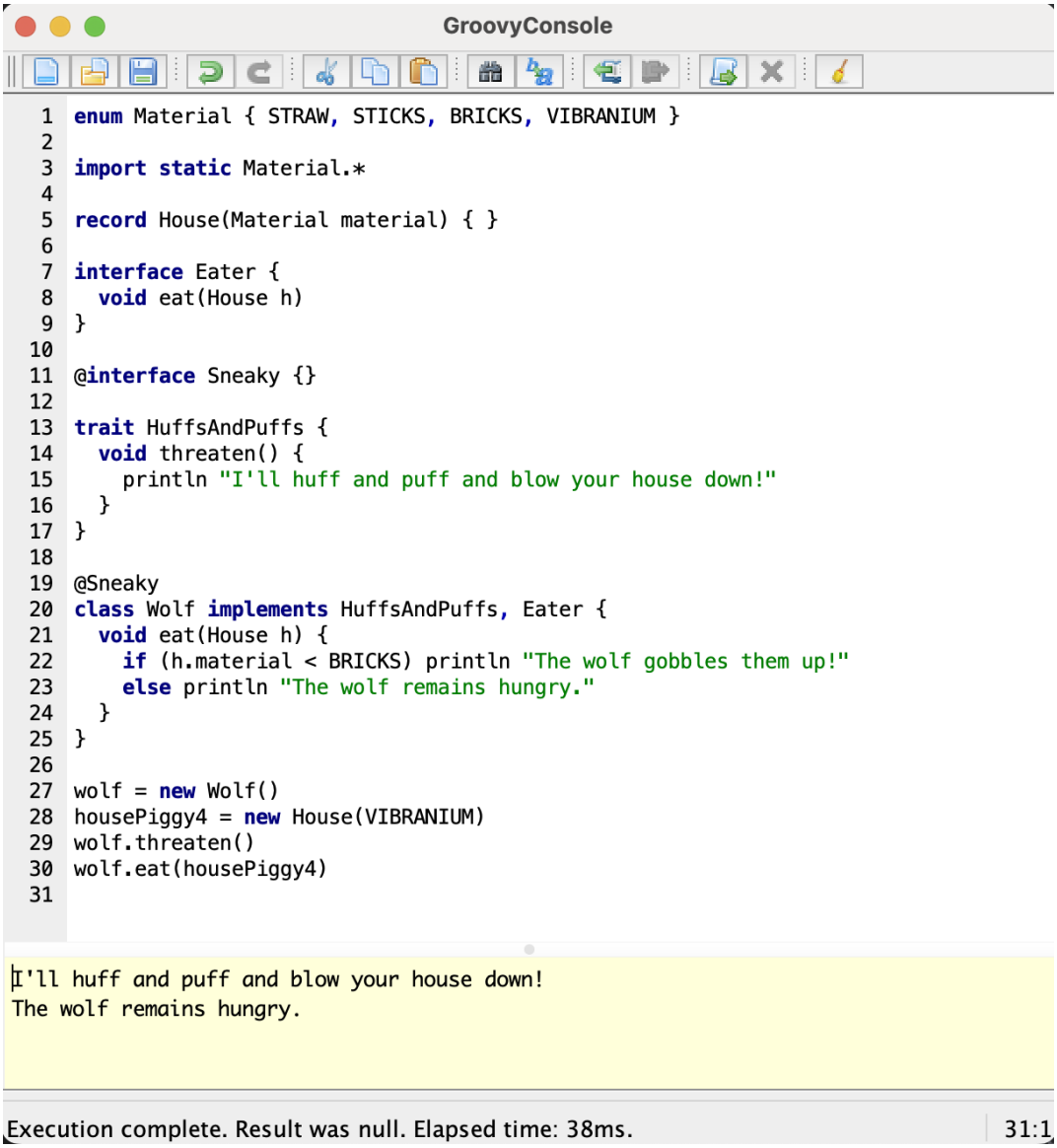
Here is an example for `LS_COLORS`:



See the REPL Console Customization documentation for more details about such customization.

`/console`

Displays the current buffer in the GroovyConsole.



/date

Groovy has numerous classes for working with dates and time.

Here are some examples using the legacy date/calendar classes:

```
groovy> println new Date()
Tue Aug 19 14:35:22 AEST 2025
groovy> printf '%1ta %1$tb %1$td %1$tT %1$tZ %1$tY%n', new Date()
Tue Aug 19 14:35:22 AEST 2025
groovy> println Calendar.instance.time
Tue Aug 19 14:35:22 AEST 2025
groovy> println new Date().format('dd-MMM-yyyy HH:mm')
19-Aug-2025 14:35
```

Here are some examples using the `java.time` classes:

```
groovy> println LocalDateTime.now()
2025-08-19T14:17:35.383665
```

```
groovy> println Instant.now()
2025-08-19T04:17:35.590829Z
groovy> printf '%1ta %1$tb %1$td %1$tT %1$tY%n', LocalDateTime.now()
Tue Aug 19 14:47:35 2025
groovy> println LocalDateTime.now().format('dd-MMM-yyyy HH:mm')
19-Aug-2025 14:47
```

The `/date` command is an alternative for those familiar with the unix `date` commandL:

```
groovy> /date -R
Tue, 19 Aug 2025 14:17:05 +10
groovy> /date --rfc-3339=seconds
2025-08-19 14:17:05+10
groovy> /date --rfc-3339=time
Aug Tue 19 14:17:05 2025
```

/del

Deletes objects from the shell. The command looks for any variable, method, type, shared data, or import with the given name and deletes it.

Here we create two variables, `x` and `y`, then delete the `x` variable:

```
groovy> def x = 4
4
groovy> def y = 8
8
groovy> /del x
groovy> /vars
def y = 8
groovy>
```

See also, the `/vars`, `/methods`, `/types`, and `/imports` commands. These offer a `-d|--delete` option to delete the respective objects with completion.

/doc

Open documentation in a browser.

Documentation can be found for Groovy classes like this:

```
groovy> /doc groovy.lang.Tuple
```

For Java classes, the `/doc` command will look for the class in the JDK documentation as well as in the groovy-jdk if Groovy has any extensions for that class:

```
groovy> /doc java.util.List
```

Documentation is also extensible, so you can add your own documentation using regex patterns. For example, if you add the following line to your `~/.groovy/groovysh_init.groovy` file:

```
CONSOLE_OPTIONS.docs.groovy='https://groovy-lang.org/documentation.html'
CONSOLE_OPTIONS.docs.'org/jline/.*'='https://www.javadoc.io/doc/org.jline/jline/latest
/'
```

Then, the following commands will open the Groovy documentation page and the JLine documentation for `Terminal` respectively in your browser:

```
groovy> /doc groovy
groovy> /doc org.jline.terminal.Terminal
```

/echo

The `/echo` command outputs its arguments to the console. Arguments are output verbatim, but variable expansion is also supported.

```
groovy> answer = [universe: 42]
groovy> /echo answer to the universe
answer to the universe
groovy> /echo $answer is the answer
{universe=42} is the answer
groovy> /echo ${answer}.universe
42
groovy> /echo ${answer.universe}
42
groovy> /echo ${answer.universe / 7}
6
groovy> /echo ${answer.keySet()[0].capitalize()}
Universe
```

See also the `/prnt` command, which is similar but may perform additional formatting on the output(s).

/exit

Exit the shell.

This is the **only** way to exit the shell. Well, you can still `CTRL-Z` on unix platforms, but things like `CTRL_C` are trapped. (See JLine3 documentation for more details.)

/grab

Grab a dependency (Maven, Ivy, etc.) from Internet sources or cache, and add it to the Groovy Shell environment.

```
groovy> /grab org.apache.commons:commons-collections4:4.5.0
groovy> import org.apache.commons.collections4.bidimap.TreeBidiMap
groovy> TreeBidiMap t = [apple: 'red']
{apple=red}
groovy> t.inverseBidiMap()
```

```
{red=apple}
```

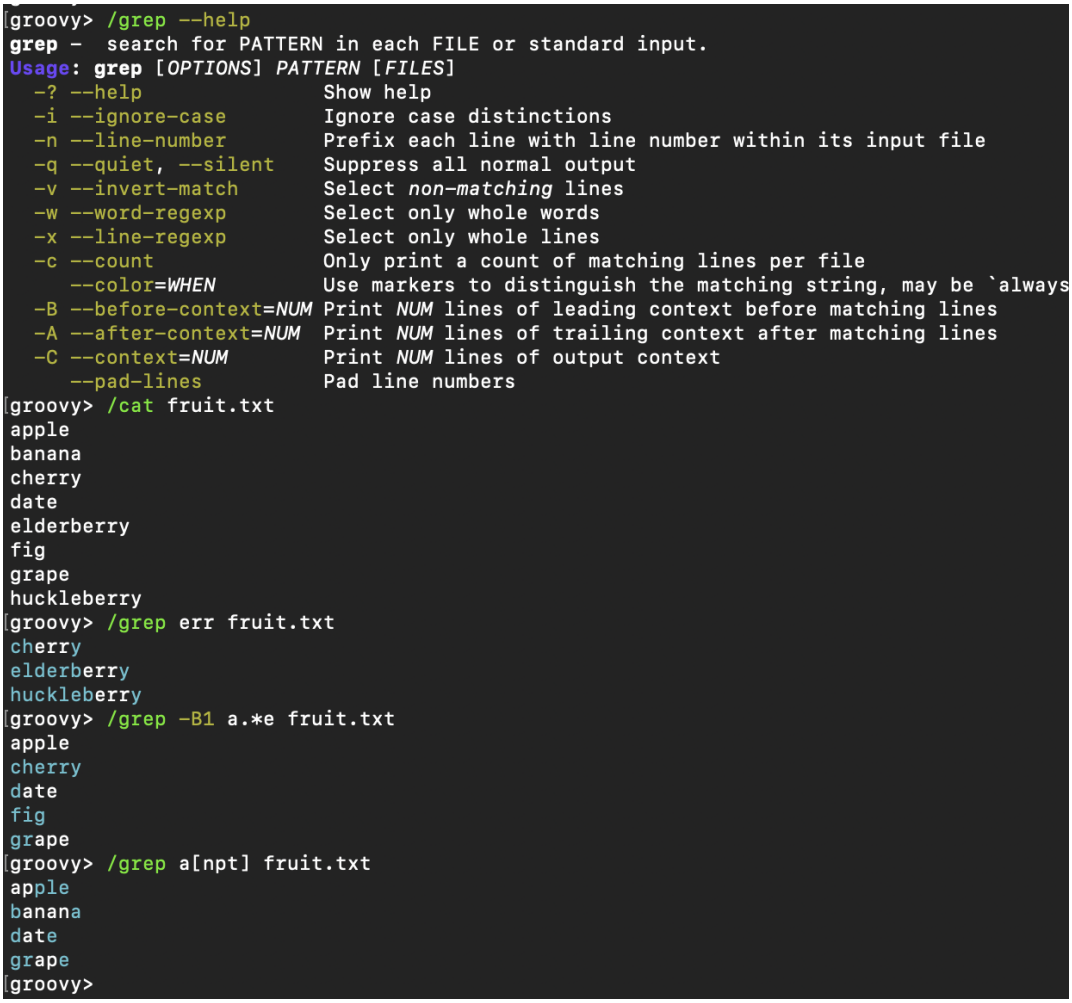
Completion is available. Currently, completion options are populated by known artifacts in the local Maven (~/.m2) and Grape (~/.groovy/grapes) repositories. In the future, completion from a remote repositories may be supported.

```
groovy> /grab org.apache.commons:commons-<TAB>
commons-collections4:     commons-dbcp2:            commons-lang3:
commons-compress:         commons-digester3:        commons-math3:
commons-configuration2:   commons-email:            commons-parent:
commons-crypto:           commons-exec:             commons-pool2:
commons-csv:              commons-imaging:          commons-text:
```

This command can be given at any time to add new dependencies.

/grep

The /grep command searches for patterns in files or standard input.



/head

The /head command displays the start of a file.

```
groovy> /head -n2 fruit.txt
apple
banana
```

/highlighter

The /highlighter command manages the nanorc theme system. Useful options include --list (show available themes), --view=THEME (preview a theme), --switch=THEME (change the active theme by rewriting your user-config jnanorc), and --refresh (reapply highlighting after a switch).

Note: --view and --switch take **full filenames including the .nanorctheme extension**, not bare theme names. /highlighter --list shows the canonical form:

```
groovy> /highlighter --list
/Users/<you>/.groovy:
dark.nanorctheme
light.nanorctheme
groovy> /highlighter --switch=light.nanorctheme
groovy> /highlighter --refresh
```

Theme switching operates on user-config copies of jnanorc (used by the line reader and most commands) **and** jlessrc (used by /less); if a file is missing, that path silently no-ops on --switch. So a partial setup with only jnanorc will switch the prompt highlighting but leave /less permanently on whichever theme the classpath copy of jlessrc ships with.

Once you create user-config copies, JLine resolves both the theme directive and the include .nanorc entries *relative to the user-config directory* — the classpath copies are no longer consulted. So you also need the per-language syntax files (groovy.nanorc, java.nanorc, json.nanorc, xml.nanorc, yaml.nanorc, etc.) in the same directory, or you'll see the theme palette loaded but no actual highlighting (every file rendered as plain text). Easiest setup is to copy the entire nanorc/ resource directory in one go:

```
cp <groovysh-src>/src/main/resources/nanorc/* ~/.groovy/
```

/history

Display, manage and recall edit-line history. The /history command has numerous options which let you list (with various options), save, read, and clear the edit-line history.

```
groovy> /history -?
history -  list history of commands
Usage: history [-dnrfEie] [-m match] [first] [last]
       history -ARWI [filename]
       history -s [old=new] [command]
       history --clear
       history --save
  -? --help                       Displays command help
```

```
      --clear                    Clear history
      --save                     Save history
   -m match                      If option -m is present the first argument is taken
as a pattern
                                 and only the history events matching the pattern will
be shown
   -d                            Print timestamps for each event
   -f                            Print full time date stamps in the US format
   -E                            Print full time date stamps in the European format
   -i                            Print full time date stamps in ISO8601 format
   -n                            Suppresses command numbers
   -r                            Reverses the order of the commands
   -A                            Appends the history out to the given file
   -R                            Reads the history from the given file
   -W                            Writes the history out to the given file
   -I                            If added to -R, only the events that are not
contained within the internal list are added
                                 If added to -W or -A, only the events that are new
since the last incremental operation
                                 to the file are added
   [first] [last]                These optional arguments may be specified as a number
or as a string. A negative number
                                 is used as an offset to the current history event
number. A string specifies the most
                                 recent event beginning with the given string.
   -e                            Uses the nano editor to edit the commands before
executing
   -s                            Re-executes the command without invoking an editor
```

Here is an example of using the `/history` command:

```
groovy> /history
    0  /clear
    1  /load pigs3.groovy
    2  /types Material
    3  /types -d Material
    4  enum Material { STRAW, STICKS, BRICKS, VIBRANIUM }
    5  /nano
    6  /types
    7  /imports
    8  /save -o pigs4.groovy
    9  wolf = new Wolf()
   10  housePiggy4 = new House(VIBRANIUM)
   11  wolf.threaten()
   12  wolf.eat(housePiggy4)
   13  /history
groovy> /history -d -m *[Pp]ig*
    1  18:12:34  /load pigs3.groovy
    8  18:13:45  /save -o pigs4.groovy
   10  18:14:12  housePiggy4 = new House(VIBRANIUM)
   12  18:14:36  wolf.eat(housePiggy4)
groovy> /history -s 12
wolf.eat(housePiggy4)
The wolf remains hungry.
```

import

Add a custom import which will be included for all shell evaluations.

```
groovy> import java.util.concurrent.BlockingDeque
```

Module imports (JEP 476) are also supported, which import all exported packages from a named module:

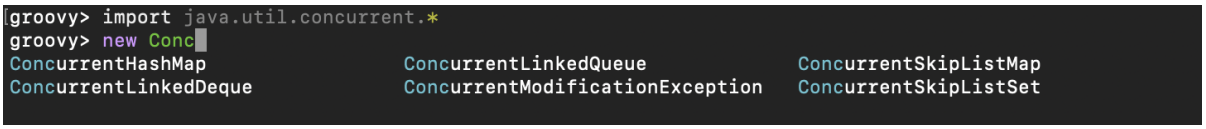
```
groovy> import module java.sql
groovy> Connection.name
===> java.sql.Connection
```

This command can be given at any time to add new imports.

Completion is available and prompts a level at a time using the package structure of all known classes.

```
groovy> import java.util.concurrent.<TAB>
others
atomic                          locks
Classes
AbstractExecutorService         ConcurrentSkipListMap           ForkJoinPool
ArrayBlockingQueue              ConcurrentSkipListSet           ForkJoinTask
...
```

Once an import statement has been executed, relevant classes will become available for completion:



/imports

You can use this to list and delete existing imports.

```
groovy> /imports
import java.util.concurrent.BlockingQueue
```

/img

The `/img` command displays an image inline using JLine's terminal-graphics support (Sixel, Kitty, or iTerm2 protocols, auto-detected). The argument is either a local file path, an `http(s)://` URL, or a Groovy variable reference using the standard `$` syntax.

```
groovy> /img chart.png
```

```
groovy> /img --width=80 https://example.com/diagram.png
groovy> img = new java.awt.image.BufferedImage(200, 100, 1)  // some BufferedImage
groovy> /img $img
```

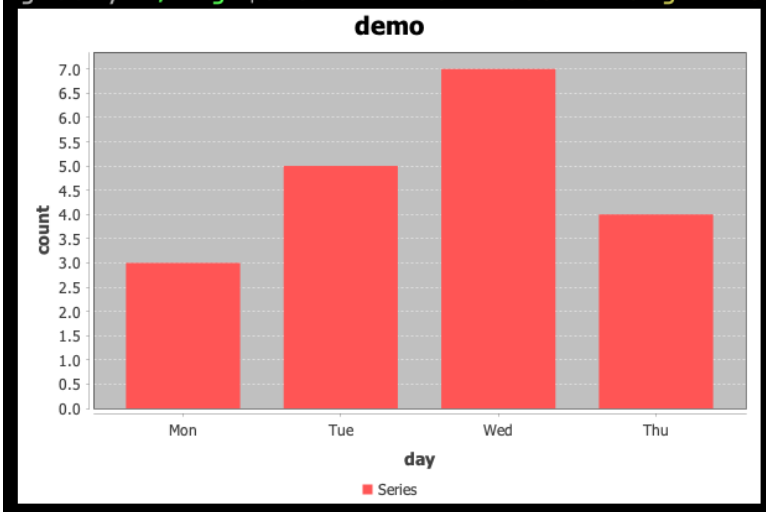
When the argument resolves to a Groovy value (rather than a path), `/img` will accept any of:

- a `java.awt.image.BufferedImage` — used as-is
- a `java.awt.image.RenderedImage` — drawn into a `BufferedImage`
- anything with a `createBufferedImage(int, int)` method (e.g. `org.jfree.chart.JFreeChart`) — duck-typed, no compile-time dependency
- anything with a `toBufferedImage(int, int)` method (e.g. Smile's `smile.plot.swing.Figure`) — duck-typed sibling for libraries that follow the `to⋯` rather than `create⋯` naming convention
- a `javax.swing.JComponent` (e.g. Smile's `Canvas` and `MultiFigurePane`) — laid out and painted to a `BufferedImage` at the requested size

So once you've grabbed JFreeChart you can render charts in the REPL without saving to a file:

```
groovy> /grab org.jfree:jfreechart:1.5.6
groovy> import org.jfree.data.category.DefaultCategoryDataset
groovy> ds = new DefaultCategoryDataset()
groovy> ds.addValue(3d, 'Series', 'Mon'); ds.addValue(5d, 'Series', 'Tue')
groovy> ds.addValue(7d, 'Series', 'Wed'); ds.addValue(4d, 'Series', 'Thu')
groovy> import org.jfree.chart.ChartFactory
groovy> chart = ChartFactory.createBarChart('demo', 'day', 'count', ds)
groovy> /img $chart --width=600 --height=400
```

```
groovy> /grab org.jfree:jfreechart:1.5.6
groovy> import org.jfree.data.category.DefaultCategoryDataset
groovy> ds = new DefaultCategoryDataset()
groovy> ds.addValue(3d, 'Series', 'Mon'); ds.addValue(5d, 'Series', 'Tue')
groovy> ds.addValue(7d, 'Series', 'Wed'); ds.addValue(4d, 'Series', 'Thu')
groovy> import org.jfree.chart.ChartFactory
groovy> chart = ChartFactory.createBarChart('demo', 'day', 'count', ds)
groovy> /img $chart --width=600 --height=400
```



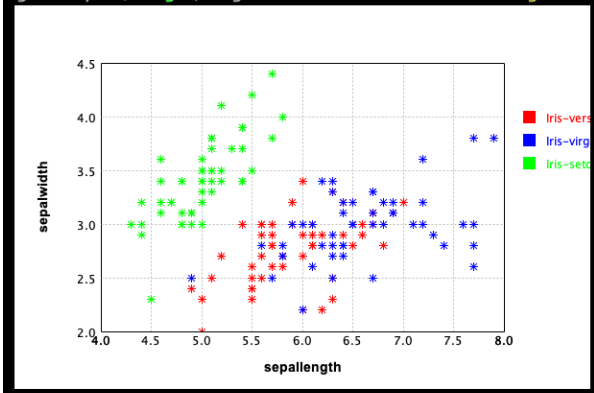

```
groovy>
```

Smile is similar — `Figure.toBufferedImage(int, int)` matches the second duck-type, so a fresh `Figure` can be handed straight to `/img`. The following draws the classic Iris scatter plot:

```
groovy> /grab com.github.haifengl:smile-io:3.1.0
groovy> import smile.io.Read
groovy> iris = Read.arff('iris.arff')
groovy> import smile.plot.swing.ScatterPlot
groovy> figure = ScatterPlot.of(iris, 'sepallength', 'sepalwidth', 'class', '*' as
char).figure()
groovy> figure.setAxisLabels('sepallength', 'sepalwidth')
groovy> /img $figure --width=600 --height=400
```

```
groovy> /grab com.github.haifengl:smile-plot:6.1.0
groovy> import smile.io.Read
groovy> val iris = Read.arff('iris.arff')
+---+-----------+----------+-----------+----------+-----------+
|   |sepallength|sepalwidth|petallength|petalwidth|      class|
+---+-----------+----------+-----------+----------+-----------+
|  0|        5.1|       3.5|        1.4|       0.2|Iris-setosa|
|  1|        4.9|         3|        1.4|       0.2|Iris-setosa|
|  2|        4.7|       3.2|        1.3|       0.2|Iris-setosa|
|  3|        4.6|       3.1|        1.5|       0.2|Iris-setosa|
|  4|          5|       3.6|        1.4|       0.2|Iris-setosa|
|  5|        5.4|       3.9|        1.7|       0.4|Iris-setosa|
|  6|        4.6|       3.4|        1.4|       0.3|Iris-setosa|
|  7|          5|       3.4|        1.5|       0.2|Iris-setosa|
|  8|        4.4|       2.9|        1.4|       0.2|Iris-setosa|
|  9|        4.9|       3.1|        1.5|       0.1|Iris-setosa|
+---+-----------+----------+-----------+----------+-----------+
140 more rows...

groovy> import smile.plot.swing.ScatterPlot
groovy> figure = ScatterPlot.of(iris, 'sepallength', 'sepalwidth', 'class', '*' as char).figure()
groovy> figure.setAxisLabels('sepallength', 'sepalwidth')
smile.plot.swing.Figure@7be94cd6
groovy> /img $figure --width=600 --height=400
```




```
groovy>
```

Smile's `Canvas` and `MultiFigurePane` are themselves `JComponent` subclasses, so the SPLOM (scatter-plot matrix) goes through the `JComponent` path with no extra wrapping:

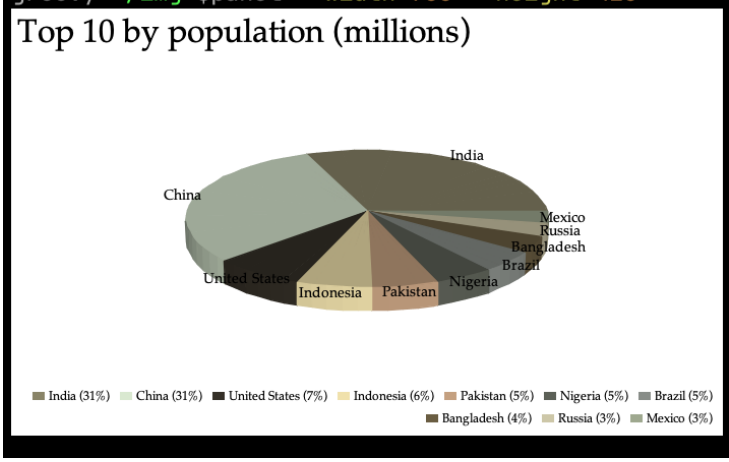
```
groovy> import smile.plot.swing.MultiFigurePane
groovy> splom = MultiFigurePane.splom(iris, '*' as char, 'class')
groovy> /img $splom --width=900 --height=900
```

Orson Charts gives 3D chart shapes (pie, bar, scatter, surface, …). `Chart3D` doesn't expose `createBufferedImage` directly, but pairs with `Chart3DPanel` (a `JComponent`) which reaches `/img` through the same JComponent path as Smile's SPLOM. Pairing that with Apache Commons CSV — handed a `population.csv` whose columns are `country` and `millions` — gives a 3D pie of the world's ten most populous countries:

```
groovy> /grab org.apache.commons:commons-csv:1.14.1
groovy> import static org.apache.commons.csv.CSVFormat.RFC4180 as CSV
groovy> parser = CSV.builder().setHeader().setSkipHeaderRecord(true).build()
groovy> records = parser.parse(new FileReader('population.csv'))
groovy> /grab org.jfree:org.jfree.chart3d:2.1.1
```

```
groovy> import module org.jfree.chart3d
groovy> data = new StandardPieDataset3D()
groovy> records.each { data.add(it.country, it.millions as double) }
groovy> chart = Chart3DFactory.createPieChart('Top 10 by population (millions)', null,
data)
groovy> panel = new Chart3DPanel(chart)
groovy> /img $panel --width=700 --height=420
```

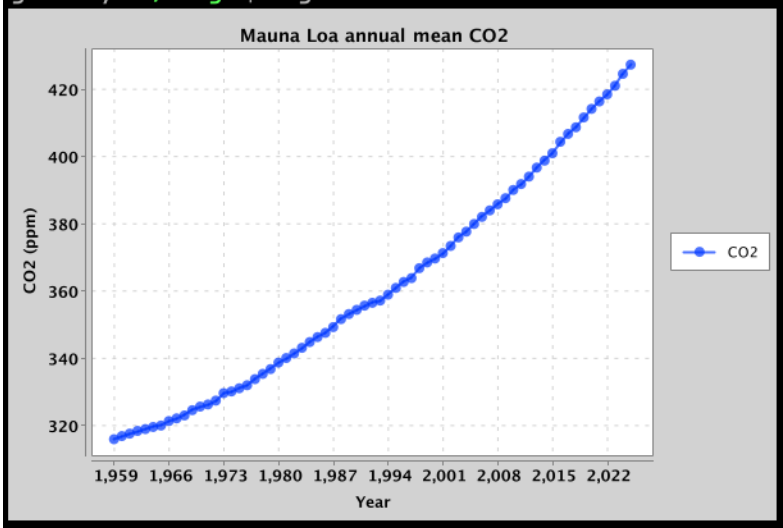
```
groovy> /grab org.apache.commons:commons-csv:1.14.1
groovy> import static org.apache.commons.csv.CSVFormat.RFC4180 as CSV
groovy> parser = CSV.builder().setHeader().setSkipHeaderRecord(true).build()
groovy> records = parser.parse(new FileReader('population.csv'))
groovy> /grab org.jfree:org.jfree.chart3d:2.1.1
groovy> import module org.jfree.chart3d
groovy> data = new StandardPieDataset3D()
groovy> records.each { data.add(it.country, it.millions as double) }
org.apache.commons.csv.CSVParser@4168f3d9
groovy> chart = Chart3DFactory.createPieChart('Top 10 by population (millions)', null, data)
groovy> panel = new Chart3DPanel(chart)
groovy> /img $panel --width=700 --height=420
```



XChart's `BitmapEncoder.getBufferedImage(chart)` yields a plain `BufferedImage`, which is the simplest path of all — assign it to a variable and hand it straight to `/img`. Combining that with the `/slurp` command (which prefers `groovy-csv`'s `CsvSlurper` and falls back to Apache Commons CSV) gives a declarative CSV-to-chart pipeline. Reading a `co2-mlo.csv` whose columns are `Year` and `Mean` (annual mean Mauna Loa CO2 in ppm) draws the iconic Keeling-curve line:

```
groovy> /grab org.knowm.xchart:xchart:3.8.8
groovy> rows = /slurp co2-mlo.csv
groovy> import org.knowm.xchart.XYChartBuilder
groovy> chart = new XYChartBuilder().title('Mauna Loa annual mean
CO2').xAxisTitle('Year').yAxisTitle('CO2 (ppm)').width(600).height(400).build()
groovy> chart.addSeries('CO2', rows.collect { it.Year as int }, rows.collect { it.Mean
as double })
groovy> import org.knowm.xchart.BitmapEncoder
groovy> img = BitmapEncoder.getBufferedImage(chart)
groovy> /img $img
```

```
groovy> /grab org.knowm.xchart:xchart:3.8.8
groovy> rows = /slurp co2-mlo.csv
groovy> import org.knowm.xchart.XYChartBuilder
groovy> chart = new XYChartBuilder().title('Mauna Loa annual mean CO2')\
      > .xAxisTitle('Year').yAxisTitle('CO2 (ppm)')\
      > .width(600).height(400).build()
groovy> chart.addSeries('CO2', rows.collect { it.Year as int },
add: )>   rows.collect { it.Mean as double })
org.knowm.xchart.XYSeries@7b5ac347
groovy> import org.knowm.xchart.BitmapEncoder
groovy> img = BitmapEncoder.getBufferedImage(chart)
groovy> /img $img
```




```
groovy>
```

The `--width` and `--height` options have two distinct meanings depending on the input. For raw-pixel inputs (file path, `http(s)://` URL, `BufferedImage`, `RenderedImage`), they control the rendered size in **terminal character cells**. For inputs that generate the image at the requested size (`createBufferedImage(int,int)`, `toBufferedImage(int,int)`, `JComponent`), they are **source-image pixels** — the terminal then renders the image at its natural fit. Aspect ratio is preserved by default (override with `--no-preserve-aspect-ratio`).

If the active terminal doesn't speak any of the supported graphics protocols (common cases: macOS Terminal.app, VS Code's built-in terminal, JetBrains terminals, plain Windows console), `/img` prints a `[image: WxH, label]` summary line instead. Pass `--gui` to open a Swing window with the image regardless of terminal capability.

The command requires the `java.desktop` module at runtime; on a JVM where that module is unavailable, `/img` is not registered.

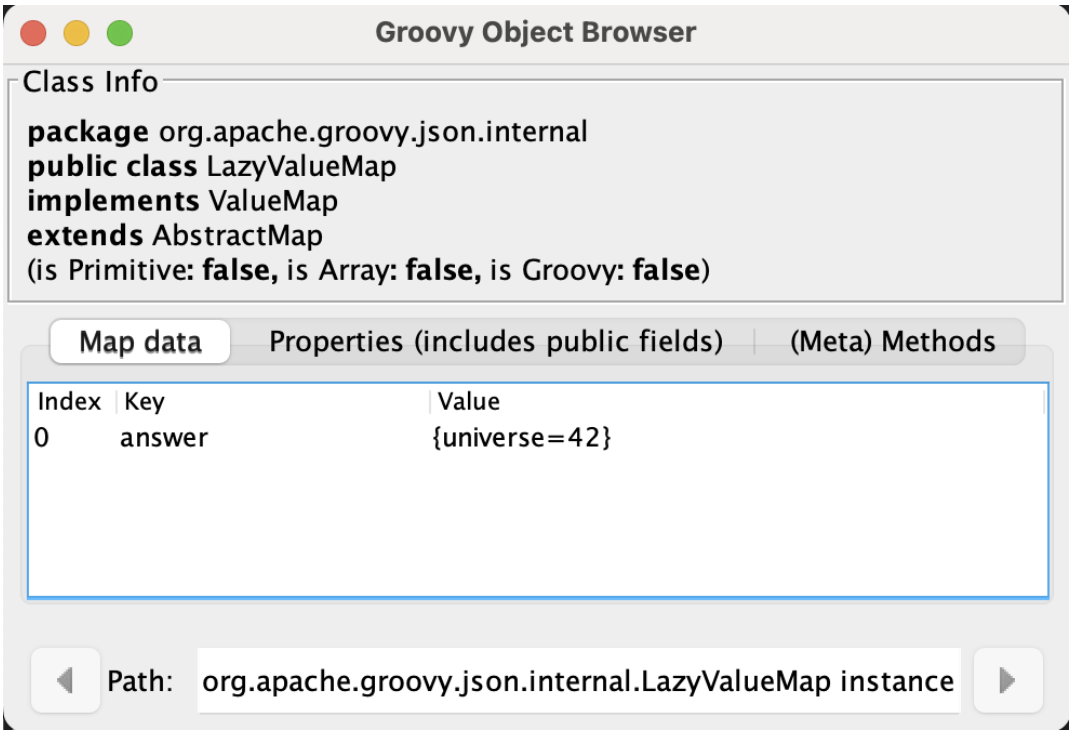
`/inspect`

Display or browse object info on the terminal or object browser.

```
groovy> json = /slurp answers.json
groovy> /inspect -?
/inspect -  display/browse object info on terminal/object browser
Usage: /inspect [OPTIONS] OBJECT
  -? --help                    Displays command help
  -g --gui                     Launch object browser
  -i --info                    Object class info
  -m --methods                 List object methods
  -n --metaMethods             List object metaMethods

groovy> /inspect $json
propertyInfo
    [GROOVY, public, n/a, Map, map, ['answer':['universe':42]]]
    [GROOVY, public, n/a, [Ljava.util.Map$Entry;, items, null]
    [GROOVY, public, n/a, int, len, 0]
    [GROOVY, public, n/a, boolean, lazyChop, true]
    [GROOVY, public, n/a, boolean, mapChopped, false]
classProps
    package org.apache.groovy.json.internal
    public class LazyValueMap
    implements ValueMap
    extends AbstractMap
    is Primitive: false, is Array: false, is Groovy: false
publicFields []
groovy> /inspect -g $json
groovy>
```

Using the `--gui/-g` option displays the object in Groovy's object browser:



`/less`

Display the contents of a file (usually a page at a time). Formatting of common file types is supported.

```
groovy> /less --help
less -  file pager
Usage: less [OPTIONS] [FILES]
  -? --help                    Show help
  -e --quit-at-eof             Exit on second EOF
  -E --QUIT-AT-EOF             Exit on EOF
  -F --quit-if-one-screen      Exit if entire file fits on first screen
  -q --quiet --silent          Silent mode
  -Q --QUIET --SILENT          Completely silent
  -S --chop-long-lines         Do not fold long lines
  -i --ignore-case             Search ignores lowercase case
  -I --IGNORE-CASE             Search ignores all case
  -x --tabs=N[,...]            Set tab stops
  -N --LINE-NUMBERS            Display line number for each line
  -Y --syntax=name             The name of the syntax highlighting to use.
     --no-init                 Disable terminal initialization
     --no-keypad               Disable keypad handling
     --ignorercfiles           Don't look at the system's lessrc nor at the user's lessrc.
  -H --historylog=name         Log search strings to file, so they can be retrieved in later sessions
groovy> /less -F answers.json
{
    "answer": {
        "universe":
        42
    }
}
groovy> /less -F answers.xml
<?xml version="1.0" encoding="UTF-8" ?>
<answers>
    <answer question="universe">42</answer>
</answers>
groovy> /less -F answers.yaml
answer:
  universe: 42
groovy> /less -F answers.toml
[answer]
universe = 42
groovy> /less -F answers.properties
# answers
universe=42
groovy> /less -F answers.csv
"Question","Answer"
universe,42
groovy> /less -F -N answers.groovy
      1 def answers = [universe: 42]
      2 println "The answer to the ultimate question is: ${answers.universe}"
groovy> answers.groovy
The answer to the ultimate question is: 42
```

If no filename is given, the contents of the current buffer are displayed.

`/load`

Load a file into the buffer.

If no filename is given as an argument, the current shared variables are loaded from the `.groovy/groovysh.ser` file in the user home directory.

`/ls`

The `/ls` command lists files.

```
groovy> /ls subprojects
                     groovy-dateutil      groovy-swing
..                   groovy-docgenerator  groovy-templates
binary-compatibility groovy-ginq          groovy-test
groovy-all           groovy-groovydoc     groovy-test-junit5
groovy-ant           groovy-groovysh      groovy-testng
groovy-astbuilder    groovy-jmx           groovy-toml
groovy-binary        groovy-json          groovy-typecheckers
groovy-bom           groovy-jsr223        groovy-xml
```

```
groovy-cli-commons   groovy-macro         groovy-yaml
groovy-cli-picocli   groovy-macro-library performance
groovy-console       groovy-nio           stress
groovy-contracts     groovy-servlet       tests-preview
groovy-datetime      groovy-sql
```

Wildcard support is available, so you can use `/ls *.groovy` to list all Groovy files in the current directory. File patterns which involve asterisks as well as slashes can sometimes be confused with Groovy's comment syntax, so quote the pattern if in doubt:

```
groovy> /ls '**/bu*.gradle'
bootstrap/build.gradle                          subprojects/groovy-datetime/build.gradle
subprojects/groovy-swing/build.gradle
build-logic/build.gradle                        subprojects/groovy-dateutil/build.gradle
subprojects/groovy-templates/build.gradle
build.gradle                                    subprojects/groovy-
docgenerator/build.gradle  subprojects/groovy-test-junit5/build.gradle
gradle/build-scans.gradle                       subprojects/groovy-ginq/build.gradle
subprojects/groovy-test/build.gradle
subprojects/binary-compatibility/build.gradle subprojects/groovy-
groovydoc/build.gradle     subprojects/groovy-testng/build.gradle
subprojects/groovy-all/build.gradle             subprojects/groovy-groovysh/build.gradle
subprojects/groovy-toml/build.gradle
subprojects/groovy-ant/build.gradle             subprojects/groovy-jmx/build.gradle
subprojects/groovy-typecheckers/build.gradle
subprojects/groovy-astbuilder/build.gradle      subprojects/groovy-json/build.gradle
subprojects/groovy-xml/build.gradle
subprojects/groovy-binary/build.gradle          subprojects/groovy-jsr223/build.gradle
subprojects/groovy-yaml/build.gradle
subprojects/groovy-bom/build.gradle             subprojects/groovy-macro-
library/build.gradle subprojects/performance/build.gradle
subprojects/groovy-cli-commons/build.gradle     subprojects/groovy-macro/build.gradle
subprojects/stress/build.gradle
subprojects/groovy-cli-picocli/build.gradle     subprojects/groovy-nio/build.gradle
subprojects/tests-preview/build.gradle
subprojects/groovy-console/build.gradle         subprojects/groovy-servlet/build.gradle
subprojects/groovy-contracts/build.gradle       subprojects/groovy-sql/build.gradle
```

/nano

Edit files or the current buffer.

The `/nano` command has numerous options:

```
groovy> /nano --help
nano -  edit files
Usage: nano [OPTIONS] [FILES]
  -? --help                  Show help
  -B --backup                When saving a file, back up the previous version of it, using the current filename
                             suffixed with a tilde (~).
  -I --ignorercfiles         Don't look at the system's nanorc nor at the user's nanorc.
  -Q --quotestr=regex        Set the regular expression for matching the quoting part of a line.
  -T --tabsize=number        Set the size (width) of a tab to number columns.
  -U --quickblank            Do quick status-bar blanking: status-bar messages will disappear after 1 keystroke.
  -c --constantshow          Constantly show the cursor position on the status bar.
  -e --emptyline             Do not use the line below the title bar, leaving it entirely blank.
  -j --jumpyscrolling        Scroll the buffer contents per half-screen instead of per line.
  -l --linenumbers           Display line numbers to the left of the text area.
  -m --mouse                 Enable mouse support, if available for your system.
  -$ --softwrap              Enable 'soft wrapping'.
  -a --atblanks              Wrap lines at whitespace instead of always at the edge of the screen.
  -R --restricted            Restricted mode: don't allow suspending; don't allow a file to be appended to,
                             prepended to, or saved under a different name if it already has one;
                             and don't use backup files.
  -Y --syntax=name           The name of the syntax highlighting to use.
  -z --suspend               Enable the ability to suspend nano using the system's suspend keystroke (usually ^Z).
  -v --view                  Don't allow the contents of the file to be altered: read-only mode.
  -k --cutfromcursor         Make the 'Cut Text' command cut from the current cursor position to the end of the line
  -t --tempfile              Save a changed buffer without prompting (when exiting with ^X).
  -H --historylog=name       Log search strings to file, so they can be retrieved in later sessions
  -E --tabstospaces          Convert typed tabs to spaces.
  -i --autoindent            Indent new lines to the previous line's indentation.
```

You can use the `/nano` command to edit files or the current buffer:

```
  JLine Nano 3.0.0
enum Material { STRAW, STICKS, BRICKS, VIBRANIUM }

import static Material.*

record House(Material material) { }

interface Eater {
  void eat(House h)
}

@interface Sneaky {}

trait HuffsAndPuffs {
  void threaten() {
    println "I'll huff and puff and blow your house down!"
  }
}

@Sneaky
class Wolf implements HuffsAndPuffs, Eater {
  void eat(House h) {
    if (h.material < BRICKS) println "The wolf gobbles them up!"
    else println "The wolf remains hungry."
  }
}

^G Get Help          ^O WriteOut          ^R Read File
^X Exit              ^J Justify           ^W Where Is
```

If editing the current buffer, when you exit and then save, the buffer will be reloaded with the edited contents.

`/pipe`

The `/pipe` command lets you create and delete custom pipe operators.

There are several builtin pipe and pipe-like operators:

- The *pipe and* operator (|&&) executes the right hand side only if the execution of the left hand

side succeeds

- The *pipe or* operator (`|||`) executes the right hand side only if the execution of the left hand side fails
- The *pipe flip* operator (`|;`) flips the left and right hand side expressions
- The *output redirect* operator (`|>`) redirects the output to a file
- The *append output redirect* operator (`|>>`) redirects the output by appending to a file

For the pipe *and* and *or* operators, *succeeds* means executing without throwing an exception and returning a non-null, non-empty, non-zero result, i.e. `true` according to *Groovy truth*.

Here are some examples for *and* and *or*:

```
groovy> x = 0 ||| x = 42
groovy> /show x
x 42
groovy> x = 0 |&& x = 42
groovy> /show x
x 0
groovy> x = 42 |&& x = 0
groovy> /show x
x 0
groovy> 1/0 ||| 42
ArithmeticException: Division by zero
42
groovy> 42 ||| 1/0
groovy> 1/0 |&& 42
ArithmeticException: Division by zero
groovy> x = 5 |&& x++ |&& ++x
7
groovy>
```

Pipe operators are handled by the shell. The left and right hand sides will be either repl commands or code passed to the Groovy engine for execution.

The pipe *flip* operator is most commonly used with commands:

```
groovy> /widget -l |; /prnt -s JSON
[
    "_autosuggest-end-of-line (_autosuggest-end-of-line)",
    "_autosuggest-forward-char (_autosuggest-forward-char)",
    ...
]
```

```
groovy> /ls --1 subprojects |; /head -n12 |; /tail -n5
groovy-bom
groovy-cli-commons
groovy-cli-picocli
groovy-console
groovy-contracts
```

It is less common, but it can also be used with code:

```
groovy> 4 |; 8 -
4
groovy> 4 |; 8 *
32
groovy> 4 |; 8 /
2
groovy> 4 |; 8 +
12
groovy> 4 |; 8 %
0
groovy> 4 |; 8 **
4096
```

You can also define your own custom pipe operators, defined using `/pipe [OPERATOR] [PREFIX] [POSTFIX]`, as these examples show:

```
groovy> /pipe |.  '.collect{' '}'
groovy> /pipe |:  '.collectEntries{' '}'
groovy> /pipe |?  '.findAll{' '}'
groovy> /pipe |#  '.take(' ')'
groovy> /pipe --list
 |. [.collect{, }]
 |? [.findAll{, }]
 |# [.take(, )]
 |: [.collectEntries{, }]
groovy> fruit=/slurp fruit.txt
groovy> /show fruit
 fruit [apple, banana, cherry, date, elderberry]
groovy> fruit |. it.size()
 [5, 6, 6, 4, 10]
groovy> fruit |: [it, it.size()]
 [apple:5, banana:6, cherry:6, date:4, elderberry:10]
groovy> fruit |? it.size() < 6
 [apple, date]
groovy> fruit |# 3
 [apple, banana, cherry]
groovy>
```

JLine functionality includes support for *named pipes* and *pipe aliases*.

Here is a named pipe for finding fruit with small names:

```
groovy> /pipe smallFruit '.findAll{ it.size() <' '}'
groovy> fruit | smallFruit 6
[apple, date]
```

Here is a pipe alias:

```
groovy> /alias sizes '|. it.size()'
```

```
groovy> fruit | sizes
[5, 6, 6, 4, 10]
```

The pipe alias always has the pipe it is aliasing at the start, `|.` in the above example.

NOTE

Care should be taken when using named pipes along with Groovy code that uses Groovy's pipe operator (`|`), AKA binary OR operator, as this can lead to conflicts.

Is the expression `x | y` the binary OR of two variables, or making use of a named pipe with name `y`?

Spaces in Groovy are normally not significant, but the Groovy Shell is using JLine console functionality which requires spaces around the pipe operator before a named pipe and is bracket aware. So, `x| y`, `x |y`, `x|y`, and `(x | y)` will be passed to Groovy for execution. The Groovy Shell also passes `x | y` to Groovy if `y` is not a known named pipe.

```
groovy> x = 4
groovy> y = 8
groovy> x | y
12
groovy> /pipe y '+1' ''
groovy> x | y
5
```

Getting back to how you can avoid conflicts, one convention you could use to avoid conflicts is to use brackets or avoid spaces when wanting the binary OR operator, and use spaces and no brackets when wanting to use a named pipe.

Another alternative, is to define a custom pipe name instead of a named pipe:

```
groovy> /pipe |smallFruit '.findAll{ it.size() <' '}'
groovy> fruit |smallFruit 6
[apple, date, fig, grape]
groovy> /pipe
smallFruit  [.findAll{ it.size() <, }]
|smallFruit [.findAll{ it.size() <, }]
```

Here, the custom pipe named `|smallFruit`, is different to the named pipe `smallFruit`. The former mustn't have a space, the latter must have one. If you use only the first of these, you can have a convention of using spaces when you want a binary OR.

As a final example, let's use a custom pipe with the output redirection operators to find fruit with large names:

```
groovy> fruit=/slurp fruit.txt
groovy> /show fruit
fruit [apple, banana, cherry, date, elderberry, fig, grape, huckleberry]
groovy> /pipe |? '.findAll{' '}.each{ println it }'
groovy> fruit |? it.size() > 6 |> bigFruit.txt
[elderberry, huckleberry]
groovy> println 'watermelon' |>> bigFruit.txt
groovy> /cat bigFruit.txt
elderberry
huckleberry
watermelon
```

/prnt

The /prnt command outputs its argument to the console. Both variable expansion and formatting are supported.

```
groovy> answer = [universe: 42]
groovy> /prnt answer
answer
groovy> /prnt $answer
universe 42
groovy> /prnt ${answer}
universe 42
groovy> /prnt ${answer}.universe
42
groovy> /prnt ${answer.universe}
42
groovy> /prnt ${answer.universe / 7}
6
groovy> /prnt ${answer.keySet()[0].capitalize()}
Universe
groovy> /prnt -s JSON '["answer":["universe", 42]]'
{
    "answer": [
        "universe",
        42
    ]
}
```

See also the /echo command, which is similar but takes multiple arguments. It also supports variable expansion but doesn't support formatting.

/pwd

The /pwd command prints the current working directory.

```
groovy> /pwd
/Users/paulk/Projects/groovy
```

/reset

Clears the current buffer and shared variables.

/save

Saves the buffer's contents to a file.

If no filename is given as an argument, the current shared variables are saved into the .groovy/groovysh.ser file in the user home directory.

/setopt

Set options.

```
groovy> /setopt -?
setopt -  set options
Usage: setopt [-m] option ...
setopt
-? --help                       Displays command help
-m                              Use pattern matching
```

If no option is given, the current set options are displayed.

```
groovy> /setopt
disable-event-expansion
use-forward-slash
insert-bracket
no-empty-word-options
groovy>
```

See also `/unsetopt` to unset options.

/setvar

Set linereader variable values.

```
groovy> /setvar
history-file: /Users/paulk/.groovy/groovysh_history
indentation: 2
list-max: 100
secondary-prompt-pattern: %M%P >
groovy>
```

/show

Show the shared variables (the binding). These include your shared variables as well as a few used for configuring various shell features.

```
groovy> /show
CONSOLE_OPTIONS  [docs:[:]]
ROOT             jar:file:/Users/paulk/Projects/gr
nanorcValue      jar:file:/Users/paulk/Projects/gr
GROOVYSH_OPTIONS [interpreterMode:true]
groovy> x = 3.0
groovy> y = 4.0
groovy> z = Math.sqrt(x ** 2 + y ** 2)
groovy> /show
CONSOLE_OPTIONS  [docs:[:]]
ROOT             jar:file:/Users/paulk/Projects/gr
x                3.0
y                4.0
z                5.0
nanorcValue      jar:file:/Users/paulk/Projects/gr
GROOVYSH_OPTIONS [interpreterMode:true]
```

`/slurp`

Slurp files to shared variables. Groovy has a bunch of slurpers for various formats like XML, JSON, YAML, etc. You can use those in your code if you like, but the `/slurp` command can be a convenience shortcut. It supports most of the common formats, including JSON, XML, YAML, CSV, TOML, Markdown and property files.

```
groovy> csv = /slurp answers.csv
groovy> assert csv.find{ row -> row.Question == 'universe' }.Answer == '42'
groovy> json = /slurp answers.json
groovy> assert json.answer.universe == 42
groovy> props = /slurp answers.properties
groovy> assert props.universe == '42'
groovy> toml = /slurp answers.toml
groovy> assert toml.answer.universe == 42
groovy> xml = /slurp answers.xml
groovy> assert xml.answer.@question == ['universe'] && xml.answer.text() == '42'
groovy> yml = /slurp answers.yaml
groovy> assert yml.answer.universe == 42
groovy>
groovy> /slurp -?
/slurp -  slurp file or string variable context to object
Usage: /slurp [OPTIONS] file|variable
  -? --help                  Displays command help
  -e --encoding=ENCODING     Encoding (default UTF-8)
  -f --format=FORMAT         Serialization format
```

As you can see in the usage information at the end of the above image, you can also provide an encoding and a format. If no encoding is given, `UTF_8` is the default. If no format is given, the shell will try to determine the format from the extension of the file (if given). If no format or file extension is given, the `/slurp` command makes a rudimentary attempt to determine the format by looking at the content, e.g. the json format is determined in the following example:

```
groovy> json = /slurp https://api.github.com/repos/apache/groovy
groovy> /prnt ${json.license}
name    Apache License 2.0
spdx_id Apache-2.0
key     apache-2.0
url     https://api.github.com/licenses/apache-2.0
node_id MDc6TGljZW5zZTI=
groovy> /q
```

That content-based fallback **parses**; it never evaluates. It recognises JSON and otherwise hands back the text unchanged, so a document cannot decide for itself that it should be run. Evaluating content is a separate, explicit request: pass `-f GROOVY`, or name a local `.groovy` file.

For a URL, the extension of the URL path selects the format in the same way as a filename does, but only ever a data format — `GROOVY` is not inferred from an address, because the extension is part of the URL you typed while the bytes are whatever the server chooses to return.

WARNING	`-f GROOVY` against a URL runs whatever that server returns, with your privileges, in this session. It is not a safer way to evaluate remote content than the old content sniffing was — it is the **authorised** way, and the difference is only that you chose it rather than the payload choosing for you. Redirects are followed, so the address you typed is not necessarily where the code came from; `/slurp` prints the URL it was actually served from before evaluating it. Treat it as you would piping a download straight into a shell, and prefer fetching the content first (without `-f`) so you can read it before deciding to run it.

Although the details of the exact object returned are an implementation detail which may change in the future, the current behavior is as follows:

Format	Notes
JSON	Uses JsonSlurper, returns a lazy Map.
TOML	Uses TomlSlurper, returns a lazy Map.
YAML	Uses YamlSlurper, returns a lazy Map.
XML	Uses XmlParser, returns a Node.
PROPERTIES	Returns a Properties file, which is a Map-like object.
CSV	Uses Apache Commons CSV if on the classpath. Assumes a header row which is used to create a list (the rows) of maps from column name to value.
MARKDOWN	Uses MarkdownSlurper from `groovy-markdown` if on the classpath. Returns a `MarkdownDocument` (an `Iterable<Map<String, Object>>` of structured elements). Triggered by `.md` and `.markdown` extensions.
TEXT	Reads the file as raw lines (or argument as a line).
GROOVY	**Evaluates** the content as a Groovy script and returns its result. Selected by `-f GROOVY`, or by a local `.groovy` file extension. Never selected automatically from content, nor from a URL.
NONE	If you want the raw text rather than parsed content.

If these do change in the future, the replacement will be a compatible implementation, that follows the normal GPath conventions.

If you want more advanced control over the slurping, you can use Groovy's "Slurper" classes directly, e.g. `new XmlSlurper().parseText(…)`.

`/sort`

The `/sort` command sorts files or standard input.

```
groovy> /sort --help
/sort -  writes sorted standard input to standard output.
Usage: /sort [OPTIONS] [FILES]
  -? --help                    show help
  -f --ignore-case             fold lower case to upper case characters
  -r --reverse                 reverse the result of comparisons
  -u --unique                  output only the first of an equal run
  -t --field-separator=SEP     use SEP instead of non-blank to blank tra
  -b --ignore-leading-blanks   ignore leading blancks
     --numeric-/sort            compare according to string numerical va
  -k --key=KEY                 fields to use for sorting separated by wh
groovy> /sort -r fruit.txt
elderberry
date
cherry
banana
apple
```

/tail

The /tail command displays the end of a file.

```
groovy> /tail -n2 fruit.txt
date
elderberry
```

/types

Show the declared types (enums, interfaces, classes, traits, annotation definitions, and records).

```
groovy> /imports
import static Material.*
groovy> /types
enum Material { STRAW, STICKS, BRICKS, VIBRANIUM }

record House(Material material) { }

interface Eater {
  void eat(House h)
}

@interface Sneaky {}

trait HuffsAndPuffs {
  void threaten() {
    println "I'll huff and puff and blow your house down!"
  }
}

@Sneaky
class Wolf implements HuffsAndPuffs, Eater {
  void eat(House h) {
    if (h.material < BRICKS) println "The wolf gobbles them up!"
    else println "The wolf remains hungry."
  }
}
groovy> housePiggy4 = new House(VIBRANIUM)
groovy> wolf = new Wolf()
groovy> wolf.threaten()
I'll huff and puff and blow your house down!
groovy> wolf.eat(housePiggy4)
The wolf remains hungry.
```

Types can be deleted using `/types -d` (completion is available) but see also the `/del` command:

```
groovy> /types -d
Eater           House           HuffsAndPuffs   Material        Sneaky          Wolf
```

/ttop

Display information about threads.

```
ttop - 18:57:23. OS: Mac OS X 15.5, aarch64, 12 cpus.
Classes: 5877 loaded, 0 unloaded, 5877 loaded total. Threads: 14, peak: 21, started: 139.
Memory: heap: 106.0 MiB / 16.0 GiB, non heap: 47.4 MiB.
GC: G1 Young Generation: 7 col. / 0.031 s, G1 Old Generation: 0 col. / 0.000 s.

TID NAME                                    STATE         T-CPU LOCK-OWNER-ID
 69 My Worker Thread 4                      timed_waiting 0.046
 68 My Worker Thread 3                      timed_waiting 0.044
 67 My Worker Thread 2                      timed_waiting 0.046
 66 My Worker Thread 1                      timed_waiting 0.042
 65 My Worker Thread 0                      timed_waiting 0.045
 31 DFA-cache-cleaner[ParserAtnManager]     waiting       0.000
 30 DFA-cache-cleaner[LexerAtnManager]      waiting       0.000
 26 PIC-Cleaner                             waiting       0.000
 13 Common-Cleaner                          timed_waiting 0.001
 12 Notification Thread                     runnable      0.000
  4 Signal Dispatcher                       runnable      0.007
  3 Finalizer                               waiting       0.000
  2 Reference Handler                       runnable      0.000
  1 main                                    runnable      3.797
```

Various options are available:

```
groovy> /ttop --help
ttop -  display and update sorted information about threads
Usage: ttop [OPTIONS]
  -? --help                    Show help
  -o --order=ORDER             Comma separated list of sorting keys
  -t --stats=STATS             Comma separated list of stats to display
  -s --seconds=SECONDS         Delay between updates in seconds
  -m --millis=MILLIS           Delay between updates in milliseconds
  -n --nthreads=NTHREADS       Only display up to NTHREADS threads
```

/unsetopt

Unset options.

```
groovy> /unsetopt -?
unsetopt -  unset options
Usage: unsetopt [-m] option ...
       unsetopt
  -? --help                       Displays command help
  -m                              Use pattern matching
```

If no option is given, the current unset options are displayed.

```
groovy> /unsetopt
complete-in-word
```

```
complete-matcher-camelcase
no-complete-matcher-typo
history-verify
no-history-ignore-space
no-history-ignore-dups
no-history-reduce-blanks
no-history-beep
no-history-incremental
no-history-timestamped
no-auto-group
no-auto-menu
no-auto-list
auto-menu-list
recognize-exact
no-group
group-persist
case-insensitive
list-ambiguous
list-packed
list-rows-first
glob-complete
menu-complete
auto-fresh-line
delay-line-wrap
no-auto-param-slash
no-auto-remove-slash
insert-tab
mouse
disable-highlighter
no-bracketed-paste
erase-line-on-finish
case-insensitive-search
disable-undo
groovy>
```

See also `/setopt` to set options.

/wc

The `/wc` command displays word, line, character, and byte counts.

```
groovy> /wc LICENSE
     262    1754   13493 LICENSE
```

/unalias

Allows aliases to be removed.

```
groovy> /alias
[/cat:/less -F, now:println new Date(), /x:exit, /h:help]
groovy> /unalias /cat now
groovy> /alias
[/x:exit, /h:help]
```

Removed aliases will also be removed from persistent storage.

State

History

History is stored in the file `$HOME/.groovy/groovysh_history`.

Widgets

JLine provides a powerful widget system that lets you extend the functionality of its line reader. A number of builtin widgets are available including `end-of-line`, `beginning-of-line`, `forward-word`, `backward-word`, `kill-word`, `backward-kill-word`, `capitalize-word`, `transpose-words`, and `yank-pop`, just to name a few. You can use the `/keymap` command to see the key bindings for these widgets.

Groovy also includes JLine's *tailtip* and *autosuggest* widget functionality. You can see the related widgets by using the `/widget -l` command, which lists custom widgets.

```
groovy> /widget -l
_autosuggest-end-of-line (_autosuggest-end-of-line)
_autosuggest-forward-char (_autosuggest-forward-char)
_autosuggest-forward-word (_autosuggest-forward-word)
_tailtip-accept-line (_tailtip-accept-line)
_tailtip-backward-delete-char (_tailtip-backward-delete-char)
_tailtip-delete-char (_tailtip-delete-char)
_tailtip-expand-or-complete (_tailtip-expand-or-complete)
_tailtip-kill-line (_tailtip-kill-line)
_tailtip-kill-whole-line (_tailtip-kill-whole-line)
_tailtip-redisplay (_tailtip-redisplay)
_tailtip-self-insert (_tailtip-self-insert)
autosuggest-toggle (autosuggest-toggle)
tailtip-toggle (tailtip-toggle)
tailtip-window (tailtip-window)
```

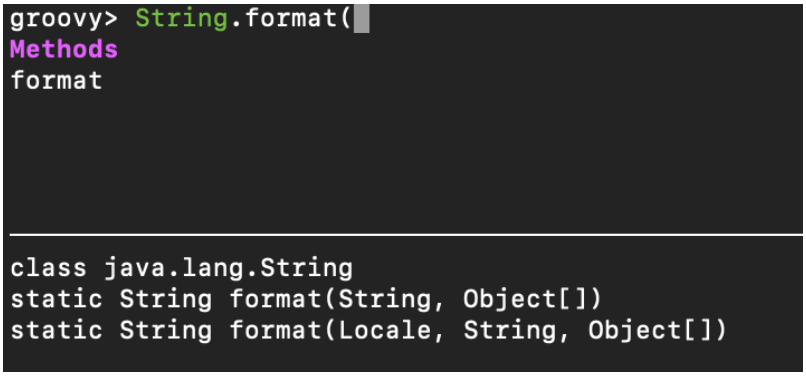
These are available but not enabled by default. You can enable them using the related *toggle* widgets. You can see what key bindings are associated with these widgets by using the `/keymap` command.

```
groovy> /keymap
...
"^[s" tailtip-toggle
"^[v" autosuggest-toggle
...
```

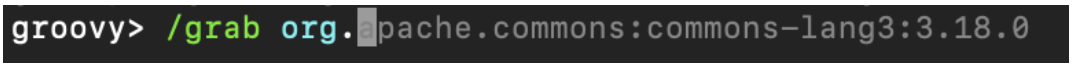
Normally, completions are shown when you hit the 'TAB' key, but with the tailtip widget enabled, you can see completions as you type., as well as additional usage information given in the tailtip window as seen here for a command:



And here for some code:



With the autosuggest widget enabled, you can see suggestions for what to type next as you type, based on your history, as seen here:



You can accept the entire suggestion or a word at a time. Both widgets can be enabled.

Embedding groovysh

Applications that host groovysh — a domain-specific console, for example — should not copy `Main`. The embedding API is `Main.start(GroovyshOptions, String[])`, plus constructors on `GroovyEngine` for compile-time customizers.

Do not put embedding hooks into the binding `Map` accepted by `Main.start(Map, String[])`. That map is only for variables visible to evaluated scripts.

Compiler configuration

`GroovyEngine` always evaluates with an explicit `CompilerConfiguration` (`CompilerConfiguration.DEFAULT` when you pass `null`). Use a **mutable copy** (`new CompilerConfiguration()`), not `CompilerConfiguration.DEFAULT`, when adding customizers. Typical embedder customizers are `ImportCustomizer` (star and static-star imports that would otherwise be prepended as snippets on every evaluation) and `ASTTransformationCustomizer(ThreadInterrupt)` (so Ctrl+C can stop a tight loop after `Main` interrupts the execute thread).

Compile-time customizers on the engine

```
def imports = new ImportCustomizer()
imports.addStarImports('java.util.concurrent.atomic')
def config = new CompilerConfiguration()
config.addCompilationCustomizers(imports)

def engine = new GroovyEngine(config)
assert engine.execute('new AtomicInteger(7).get()') == 7
```

Subclasses that need a different `GroovyShell` can override `protected GroovyShell createShell(ClassLoader, Binding, CompilerConfiguration)`. That method is called from the constructor: do not read subclass fields that are assigned after `super(…)`.

`GroovyEngine.getClassLoader()` is the public accessor for the engine class loader (the field is `protected`; Java callers cannot read it without a getter).

The engine class loader

```
def engine = new GroovyEngine()
assert engine.classLoader instanceof GroovyEngine.EngineClassLoader
```

GroovyshOptions

`Main.start(GroovyshOptions, String[])` is the programmatic entry point that does not call `System.exit`. The options object is built with `GroovyshOptions.builder()` and is incubating in 7.0.

Where a setting has an equivalent on JLine's `ShellBuilder`, it deliberately carries the same name and parameter type, so that a future move of groovysh onto `org.jline.shell` does not break this API.

Recognized hooks:

- `compilerConfiguration` / `engine` — compile-time customizers, or a pre-built `GroovyEngine`. If both are set, the engine wins.
- `bindings` / `binding(name, value)` — variables installed on the engine before the REPL starts.
- `groups` — additional JLine `CommandGroup` instances (`org.jline.shell`, the undeprecated successor to `CommandRegistry`; groovysh adapts them internally). They are resolved **before** groovysh's own commands, so a group may override a built-in command name.
- `prompt` — `Supplier<String>` invoked on each REPL iteration (default `{ 'groovy> ' }`).
- `resultHandler` — how values are rendered, given the session `Printer` (default: `printer.println(result?.toString())`).
- `errorHandler` — how execution failures are rendered, given the error and groovysh's own renderer to delegate to (default: the latter). `UserInterruptException` and `EndOfFileException` stay in `Main` as control flow.
- `showBanner` — set `false` to suppress groovysh's banner so the host can print its own.

- `rightPrompt` — optional right-hand prompt.
- `terminal` — optional JLine `Terminal`; when omitted, `Main` builds one from the CLI flags.
- `historyFile` — where to persist history; when omitted, groovysh's own `groovysh_history`.
- `onReaderReady` — called once the reader and the command registry are both wired, immediately before the REPL starts, for settings the builder does not model: the continuation prompt (`SECONDARY_PROMPT_PATTERN`), key bindings, reader options, and completion. Takes a `Consumer<LineReader>`, or a `BiConsumer<LineReader, GroovySystemRegistry>` when the registry is needed too. groovysh has already assigned `reader.completer = registry.completer()` by then, so an embedder contributing domain completion wraps it:

  ```
  reader.completer = new AggregateCompleter(registry.completer(), myCompleter)
  ```

  `AggregateCompleter` merges candidates, so a completer added that way supplements groovysh's rather than replacing it. Assigning a completer of your own replaces it outright. Per-command completion needs none of this — it comes from `Command.completer()` on a contributed `group`.

Embedding with `GroovyshOptions`

```
int rc = Main.start(GroovyshOptions.builder()
    .terminal(terminal)                                  // host-supplied terminal
    .showBanner(false)                                   // the host prints its own
    .prompt { 'gremlin> ' }                              // primary prompt
    .onReaderReady { LineReader reader ->                // continuation prompt, key
bindings
        reader.setVariable(LineReader.SECONDARY_PROMPT_PATTERN, '.......> ')
    }
    .historyFile(Path.of(System.getProperty('user.home'), '.myapp_history'))
    .binding('answer', 42)                               // visible to evaluated
code
    .groups(new SimpleCommandGroup('MyApp', new AbstractCommand('/ping') {
        @Override
        Object execute(CommandSession session, String[] args) { 'pong' }
    }))
    .resultHandler { printer, result -> results << result }  // render values your way
    .errorHandler { error, defaultTrace -> defaultTrace.accept(error) }
    .build(), '-e', 'answer')
assert rc == 0
assert 42 in results
```

`Main.start(Map, String[])` remains the binding-only overload and delegates to `GroovyshOptions`.

Extra console commands

`/clear`, `/pwd`, `/cd`, `/ls`, `/cat`, `/grep`, `/head`, `/tail`, `/wc`, `/sort`, `/date`, `/echo` and `/!` live in the public `ExtraConsoleCommands` class. It implements JLine's deprecated `org.jline.console.CommandRegistry`, as groovysh's own registries still do; prefer `GroovyshOptions.groups` and `org.jline.shell` for new embedder commands. `Main` still registers it by default. Embedders that build their own

`SystemRegistry` can construct it directly. The former `Main.ExtraConsoleCommands` nested class remains as a deprecated subclass for Groovy callers that compiled against it.

Advanced Topics

REPL Model Details

The style of coding you use in a REPL is slightly different from what you might use in a script. The style you use may also depend on whether you are using the Groovy Shell in *interpreter mode* or not.

Using binding (shared) variables

Regardless of interpreter mode you can use shared variables:

```
groovy> fruit = []
groovy> fruit << 'peach'
[peach]
groovy> fruit << 'pear'
[peach, pear]
groovy> assert fruit == ['peach', 'pear']
```

Use the `/show` command to see the shared variables.

Using local variables

You should treat local variables as if you were using immutable data structures. An input which mutates a local variable will likely be undone by subsequent statements.

Only in interpreter mode:

```
groovy> def noFruit = []
[]
groovy> def oneFruit = noFruit << 'peach'
[peach]
groovy> def twoFruit = oneFruit << 'pear'
[peach, pear]
groovy> assert twoFruit == ['peach', 'pear']
```

By the time you get to the last statement, the previous three local variable definitions are remembered, so the assertion will pass.

Avoid this (relevant to interpreter mode):

```
groovy> def fruit = []
[]
groovy> fruit << 'peach'
[peach]
```

```
groovy> fruit << 'pear'
[pear]
groovy> assert fruit == []
```

The `def fruit = []` will be *remembered* before executing each of the next three statements.

Gotchas

The Groovy Shell attempts to blend a Groovy programming environment with a shell-like scripting environment. Sometimes the two are at odds. As one example, don't try `/ls /tmp/.txt. The /` will be interpreted as the start of a comment. Instead, quote the argument like this `/ls '/tmp/*.txt'`.

Platform Problems

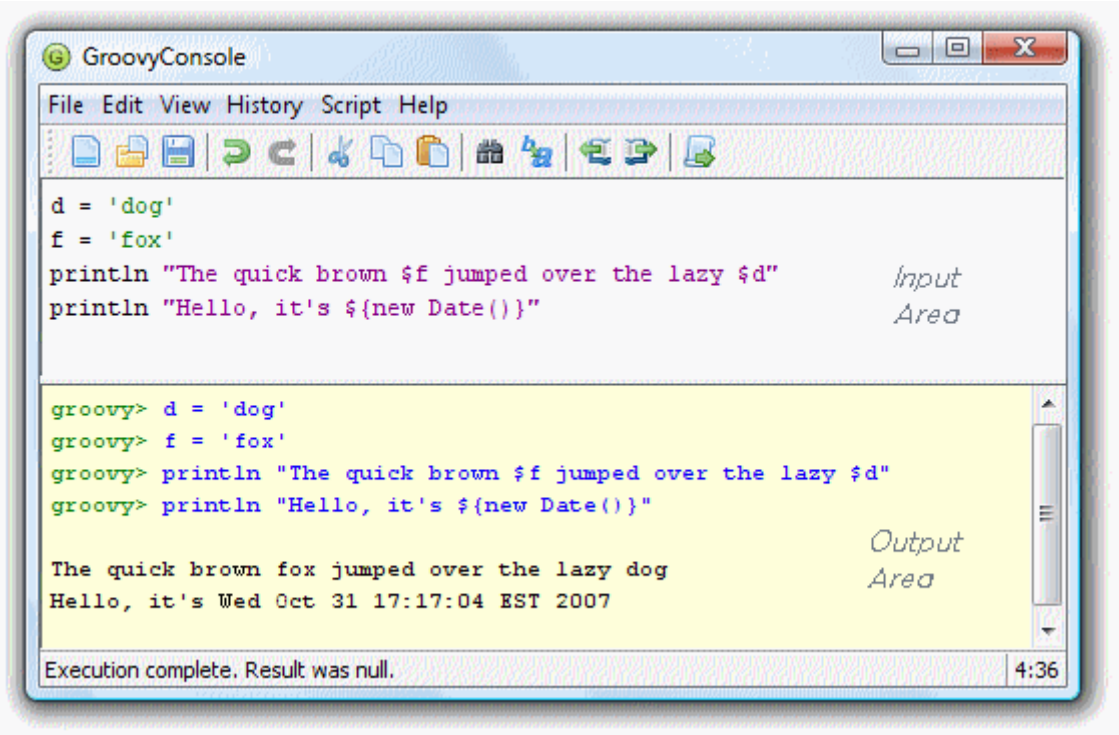
The Groovy Shell relies heavily on the JLine3 library for its platform support. If you have specific platform problems, please refer to the JLine documentation.

groovyConsole, the Groovy swing console

Groovy : Groovy Console

The Groovy Swing Console allows a user to enter and run Groovy scripts. This page documents the features of this user interface.

Basics



1. Groovy Console is launched via `groovyConsole` or `groovyConsole.bat`, both located in `$GROOVY_HOME/bin`
2. The Console has an input area and an output area.

3. You type a Groovy script in the input area.
4. When you select `Run` from the `Actions` menu, the console compiles the script and runs it.
5. Anything that would normally be printed on `System.out` is printed in the output area.
6. If the script returns a non-null result, that result is printed.

Features

Command-line Options and Arguments

The Groovy Console supports several options to control classpath and other features.

```
./bin/groovyConsole --help
Usage: groovyConsole [options] [filename]
The Groovy Swing Console allows a user to enter and run Groovy scripts.
      --configscript=PARAM   A script for tweaking the compiler configuration options
      -cp, -classpath, --classpath
                             Specify where to find the class files - must be first
                               argument
  -D, --define=<name=value>  Define a system property
  -h, --help                 Display this help message
  -pa, --parameters          Generate metadata for reflection on method parameter
                               names (jdk8+ only)
  -pr, --enable-preview      Enable preview Java features (jdk12+ only)
  -V, --version              Display the version
```

Running Scripts

There are several shortcuts that you can use to run scripts or code snippets:

- `Ctrl+Enter` and `Ctrl+R` are both shortcut keys for `Run Script`.
- If you highlight just part of the text in the input area, then Groovy runs just that text.
- The result of a script is the value of the last expression executed.
- You can turn the System.out capture on and off by selecting `Capture System.out` from the `Actions` menu

Running and compiling as Java

The console can also treat the editor contents as Java source rather than Groovy:

- `Script > Run as Java` (shortcut `Alt+R`) compiles the input as Java and invokes its `main` method, with any program arguments taken from `Script > Set Script Arguments`.
- `Script > Run Selection as Java` does the same for just the highlighted text.
- `Script > Compile as Java` compiles the input without running it — useful as a quick syntax check.

The source must declare a single top-level public class whose name matches a fully-qualified binary

name in any `package` declaration; the console parses the source to locate that class and reports if none is found. Compilation and execution go through the `org.apache.groovy.util.JavaShell` API, so the same `javac` diagnostics surface in the output area.

Editing Files

You can open any text file, edit it, run it (as a Groovy Script) and then save it again when you are finished.

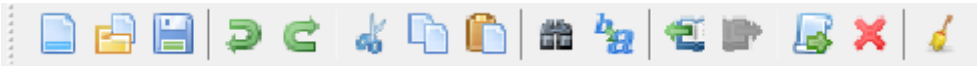
- Select `File > Open` (shortcut key `ctrl+O`) to open a file
- Select `File > Save` (shortcut key `ctrl+S`) to save a file
- Select `File > New File` (shortcut key `ctrl+Q`) to start again with a blank input area

History and results

- You can pop up a gui inspector on the last (non-null) result by selecting `Inspect Last` from the `Actions` menu. The inspector is a convenient way to view lists and maps.
- The console remembers the last ten script runs. You can scroll back and forth through the history by selecting `Next` and `Previous` from the `Edit` menu. `Ctrl-N` and `ctrl-P` are convenient shortcut keys.
- The last (non-null) result is bound to a variable named `_` (an underscore).
- The last result (null and non-null) for every run in the history is bound into a list variable named `__` (two underscores). The result of the last run is `__[-1]`, the result of the second to last run is `__[-2]` and so forth.

Interrupting a script

The Groovy console is a very handy tool to develop scripts. Often, you will find yourself running a script multiple times until it works the way you want it to. However, what if your code takes too long to finish or worse, creates an infinite loop? Interrupting script execution can be achieved by clicking the `interrupt` button on the small dialog box that pops up when a script is executing or through the `interrupt` icon in the toolbar.

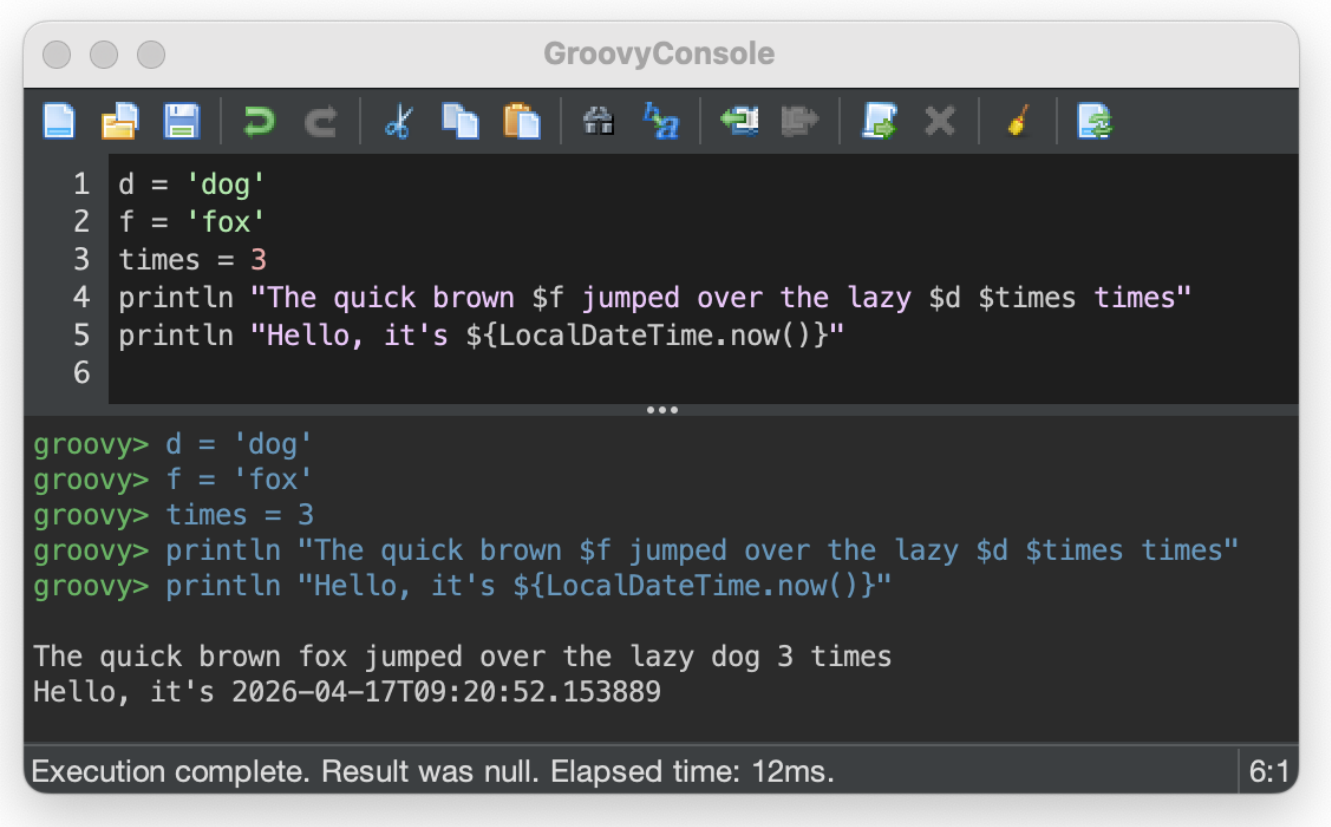


However, this may not be sufficient to interrupt a script: clicking the button will interrupt the execution thread, but if your code doesn't handle the interrupt flag, the script is likely to keep running without you being able to effectively stop it. To avoid that, you have to make sure that the `Script > Allow interruption` menu item is flagged. This will automatically apply an AST transformation to your script which will take care of checking the interrupt flag (`@ThreadInterrupt`). This way, you guarantee that the script can be interrupted even if you don't explicitly handle interruption, at the cost of extra execution time.

And more

- You can change the font size by selecting `Smaller Font` or `Larger Font` from the `Actions menu`
- You can switch between Light, Dark, and System (if supported by your OS) themes from the `View`

→ Theme menu or by clicking the theme cycle button in the toolbar. The theme preference is remembered across sessions. The System option follows your operating system's dark mode setting.



- Code is auto indented when you hit return
- You can drag'n'drop a Groovy script over the text area to open a file
- You can modify the classpath with which the script in the console is being run by adding a new JAR or a directory to the classpath from the Script menu
- Error hyperlinking from the output area when a compilation error is expected or when an exception is thrown

Embedding the Console

To embed a Swing console in your application, simply create the Console object, load some variables, and then launch it. The console can be embedded in either Java or Groovy code. The Java code for this is:

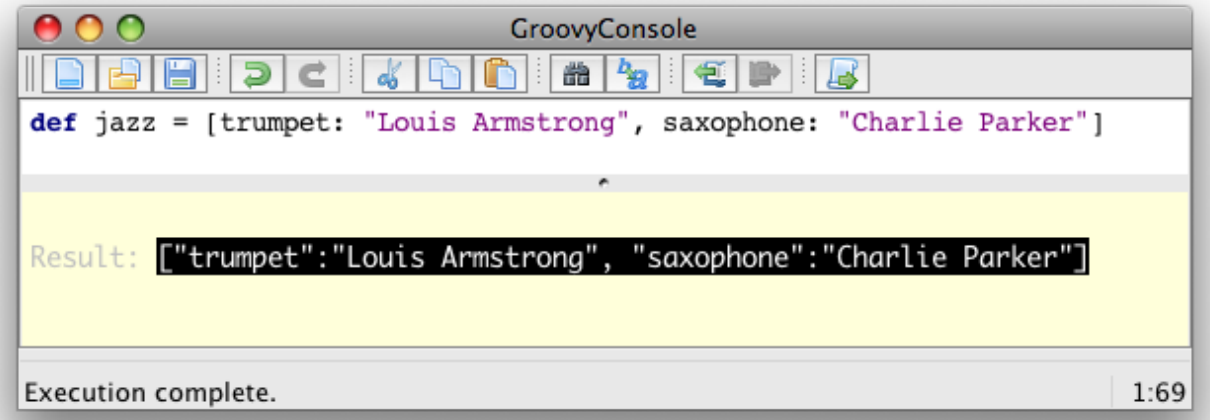
```
import groovy.ui.Console;

    ...
    Console console = new Console();
    console.setVariable("var1", getValueOfVar1());
    console.setVariable("var2", getValueOfVar2());
    console.run();
    ...
```

Once the console is launched, you can use the variable values in Groovy code.

Visualizing script output results

You can customize the way script output results are visualized. Let's see how we can customize this. For example, viewing a map result would show something like this:

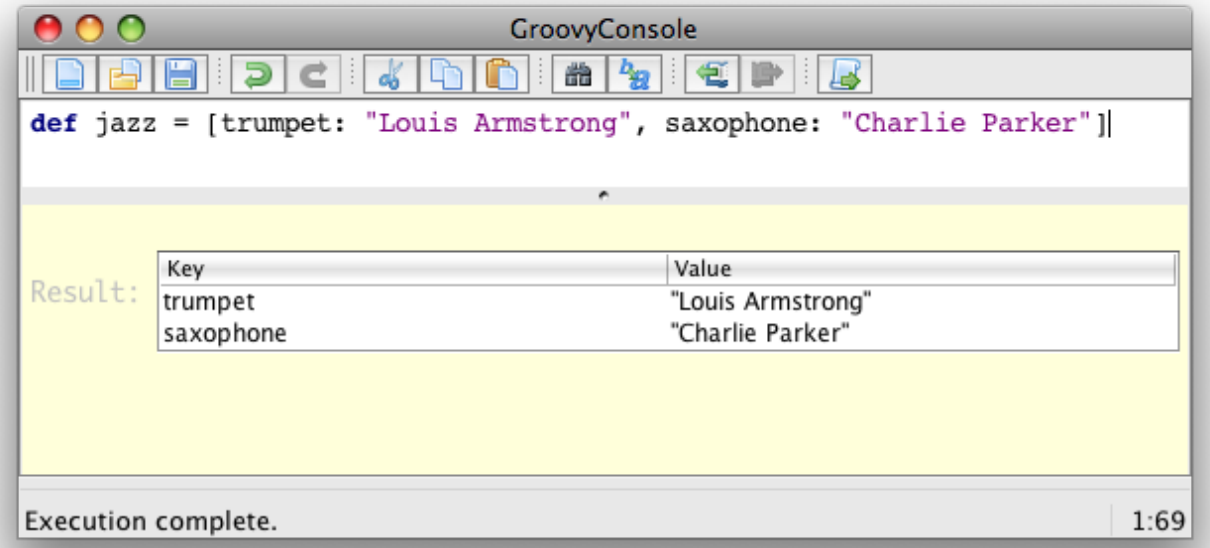


What you see here is the usual textual representation of a Map. But, what if we enabled custom visualization of certain results? The Swing console allows you to do just that. First of all, you have to ensure that the visualization option is ticked: `View → Visualize Script Results` — for the record, all settings of the Groovy Console are stored and remembered thanks to the Preference API. There are a few result visualizations built-in: if the script returns a `java.awt.Image`, a `javax.swing.Icon`, or a `java.awt.Component` with no parent, the object is displayed instead of its `toString()` representation. Otherwise, everything else is still just represented as text. Now, create the following Groovy script in `~/.groovy/OutputTransforms.groovy`:

```
import javax.swing.*

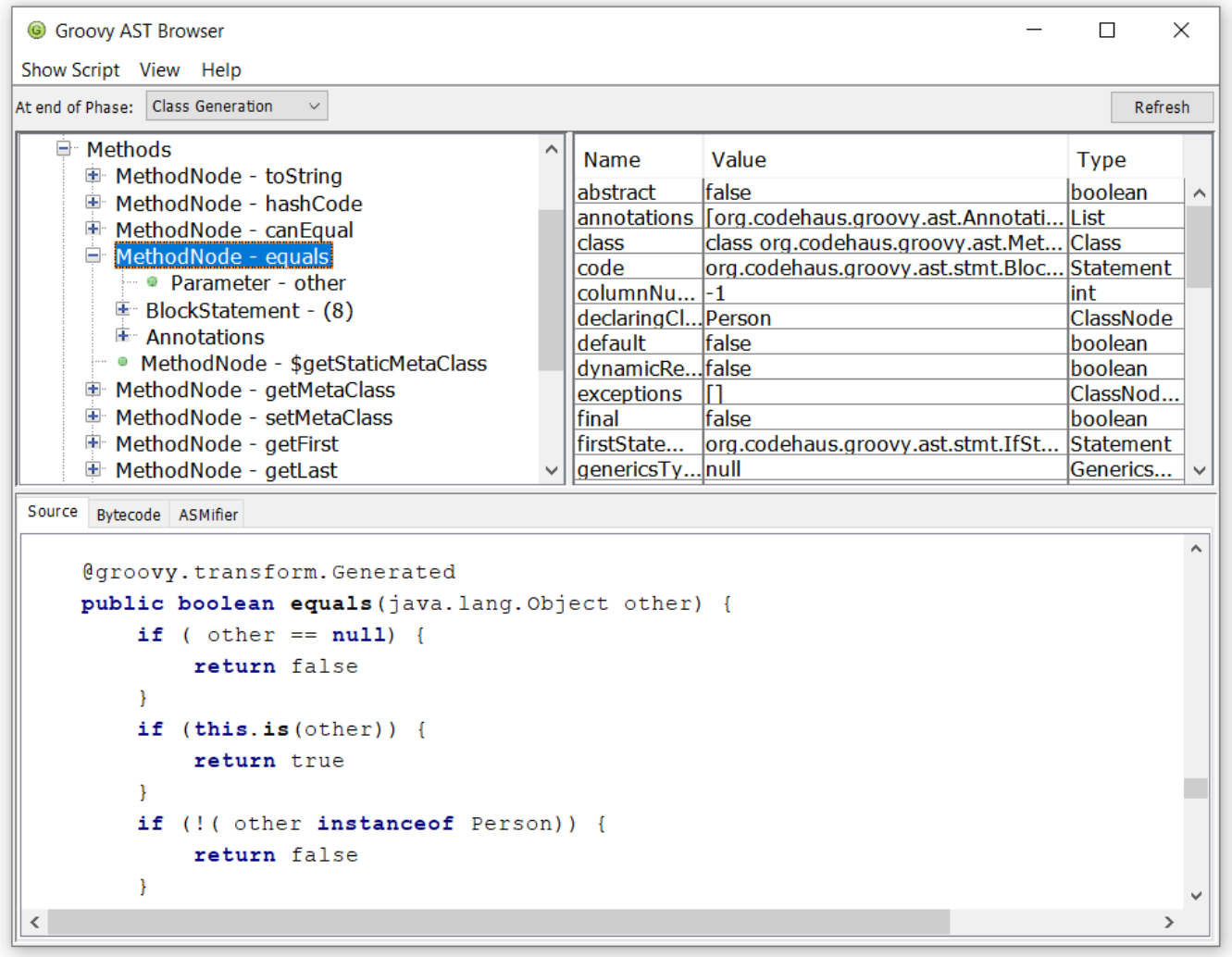
transforms << { result ->
    if (result instanceof Map) {
        def table = new JTable(
            result.collect{ k, v ->
                [k, v?.inspect()] as Object[]
            } as Object[][],
            ['Key', 'Value'] as Object[])
        table.preferredViewportSize = table.preferredSize
        return new JScrollPane(table)
    }
}
```

The Groovy Swing console will execute that script on startup, injecting a transforms list in the binding of the script, so that you can add your own script results representations. In our case, we transform the Map into a nice-looking Swing JTable. We're now able to visualize maps in a friendly and attractive fashion, as the screenshot below shows:

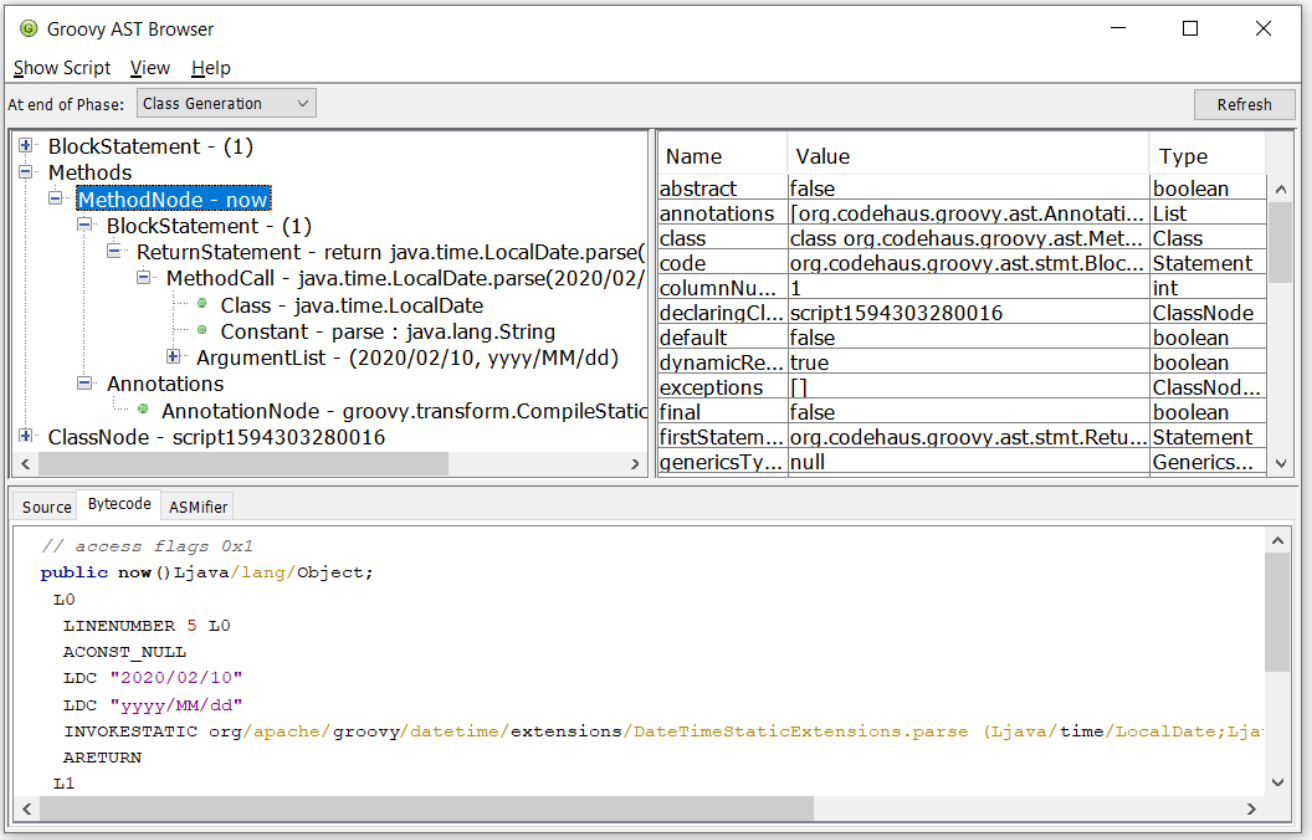


Advanced debugging: AST browser

Groovy Console can visualize the AST (Abstract Syntax Tree) representing the currently edited script, as shown by the screenshot below. This is useful when you want to understand how an AST transformation is working and particularly handy if you are developing your own AST transform. In the example below, we have annotated our class with the `@Immutable` annotation and the Groovy compiler has generated a lot of boilerplate code for us. We can see the code for the generated equals method in the `Source` tab.

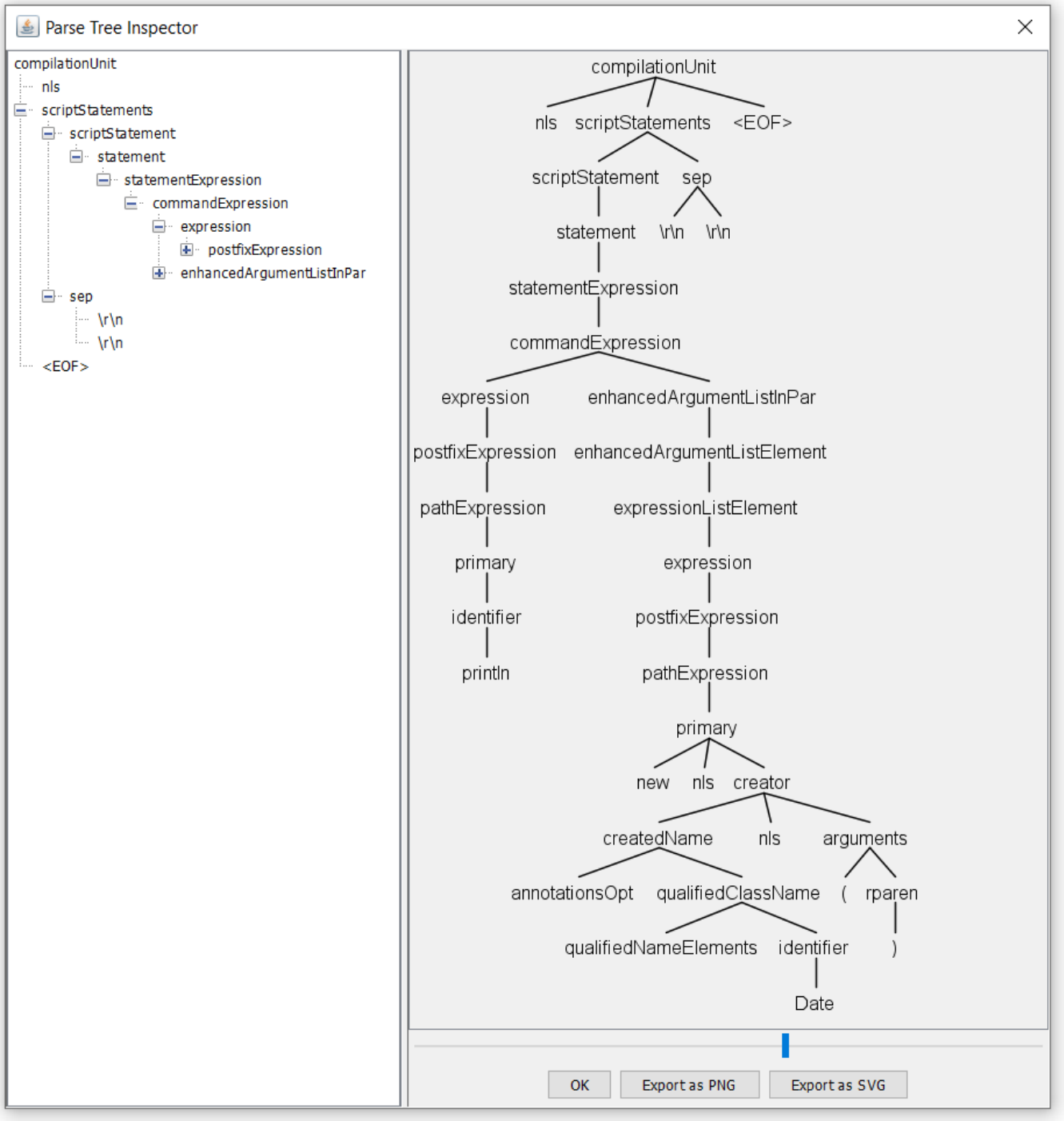


We can even examine the JVM bytecode generated by the compiler. In the image below we are looking at the bytecode for the Groovy expression `LocalDate.parse('2020/02/10', 'yyyy/MM/dd')`.



Advanced debugging: CST browser

Groovy Console can visualize the CST (Concrete Syntax Tree) representing the initial parsing of the script. This is mainly useful for parsing gurus.



groovydoc, the Groovy & Java documentation generator

GroovyDoc generates API documentation from your source code. It plays the role the Javadoc tool plays in the Java ecosystem, but accepts both `.groovy` and `.java` sources and adds conveniences suited to Groovy.

The documentation generator is driven from the command line, from Apache Ant, or from build-tool wrappers such as the GMavenPlus Maven plugin. The sections below cover the everyday authoring and invocation patterns; Advanced topics collects the lesser-used features.

Writing doc comments

GroovyDoc recognises three forms of doc comment, all of which can coexist in the same source file:

- Traditional Javadoc `/* … */` block comments.
- JEP 467-style Markdown comments using runs of `///` line prefixes.
- Runtime-retained Javadoc `/*@ … */` block comments, whose content is reflectable at runtime via the `@Groovydoc` annotation (see Runtime Groovydoc).

Comment forms

Traditional Javadoc

```
/**
 * Summary sentence. More description.
 *
 * @param name the subject to greet
 * @return the greeting
 */
String greet(String name) { "Hello, $name" }
```

Leading `*` prefixes on each line are stripped. The first sentence (up to the first period followed by whitespace) is used as the short summary on index/package pages.

JEP 467 Markdown (`///`)

Each line of the run starts with `///`. Inside the comment, CommonMark Markdown is supported — headings, bullets, emphasis, links, and fenced code blocks all render as you would expect:

```
/// # Greet
///
/// Returns a friendly greeting. Supports the usual inline tags:
/// {@link String} and {@code greet} still work.
///
/// ```groovy
/// assert greeter.greet('world') == 'Hello, world'
/// ```
///
/// @param name the subject to greet
/// @return the greeting
String greet(String name) { "Hello, $name" }
```

Markdown is rendered through CommonMark. Headings are shifted down by two levels (so `#` becomes `<h3>`) to fit under the page's existing title and section banners. Inline Javadoc tags like `{@link …}` are preserved inside Markdown bodies. `{@link …}` tokens appearing inside fenced code blocks are left literal — the fence is treated as code, not documentation.

For the full Markdown syntax and semantics, see JEP 467 and CommonMark.

Runtime Groovydoc (`/**@`)

If the comment opens with `/@ instead of /`, Groovy's parser can retain the raw comment text as a `groovy.lang.@Groovydoc` annotation on the enclosing class/method/field. Enable the feature through the compiler option `runtimeGroovydoc`; see Runtime Groovydoc.

Block tags

Block tags appear at the end of the comment, one per paragraph, leading with `@`. Common tags:

Tag	Purpose
`@param <name> <description>`	Describes a parameter. Use `@param <T>` for a type parameter.
`@return <description>`	Describes the return value.
`@throws <exception> <description>`, `@exception <exception> <description>`	Describes an exception the method might throw.
`@since <version>`	Indicates when the feature was added.
`@author <name>`	Author of the element. Rendered only when `-author` is enabled.
`@version <text>`	Version information.
`@deprecated <explanation>`	Marks the element as deprecated; recommend a replacement.
`@see <reference>`	Adds a "See Also" entry. The reference is a class or member.
`@apiNote <text>`	Guidance intended for callers of the API.
`@implSpec <text>`	Contract that overriders must preserve.
`@implNote <text>`	Non-contract implementation detail.

Multiple `@param`, `@throws`, and `@see` entries are collated under a single heading in the rendered output.

Inline tags

Inline tags appear within a comment body and are expanded in place.

Tag	Purpose
`{@link <reference>}`	Cross-reference to a class or member. Produces a hyperlink.
`{@code <text>}`	Renders the text in a monospaced code style. HTML metacharacters are escaped. The body is brace-balanced with a linear scan (GROOVY-12095), so nested braces — Groovy closures, map/type literals, etc. — are kept intact, including inside `<pre>{@code ⋯}</pre>` examples. Braces inside ordinary quoted string/character literals (e.g. `System.out.println(" } ")`), multi-line text blocks (`"""…"""` / `'''…'''`), `/* ⋯ */` block comments, and `// ⋯` line comments (e.g. `// }`) are ignored while matching.

Tag	Purpose
{@literal <text>}	Renders the text verbatim without code styling. Same brace-balanced body rules as {@code}.
{@value #FIELD}, {@value Class#FIELD}	Inlines the value of a compile-time-constant field. Same-class references and cross-class references both work. On a constant field's own comment, bare {@value} inlines that field's own value.
{@inheritDoc}	Inherits the parent class's/interface's corresponding doc.
{@snippet ⋯}	Embeds a code snippet. See {@snippet} under advanced topics.
{@docRoot}	Path to the root of the generated documentation. Useful when writing a relative link that must work from pages at any depth.

Running groovydoc

The groovydoc command line tool

The groovydoc command line invocation:

```
groovydoc [options] [packagenames] [sourcefiles]
```

Most commonly-used options:

Short version	Long version	Description
-d	--destdir <dir>	Destination directory for output files
-sourcepath <pathlist>		Directories to find source files (platform path separator)
-classpath, -cp	--classpath	Where to find class files (must be first argument)
-public		Show only public classes and members
-protected		Show protected/public classes and members (default)
-package		Show package/protected/public classes and members
-private		Show all classes and members
-charset <charset>		Charset for cross-platform viewing of generated HTML
-fileEncoding <charset>		Charset for generated documentation files
-doctitle <html>		Title for the overview page
-windowtitle <text>		Browser window title
-header <html>		Include header text on each page
-footer <html>		Include footer text on each page
-overview <file>		Read overview documentation from HTML file

Short version	Long version	Description
-exclude <pkglist>		Exclude listed packages (colon-separated)
-noscripts		Don't process Groovy Scripts
-author		Include `@author` paragraphs
-stylesheetfile <path>		Replace the default stylesheet with a supplied one
-addStylesheet <path>		Add a stylesheet **alongside** the default (can be given multiple times with comma-separated values)
-theme <mode>		Theme: `auto` (default, follow OS), `light`, or `dark`
-syntaxHighlighter <name>		Client-side syntax highlighter: `prism` (bundled) or `none` (default)
-preLanguage <lang>		Default Prism language id for unattributed `<pre>` blocks in doc comments. When set, a post-render pass wraps their bodies so Prism highlights them (see Highlighting plain `<pre>` blocks).
-showInternal		Include members annotated with `groovy.transform.@Internal` (hidden by default)
-noindex		Skip generating the alphabetical index page
-nodeprecatedlist		Skip generating the deprecated list
-nohelp		Skip generating the help page
-notimestamp		Don't include generation timestamp in output
-noversionstamp		Don't include Groovy version in output
-nomainforscripts		Don't include the implicit `public static void main` method for scripts
-help, -h	--help	Display help message
-verbose, -quiet, --debug		Adjust logging verbosity
	--version	Display the version
-javaVersion <version>		Language version for parsed Java files (e.g. `JAVA_17`)

The `groovydoc` Ant task

The Ant task exposes the same feature set. You will need the task defined before invoking it:

```
<taskdef name        = "groovydoc"
         classname   = "org.codehaus.groovy.ant.Groovydoc"
         classpathref = "my.classpath"/>
```

where *my.classpath* references the Groovy distribution jars (`groovy-VERSION.jar`, `groovy-ant-`

`VERSION.jar`, `groovy-groovydoc-VERSION.jar`, plus any modules and transitive dependencies you use).

`<groovydoc>` attributes

Attribute	Description	Required
destdir	Location to write generated documentation to.	Yes
sourcepath	Directories to scan for source files.	No
packagenames	Comma-separated list of package specs (wildcards allowed, e.g. `com.foo.**`).	No
excludePackageNames	Comma-separated (or colon-separated) list of package specs to exclude.	No
windowtitle	Browser window title.	No
doctitle	HTML shown at the top of the overview page.	No
header, footer	HTML shown at the top/bottom of every page.	No
overview	HTML file read as the overview page.	No
charset	Charset for cross-platform viewing of generated HTML.	No
fileEncoding	Charset used when writing the generated files.	No
author	Include `@author` paragraphs. Defaults to `true`.	No
processScripts	Process Groovy scripts. Defaults to `true`; set to `false` to mirror the CLI `-noscripts` flag.	No
includeMainForScripts	Include the synthetic `public static void main` method when documenting scripts. Defaults to `true`; set to `false` to mirror the CLI `-nomainforscripts` flag.	No
stylesheetFile	Replace the default stylesheet with a supplied one.	No
private	Show all classes and members (including private ones).	No
public, protected, package	Set individually to control the visibility scope; equivalent to the CLI `-public`, `-protected`, `-package` flags.	No
access	Alternative to the boolean scope attributes above. Accepts `public`, `protected`, `package`, or `private`.	No
javaVersion	Language version for Java source files.	No
extensions	Colon-separated source extensions (default `.java:.groovy:.gv:.gvy:.gsh`).	No
use	Legacy Javadoc option accepted for parity. Has no effect on groovydoc output.	No
showInternal	Include members annotated with `groovy.transform.@Internal`.	No
noIndex, noDeprecatedList, noHelp	Suppress the matching auxiliary page.	No

Attribute	Description	Required
syntaxHighlighter	`prism` or `none` (default).	No
theme	Theme lock: `auto` (default), `light`, or `dark`.	No
preLanguage	Default Prism language for unattributed `<pre>` blocks in doc comments. See Highlighting plain `<pre>` blocks.	No
noTimestamp, noVersionStamp	Omit the matching stamp from output.	No

`<groovydoc>` nested elements

`<link>`

Create cross-references to external Javadoc/groovydoc output. See Linking to external APIs.

Attribute	Description	Required
packages	Comma-separated list of package prefixes.	Yes
href	Base URL of the external site.	Yes
module	JPMS module name; inserted as a path segment for modular Javadoc layouts.	No

`<addStylesheet>`

Add an extra stylesheet alongside the default. Repeat the element to add multiple stylesheets.

Attribute	Description	Required
file	Path to the stylesheet file.	Yes

Ant task example

```
<taskdef name           = "groovydoc"
         classname      = "org.codehaus.groovy.ant.Groovydoc"
         classpathref   = "path_to_groovy_all"/>

<groovydoc destdir      = "${docsDirectory}/gapi"
           sourcepath   = "${mainSourceDirectory}"
           packagenames = "**.*"
           use          = "true"
           windowtitle  = "${title}"
           doctitle     = "${title}"
           header       = "${title}"
           footer       = "${docFooter}"
           overview     = "src/main/overview.html"
           private      = "false">
        <link packages="java.,org.xml.,javax.,org.xml."
href="https://docs.oracle.com/javase/8/docs/api/"/>
        <link packages="org.apache.tools.ant."          href="https://docs.groovy-
```

```
lang.org/docs/ant/api/"/>
        <link packages="org.junit.,junit.framework."
href="https://junit.org/junit4/javadoc/latest/"/>
        <link packages="groovy.,org.codehaus.groovy."   href="https://docs.groovy-
lang.org/latest/html/api/"/>
</groovydoc>
```

Running the task from Groovy via `AntBuilder` works the same way.

Linking to external APIs

`<link>` (Ant) and `-link` (CLI) let cross-references resolve to an external Javadoc/groovydoc site. Packages are matched by the `packages` attribute as a comma-separated list of prefixes; when a documented reference matches one of the prefixes the link goes to the configured `href` base.

The `module` attribute on `<link>` supports modular Javadoc layouts, where an external site places each JPMS module's docs under a module-named subdirectory. With `module="java.base"` and `href="https://docs.oracle.com/en/java/javase/17/docs/api/"`, a link to `java.lang.Object` resolves to `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html`.

Customising the output

Stylesheets

Two mechanisms cover most customisation needs:

- `-stylesheetfile <path>` (CLI) / `stylesheetFile="…"` (Ant) — **replaces** the default stylesheet. The supplied file is copied verbatim over `stylesheet.css` in the output. If the file has no dark-mode support of its own, the generated docs will be light-only regardless of the theme settings below.
- `-addStylesheet <path>` (CLI) / `<addStylesheet file="…"/>` (Ant) — **adds** an extra stylesheet alongside the default. The default is still loaded and supplies the baseline look; the added stylesheet layers on top. This is the safer choice for theme tweaks.

Author stylesheets can opt into the baseline palette by referencing the default stylesheet's CSS custom properties (`--fg`, `--bg`, `--link`, `--bg-panel`, `--border`, `--caption-bg`, and related semantic tokens). Doing so makes the added stylesheet automatically dark-mode aware.

Theme

`-theme <mode>` / `theme="…"` controls the palette strategy:

- `auto` (default) — the generated stylesheet emits a `prefers-color-scheme` media query. Each reader's browser picks the light or dark palette that matches their OS preference. Syntax highlighting (if enabled) switches Prism themes in sync.
- `light` — locks the output to the light palette regardless of OS preference.
- `dark` — locks the output to the dark palette regardless of OS preference.

`auto` is the right choice for public-facing docs that should adapt to the reader. `light` and `dark` are

useful for branded sites where a consistent look matters more than honouring OS preference.

Syntax highlighting

`-syntaxHighlighter prism` / `syntaxHighlighter="prism"` enables client-side syntax highlighting for `{@snippet}` blocks and Markdown fenced code blocks. Prism.js is bundled and covers at minimum Groovy, Java, XML, JSON, YAML, TOML, SQL, CSV, Markdown, JavaScript, and regex. The light/dark theme selection tracks the theme setting automatically. The default is `none` (no highlighting); code blocks are rendered as plain preformatted text.

Highlighting plain `<pre>` blocks

Doc comments written before the `{@snippet}` era often use bare `<pre>…</pre>` wrappers for code samples. Prism only walks `<code>` descendants of language-classed elements, so a bare `<pre>` with prose inside is skipped even when `-syntaxHighlighter=prism` is on. The `-preLanguage <lang>` option (CLI) or `preLanguage="lang"` attribute (Ant task) closes that gap without touching source files:

```
groovydoc -syntaxHighlighter prism -preLanguage groovy ...
```

When set, a post-render pass rewrites each unattributed `<pre>body</pre>` to `<pre class="language-xxx"><code>body</code></pre>`, using the configured language id. A `<pre class="language-xxx">` that lacks an inner `<code>` is also completed with a `<code>` wrapper so hand-authored language-classed blocks highlight whether or not the author remembered the nested `<code>`. `<pre>` tags with any existing attribute (authored `class="language-sql"`, `id="…"`, or similar) are left untouched.

The option is off by default; unset, no rewriting occurs.

Disable optional pages

GroovyDoc generates auxiliary pages by default:

- `index-all.html` — alphabetical index of every class and member
- `deprecated-list.html` — everything marked `@deprecated`
- `help-doc.html` — explanation of the page layout

Any of these can be suppressed with `-noindex`, `-nodeprecatedlist`, `-nohelp` (CLI) or the corresponding `noIndex`, `noDeprecatedList`, `noHelp` Ant attributes. When suppressed, the page is not generated and navigation bars on other pages omit the corresponding link.

GMavenPlus Maven plugin

GMavenPlus is a Maven plugin with goals that drive groovydoc. Consult the plugin documentation for the current list of options it exposes.

Advanced topics

`{@snippet}` code snippets

`{@snippet}` embeds a code sample that renders as a preformatted block and, when syntax highlighting is enabled, is coloured according to its declared language. Three forms are supported:

Inline body

```
/**
 * {@snippet lang="groovy" :
 * def greet(name) { "Hello, $name" }
 * greet('world')
 * }
 */
```

The language is optional; when set, it becomes the Prism language class used for syntax highlighting. Any text from the opening brace to the matching closing brace is the snippet body.

Brace balancing in the inline form

Body parsing is a linear brace-depth scan: depth starts at 1 on the opening `{` of the tag, increments on each structural `{`, decrements on each structural `}`, and the matching close is reached when depth returns to zero. While scanning, ordinary quoted string/character literals (backslash escapes included), multi-line text blocks (`"""…"""` / `'''…'''`), `/* … */` block comments, and `// …` line comments (to end of line) are skipped so braces inside them do not affect depth. Nested braces in real code — closures, `catalog.each { book -> … }`, GString `"value=${x}"`, and unbalanced braces *inside* quotes, text blocks, or comments such as `System.out.println(" } ")`, `"""\n}\n"""`, or `// }` — are therefore handled correctly and do not terminate the tag early.

```
/**
 * {@snippet lang="groovy" :
 * System.out.println(" } ")   // } inside the string is fine
 * // } inside a line comment is fine too
 * def block = """
 *   multi-line with }
 * """
 * if (ok) { work() }
 * }
 */
```

Because `//` comments run to end of line, put the tag-closing `}` on its own line (or after real code), not after a trailing `//` on the same line — otherwise the closer is swallowed by the comment. Braces that are not inside a skipped span and are not balanced will still confuse the parser; workarounds:

- Rewrite the example so structural braces balance, or keep unbalanced braces inside quotes, text blocks, or comments (already ignored).
- Move the sample into a file and use the external form (`{@snippet file="Example.groovy"}`), which reads the file verbatim without brace-balanced parsing.
- Write a problematic `}` as the HTML entity `}`, which is not a brace character during body

extraction but renders as `}` in the output.

External file reference

Place the source of the snippet in a `snippet-files/` subdirectory of the current package, then reference it:

```
/** {@snippet file="GreetExample.groovy"} */
```

The `snippet-files/` directory is discovered via the `-sourcepath`. The file is copied to the output site verbatim so that the snippet remains browsable in the rendered docs.

A `region="name"` attribute selects a slice of the file bounded by `// @start region="name"` and `// @end region="name"` marker comments.

Two author-facing conveniences apply to the external form:

- **Language inference from file extension** — when no explicit `lang="…"` attribute is given, the Prism language class is derived from the referenced file's extension. The recognised extensions are `.groovy` / `.gvy` / `.gy` / `.gsh` (groovy), `.java`, `.xml` / `.html` / `.htm` (xml-doc), `.json`, `.yaml` / `.yml`, `.toml`, `.sql`, `.csv`, `.md` / `.markdown`, `.properties`, and `.js` (javascript). Explicit `lang=` still wins when both are present.
- **License-header auto-strip** — external snippet files typically carry a license or copyright notice as their first block comment, which isn't useful in the rendered output. If the file begins with a leading `/* … */` block comment whose text contains either "Licensed" or "Copyright", the block is stripped before rendering (covers Apache, MIT, BSD, GPL and most proprietary notices). Files whose first content isn't a license-looking block comment are left verbatim. To preserve a matching header (or suppress the strip for a file that happens to match), add `keepHeader=true`:

  ```
  /** {@snippet file="GreetExample.groovy" keepHeader=true} */
  ```

Markup comments inside a snippet body

Markup comments let snippet authors highlight, replace, or link portions of the code without changing the source itself:

Directive	Effect
`// @highlight [substring="text"|regex="pattern"] [type="bold|italic|highlighted"]`	Wrap matches in `<b>`/`<i>`/`<mark>`. Type defaults to `bold`.
`// @replace [substring="text"|regex="pattern"] replacement="text"`	Replace matching text in the rendered output.

Directive	Effect
`// @link [substring="text"|regex="pattern"] target="Class[#member]"`	Turn matches into anchors pointing at the referenced element.
`// @start region="name"`, `// @end [region="name"]`	Define a named region for the external-file form, or as a scoping bound for any of the above directives.

A directive written on its own line applies to the next code line. A directive written at the end of a code line (after the code) applies to that line's code portion. Adding a `region="name"` attribute to a directive activates it until the matching `// @end`. Directive comment lines are stripped from the rendered output.

For the complete semantics see JEP 413.

Class hierarchy tree pages

GroovyDoc generates a global `overview-tree.html` and per-package `package-tree.html` showing class and interface hierarchies. Classes are rooted at `java.lang.Object`; interfaces and traits form a separate tree. External ancestors (types the tool can't see) render as plain text joining documented classes up the chain.

Runtime Groovydoc (`/**@`)

A doc comment whose opening marker is `/@` instead of `/` can be retained as runtime metadata. Enable it by setting the compiler option:

```
compilerConfiguration.setOptimizationOptions(
    [(CompilerConfiguration.RUNTIME_GROOVYDOC): true]
)
```

When enabled, the parser attaches the raw comment text (still in source form — including `/*@`, per-line, and `*/`) to the declaration as a `groovy.lang.@Groovydoc` annotation. Readers access the text reflectively:

```
assert SomeClass.class.groovydoc.content.contains('class summary')
```

`/@` comments do not need to be marked specially to participate in normal groovydoc HTML generation; they render identically to `/`.

`doc-files/` and `snippet-files/`

Each package may contain a `doc-files/` subdirectory whose content is copied verbatim into the rendered output under the same package path. Useful for bundling images, diagrams, and supplementary HTML alongside the generated pages.

The sibling convention `snippet-files/` is used by `{@snippet file=⋯}`. Both conventions are sourced from `-sourcepath` directories and preserve nested subdirectories.

Java source version

`-javaVersion` (CLI) / `javaVersion` (Ant attribute) declares the language level at which `.java` source files are parsed. The accepted values are the names of the `ParserConfiguration.LanguageLevel` enum — for example `JAVA_8`, `JAVA_17`, `JAVA_21`.

Custom templates

The default template set can be replaced by extending `Groovydoc` in a small Java class:

```
package org.codehaus.groovy.tools.groovydoc;

import org.codehaus.groovy.ant.Groovydoc;

public class CustomGroovyDoc extends Groovydoc {

    @Override
    protected String[] getClassTemplates() {
        return new String[]{"path/to/custom/classDocName.html"};
    }
}
```

The following methods can be overridden:

- `getClassTemplates` — class-level templates
- `getPackageTemplates` — package-level templates
- `getDocTemplates` — top-level (overview, index, tree, etc.) templates

The list of default templates is in `org.codehaus.groovy.tools.groovydoc.gstringTemplates.GroovyDocTemplateInfo`.

Having extended the task, you then register it as an Ant task under the same name:

```
<taskdef name        = "groovydoc"
         classname   = "org.codehaus.groovy.tools.groovydoc.CustomGroovyDoc"
         classpathref = "path_to_groovy_all"/>
```

Template customisation is provided as is — the template APIs are subject to change between releases.

IDE integration

Many IDEs and text editors support the Groovy programming language.

Editor	Syntax highlighting	Code completion	Refactoring
Groovy Eclipse Plugin	Yes	Yes	Yes
IntelliJ IDEA	Yes	Yes	Yes
Netbeans	Yes	Yes	Yes
Groovy Emacs Modes	Yes	No	No
TextMate	Yes	No	No
vim	Yes	No	No
UltraEdit	Yes	No	No
SlickEdit	Yes	No	No
EditRocket	Yes	No	No
VSCode	Yes	No	Yes

User Guides

Getting started

Download

From the download page, you will be able to download the distribution (binary and source), the Windows installer (a community artifact) and the documentation for **Groovy**.

For a quick and effortless start on Mac OSX, Linux, WSL2 or Cygwin, you can use SDKMAN! (The Software Development Kit Manager) to download and configure any **Groovy** version of your choice. Basic instructions can be found below.

Stable

- **Download zip**: **Binary Release** | Source Release
- **Download documentation**: **JavaDoc and zipped online documentation**
- **Combined binary / source / documentation bundle**: **Distribution bundle**

You can learn more about this version in the release notes or in the changelog.

If you plan on using invokedynamic support, read those notes.

Snapshots

For those who want to test the very latest versions of Groovy and live on the bleeding edge, you can use our snapshot builds. As soon as a build succeeds on our continuous integration server a snapshot is deployed to this repository. These snapshots are not official releases and are intended for integration testing by the development community prior to official versions being released. We welcome any feedback.

Prerequisites

Groovy {groovy-short-version} requires Java 17+ with support for up to Java 25.

Various Groovy CI servers run the test suite (with more than 10000 tests) across numerous versions of Java. Those servers are also useful to look at to confirm supported Java versions for different Groovy releases.

Maven Repository

If you wish to embed Groovy in your application, you may just prefer to point your build to your favourite maven repository or the Groovy artifactory instance. Please see the download page for available modules for each Groovy version.

SDKMAN! (The Software Development Kit Manager)

This tool makes installing Groovy on any Bash platform (Mac OSX, Linux, Cygwin, Solaris or

FreeBSD) very easy.

Simply open a new terminal and enter:

```
$ curl -s "https://get.sdkman.io" | bash
```

Follow the instructions on-screen to complete installation.

Open a new terminal or type the command:

```
$ source "$HOME/.sdkman/bin/sdkman-init.sh"
```

Then install the latest stable Groovy:

```
$ sdk install groovy
```

After installation is complete and you've made it your default version, test it with:

```
$ groovy -version
```

That's all there is to it!

Other ways to get Groovy

Installation on Mac OS X

MacPorts

If you're on macOS and have MacPorts installed, you can run:

```
sudo port install groovy
```

Homebrew

If you're on macOS and have Homebrew installed, you can run:

```
brew install groovy
```

Installation on Windows

If you're on Windows, you can also use the Windows installer.

Other Distributions

You may download other distributions of Groovy from the ASF archive repository or from the Groovy artifactory instance (also includes pre-ASF versions).

Source Code

If you prefer to live on the bleeding edge, you can also grab the source code from GitHub.

IDE plugin

If you are an IDE user, you can just grab the latest IDE plugin and follow the plugin installation instructions.

Install Binary

These instructions describe how to install a binary distribution of **Groovy**:

- Download a binary distribution of Groovy and unpack it into some folder on your local file system.
- Set your `GROOVY_HOME` environment variable to the directory where you unpacked the distribution.
- Add `GROOVY_HOME/bin` to your `PATH` environment variable.
- Set your `JAVA_HOME` environment variable to point to your JDK. On OS X this is `/Library/Java/Home`, on other unixes its often `/usr/java` etc. If you've already installed tools like Ant or Maven you've probably already done this step.

You should now have Groovy installed properly. You can test this by typing the following in a command shell:

```
groovysh
```

Which should create an interactive groovy shell where you can type Groovy statements. Or to run the Swing interactive console type:

```
groovyConsole
```

To run a specific Groovy script type:

```
groovy SomeScript
```

Differences with Java

Groovy tries to be as natural as possible for Java developers. We've tried to follow the principle of least surprise when designing Groovy, particularly for developers learning Groovy who've come

from a Java background.

Here we list all the major differences between Java and Groovy.

Default imports

All these packages and classes are imported by default, i.e. you do not have to use an explicit `import` statement to use them:

- java.io.*
- java.lang.*
- java.math.BigDecimal
- java.math.BigInteger
- java.net.*
- java.time.*
- java.util.*
- groovy.lang.*
- groovy.util.*

Multi-methods

In Groovy, the methods which will be invoked are chosen at runtime. This is called runtime dispatch or multi-methods. It means that the method will be chosen based on the types of the arguments at runtime. In Java, this is the opposite: methods are chosen at compile time, based on the declared types.

The following code, written as Java code, can be compiled in both Java and Groovy, but it will behave differently:

```
int method(String arg) {
    return 1;
}
int method(Object arg) {
    return 2;
}
Object o = "Object";
int result = method(o);
```

In Java, you would have:

```
assertEquals(2, result);
```

Whereas in Groovy:

```
assertEquals(1, result);
```

That is because Java will use the static information type, which is that `o` is declared as an `Object`, whereas Groovy will choose at runtime, when the method is actually called. Since it is called with a `String`, then the `String` version is called.

Array initializers

In Java, array initializers take either of these two forms:

```
int[] array = {1, 2, 3};             // Java array initializer shorthand syntax
int[] array2 = new int[] {4, 5, 6};  // Java array initializer long syntax
```

In Groovy, the `{ ⋯ }` block is reserved for closures. That means that you cannot create array literals using Java's array initializer shorthand syntax. You instead borrow Groovy's literal list notation like this:

```
int[] array = [1, 2, 3]
```

For Groovy 3+, you can optionally use the Java's array initializer long syntax:

```
def array2 = new int[] {1, 2, 3} // Groovy 3.0+ supports the Java-style array
initialization long syntax
```

Package scope visibility

In Groovy, omitting a modifier on a field doesn't result in a package-private field like in Java:

```
class Person {
    String name
}
```

Instead, it is used to create a *property*, that is to say a *private field*, an associated *getter* and an associated *setter*.

It is possible to create a package-private field by annotating it with `@PackageScope`:

```
class Person {
    @PackageScope String name
}
```

ARM blocks

Java 7 introduced ARM (Automatic Resource Management) blocks (also know as try-with-resources) blocks like this:

```
Path file = Paths.get("/path/to/file");
Charset charset = Charset.forName("UTF-8");
try (BufferedReader reader = Files.newBufferedReader(file, charset)) {
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }

} catch (IOException e) {
    e.printStackTrace();
}
```

Such blocks are supported from Groovy 3+. However, Groovy provides various methods relying on closures, which have the same effect while being more idiomatic. For example:

```
new File('/path/to/file').eachLine('UTF-8') {
   println it
}
```

or, if you want a version closer to Java:

```
new File('/path/to/file').withReader('UTF-8') { reader ->
   reader.eachLine {
       println it
   }
}
```

Inner classes

WARNING	The implementation of anonymous inner classes and nested classes follow Java closely, but there are some differences, e.g. local variables accessed from within such classes don't have to be final. We piggyback on some implementation details we use for `groovy.lang.Closure` when generating inner class bytecode.

Static inner classes

Here's an example of static inner class:

```
class A {
    static class B {}
```

```
}

new A.B()
```

The usage of static inner classes is the best supported one. If you absolutely need an inner class, you should make it a static one.

Anonymous Inner Classes

```
import java.util.concurrent.CountDownLatch
import java.util.concurrent.TimeUnit

CountDownLatch called = new CountDownLatch(1)

Timer timer = new Timer()
timer.schedule(new TimerTask() {
    void run() {
        called.countDown()
    }
}, 0)

assert called.await(10, TimeUnit.SECONDS)
```

Creating Instances of Non-Static Inner Classes

In Java you can do this:

```
public class Y {
    public class X {}
    public X foo() {
        return new X();
    }
    public static X createX(Y y) {
        return y.new X();
    }
}
```

Before 3.0.0, Groovy doesn't support the `y.new X()` syntax. Instead, you have to write `new X(y)`, like in the code below:

```
public class Y {
    public class X {}
    public X foo() {
        return new X()
    }
    public static X createX(Y y) {
        return new X(y)
```

```
    }
}
```

WARNING	Caution though, Groovy supports calling methods with one parameter without giving an argument. The parameter will then have the value null. Basically the same rules apply to calling a constructor. There is a danger that you will write new X() instead of new X(this) for example. Since this might also be the regular way we have not yet found a good way to prevent this problem.

NOTE	Groovy 3.0.0 supports Java style syntax for creating instances of non-static inner classes.

Lambda expressions and the method reference operator

Java 8+ supports lambda expressions and the method reference operator (`::`):

```
Runnable run = () -> System.out.println("Run");  // Java
list.forEach(System.out::println);
```

Groovy 3 and above also support these within the Parrot parser. In earlier versions of Groovy you should use closures instead:

```
Runnable run = { println 'run' }
list.each { println it } // or list.each(this.&println)
```

Since Groovy 6.0, intersection-type casts (JLS §15.16) are supported on lambdas and method references, so a lambda can opt into `Serializable` or other marker interfaces, just like in Java:

```
Runnable r = (Runnable & Serializable) () -> println('hi') // serializable lambda
```

Groovy additionally accepts the `as` form, where parentheses are required around the intersection:

```
def r = { -> println 'hi' } as (Runnable & Serializable)
```

See the Intersection-type cast and coercion section in the semantics guide.

GStrings

As double-quoted string literals are interpreted as `GString` values, Groovy may fail with compile error or produce subtly different code if a class with `String` literal containing a dollar character is compiled with Groovy and Java compiler.

While typically, Groovy will auto-cast between `GString` and `String` if an API declares the type of a parameter, beware of Java APIs that accept an `Object` parameter and then check the actual type.

String and Character literals

Singly-quoted literals in Groovy are used for `String`, and double-quoted result in `String` or `GString`, depending whether there is interpolation in the literal.

```
assert 'c'.class == String
assert "c".class == String
assert "c${1}".class in GString
```

Groovy will automatically cast a single-character `String` to `char` only when assigning to a variable of type `char`. When calling methods with arguments of type `char` we need to either cast explicitly or make sure the value has been cast in advance.

```
char a = 'a'
assert Character.digit(a, 16) == 10: 'But Groovy does boxing'
assert Character.digit((char) 'a', 16) == 10

try {
  assert Character.digit('a', 16) == 10
  assert false: 'Need explicit cast'
} catch(MissingMethodException e) {
}
```

Groovy supports two styles of casting and in the case of casting to `char` there are subtle differences when casting a multi-char strings. The Groovy style cast is more lenient and will take the first character, while the C-style cast will fail with exception.

```
// for single char strings, both are the same
assert ((char) "c").class == Character
assert ("c" as char).class == Character

// for multi char strings they are not
try {
  ((char) 'cx') == 'c'
  assert false: 'will fail - not castable'
} catch(GroovyCastException e) {
}
assert ('cx' as char) == 'c'
assert 'cx'.asType(char) == 'c'
```

Behaviour of `==`

In Java, `==` means equality of primitive types or identity for objects. In Groovy, `==` means equality in all places. For non-primitives, it translates to `a.compareTo(b) == 0`, when evaluating equality for `Comparable` objects, and `a.equals(b)` otherwise.

To check for identity (reference equality), use the `is` method: `a.is(b)`. From Groovy 3, you can also

use the `===` operator (or negated version): `a === b` (or `c !== d`).

Primitives and wrappers

In a pure object-oriented language, everything would be an object. Java takes the stance that primitive types, such as int, boolean and double, are used very frequently and worthy of special treatment. Primitives can be efficiently stored and manipulated but can't be used in all contexts where an object could be used. Luckily, Java auto boxes and unboxes primitives when they are passed as parameters or used as return types:

```
public class Main {           // Java

   float f1 = 1.0f;
   Float f2 = 2.0f;

   float add(Float a1, float a2) { return a1 + a2; }

   Float calc() { return add(f1, f2); } ①

    public static void main(String[] args) {
       Float calcResult = new Main().calc();
       System.out.println(calcResult); // => 3.0
    }
}
```

① The `add` method expects wrapper then primitive type arguments, but we are supplying parameters with a primitive then wrapper type. Similarly, the return type from `add` is primitive, but we need the wrapper type.

Groovy does the same:

```
class Main {

    float f1 = 1.0f
    Float f2 = 2.0f

    float add(Float a1, float a2) { a1 + a2 }

    Float calc() { add(f1, f2) }
}

assert new Main().calc() == 3.0
```

Groovy, also supports primitives and object types, however, it goes a little further in pushing OO purity; it tries hard to treat *everything* as an object. Any primitive typed variable or field can be treated like an object, and it will be auto-wrapped as needed. While primitive types might be used under the covers, their use should be indistinguishable from normal object use whenever possible, and they will be boxed/unboxed as needed.

Here is a little example using Java trying to (incorrectly for Java) dereference a primitive `float`:

```
public class Main {           // Java

    public float z1 = 0.0f;

    public static void main(String[] args){
      new Main().z1.equals(1.0f); // DOESN'T COMPILE, error: float cannot be
dereferenced
    }
}
```

The same example using Groovy compiles and runs successfully:

```
class Main {
    float z1 = 0.0f
}
assert !(new Main().z1.equals(1.0f))
```

Because of Groovy's additional use of un/boxing, it does not follow Java's behavior of widening taking priority over boxing. Here's an example using `int`

```
int i
m(i)

void m(long l) {           ①
    println "in m(long)"
}

void m(Integer i) {        ②
    println "in m(Integer)"
}
```

① This is the method that Java would call, since widening has precedence over unboxing.

② This is the method Groovy actually calls, since all primitive references use their wrapper class.

Numeric Primitive Optimisation with `@CompileStatic`

Since Groovy converts to wrapper classes in more places, you might wonder whether it produces less efficient bytecode for numeric expressions. Groovy has a highly optimised set of classes for doing math computations. When using `@CompileStatic`, expressions involving only primitives uses the same bytecode that Java would use.

Positive/Negative zero edge case

Java float/double operations for both primitives and wrapper classes follow the IEEE 754 standard but there is an interesting edge case involving positive and negative zero. The standard supports

distinguishing between these two cases and while in many scenarios programmers may not care about the difference, in some mathematical or data science scenarios it is important to cater for the distinction.

For primitives, Java maps down onto a special bytecode instruction when comparing such values which has the property that "Positive zero and negative zero are considered equal".

```
jshell> float f1 = 0.0f
f1 ==> 0.0

jshell> float f2 = -0.0f
f2 ==> -0.0

jshell> f1 == f2
$3 ==> true
```

For the wrapper classes, e.g. java.base/java.lang.Float#equals(java.lang.Object), the result is `false` for this same case.

```
jshell> Float f1 = 0.0f
f1 ==> 0.0

jshell> Float f2 = -0.0f
f2 ==> -0.0

jshell> f1.equals(f2)
$3 ==> false
```

Groovy on the one hand tries to follow Java behavior closely, but on the other switches automatically between primitives and wrapped equivalents in more places. To avoid confusion we recommend the following guidelines:

- If you wish to distinguish between positive and negative zero, use the `equals` method directly or cast any primitives to their wrapper equivalent before using `==`.
- If you wish to ignore the difference between positive and negative zero, use the `equalsIgnoreZeroSign` method directly or cast any non-primitives to their primitive equivalent before using `==`.

These guidelines are illustrated in the following example:

```
float f1 = 0.0f
float f2 = -0.0f
Float f3 = 0.0f
Float f4 = -0.0f

assert f1 == f2
assert (Float) f1 != (Float) f2
```

```
assert f3 != f4         ①
assert (float) f3 == (float) f4

assert !f1.equals(f2)
assert !f3.equals(f4)

assert f1.equalsIgnoreZeroSign(f2)
assert f3.equalsIgnoreZeroSign(f4)
```

① Recall that for non-primitives, `==` maps to `.equals()`

Conversions

Java does automatic widening and narrowing conversions.

Table 10. Java Conversions

	Converts to							
Converts from	**boolean**	**byte**	**short**	**char**	**int**	**long**	**float**	**double**
boolean	-	N	N	N	N	N	N	N
byte	N	-	Y	C	Y	Y	Y	Y
short	N	C	-	C	Y	Y	Y	Y
char	N	C	C	-	Y	Y	Y	Y
int	N	C	C	C	-	Y	T	Y
long	N	C	C	C	C	-	T	T
float	N	C	C	C	C	C	-	Y
double	N	C	C	C	C	C	C	-

- 'Y' indicates a conversion Java can make
- 'C' indicates a conversion Java can make when there is an explicit cast
- 'T` indicates a conversion Java can make but data is truncated
- 'N' indicates a conversion Java can't make

Groovy expands greatly on this.

Table 11. Groovy Conversions

	Converts to

Converts from	boolean	Boolean	byte	Byte	short	Short	char	Character	int	Integer	long	Long	BigInteger	float	Float	double	Double	BigDecimal
boolean	-	B	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Boolean	B	-	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
byte	T	T	-	B	Y	Y	Y	D	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Byte	T	T	B	-	Y	Y	Y	D	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
short	T	T	D	D	-	B	Y	D	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Short	T	T	D	T	B	-	Y	D	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
char	T	T	Y	D	Y	D	-	D	Y	D	Y	D	D	Y	D	Y	D	D
Character	T	T	D	D	D	D	D	-	D	D	D	D	D	D	D	D	D	D
int	T	T	D	D	D	D	Y	D	-	B	Y	Y	Y	Y	Y	Y	Y	Y
Integer	T	T	D	D	D	D	Y	D	B	-	Y	Y	Y	Y	Y	Y	Y	Y
long	T	T	D	D	D	D	Y	D	D	D	-	B	Y	T	T	T	T	Y
Long	T	T	D	D	D	T	Y	D	D	T	B	-	Y	T	T	T	T	Y
BigInteger	T	T	D	D	D	D	D	D	D	D	D	D	-	D	D	D	D	T
float	T	T	D	D	D	D	T	D	D	D	D	D	D	-	B	Y	Y	Y

Float	T	T	D	T	D	T	T	D	D	T	D	T	D	B	-	Y	Y	Y
double	T	T	D	D	D	D	T	D	D	D	D	D	D	D	D	-	B	Y
Double	T	T	D	T	D	T	T	D	D	T	D	T	D	D	T	B	-	Y
BigDecimal	T	T	D	D	D	D	D	D	D	D	D	D	D	T	D	T	D	-

- 'Y' indicates a conversion Groovy can make
- 'D' indicates a conversion Groovy can make when compiled dynamically or explicitly cast
- 'T' indicates a conversion Groovy can make but data is truncated
- 'B' indicates a boxing/unboxing operation
- 'N' indicates a conversion Groovy can't make.

The truncation uses Groovy Truth when converting to `boolean`/`Boolean`. Converting from a number to a character casts the `Number.intvalue()` to `char`. Groovy constructs `BigInteger` and `BigDecimal` using `Number.doubleValue()` when converting from a `Float` or `Double`, otherwise it constructs using `toString()`. Other conversions have their behavior defined by `java.lang.Number`.

Extra keywords

Groovy has many of the same keywords as Java and Groovy 3 and above also has the same `var` reserved type as Java. In addition, Groovy has the following keywords:

- `as`
- `def`
- `in`
- `trait`
- `it` // within closures

Groovy is less stringent than Java in that it allows some keywords to appear in places that would be illegal in Java, e.g. the following is valid: `var var = [def: 1, as: 2, in: 3, trait: 4]`. Nevertheless, you are discouraged from using the above keywords in places that might cause confusion even when the compiler might be happy. In particular, avoid using them for variable, method and class names, so our previous `var var` example would be considered poor style.

Additional documentation is available for keywords.

Groovy Development Kit

Working with IO

Groovy provides a number of helper methods for working with I/O. While you could use standard Java code in Groovy to deal with those, Groovy provides much more convenient ways to handle files, streams, readers, …

In particular, you should take a look at methods added to:

- the `java.io.File` class : http://docs.groovy-lang.org/latest/html/groovy-jdk/java/io/File.html
- the `java.io.InputStream` class: http://docs.groovy-lang.org/latest/html/groovy-jdk/java/io/InputStream.html
- the `java.io.OutputStream` class: http://docs.groovy-lang.org/latest/html/groovy-jdk/java/io/OutputStream.html
- the `java.io.Reader` class: http://docs.groovy-lang.org/latest/html/groovy-jdk/java/io/Reader.html
- the `java.io.Writer` class: http://docs.groovy-lang.org/latest/html/groovy-jdk/java/io/Writer.html
- the `java.nio.file.Path` class: http://docs.groovy-lang.org/latest/html/groovy-jdk/java/nio/file/Path.html

The following section focuses on sample idiomatic constructs using helper methods available above but is not meant to be a complete description of all available methods. For that, please read the GDK API.

Reading files

As a first example, let's see how you would print all lines of a text file in Groovy:

```
new File(baseDir, 'haiku.txt').eachLine { line ->
    println line
}
```

The `eachLine` method is a method added to the `File` class automatically by Groovy and has many variants, for example if you need to know the line number, you can use this variant:

```
new File(baseDir, 'haiku.txt').eachLine { line, nb ->
    println "Line $nb: $line"
}
```

If for whatever reason an exception is thrown in the `eachLine` body, the method makes sure that the resource is properly closed. This is true for all I/O resource methods that Groovy adds.

For example in some cases you will prefer to use a `Reader`, but still benefit from the automatic resource management from Groovy. In the next example, the reader **will** be closed even if the exception occurs:

```
def count = 0, MAXSIZE = 3
new File(baseDir,"haiku.txt").withReader { reader ->
    while (reader.readLine()) {
        if (++count > MAXSIZE) {
            throw new RuntimeException('Haiku should only have 3 verses')
        }
    }
}
```

Should you need to collect the lines of a text file into a list, you can do:

```
def list = new File(baseDir, 'haiku.txt').collect {it}
```

Or you can even leverage the `as` operator to get the contents of the file into an array of lines:

```
def array = new File(baseDir, 'haiku.txt') as String[]
```

How many times did you have to get the contents of a file into a `byte[]` and how much code does it require? Groovy makes it very easy actually:

```
byte[] contents = file.bytes
```

Groovy also provides asynchronous variants for reading and writing files via the `groovy-nio` module. These return a `CompletableFuture`, making them composable and easy to use with async frameworks.

```
def path = file.toPath()
def future = path.textAsync          // returns CompletableFuture<String>
println future.get()                 // blocks until content is available
```

Similarly, `path.bytesAsync` returns a `CompletableFuture<byte[]>`, and `path.writeAsync(text)` writes content asynchronously.

Working with I/O is not limited to dealing with files. In fact, a lot of operations rely on input/output streams, hence why Groovy adds a lot of support methods to those, as you can see in the documentation.

As an example, you can obtain an `InputStream` from a `File` very easily:

```
def is = new File(baseDir,'haiku.txt').newInputStream()
// do something ...
is.close()
```

However you can see that it requires you to deal with closing the inputstream. In Groovy it is in general a better idea to use the `withInputStream` idiom that will take care of that for you:

```
new File(baseDir,'haiku.txt').withInputStream { stream ->
    // do something ...
}
```

Writing files

Of course in some cases you won't want to read but write a file. One of the options is to use a `Writer`:

```
new File(baseDir,'haiku.txt').withWriter('utf-8') { writer ->
    writer.writeLine 'Into the ancient pond'
    writer.writeLine 'A frog jumps'
    writer.writeLine "Water's sound!"
}
```

But for such a simple example, using the `<<` operator would have been enough:

```
new File(baseDir,'haiku.txt') << '''Into the ancient pond
A frog jumps
Water's sound!'''
```

Of course we do not always deal with text contents, so you could use the `Writer` or directly write bytes as in this example:

```
file.bytes = [66,22,11]
```

Of course you can also directly deal with output streams. For example, here is how you would create an output stream to write into a file:

```
def os = new File(baseDir,'data.bin').newOutputStream()
// do something ...
os.close()
```

However you can see that it requires you to deal with closing the output stream. Again it is in general a better idea to use the `withOutputStream` idiom that will handle the exceptions and close the stream in any case:

```
new File(baseDir,'data.bin').withOutputStream { stream ->
    // do something ...
}
```

Traversing file trees

In scripting contexts it is a common task to traverse a file tree in order to find some specific files and do something with them. Groovy provides multiple methods to do this. For example you can perform something on all files of a directory:

```
dir.eachFile { file ->                          ①
    println file.name
}
dir.eachFileMatch(~/.*\.txt/) { file ->         ②
    println file.name
}
```

① executes the closure code on each file found in the directory

② executes the closure code on files in the directory matching the specified pattern

Often you will have to deal with a deeper hierarchy of files, in which case you can use `eachFileRecurse`:

```
dir.eachFileRecurse { file ->                       ①
    println file.name
}

dir.eachFileRecurse(FileType.FILES) { file ->       ②
    println file.name
}
```

① executes the closure code on each file or directory found in the directory, recursively

② executes the closure code only on files, but recursively

For more complex traversal techniques you can use the `traverse` method, which requires you to set a special flag indicating what to do with the traversal:

```
dir.traverse { file ->
    if (file.directory && file.name=='bin') {
        FileVisitResult.TERMINATE                   ①
    } else {
        println file.name
        FileVisitResult.CONTINUE                    ②
    }

}
```

① if the current file is a directory and its name is `bin`, stop the traversal

② otherwise print the file name and continue

Data and objects

In Java it is not uncommon to serialize and deserialize data using the `java.io.DataOutputStream` and `java.io.DataInputStream` classes respectively. Groovy will make it even easier to deal with them. For example, you could serialize data into a file and deserialize it using this code:

```
boolean b = true
String message = 'Hello from Groovy'
// Serialize data into a file
file.withDataOutputStream { out ->
    out.writeBoolean(b)
    out.writeUTF(message)
}
// ...
// Then read it back
file.withDataInputStream { input ->
    assert input.readBoolean() == b
    assert input.readUTF() == message
}
```

And similarly, if the data you want to serialize implements the `Serializable` interface, you can proceed with an object output stream, as illustrated here:

```
Person p = new Person(name:'Bob', age:76)
// Serialize data into a file
file.withObjectOutputStream { out ->
    out.writeObject(p)
}
// ...
// Then read it back
file.withObjectInputStream { input ->
    def p2 = input.readObject()
    assert p2.name == p.name
    assert p2.age == p.age
}
```

Executing External Processes

The previous section described how easy it was to deal with files, readers or streams in Groovy. However in domains like system administration or devops it is often required to communicate with external processes.

Groovy provides `execute` and `toProcessBuilder` methods as two mechanisms to execute external processes. In simple cases, calls to `execute()` are invoked directly on a command string:

```
"javac src/main/java/*.java".execute()
```

For more sophisticated workloads, a *named-parameter* variant configures the process with options like working directory, environment, and stream redirection:

```
def result = "javac src/main/java/*.java".execute(
    dir: new File('/myproject'),
    redirectErrorStream: true
).waitForResult(2, TimeUnit.MINUTES)

if (!result.ok) System.err.println result.out
```

The `toProcessBuilder()` method lets you piggyback directly onto the `ProcessBuilder` fluent API:

```
def result = "javac src/main/java/*.java".toProcessBuilder()
    .directory(new File('/myproject'))
    .redirectErrorStream(true)
    .start()
    .waitForResult(2, TimeUnit.MINUTES)

if (!result.ok) System.err.println result.out
```

The `execute()` and `start()` methods return a `java.lang.Process` instance, and `waitForResult()` returns a `ProcessResult` capturing stdout, stderr, and the exit code. The following sections cover each approach in detail.

Simple execution

Simply write the command line as a string and call the `execute()` method. E.g., on a *nix machine (or a Windows machine with appropriate *nix commands installed), you can execute this:

```
def process = "ls -l".execute()       ①
println "Files: $process.text"                ②
```

① executes the `ls` command in an external process

② consume the output of the command and retrieve the text

Here is the same command as above but we will now process the resulting stream a line at a time:

```
def process = "ls -l".execute()             ①
process.in.eachLine { line ->               ②
    println line                            ③
}
```

① executes the `ls` command in an external process

② for each line of the input stream of the process

③ print the line

It is worth noting that `in` corresponds to an input stream to the standard output of the command. `out` will refer to a stream where you can send data to the process (its standard input).

Remember that many commands are shell built-ins and need special handling. So if you want a listing of files in a directory on a Windows machine and write:

```
def process = "dir".execute()
println "Files: $process.text"
```

you will receive an `IOException` saying *Cannot run program "dir": CreateProcess error=2, The system cannot find the file specified.*

This is because `dir` is built-in to the Windows shell (`cmd.exe`) and can't be run as a simple executable. Instead, you will need to write:

```
def process = "cmd /c dir".execute()
println "Files: $process.text"
```

Named parameters for execute

The `execute()` method accepts named parameters to configure the process. Supported options include `dir` (working directory), `env` (environment variables), `redirectErrorStream`, `inheritIO`, `outputFile`, `errorFile`, `inputFile`, `appendOutput`, and `appendError`. The file-related options (`dir`, `outputFile`, `errorFile`, `inputFile`, `appendOutput`, `appendError`) accept a `File`, `Path`, or `String`:

```
def process = "ls".execute(dir: tmpDir)
def result = process.waitForResult()
assert result.ok
assert result.out.contains('test.txt')
```

To merge stderr into stdout:

```
def process = "ls /nonexistent_path_xxx".execute(redirectErrorStream: true)
def output = process.text
assert output.length() > 0  // error message now in stdout
```

Using ProcessBuilder directly

For full control over process construction, the `toProcessBuilder()` method creates a `java.lang.ProcessBuilder` from a command string, array, or list:

```
def pb = "echo Hello".toProcessBuilder()
pb.redirectErrorStream(true)
def process = pb.start()
assert process.text.trim() == 'Hello'
```

Capturing output, error, and exit code

A common need is to capture a process's standard output, standard error, and exit code all at once. The `waitForResult()` method returns a `ProcessResult` with `out`, `err`, `exitCode`, and `ok` properties:

```
def result = "echo Hello World".execute().waitForResult()
assert result.ok
assert result.out.trim() == 'Hello World'
assert result.exitCode == 0
```

If the process produces an error:

```
def result = "ls /nonexistent_path_xxx".execute().waitForResult()
assert !result.ok
assert result.err.length() > 0
```

A timeout variant is available that forcibly kills the process if it exceeds the given duration:

```
def result = "echo Fast".execute().waitForResult(10, TimeUnit.SECONDS)
assert result.ok
assert result.out.trim() == 'Fast'
```

Stream handling

Because some native platforms only provide limited buffer size for standard input and output streams, failure to promptly write the input stream or read the output stream of the subprocess may cause the subprocess to block, and even deadlock.

Because of this, Groovy provides some additional helper methods which make stream handling for processes easier.

Here is how to gobble all of the output (including the error stream output) from your process:

```
def p = "rm -f foo.tmp".execute([], tmpDir)
p.consumeProcessOutput()
p.waitFor()
```

There are also variations of `consumeProcessOutput` that make use of `StringBuffer`, `InputStream`, `OutputStream` etc... For a complete list, please read the GDK API for java.lang.Process

Piping

The `pipeTo` command (mapped to `|` to allow overloading) lets the output stream of one process be fed into the input stream of another process.

Here are some examples of use:

Pipes in action

```
proc1 = 'ls'.execute()
proc2 = 'tr -d o'.execute()
proc3 = 'tr -d e'.execute()
proc4 = 'tr -d i'.execute()
proc1 | proc2 | proc3 | proc4
proc4.waitFor()
if (proc4.exitValue()) {
    println proc4.err.text
} else {
    println proc4.text
}
```

Consuming errors

```
def sout = new StringBuilder()
def serr = new StringBuilder()
proc2 = 'tr -d o'.execute()
proc3 = 'tr -d e'.execute()
proc4 = 'tr -d i'.execute()
proc4.consumeProcessOutput(sout, serr)
proc2 | proc3 | proc4
[proc2, proc3].each { it.consumeProcessErrorStream(serr) }
proc2.withWriter { writer ->
    writer << 'testfile.groovy'
}
proc4.waitForOrKill(1000)
println "Standard output: $sout"
println "Standard error: $serr"
```

For more robust piping, the `pipeline()` method uses native OS-level piping via `ProcessBuilder.startPipeline()`:

```
def procs = ["echo one two three", "wc -w"].pipeline()
def result = procs.last().waitForResult()
assert result.ok
assert result.out.trim() == '3'
```

Asynchronous notifications

The `onExit` method registers a closure to be called when the process terminates:

```
def latch = new CountDownLatch(1)
def exitCode = -1
"echo done".execute().onExit { proc ->
    exitCode = proc.exitValue()
    latch.countDown()
```

```
}
latch.await(10, TimeUnit.SECONDS)
assert exitCode == 0
```

Working with collections

Groovy provides native support for various collection types, including lists, maps or ranges. Most of those are based on the Java collection types and decorated with additional methods found in the Groovy development kit.

Lists

List literals

You can create lists as follows. Notice that `[]` is the empty list expression.

```
def list = [5, 6, 7, 8]
assert list.get(2) == 7
assert list[2] == 7
assert list instanceof java.util.List

def emptyList = []
assert emptyList.size() == 0
emptyList.add(5)
assert emptyList.size() == 1
```

Each list expression creates an implementation of java.util.List.

Of course lists can be used as a source to construct another list:

```
def list1 = ['a', 'b', 'c']
//construct a new list, seeded with the same items as in list1
def list2 = new ArrayList<String>(list1)

assert list2 == list1 // == checks that each corresponding element is the same

// clone() can also be called
def list3 = list1.clone()
assert list3 == list1
```

A list is an ordered collection of objects:

```
def list = [5, 6, 7, 8]
assert list.size() == 4
assert list.getClass() == ArrayList     // the specific kind of list being used

assert list[2] == 7                     // indexing starts at 0
```

```
assert list.getAt(2) == 7              // equivalent method to subscript operator []
assert list.get(2) == 7                // alternative method

list[2] = 9
assert list == [5, 6, 9, 8,]          // trailing comma OK

list.putAt(2, 10)                      // equivalent method to [] when value being
changed
assert list == [5, 6, 10, 8]
assert list.set(2, 11) == 10           // alternative method that returns old value
assert list == [5, 6, 11, 8]

assert ['a', 1, 'a', 'a', 2.5, 2.5f, 2.5d, 'hello', 7g, null, 9 as byte]
//objects can be of different types; duplicates allowed

assert [1, 2, 3, 4, 5][-1] == 5            // use negative indices to count from the
end
assert [1, 2, 3, 4, 5][-2] == 4
assert [1, 2, 3, 4, 5].getAt(-2) == 4      // getAt() available with negative
index...
try {
    [1, 2, 3, 4, 5].get(-2)                // but negative index not allowed with
get()
    assert false
} catch (e) {
    assert e instanceof IndexOutOfBoundsException
}
```

List as a boolean expression

Lists can be evaluated as a `boolean` value:

```
assert ![]            // an empty list evaluates as false

//all other lists, irrespective of contents, evaluate as true
assert [1] && ['a'] && [0] && [0.0] && [false] && [null]
```

Iterating on a list

Iterating on elements of a list is usually done calling the `each` and `eachWithIndex` methods, which execute code on each item of a list:

```
[1, 2, 3].each {
    println "Item: $it" // `it` is an implicit parameter corresponding to the current
element
}
['a', 'b', 'c'].eachWithIndex { it, i -> // `it` is the current element, while `i` is
the index
    println "$i: $it"
```

```
}
```

In addition to iterating, it is often useful to create a new list by transforming each of its elements into something else. This operation, often called mapping, is done in Groovy thanks to the `collect` method:

```
assert [1, 2, 3].collect { it * 2 } == [2, 4, 6]

// shortcut syntax instead of collect
assert [1, 2, 3]*.multiply(2) == [1, 2, 3].collect { it.multiply(2) }

def list = [0]
// it is possible to give `collect` the list which collects the elements
assert [1, 2, 3].collect(list) { it * 2 } == [0, 2, 4, 6]
assert list == [0, 2, 4, 6]
```

When each element maps to a collection of results, `collectMany` transforms and flattens in a single step. It is the idiomatic (and more efficient) alternative to `collect { ⋯ }.flatten()`:

```
def words = ['a', 'bb', 'ccc']
// collectMany transforms each element into a collection, then flattens one level;
// it is the idiomatic (and more efficient) alternative to collect {}.flatten()
assert words.collectMany { [it, it.size()] } == ['a', 1, 'bb', 2, 'ccc', 3]

// handy for expanding each element into zero, one or many results
assert [1, 2, 3].collectMany { it % 2 == 0 ? [it, it] : [] } == [2, 2]
```

To build a `Map` from a list, `collectEntries` collects a key/value pair for each element:

```
def languages = ['Groovy', 'Java', 'Kotlin']
// collectEntries builds a map from a list; the closure returns a [key, value] pair
per element
assert languages.collectEntries { [it, it.size()] } == [Groovy: 6, Java: 4, Kotlin: 6]

// a map literal can be returned instead of a two-element list
assert languages.collectEntries { [(it): it.toUpperCase()] } ==
       [Groovy: 'GROOVY', Java: 'JAVA', Kotlin: 'KOTLIN']
```

When you need each element together with its position, `withIndex` provides a more composable alternative to `eachWithIndex`, while `indexed` returns a map from index to element:

```
def vowels = ['a', 'e', 'i', 'o', 'u']

// withIndex pairs each element with its index (starting at 0 by default)
assert vowels.withIndex() == [['a', 0], ['e', 1], ['i', 2], ['o', 3], ['u', 4]]
```

```
// an optional offset sets the starting index
assert vowels.withIndex(1) == [['a', 1], ['e', 2], ['i', 3], ['o', 4], ['u', 5]]

// often chained with collect for a cleaner alternative to eachWithIndex
assert vowels.withIndex().collect { v, i -> "$i:$v" } == ['0:a', '1:e', '2:i', '3:o',
'4:u']

// indexed returns a map from index to element
assert vowels.indexed(1) == [1: 'a', 2: 'e', 3: 'i', 4: 'o', 5: 'u']
```

Manipulating lists

Filtering and searching

The Groovy development kit contains a lot of methods on collections that enhance the standard collections with pragmatic methods, some of which are illustrated here:

```
assert [1, 2, 3].find { it > 1 } == 2          // find 1st element matching criteria
assert [1, 2, 3].findAll { it > 1 } == [2, 3]  // find all elements matching
critieria
assert ['a', 'b', 'c', 'd', 'e'].findIndexOf {     // find index of 1st element
matching criteria
    it in ['c', 'e', 'g']
} == 2

assert ['a', 'b', 'c', 'd', 'c'].indexOf('c') == 2  // index returned
assert ['a', 'b', 'c', 'd', 'c'].indexOf('z') == -1 // index -1 means value not in
list
assert ['a', 'b', 'c', 'd', 'c'].lastIndexOf('c') == 4

assert [1, 2, 3].every { it < 5 }              // returns true if all elements match
the predicate
assert ![1, 2, 3].every { it < 3 }
assert [1, 2, 3].any { it > 2 }                // returns true if any element matches
the predicate
assert ![1, 2, 3].any { it > 3 }

assert [1, 2, 3, 4, 5, 6].sum() == 21               // sum anything with a plus()
method
assert ['a', 'b', 'c', 'd', 'e'].sum {
    it == 'a' ? 1 : it == 'b' ? 2 : it == 'c' ? 3 : it == 'd' ? 4 : it == 'e' ? 5 : 0
    // custom value to use in sum
} == 15
assert ['a', 'b', 'c', 'd', 'e'].sum { ((char) it) - ((char) 'a') } == 10
assert ['a', 'b', 'c', 'd', 'e'].sum() == 'abcde'
assert [['a', 'b'], ['c', 'd']].sum() == ['a', 'b', 'c', 'd']

// an initial value can be provided
assert [].sum(1000) == 1000
assert [1, 2, 3].sum(1000) == 1006
```

```
assert [1, 2, 3].join('-') == '1-2-3'           // String joining
assert [1, 2, 3].inject('counting: ') {
    str, item -> str + item                     // reduce operation
} == 'counting: 123'
assert [1, 2, 3].inject(0) { count, item ->
    count + item
} == 6
```

And here is idiomatic Groovy code for finding the maximum and minimum in a collection:

```
def list = [9, 4, 2, 10, 5]
assert list.max() == 10
assert list.min() == 2

// we can also compare single characters, as anything comparable
assert ['x', 'y', 'a', 'z'].min() == 'a'

// we can use a closure to specify the sorting behaviour
def list2 = ['abc', 'z', 'xyzuvw', 'Hello', '321']
assert list2.max { it.size() } == 'xyzuvw'
assert list2.min { it.size() } == 'z'
```

In addition to closures, you can use a `Comparator` to define the comparison criteria:

```
Comparator mc = { a, b -> a == b ? 0 : (a < b ? -1 : 1) }

def list = [7, 4, 9, -6, -1, 11, 2, 3, -9, 5, -13]
assert list.max(mc) == 11
assert list.min(mc) == -13

Comparator mc2 = { a, b -> a == b ? 0 : (Math.abs(a) < Math.abs(b)) ? -1 : 1 }


assert list.max(mc2) == -13
assert list.min(mc2) == -1

assert list.max { a, b -> a.equals(b) ? 0 : Math.abs(a) < Math.abs(b) ? -1 : 1 } == -
13
assert list.min { a, b -> a.equals(b) ? 0 : Math.abs(a) < Math.abs(b) ? -1 : 1 } == -1
```

Adding or removing elements

We can use `[]` to assign a new empty list and `<<` to append items to it:

```
def list = []
assert list.empty
```

```
list << 5
assert list.size() == 1

list << 7 << 'i' << 11
assert list == [5, 7, 'i', 11]

list << ['m', 'o']
assert list == [5, 7, 'i', 11, ['m', 'o']]

//first item in chain of << is target list
assert ([1, 2] << 3 << [4, 5] << 6) == [1, 2, 3, [4, 5], 6]

//using leftShift is equivalent to using <<
assert ([1, 2, 3] << 4) == ([1, 2, 3].leftShift(4))
```

We can add to a list in many ways:

```
assert [1, 2] + 3 + [4, 5] + 6 == [1, 2, 3, 4, 5, 6]
// equivalent to calling the `plus` method
assert [1, 2].plus(3).plus([4, 5]).plus(6) == [1, 2, 3, 4, 5, 6]

def a = [1, 2, 3]
a += 4      // creates a new list and assigns it to `a`
a += [5, 6]
assert a == [1, 2, 3, 4, 5, 6]

assert [1, *[222, 333], 456] == [1, 222, 333, 456]
assert [*[1, 2, 3]] == [1, 2, 3]
assert [1, [2, 3, [4, 5], 6], 7, [8, 9]].flatten() == [1, 2, 3, 4, 5, 6, 7, 8, 9]

def list = [1, 2]
list.add(3)
list.addAll([5, 4])
assert list == [1, 2, 3, 5, 4]

list = [1, 2]
list.add(1, 3) // add 3 just before index 1
assert list == [1, 3, 2]

list.addAll(2, [5, 4]) //add [5,4] just before index 2
assert list == [1, 3, 5, 4, 2]

list = ['a', 'b', 'z', 'e', 'u', 'v', 'g']
list[8] = 'x' // the [] operator is growing the list as needed
// nulls inserted if required
assert list == ['a', 'b', 'z', 'e', 'u', 'v', 'g', null, 'x']
```

It is however important that the + operator on a list is **not mutating**. Compared to <<, it will create a

new list, which is often not what you want and can lead to performance issues.

The Groovy development kit also contains methods allowing you to easily remove elements from a list by value:

```
assert ['a','b','c','b','b'] - 'c' == ['a','b','b','b']
assert ['a','b','c','b','b'] - 'b' == ['a','c']
assert ['a','b','c','b','b'] - ['b','c'] == ['a']

def list = [1,2,3,4,3,2,1]
list -= 3          // creates a new list by removing `3` from the original one
assert list == [1,2,4,2,1]
assert ( list -= [2,4] ) == [1,1]
```

It is also possible to remove an element by passing its index to the `remove` method, in which case the list is mutated:

```
def list = ['a','b','c','d','e','f','b','b','a']
assert list.remove(2) == 'c'        // remove the third element, and return it
assert list == ['a','b','d','e','f','b','b','a']
```

In case you only want to remove the first element having the same value in a list, instead of removing all elements, you can call the `remove` method passing the value:

```
def list= ['a','b','c','b','b']
assert list.remove('c')             // remove 'c', and return true because element
removed
assert list.remove('b')             // remove first 'b', and return true because
element removed

assert ! list.remove('z')           // return false because no elements removed
assert list == ['a','b','b']
```

As you can see, there are two `remove` methods available. One that takes an integer and removes an element by its index, and another that will remove the first element that matches the passed value. So what should we do when we have a list of integers? In this case, you may wish to use `removeAt` to remove an element by its index, and `removeElement` to remove the first element that matches a value.

```
def list = [1,2,3,4,5,6,2,2,1]

assert list.remove(2) == 3          // this removes the element at index 2, and
returns it
assert list == [1,2,4,5,6,2,2,1]

assert list.removeElement(2)        // remove first 2 and return true
```

```
assert list == [1,4,5,6,2,2,1]

assert ! list.removeElement(8)      // return false because 8 is not in the list
assert list == [1,4,5,6,2,2,1]

assert list.removeAt(1) == 4        // remove element at index 1, and return it
assert list == [1,5,6,2,2,1]
```

Of course, `removeAt` and `removeElement` will work with lists of any type.

Additionally, removing all the elements in a list can be done by calling the `clear` method:

```
def list= ['a',2,'c',4]
list.clear()
assert list == []
```

Set operations

The Groovy development kit also includes methods making it easy to reason on sets:

```
assert 'a' in ['a','b','c']            // returns true if an element belongs to the
list
assert ['a','b','c'].contains('a')     // equivalent to the `contains` method in Java
assert [1,3,4].containsAll([1,4])      // `containsAll` will check that all elements
are found

assert [1,2,3,3,3,3,4,5].count(3) == 4  // count the number of elements which have
some value
assert [1,2,3,3,3,3,4,5].count {
    it%2==0                            // count the number of elements which match
the predicate
} == 2

assert [1,2,4,6,8,10,12].intersect([1,3,6,9,12]) == [1,6,12]

assert [1,2,3].disjoint( [4,6,9] )
assert ![1,2,3].disjoint( [2,4,6] )
```

Sorting

Working with collections often implies sorting. Groovy offers a variety of options to sort lists, from using closures to comparators, as in the following examples:

```
assert [6, 3, 9, 2, 7, 1, 5].sort() == [1, 2, 3, 5, 6, 7, 9]

def list = ['abc', 'z', 'xyzuvw', 'Hello', '321']
assert list.sort {
```

```
    it.size()
} == ['z', 'abc', '321', 'Hello', 'xyzuvw']

def list2 = [7, 4, -6, -1, 11, 2, 3, -9, 5, -13]
assert list2.sort { a, b -> a == b ? 0 : Math.abs(a) < Math.abs(b) ? -1 : 1 } ==
        [-1, 2, 3, 4, 5, -6, 7, -9, 11, -13]

Comparator mc = { a, b -> a == b ? 0 : Math.abs(a) < Math.abs(b) ? -1 : 1 }

// JDK 8+ only
// list2.sort(mc)
// assert list2 == [-1, 2, 3, 4, 5, -6, 7, -9, 11, -13]

def list3 = [6, -3, 9, 2, -7, 1, 5]

Collections.sort(list3)
assert list3 == [-7, -3, 1, 2, 5, 6, 9]

Collections.sort(list3, mc)
assert list3 == [1, 2, -3, 5, 6, -7, 9]
```

Removing duplicates

The `unique` method removes duplicate elements **in place**, while `toUnique` returns a **new** list, leaving the original untouched. Both accept an optional closure or comparator to define what "the same" means:

```
// unique removes duplicates IN PLACE and returns the (now mutated) list
def list = [3, 2, 1, 3, 3, 2]
assert list.unique() == [3, 2, 1]
assert list == [3, 2, 1]                          // the original list was mutated

// toUnique returns a NEW list, leaving the original untouched
def fruits = ['apple', 'apricot', 'banana', 'blueberry']
assert fruits.toUnique { it[0] } == ['apple', 'banana']  // uniqueness by first letter
assert fruits.size() == 4                         // original list unchanged
```

Taking and dropping elements

Groovy can select or discard elements from either end of a list, either by count (`take`, `drop`, `takeRight`, `dropRight`) or by predicate (`takeWhile`, `dropWhile`). None of these methods mutate the original list:

```
def list = [1, 2, 3, 4, 5]

// take/drop select or discard a number of leading elements
assert list.take(2) == [1, 2]
assert list.drop(2) == [3, 4, 5]
```

```
// takeRight/dropRight work from the end of the list
assert list.takeRight(2) == [4, 5]
assert list.dropRight(2) == [1, 2, 3]

// takeWhile/dropWhile use a predicate instead of a count, stopping at the first
mismatch
assert list.takeWhile { it < 3 } == [1, 2]
assert list.dropWhile { it < 3 } == [3, 4, 5]

assert list == [1, 2, 3, 4, 5]                    // none of the above mutate the
original
```

Duplicating elements

The Groovy development kit also takes advantage of operator overloading to provide methods allowing duplication of elements of a list:

```
assert [1, 2, 3] * 3 == [1, 2, 3, 1, 2, 3, 1, 2, 3]
assert [1, 2, 3].multiply(2) == [1, 2, 3, 1, 2, 3]
assert Collections.nCopies(3, 'b') == ['b', 'b', 'b']

// nCopies from the JDK has different semantics than multiply for lists
assert Collections.nCopies(2, [1, 2]) == [[1, 2], [1, 2]] //not [1,2,1,2]
```

Maps

Map literals

In Groovy, maps (also known as associative arrays) can be created using the map literal syntax: `[:]`:

```
def map = [name: 'Gromit', likes: 'cheese', id: 1234]
assert map.get('name') == 'Gromit'
assert map.get('id') == 1234
assert map['name'] == 'Gromit'
assert map['id'] == 1234
assert map instanceof java.util.Map

def emptyMap = [:]
assert emptyMap.size() == 0
emptyMap.put("foo", 5)
assert emptyMap.size() == 1
assert emptyMap.get("foo") == 5
```

Map keys are strings by default: `[a:1]` is equivalent to `['a':1]`. This can be confusing if you define a variable named `a` and that you want the **value** of `a` to be the key in your map. If this is the case, then you **must** escape the key by adding parenthesis, like in the following example:

```
def a = 'Bob'
def ages = [a: 43]
assert ages['Bob'] == null // `Bob` is not found
assert ages['a'] == 43     // because `a` is a literal!

ages = [(a): 43]            // now we escape `a` by using parenthesis
assert ages['Bob'] == 43   // and the value is found!
```

In addition to map literals, it is possible, to get a new copy of a map, to clone it:

```
def map = [
        simple : 123,
        complex: [a: 1, b: 2]
]
def map2 = map.clone()
assert map2.get('simple') == map.get('simple')
assert map2.get('complex') == map.get('complex')
map2.get('complex').put('c', 3)
assert map.get('complex').get('c') == 3
```

The resulting map is a **shallow** copy of the original one, as illustrated in the previous example.

Map property notation

Maps also act like beans so you can use the property notation to get/set items inside the `Map` as long as the keys are strings which are valid Groovy identifiers:

```
def map = [name: 'Gromit', likes: 'cheese', id: 1234]
assert map.name == 'Gromit'     // can be used instead of map.get('name')
assert map.id == 1234

def emptyMap = [:]
assert emptyMap.size() == 0
emptyMap.foo = 5
assert emptyMap.size() == 1
assert emptyMap.foo == 5
```

Note: by design `map.foo` will always look for the key `foo` in the map. This means `foo.class` will return `null` on a map that doesn't contain the `class` key. Should you really want to know the class, then you must use `getClass()`:

```
def map = [name: 'Gromit', likes: 'cheese', id: 1234]
assert map.class == null
assert map.get('class') == null
assert map.getClass() == LinkedHashMap // this is probably what you want

map = [1      : 'a',
```

```
        (true) : 'p',
        (false): 'q',
        (null) : 'x',
        'null' : 'z']
assert map.containsKey(1) // 1 is not an identifier so used as is
assert map.true == null
assert map.false == null
assert map.get(true) == 'p'
assert map.get(false) == 'q'
assert map.null == 'z'
assert map.get(null) == 'x'
```

Property access versus subscript and `get`

Because `map.k` reads the **key** `k` (rather than a bean property), property access, the subscript operator and `get` can diverge once a key collides with a real property. Every cell in the tables below is included from a passing assertion in `WorkingWithCollectionsTest`, so they always reflect the current runtime.

For a **plain** map there are only entries, so all four forms agree:

Table 12. Plain map (`[foo:'bar']`, `[class:'C']`, `[:]`)

key	`m.k`	`m.k` (`@CompileStatic`)	`m['k']`	`m.get('k')`
ordinary (`'foo'`)	'bar'	'bar'	'bar'	'bar'
meta-name present (`'class'`)	'C'	'C'	'C'	'C'
meta-name absent (`'class'`)	null	null	null	null

A map **subclass** can declare real properties. Then `m.k` reads the property (identically under dynamic and `@CompileStatic` compilation), while `m['k']` and `m.get('k')` read the entry:

```
static class MapWithProps extends HashMap<String, Object> {
    def foo = 1
    def bar = 2
}
```

then, after `m.put('foo', 11)` and `m.put('baz', 33)`:

Table 13. Map subclass with declared properties `foo`, `bar`

key	`m.k`	`m.k` (`@CompileStatic`)	`m['k']`	`m.get('k')`
`'foo'` (property=1, entry=11)	1	1	11	11
`'bar'` (property=2, no entry)	2	2	null	null

key	`m.k`	`m.k` (`@CompileStatic`)	`m['k']`	`m.get('k')`
`'baz'` (no property, entry=33)	33	33	33	33

Two things to take away: after the addition of `getAt(Map, String)`/`putAt(Map, String, V)`, `m['k']` is equivalent to `m.get('k')` in every row and both compilation modes; and `m.k` resolves to a declared property when one exists (use `['k']`/`get` to be sure of reading the entry).

The tables above all use an explicit receiver `m`. A separate gotcha applies to an **unqualified** name read inside the body of a `Map` subclass — a method, a `with(map) { … }` block, or an anonymous-inner-class initializer. There the implicit receiver is the map itself, so the name is a property read on `this` and resolves to the **entry**, not to any binding in an enclosing scope. From Groovy 5, this self property takes precedence over an enclosing-class field of the same name (see GROOVY-10985), so such a reference silently yields `null` for a missing entry rather than reading the field:

```
class Outer {
    static final String NAME = 'from-enclosing-field'

    // An unqualified name in the body of a Map subclass is a property read on
    // `this`, so it reads the (missing) entry and shadows the enclosing field:
    static Map<String, String> shadowed  = new HashMap<String, String>() { { put('k',
NAME)       } }

    // Qualifying the reference resolves the enclosing field:
    static Map<String, String> qualified = new HashMap<String, String>() { { put('k',
Outer.NAME) } }
}

assert Outer.shadowed['k']  == null                    // 'NAME' read as a (missing)
entry
assert Outer.qualified['k'] == 'from-enclosing-field'   // the enclosing field
```

Qualify the outer reference (`Outer.NAME` above) to read the enclosing field.

Iterating on maps

As usual in the Groovy development kit, idiomatic iteration on maps makes use of the `each` and `eachWithIndex` methods. It's worth noting that maps created using the map literal notation are **ordered**, that is to say that if you iterate on map entries, it is guaranteed that the entries will be returned in the same order they were added in the map.

```
def map = [
        Bob  : 42,
        Alice: 54,
        Max  : 33
]
```

```
// `entry` is a map entry
map.each { entry ->
    println "Name: $entry.key Age: $entry.value"
}

// `entry` is a map entry, `i` the index in the map
map.eachWithIndex { entry, i ->
    println "$i - Name: $entry.key Age: $entry.value"
}

// Alternatively you can use key and value directly
map.each { key, value ->
    println "Name: $key Age: $value"
}

// Key, value and i as the index in the map
map.eachWithIndex { key, value, i ->
    println "$i - Name: $key Age: $value"
}
```

Manipulating maps

Adding or removing elements

Adding an element to a map can be done either using the `put` method, the subscript operator or using `putAll`:

```
def defaults = [1: 'a', 2: 'b', 3: 'c', 4: 'd']
def overrides = [2: 'z', 5: 'x', 13: 'x']

def result = new LinkedHashMap(defaults)
result.put(15, 't')
result[17] = 'u'
result.putAll(overrides)
assert result == [1: 'a', 2: 'z', 3: 'c', 4: 'd', 5: 'x', 13: 'x', 15: 't', 17: 'u']
```

Removing all the elements of a map can be done by calling the `clear` method:

```
def m = [1:'a', 2:'b']
assert m.get(1) == 'a'
m.clear()
assert m == [:]
```

Maps generated using the map literal syntax are using the object `equals` and `hashcode` methods. This means that you should **never** use an object which hash code is subject to change over time, or you wouldn't be able to get the associated value back.

It is also worth noting that you should **never** use a `GString` as the key of a map, because the hash

code of a `GString` is not the same as the hash code of an equivalent `String`:

```
def key = 'some key'
def map = [:]
def gstringKey = "${key.toUpperCase()}"
map.put(gstringKey,'value')
assert map.get('SOME KEY') == null
```

Keys, values and entries

We can inspect the keys, values, and entries in a view:

```
def map = [1:'a', 2:'b', 3:'c']

def entries = map.entrySet()
entries.each { entry ->
  assert entry.key in [1,2,3]
  assert entry.value in ['a','b','c']
}

def keys = map.keySet()
assert keys == [1,2,3] as Set
```

Mutating values returned by the view (be it a map entry, a key or a value) is highly discouraged because success of the operation directly depends on the type of the map being manipulated. In particular, Groovy relies on collections from the JDK that in general make no guarantee that a collection can safely be manipulated through `keySet`, `entrySet`, or `values`.

Filtering and searching

The Groovy development kit contains filtering, searching and collecting methods similar to those found for lists:

```
def people = [
    1: [name:'Bob', age: 32, gender: 'M'],
    2: [name:'Johnny', age: 36, gender: 'M'],
    3: [name:'Claire', age: 21, gender: 'F'],
    4: [name:'Amy', age: 54, gender:'F']
]

def bob = people.find { it.value.name == 'Bob' } // find a single entry
def females = people.findAll { it.value.gender == 'F' }

// both return entries, but you can use collect to retrieve the ages for example
def ageOfBob = bob.value.age
def agesOfFemales = females.collect {
    it.value.age
}
```

```
assert ageOfBob == 32
assert agesOfFemales == [21,54]

// but you could also use a key/pair value as the parameters of the closures
def agesOfMales = people.findAll { id, person ->
    person.gender == 'M'
}.collect { id, person ->
    person.age
}
assert agesOfMales == [32, 36]

// `every` returns true if all entries match the predicate
assert people.every { id, person ->
    person.age > 18
}

// `any` returns true if any entry matches the predicate

assert people.any { id, person ->
    person.age == 54
}
```

Grouping

We can group a list into a map using some criteria:

```
assert ['a', 7, 'b', [2, 3]].groupBy {
    it.class
} == [(String)   : ['a', 'b'],
      (Integer)  : [7],
      (ArrayList): [[2, 3]]
]

assert [
        [name: 'Clark', city: 'London'], [name: 'Sharma', city: 'London'],
        [name: 'Maradona', city: 'LA'], [name: 'Zhang', city: 'HK'],
        [name: 'Ali', city: 'HK'], [name: 'Liu', city: 'HK'],
].groupBy { it.city } == [
        London: [[name: 'Clark', city: 'London'],
                 [name: 'Sharma', city: 'London']],
        LA    : [[name: 'Maradona', city: 'LA']],
        HK    : [[name: 'Zhang', city: 'HK'],
                 [name: 'Ali', city: 'HK'],
                 [name: 'Liu', city: 'HK']],
]
```

Maps with a default value

The `withDefault` method wraps a map so that reading a missing key returns (and stores) a value computed by a closure. This is particularly handy for accumulating into per-key collections without having to pre-populate them:

```
// withDefault wraps a map so that missing keys are computed on demand
def wordsByLetter = [:].withDefault { [] }
['apple', 'banana', 'avocado', 'cherry'].each { word ->
    wordsByLetter[word[0]] << word          // no need to pre-populate the per-letter
lists
}
assert wordsByLetter == [a: ['apple', 'avocado'], b: ['banana'], c: ['cherry']]

// reading an absent key returns (and stores) the computed default
def counts = [:].withDefault { 0 }
assert counts['missing'] == 0
```

NOTE Since Groovy 6, under static compilation, calling `withDefault` on a map with statically-known key/value types automatically selects the type-checked variant: the wrapped map rejects ill-typed keys and values at runtime instead of silently growing entries that violate the map's declared generics (GROOVY-11807). Code that (usually inadvertently) relied on the lenient behaviour can defer the upgrade with the `groovy.classtag.preemption.disable` system property — see the `@ClassTag` documentation for details. Dynamic code is unaffected.

Ranges

Ranges allow you to create a list of sequential values. These can be used as `List` since Range extends java.util.List.

Ranges defined with the `..` notation are inclusive (that is the list contains the from and to value).

Ranges defined with the `..<` notation are half-open, they include the first value but not the last value.

Ranges defined with the `<..` notation are also half-open, they include the last value but not the first value.

Ranges defined with the `<..<` notation are full-open, they do not include the first value nor the last value.

```
// an inclusive range
def range = 5..8
assert range.size() == 4
assert range.get(2) == 7
assert range[2] == 7
assert range instanceof java.util.List
assert range.contains(5)
```

```
assert range.contains(8)

// lets use a half-open range
range = 5..<8
assert range.size() == 3
assert range.get(2) == 7
assert range[2] == 7
assert range instanceof java.util.List
assert range.contains(5)
assert !range.contains(8)

//get the end points of the range without using indexes
range = 1..10
assert range.from == 1
assert range.to == 10
```

Note that int ranges are implemented efficiently, creating a lightweight Java object containing a from and to value.

Ranges can be used for any Java object which implements java.lang.Comparable for comparison and also have methods `next()` and `previous()` to return the next / previous item in the range. For example, you can create a range of `String` elements:

```
// an inclusive range
def range = 'a'..'d'
assert range.size() == 4
assert range.get(2) == 'c'
assert range[2] == 'c'
assert range instanceof java.util.List
assert range.contains('a')
assert range.contains('d')
assert !range.contains('e')
```

You can iterate on a range using a `for` loop:

```
for (i in 1..10) {
    println "Hello ${i}"
}
```

but alternatively you can achieve the same effect in a more Groovy idiomatic style, by iterating a range with `each` method:

```
(1..10).each { i ->
    println "Hello ${i}"
}
```

Ranges can be also used in the `switch` statement:

```
switch (years) {
    case 1..10: interestRate = 0.076; break;
    case 11..25: interestRate = 0.052; break;
    default: interestRate = 0.037;
}
```

Syntax enhancements for collections

GPath support

Thanks to the support of property notation for both lists and maps, Groovy provides syntactic sugar making it really easy to deal with nested collections, as illustrated in the following examples:

```
def listOfMaps = [['a': 11, 'b': 12], ['a': 21, 'b': 22]]
assert listOfMaps.a == [11, 21] //GPath notation
assert listOfMaps*.a == [11, 21] //spread dot notation

listOfMaps = [['a': 11, 'b': 12], ['a': 21, 'b': 22], null]
assert listOfMaps*.a == [11, 21, null] // caters for null values
assert listOfMaps*.a == listOfMaps.collect { it?.a } //equivalent notation
// But this will only collect non-null values
assert listOfMaps.a == [11,21]
```

Spread operator

The spread operator can be used to "inline" a collection into another. It is syntactic sugar which often avoids calls to `putAll` and facilitates the realization of one-liners:

```
assert [ 'z': 900,
         *: ['a': 100, 'b': 200], 'a': 300] == ['a': 300, 'b': 200, 'z': 900]
//spread map notation in map definition
assert [*: [3: 3, *: [5: 5]], 7: 7] == [3: 3, 5: 5, 7: 7]

def f = { [1: 'u', 2: 'v', 3: 'w'] }
assert [*: f(), 10: 'zz'] == [1: 'u', 10: 'zz', 2: 'v', 3: 'w']
//spread map notation in function arguments
f = { map -> map.c }
assert f(*: ['a': 10, 'b': 20, 'c': 30], 'e': 50) == 30

f = { m, i, j, k -> [m, i, j, k] }
//using spread map notation with mixed unnamed and named arguments
assert f('e': 100, *[4, 5], *: ['a': 10, 'b': 20, 'c': 30], 6) ==
        [["e": 100, "b": 20, "c": 30, "a": 10], 4, 5, 6]
```

The star-dot `*.' operator

The "star-dot" operator is a shortcut operator allowing you to call a method or a property on all

elements of a collection:

```
assert [1, 3, 5] == ['a', 'few', 'words']*.size()

class Person {
    String name
    int age
}
def persons = [new Person(name:'Hugo', age:17), new Person(name:'Sandra',age:19)]
assert [17, 19] == persons*.age
```

Slicing with the subscript operator

You can index into lists, arrays, maps using the subscript expression. It is interesting that strings are considered as special kinds of collections in that context:

```
def text = 'nice cheese gromit!'
def x = text[2]

assert x == 'c'
assert x.class == String

def sub = text[5..10]
assert sub == 'cheese'

def list = [10, 11, 12, 13]
def answer = list[2,3]
assert answer == [12,13]
```

Notice that you can use ranges to extract part of a collection:

```
list = 100..200
sub = list[1, 3, 20..25, 33]
assert sub == [101, 103, 120, 121, 122, 123, 124, 125, 133]
```

The subscript operator can be used to update an existing collection (for collection type which are not immutable):

```
list = ['a','x','x','d']
list[1..2] = ['b','c']
assert list == ['a','b','c','d']
```

It is worth noting that negative indices are allowed, to extract more easily from the end of a collection:

```
text = "nice cheese gromit!"
x = text[-1]
assert x == "!"
```

You can use negative indices to count from the end of the List, array, String etc.

```
def name = text[-7..-2]
assert name == "gromit"
```

Eventually, if you use a backwards range (the starting index is greater than the end index), then the answer is reversed.

```
text = "nice cheese gromit!"
name = text[3..1]
assert name == "eci"
```

Enhanced Collection Methods

In addition to lists, maps or ranges, Groovy offers a lot of additional methods for filtering, collecting, grouping, counting, ... which are directly available on either collections or more easily iterables.

In particular, we invite you to read the Groovy development kit API docs and specifically:

- methods added to `Iterable` can be found here
- methods added to `Iterator` can be found here
- methods added to `Collection` can be found here
- methods added to `List` can be found here
- methods added to `Map` can be found here

Working with arrays

Groovy provides array support based on Java arrays with several extensions found in the Groovy development kit. The overall intention is that whether you are using an array or a collection, the code for working with the aggregate remains the same.

Arrays

Array literals

You can create arrays as follows. Notice that `[]` is also used as the empty array expression when given an explicit array type.

```
Number[] nums = [5, 6, 7, 8]
assert nums[1] == 6
```

```
assert nums.getAt(2) == 7                // alternative syntax
assert nums[-1] == 8                     // negative indices
assert nums instanceof Number[]

int[] primes = [2, 3, 5, 7]              // primitives
assert primes instanceof int[]

def odds = [1, 3, 5] as int[]            // alt syntax 1
assert odds instanceof int[]

def evens = new int[]{2, 4, 6}           // alt syntax 2
assert evens instanceof int[]

// empty array examples
Number[] emptyNums = []
assert emptyNums instanceof Number[] && emptyNums.size() == 0

var emptyObjects = new Object[0]         // alternative syntax 1
assert emptyObjects instanceof Object[] && emptyObjects.size() == 0

def emptyStrings = new String[]{}        // alternative syntax 2
assert emptyStrings instanceof String[] && emptyStrings.size() == 0

// multi-dimension examples
int[][] manyInts = [ [1,2], [3,4] ]
assert manyInts instanceof int[][] && manyInts.size() == 2
assert manyInts[0].size() == 2 && manyInts[1].size() == 2

def manyStrings = new String[][] { {'one','two'}, {'three','four'}, }
assert manyStrings instanceof String[][] && manyStrings.size() == 2
assert manyStrings[0].size() == 2 && manyStrings[1].size() == 2
assert manyStrings[0][0] == 'one'
assert manyStrings[-1][-1] == 'four'
```

Iterating on an array

Iterating on elements of an array is usually done calling the `each` and `eachWithIndex` methods, which execute code on each item of an array:

```
String[] vowels = ['a', 'e', 'i', 'o', 'u']
var result = ''
vowels.each {
    result += it
}
assert result == 'aeiou'
result = ''
vowels.eachWithIndex { v, i ->
    result += v * i          // index starts from 0
}
assert result == 'eiiooouuuu'
```

```
result = ''
int[] nums = [0, 1, 2]
nums.eachWithIndex { value, index ->
    result += value.doubleValue()
}
assert result == '0.01.02.0'
```

Other useful methods

There are numerous other GDK methods for working with arrays. Just be a little careful to read the documentation. For collections, there are some mutating methods which alter the original collection and others which produce new collections, leaving the original untouched. Since arrays are of a fixed size, we wouldn't expect mutating methods which altered an array's size. Often instead, such methods return collections. Here are some interesting array GDK methods:

```
int[] nums = [1, 2, 3]
def doubled = nums.collect { it * 2 }
assert doubled == [2, 4, 6] && doubled instanceof List
def tripled = nums*.multiply(3)
assert tripled == [3, 6, 9] && doubled instanceof List

assert nums.any{ it > 2 }
assert nums.every{ it < 4 }
assert nums.average() == 2
assert nums.min() == 1
assert nums.max() == 3
assert nums.sum() == 6
assert nums.indices == [0, 1, 2]
assert nums.swap(0, 2) == [3, 2, 1] as int[]

Integer[] moreNums = [10, 20, 30, 40]
assert moreNums.find { it > 15 } == 20              // first matching element
assert moreNums.findAll { it > 15 } == [20, 30, 40] // all matching elements (as a
List)
assert moreNums[1..2] == [20, 30]                   // slice an array with a range
(returns a List)
assert moreNums[-1] == 40                            // negative indices count from
the end

int[][] pairs = [[1, 2], [3, 4]]
assert pairs.flatten() == [1, 2, 3, 4]              // flatten nested arrays into a
single List
```

Working with legacy Date/Calendar types

The `groovy-dateutil` module supports numerous extensions for working with Java's classic `Date` and `Calendar` classes.

You can access the properties of a `Date` or `Calendar` using the normal array index notation with the constant field numbers from the `Calendar` class as shown in the following example:

```
import static java.util.Calendar.*    ①

def cal = Calendar.instance
cal[YEAR] = 2000                      ②
cal[MONTH] = JANUARY                  ②
cal[DAY_OF_MONTH] = 1                 ②
assert cal[DAY_OF_WEEK] == SATURDAY   ③
```

① Import the constants

② Setting the calendar's year, month and day of month

③ Accessing the calendar's day of week

Groovy supports arithmetic on and iteration between `Date` and `Calendar` instances as shown in the following example:

```
def utc = TimeZone.getTimeZone('UTC')
Date date = Date.parse("yyyy-MM-dd HH:mm", "2010-05-23 09:01", utc)

def prev = date - 1
def next = date + 1

def diffInDays = next - prev
assert diffInDays == 2

int count = 0
prev.upto(next) { count++ }
assert count == 3
```

You can parse strings into dates and output dates into formatted strings:

```
def orig = '2000-01-01'
def newYear = Date.parse('yyyy-MM-dd', orig)
assert newYear[DAY_OF_WEEK] == SATURDAY
assert newYear.format('yyyy-MM-dd') == orig
assert newYear.format('dd/MM/yyyy') == '01/01/2000'
```

You can also create a new Date or Calendar based on an existing one:

```
def newYear = Date.parse('yyyy-MM-dd', '2000-01-01')
def newYearsEve = newYear.copyWith(
    year: 1999,
    month: DECEMBER,
    dayOfMonth: 31
```

```
)
assert newYearsEve[DAY_OF_WEEK] == FRIDAY
```

A number-based DSL for durations

The `org.apache.groovy.dateutil.TimeCategory` category adds properties to numbers for producing durations that can be added to or subtracted from `Date` instances:

```
use (org.apache.groovy.dateutil.TimeCategory) {
    def christmas = new Date(124, 11, 25)     // 25 Dec 2024
    assert (christmas + 1.week - 3.days).date == 29
    assert (christmas - christmas).toMilliseconds() == 0

    def party = 2.hours + 30.minutes
    assert party.toMilliseconds() == (150 * 60 * 1000)
}
```

This is a dequirked, `java.util.Date`-flavored parallel to the (now deprecated) `groovy.time.TimeCategory`. Relative to the legacy class, `.ago`/`.from.now` preserve the time-of-day uniformly (they no longer floor day/month/year amounts to midnight), and `toMilliseconds()` is deterministic (years and months use fixed `java.time.temporal.ChronoUnit` estimates rather than resolving against the current date). A `java.time` flavored equivalent, `org.apache.groovy.datetime.TimeCategory`, is available in the `groovy-datetime` module.

Working with Date/Time types

The `groovy-datetime` module supports numerous extensions for working with the Date/Time API introduced in Java 8. This documentation refers to the data types defined by this API as "JSR 310 types."

Formatting and parsing

A common use case when working with date/time types is to convert them to Strings (formatting) and from Strings (parsing). Groovy provides these additional formatting methods:

Method	Description	Example
`getDateString()`	For `LocalDate` and `LocalDateTime`, formats with `DateTimeFormatter.ISO_LOCAL_DATE`	`2018-03-10`
	For `OffsetDateTime`, formats with `DateTimeFormatter.ISO_OFFSET_DATE`	`2018-03-10+04:00`

Method	Description	Example
	For `ZonedDateTime`, formats with `DateTimeFormatter.ISO_LOCAL_DATE` and appends the `ZoneId` short name	`2018-03-10EST`
`getDateTimeString()`	For `LocalDateTime`, formats with `DateTimeFormatter.ISO_LOCAL_DATE_TIME`	`2018-03-10T20:30:45`
	For `OffsetDateTime`, formats with `DateTimeFormatter.ISO_OFFSET_DATE_TIME`	`2018-03-10T20:30:45+04:00`
	For `ZonedDateTime`, formats with `DateTimeFormatter.ISO_LOCAL_DATE_TIME` and appends the `ZoneId` short name	`2018-03-10T20:30:45EST`
`getTimeString()`	For `LocalTime` and `LocalDateTime`, formats with `DateTimeFormatter.ISO_LOCAL_TIME`	`20:30:45`
	For `OffsetTime` and `OffsetDateTime`, formats with `DateTimeFormatter.ISO_OFFSET_TIME` formatter	`20:30:45+04:00`
	For `ZonedDateTime`, formats with `DateTimeFormatter.ISO_LOCAL_TIME` and appends the `ZoneId` short name	`20:30:45EST`
`format(FormatStyle style)`	For `LocalTime` and `OffsetTime`, formats with `DateTimeFormatter.ofLocalizedTime(style)`	`4:30 AM` (with style `FormatStyle.SHORT`, e.g.)
	For `LocalDate`, formats with `DateTimeFormatter.ofLocalizedDate(style)`	`Saturday, March 10, 2018` (with style `FormatStyle.FULL`, e.g.)
	For `LocalDateTime`, `OffsetDateTime`, and `ZonedDateTime` formats with `DateTimeFormatter.ofLocalizedDateTime(style)`	`Mar 10, 2019 4:30:45 AM` (with style `FormatStyle.MEDIUM`, e.g.)
`format(String pattern)`	Formats with `DateTimeFormatter.ofPattern(pattern)`	`03/10/2018` (with pattern 'MM/dd/yyyy', e.g.)

For parsing, Groovy adds a static `parse` method to many of the JSR 310 types. The method takes two arguments: the value to be formatted and the pattern to use. The pattern is defined by the `java.time.format.DateTimeFormatter` API. As an example:

```
def date = LocalDate.parse('Jun 3, 04', 'MMM d, yy')
assert date == LocalDate.of(2004, Month.JUNE, 3)

def time = LocalTime.parse('4:45', 'H:mm')
assert time == LocalTime.of(4, 45, 0)

def offsetTime = OffsetTime.parse('09:47:51-1234', 'HH:mm:ssZ')
assert offsetTime == OffsetTime.of(9, 47, 51, 0, ZoneOffset.ofHoursMinutes(-12, -34))

def dateTime = ZonedDateTime.parse('2017/07/11 9:47PM Pacific Standard Time',
'yyyy/MM/dd h:mma zzzz')
assert dateTime == ZonedDateTime.of(
        LocalDate.of(2017, 7, 11),
        LocalTime.of(21, 47, 0),
        ZoneId.of('America/Los_Angeles')
)
```

Note that these `parse` methods have a different argument ordering than the static `parse` method Groovy added to `java.util.Date`. This was done to be consistent with the existing `parse` methods of the Date/Time API.

Manipulating date/time

Addition and subtraction

`Temporal` types have `plus` and `minus` methods for adding or subtracting a provided `java.time.temporal.TemporalAmount` argument. Because Groovy maps the `+` and `-` operators to single-argument methods of these names, a more natural expression syntax can be used to add and subtract.

```
def aprilFools = LocalDate.of(2018, Month.APRIL, 1)

def nextAprilFools = aprilFools + Period.ofDays(365) // add 365 days
assert nextAprilFools.year == 2019

def idesOfMarch = aprilFools - Period.ofDays(17) // subtract 17 days
assert idesOfMarch.dayOfMonth == 15
assert idesOfMarch.month == Month.MARCH
```

Groovy provides additional `plus` and `minus` methods that accept an integer argument, enabling the above to be rewritten more succinctly:

```
def nextAprilFools = aprilFools + 365 // add 365 days
def idesOfMarch = aprilFools - 17 // subtract 17 days
```

The unit of these integers depends on the JSR 310 type operand. As evident above, integers used with `ChronoLocalDate` types like `LocalDate` have a unit of days. Integers used with `Year` and `YearMonth`

have a unit of years and months, respectively. All other types have a unit of seconds, such as LocalTime, for instance:

```
def mars = LocalTime.of(12, 34, 56) // 12:34:56 pm

def thirtySecondsToMars = mars - 30 // go back 30 seconds
assert thirtySecondsToMars.second == 26
```

Multiplication and division

The * operator can be used to multiply Period and Duration instances by an integer value; the / operator can be used to divide Duration instances by an integer value.

```
def period = Period.ofMonths(1) * 2 // a 1-month period times 2
assert period.months == 2

def duration = Duration.ofSeconds(10) / 5// a 10-second duration divided by 5
assert duration.seconds == 2
```

Incrementing and decrementing

The ++ and -- operators can be used increment and decrement date/time values by one unit. Since the JSR 310 types are immutable, the operation will create a new instance with the incremented/decremented value and reassign it to the reference.

```
def year = Year.of(2000)
--year // decrement by one year
assert year.value == 1999

def offsetTime = OffsetTime.of(0, 0, 0, 0, ZoneOffset.UTC) // 00:00:00.000 UTC
offsetTime++ // increment by one second
assert offsetTime.second == 1
```

Negation

The Duration and Period types represent a negative or positive length of time. These can be negated with the unary - operator.

```
def duration = Duration.ofSeconds(-15)
def negated = -duration
assert negated.seconds == 15
```

Interacting with date/time values

Property notation

The `getLong(TemporalField)` method of `TemporalAccessor` types (e.g. `LocalDate`, `LocalTime`, `ZonedDateTime`, etc.) and the `get(TemporalUnit)` method of `TemporalAmount` types (namely `Period` and `Duration`), can be invoked with Groovy's property notation. For example:

```
def date = LocalDate.of(2018, Month.MARCH, 12)
assert date[ChronoField.YEAR] == 2018
assert date[ChronoField.MONTH_OF_YEAR] == Month.MARCH.value
assert date[ChronoField.DAY_OF_MONTH] == 12
assert date[ChronoField.DAY_OF_WEEK] == DayOfWeek.MONDAY.value

def period = Period.ofYears(2).withMonths(4).withDays(6)
assert period[ChronoUnit.YEARS] == 2
assert period[ChronoUnit.MONTHS] == 4
assert period[ChronoUnit.DAYS] == 6
```

Ranges, `upto`, and `downto`

The JSR 310 types can be used with the range operator. The following example iterates between today and the `LocalDate` six days from now, printing out the day of the week for each iteration. As both range bounds are inclusive, this prints all seven days of the week.

```
def start = LocalDate.now()
def end = start + 6 // 6 days later
(start..end).each { date ->
    println date.dayOfWeek
}
```

The `upto` method will accomplish the same as the range in the above example. The `upto` method iterates from the earlier `start` value (inclusive) to the later `end` value (also inclusive), calling the closure with the incremented `next` value once per iteration.

```
def start = LocalDate.now()
def end = start + 6 // 6 days later
start.upto(end) { next ->
    println next.dayOfWeek
}
```

The `downto` method iterates in the opposite direction, from a later `start` value to an earlier `end` value.

The unit of iteration for `upto`, `downto`, and ranges is the same as the unit for addition and subtraction: `LocalDate` iterates by one day at a time, `YearMonth` iterates by one month, `Year` by one year, and everything else by one second. Both methods also support an optional a `TemporalUnit` argument to change the unit of iteration.

Consider the following example, where March 1st, 2018 is iterated up to March 2nd, 2018 using an

iteration unit of months.

```
def start = LocalDate.of(2018, Month.MARCH, 1)
def end = start + 1 // 1 day later

int iterationCount = 0
start.upto(end, ChronoUnit.MONTHS) { next ->
    println next
    ++iterationCount
}

assert iterationCount == 1
```

Since the `start` date is inclusive, the closure is called with a `next` date value of March 1st. The `upto` method then increments the date by one month, yielding the date, April 1st. Because this date is *after* the specified `end` date of March 2nd, the iteration stops immediately, having only called the closure once. This behavior is the same for the `downto` method except that the iteration will stop as soon as the value of `next` becomes earlier than the targeted `end` date.

In short, when iterating with the `upto` or `downto` methods with a custom unit of iteration, the current value of iteration will never exceed the end value.

Combining date/time values

The left-shift operator (`<<`) can be used to combine two JSR 310 types into an aggregate type. For example, a `LocalDate` can be left-shifted into a `LocalTime` to produce a composite `LocalDateTime` instance.

```
MonthDay monthDay = Month.JUNE << 3 // June 3rd
LocalDate date = monthDay << Year.of(2015) // 3-Jun-2015
LocalDateTime dateTime = date << LocalTime.NOON // 3-Jun-2015 @ 12pm
OffsetDateTime offsetDateTime = dateTime << ZoneOffset.ofHours(-5) // 3-Jun-2015 @
12pm UTC-5
```

The left-shift operator is reflexive; the order of the operands does not matter.

```
def year = Year.of(2000)
def month = Month.DECEMBER

YearMonth a = year << month
YearMonth b = month << year
assert a == b
```

Creating periods and durations

The right-shift operator (`>>`) produces a value representing the period or duration between the operands. For `ChronoLocalDate`, `YearMonth`, and `Year`, the operator yields a `Period` instance:

```
def newYears = LocalDate.of(2018, Month.JANUARY, 1)
def aprilFools = LocalDate.of(2018, Month.APRIL, 1)

def period = newYears >> aprilFools
assert period instanceof Period
assert period.months == 3
```

The operator produces a `Duration` for the time-aware JSR types:

```
def duration = LocalTime.NOON >> (LocalTime.NOON + 30)
assert duration instanceof Duration
assert duration.seconds == 30
```

If the value on the left-hand side of the operator is earlier than the value on the right-hand side, the result is positive. If the left-hand side is later than the right-hand side, the result is negative:

```
def decade = Year.of(2010) >> Year.of(2000)
assert decade.years == -10
```

A number-based DSL for periods and durations

The `org.apache.groovy.datetime.TimeCategory` category adds properties to numbers for producing `Duration` (time-based) and `Period` (date-based) values, giving a concise DSL for date/time manipulation:

```
import java.time.LocalDate
import java.time.LocalDateTime
use (org.apache.groovy.datetime.TimeCategory) {
    assert 1.hour == java.time.Duration.ofHours(1)
    assert 2.months == java.time.Period.ofMonths(2)

    // apply amounts to temporals via their native arithmetic
    assert LocalDate.of(2000, 1, 1) + 2.months == LocalDate.of(2000, 3, 1)

    // date-based and time-based amounts compose by chaining
    def base = LocalDateTime.of(2000, 1, 1, 0, 0, 0)
    assert base + 1.month + 3.hours == LocalDateTime.of(2000, 2, 1, 3, 0, 0)

    // relative-time terminals: a Period yields a LocalDate, a Duration a
LocalDateTime
    assert 2.days.ago instanceof LocalDate
    assert 1.hour.from.now instanceof LocalDateTime
}
```

Time-based units (`nanoseconds`, `milliseconds`, `seconds`, `minutes`, `hours`) produce a `Duration`; date-based units (`days`, `weeks`, `months`, `years`) produce a `Period`. This is the modern counterpart to the (now

deprecated) `groovy.time.TimeCategory`. A `java.util.Date` flavored equivalent, `org.apache.groovy.dateutil.TimeCategory`, is available in the `groovy-dateutil` module.

Converting between legacy and JSR 310 types

Despite the shortcomings of `Date`, `Calendar`, and `TimeZone` types in the `java.util` package they are fairly common in Java APIs (at least in those prior to Java 8). To accommodate use of such APIs, Groovy provides methods for converting between the JSR 310 types and legacy types.

Most JSR types have been fitted with `toDate()` and `toCalendar()` methods for converting to relatively equivalent `java.util.Date` and `java.util.Calendar` values. Both `ZoneId` and `ZoneOffset` have been given a `toTimeZone()` method for converting to `java.util.TimeZone`.

```
// LocalDate to java.util.Date
def valentines = LocalDate.of(2018, Month.FEBRUARY, 14)
assert valentines.toDate().format('MMMM dd, yyyy') == 'February 14, 2018'

// LocalTime to java.util.Date
def noon = LocalTime.of(12, 0, 0)
assert noon.toDate().format('HH:mm:ss') == '12:00:00'

// ZoneId to java.util.TimeZone
def newYork = ZoneId.of('America/New_York')
assert newYork.toTimeZone() == TimeZone.getTimeZone('America/New_York')

// ZonedDateTime to java.util.Calendar
def valAtNoonInNY = ZonedDateTime.of(valentines, noon, newYork)
assert valAtNoonInNY.toCalendar().getTimeZone().toZoneId() == newYork
```

Note that when converting to a legacy type:

- Nanosecond values are truncated to milliseconds. A `LocalTime`, for example, with a `ChronoUnit.NANOS` value of 999,999,999 nanoseconds translates to 999 milliseconds.
- When converting the "local" types (`LocalDate`, `LocalTime`, and `LocalDateTime`), the time zone of the returned `Date` or `Calendar` will be the system default.
- When converting a time-only type (`LocalTime` or `OffsetTime`), the year/month/day of the `Date` or `Calendar` is set to the current date.
- When converting a date-only type (`LocalDate`), the time value of the `Date` or `Calendar` will be cleared, i.e. `00:00:00.000`.
- When converting an `OffsetDateTime` to a `Calendar`, only the hours and minutes of the `ZoneOffset` convey into the corresponding `TimeZone`. Fortunately, Zone Offsets with non-zero seconds are rare.

Groovy has added a number of methods to `Date` and `Calendar` for converting into the various JSR 310 types:

```
Date legacy = Date.parse('yyyy-MM-dd HH:mm:ss.SSS', '2010-04-03 10:30:58.999')
```

```
assert legacy.toLocalDate() == LocalDate.of(2010, 4, 3)
assert legacy.toLocalTime() == LocalTime.of(10, 30, 58, 999_000_000) // 999M ns =
999ms
assert legacy.toOffsetTime().hour == 10
assert legacy.toYear() == Year.of(2010)
assert legacy.toMonth() == Month.APRIL
assert legacy.toDayOfWeek() == DayOfWeek.SATURDAY
assert legacy.toMonthDay() == MonthDay.of(Month.APRIL, 3)
assert legacy.toYearMonth() == YearMonth.of(2010, Month.APRIL)
assert legacy.toLocalDateTime().year == 2010
assert legacy.toOffsetDateTime().dayOfMonth == 3
assert legacy.toZonedDateTime().zone == ZoneId.systemDefault()
```

Handy utilities

ConfigSlurper

`ConfigSlurper` is a utility class for reading configuration files defined in the form of Groovy scripts. Like it is the case with Java `*.properties` files, `ConfigSlurper` allows a dot notation. But in addition, it allows for Closure scoped configuration values and arbitrary object types.

```
def config = new ConfigSlurper().parse('''
    app.date = new Date()  ①
    app.age  = 42
    app {                  ②
        name = "Test${42}"
    }
''')

assert config.app.date instanceof Date
assert config.app.age == 42
assert config.app.name == 'Test42'
```

① Usage of the dot notation

② Usage of Closure scopes as an alternative to the dot notation

As can be seen in the above example, the `parse` method can be used to retrieve `groovy.util.ConfigObject` instances. The `ConfigObject` is a specialized `java.util.Map` implementation that either returns the configured value or a new `ConfigObject` instance but never `null`.

```
def config = new ConfigSlurper().parse('''
    app.date = new Date()
    app.age  = 42
    app.name = "Test${42}"
''')
```

```
assert config.test != null   ①
```

① `config.test` has not been specified yet it returns a `ConfigObject` when being called.

In the case of a dot being part of a configuration variable name, it can be escaped by using single or double quotes.

```
def config = new ConfigSlurper().parse('''
    app."person.age"  = 42
''')

assert config.app."person.age" == 42
```

In addition, `ConfigSlurper` comes with support for `environments`. The `environments` method can be used to hand over a Closure instance that itself may consist of a several sections. Let's say we wanted to create a particular configuration value for the development environment. When creating the `ConfigSlurper` instance we can use the `ConfigSlurper(String)` constructor to specify the target environment.

```
def config = new ConfigSlurper('development').parse('''
  environments {
        development {
            app.port = 8080
        }

        test {
            app.port = 8082
        }

        production {
            app.port = 80
        }
  }
''')

assert config.app.port == 8080
```

NOTE The `ConfigSlurper` environments aren't restricted to any particular environment names. It solely depends on the `ConfigSlurper` client code what value are supported and interpreted accordingly.

The `environments` method is built-in but the `registerConditionalBlock` method can be used to register other method names in addition to the `environments` name.

```
def slurper = new ConfigSlurper()
slurper.registerConditionalBlock('myProject', 'developers')   ①
```

```
def config = slurper.parse('''
  sendMail = true

  myProject {
       developers {
           sendMail = false
       }
  }
''')

assert !config.sendMail
```

① Once the new block is registered `ConfigSlurper` can parse it.

For Java integration purposes the `toProperties` method can be used to convert the `ConfigObject` to a `java.util.Properties` object that might be stored to a `*.properties` text file. Be aware though that the configuration values are converted to `String` instances during adding them to the newly created `Properties` instance.

```
def config = new ConfigSlurper().parse('''
    app.date = new Date()
    app.age  = 42
    app {
        name = "Test${42}"
    }
''')

def properties = config.toProperties()

assert properties."app.date" instanceof String
assert properties."app.age" == '42'
assert properties."app.name" == 'Test42'
```

Expando

The `Expando` class can be used to create a dynamically expandable object. Despite its name it does not use the `ExpandoMetaClass` underneath. Each `Expando` object represents a standalone, dynamically-crafted instance that can be extended with properties (or methods) at runtime.

```
def expando = new Expando()
expando.name = 'John'

assert expando.name == 'John'
```

A special case occurs when a dynamic property registers a `Closure` code block. Once being registered it can be invoked as it would be done with a method call.

```
def expando = new Expando()
expando.toString = { -> 'John' }
expando.say = { String s -> "John says: ${s}" }

assert expando as String == 'John'
assert expando.say('Hi') == 'John says: Hi'
```

Observable list, map and set

Groovy comes with observable lists, maps and sets. Each of these collections trigger `java.beans.PropertyChangeEvent` events when elements are added, removed or changed. Note that a `PropertyChangeEvent` is not only signalling that a certain event has occurred, moreover, it holds information on the property name and the old/new value a certain property has been changed to.

Depending on the type of change that has happened, observable collections might fire more specialized `PropertyChangeEvent` types. For example, adding an element to an observable list fires an `ObservableList.ElementAddedEvent` event.

```
def event                                           ①
def listener = {
    if (it instanceof ObservableList.ElementEvent)  {  ②
        event = it
    }
} as PropertyChangeListener


def observable = [1, 2, 3] as ObservableList    ③
observable.addPropertyChangeListener(listener)  ④

observable.add 42                               ⑤

assert event instanceof ObservableList.ElementAddedEvent

def elementAddedEvent = event as ObservableList.ElementAddedEvent
assert elementAddedEvent.changeType == ObservableList.ChangeType.ADDED
assert elementAddedEvent.index == 3
assert elementAddedEvent.oldValue == null
assert elementAddedEvent.newValue == 42
```

① Declares a `PropertyChangeEventListener` that is capturing the fired events

② `ObservableList.ElementEvent` and its descendant types are relevant for this listener

③ Creates an `ObservableList` from the given list

④ Registers the listener

⑤ Triggers an `ObservableList.ElementAddedEvent` event

NOTE Be aware that adding an element in fact causes two events to be triggered. The first

> is of type `ObservableList.ElementAddedEvent`, the second is a plain `PropertyChangeEvent` that informs listeners about the change of property `size`.

The `ObservableList.ElementClearedEvent` event type is another interesting one. Whenever multiple elements are removed, for example when calling `clear()`, it holds the elements being removed from the list.

```
def event
def listener = {
    if (it instanceof ObservableList.ElementEvent)  {
        event = it
    }
} as PropertyChangeListener


def observable = [1, 2, 3] as ObservableList
observable.addPropertyChangeListener(listener)

observable.clear()

assert event instanceof ObservableList.ElementClearedEvent

def elementClearedEvent = event as ObservableList.ElementClearedEvent
assert elementClearedEvent.values == [1, 2, 3]
assert observable.size() == 0
```

To get an overview of all the supported event types the reader is encouraged to have a look at the JavaDoc documentation or the source code of the observable collection in use.

`ObservableMap` and `ObservableSet` come with the same concepts as we have seen for `ObservableList` in this section.

Metaprogramming

The Groovy language supports two flavors of metaprogramming: runtime and compile-time. The first allows altering the class model and the behavior of a program at runtime while the second only occurs at compile-time. Both have pros and cons that we will detail in this section.

Runtime metaprogramming

With runtime metaprogramming we can postpone to runtime the decision to intercept, inject and even synthesize methods of classes and interfaces. For a deep understanding of Groovy's metaobject protocol (MOP) we need to understand Groovy objects and Groovy's method handling. In Groovy we work with three kinds of objects: POJO, POGO and Groovy Interceptors. Groovy allows metaprogramming for all types of objects but in a different manner.

- POJO - A regular Java object whose class can be written in Java or any other language for the JVM.

- POGO - A Groovy object whose class is written in Groovy. It extends `java.lang.Object` and implements the groovy.lang.GroovyObject interface by default.
- Groovy Interceptor - A Groovy object that implements the groovy.lang.GroovyInterceptable interface and has method-interception capability which is discussed in the GroovyInterceptable section.

For every method call Groovy checks whether the object is a POJO or a POGO. For POJOs, Groovy fetches its `MetaClass` from the groovy.lang.MetaClassRegistry and delegates method invocation to it. For POGOs, Groovy takes more steps, as illustrated in the following figure:

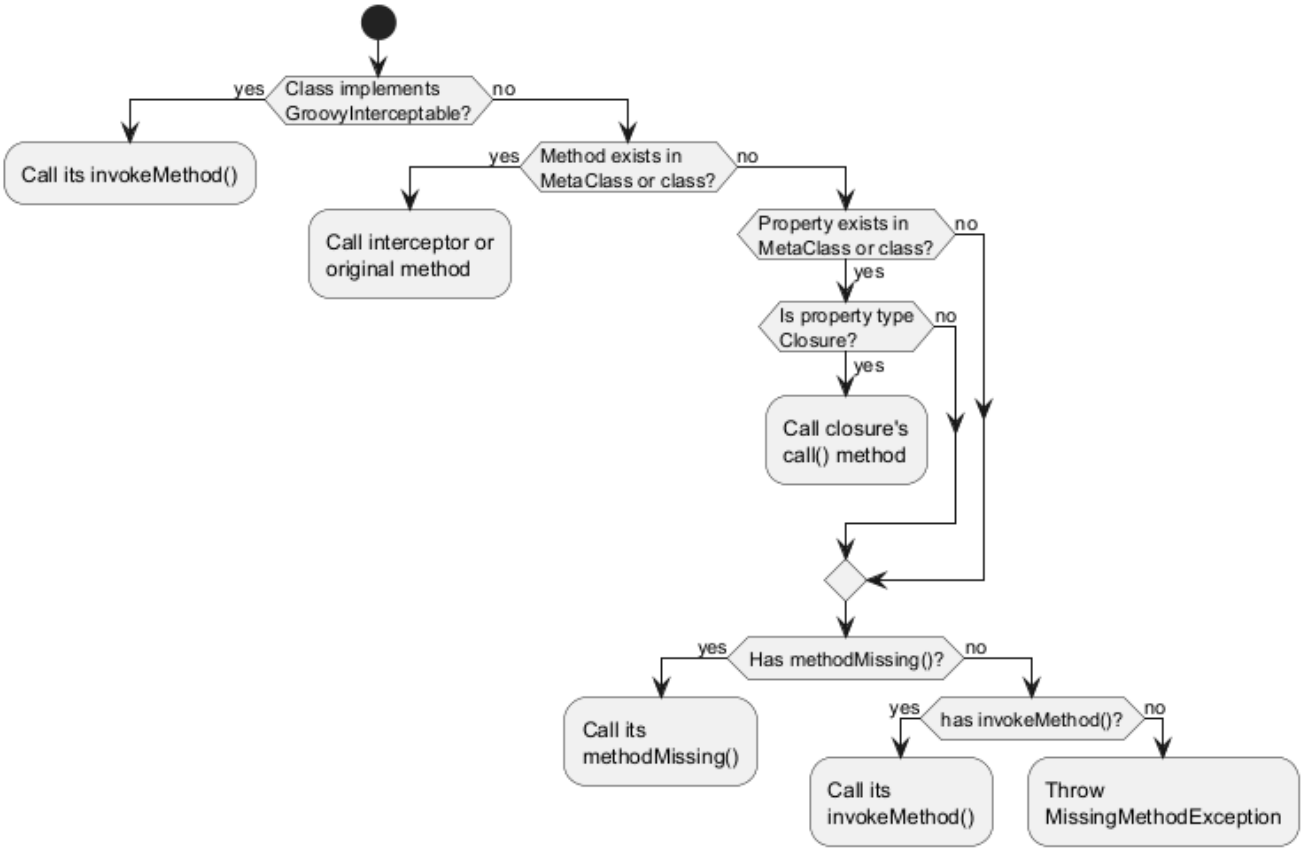


Figure 1. Groovy interception mechanism

GroovyObject interface

groovy.lang.GroovyObject is the main interface in Groovy as the `Object` class is in Java. `GroovyObject` has a default implementation in the groovy.lang.GroovyObjectSupport class and it is responsible to transfer invocation to the groovy.lang.MetaClass object. The `GroovyObject` source looks like this:

```
package groovy.lang;

public interface GroovyObject {

    Object invokeMethod(String name, Object args);

    Object getProperty(String propertyName);

    void setProperty(String propertyName, Object newValue);
```

```
    MetaClass getMetaClass();

    void setMetaClass(MetaClass metaClass);
}
```

invokeMethod

This method is primarily intended to be used in conjunction with the GroovyInterceptable interface or an object's `MetaClass` where it will intercept all method calls.

It is also invoked when the method called is not present on a Groovy object. Here is a simple example using an overridden `invokeMethod()` method:

```
class SomeGroovyClass {

    def invokeMethod(String name, Object args) {
        return "called invokeMethod $name $args"
    }

    def test() {
        return 'method exists'
    }
}

def someGroovyClass = new SomeGroovyClass()

assert someGroovyClass.test() == 'method exists'
assert someGroovyClass.someMethod() == 'called invokeMethod someMethod []'
```

However, the use of `invokeMethod` to intercept missing methods is discouraged. In cases where the intent is to only intercept method calls in the case of a failed method dispatch use methodMissing instead.

get/setProperty

Every read access to a property can be intercepted by overriding the `getProperty()` method of the current object. Here is a simple example:

```
class SomeGroovyClass {

    def property1 = 'ha'
    def field2 = 'ho'
    def field4 = 'hu'

    def getField1() {
        return 'getHa'
    }
```

```
    def getProperty(String name) {
        if (name != 'field3')
            return metaClass.getProperty(this, name) ①
        else
            return 'field3'
    }
}

def someGroovyClass = new SomeGroovyClass()

assert someGroovyClass.field1 == 'getHa'
assert someGroovyClass.field2 == 'ho'
assert someGroovyClass.field3 == 'field3'
assert someGroovyClass.field4 == 'hu'
```

① Forwards the request to the getter for all properties except `field3`.

You can intercept write access to properties by overriding the `setProperty()` method:

```
class POGO {

    String property

    void setProperty(String name, Object value) {
        this.@"$name" = 'overridden'
    }
}

def pogo = new POGO()
pogo.property = 'a'

assert pogo.property == 'overridden'
```

get/setMetaClass

You can access an object's `metaClass` or set your own `MetaClass` implementation for changing the default interception mechanism. For example, you can write your own implementation of the `MetaClass` interface and assign it to objects in order to change the interception mechanism:

```
// getMetaclass
someObject.metaClass

// setMetaClass
someObject.metaClass = new OwnMetaClassImplementation()
```

NOTE You can find an additional example in the GroovyInterceptable topic.

get/setAttribute

This functionality is related to the `MetaClass` implementation. In the default implementation you can access fields without invoking their getters and setters. The examples below demonstrates this approach:

```
class SomeGroovyClass {

    def field1 = 'ha'
    def field2 = 'ho'

    def getField1() {
        return 'getHa'
    }
}

def someGroovyClass = new SomeGroovyClass()

assert someGroovyClass.metaClass.getAttribute(someGroovyClass, 'field1') == 'ha'
assert someGroovyClass.metaClass.getAttribute(someGroovyClass, 'field2') == 'ho'
```

```
class POGO {

    private String field
    String property1

    void setProperty1(String property1) {
        this.property1 = "setProperty1"
    }
}

def pogo = new POGO()
pogo.metaClass.setAttribute(pogo, 'field', 'ha')
pogo.metaClass.setAttribute(pogo, 'property1', 'ho')

assert pogo.field == 'ha'
assert pogo.property1 == 'ho'
```

methodMissing

Groovy supports the concept of `methodMissing`. This method differs from `invokeMethod` in that it is only invoked in the case of a failed method dispatch when no method can be found for the given name and/or the given arguments:

```
class Foo {

   def methodMissing(String name, def args) {
        return "this is me"
```

```
    }
}

assert new Foo().someUnknownMethod(42l) == 'this is me'
```

Typically when using `methodMissing` it is possible to cache the result for the next time the same method is called.

For example, consider dynamic finders in GORM. These are implemented in terms of `methodMissing`. The code resembles something like this:

```
class GORM {

   def dynamicMethods = [...] // an array of dynamic methods that use regex

   def methodMissing(String name, args) {
       def method = dynamicMethods.find { it.match(name) }
       if(method) {
          GORM.metaClass."$name" = { Object[] varArgs ->
             method.invoke(delegate, name, varArgs)
          }
          return method.invoke(delegate,name, args)
       }
       else throw new MissingMethodException(name, delegate, args)
   }
}
```

Notice how, if we find a method to invoke, we then dynamically register a new method on the fly using ExpandoMetaClass. This is so that the next time the same method is called it is more efficient. This way of using `methodMissing` does not have the overhead of `invokeMethod` *and* is not expensive from the second call on.

propertyMissing

Groovy supports the concept of `propertyMissing` for intercepting otherwise failing property resolution attempts. In the case of a getter method, `propertyMissing` takes a single `String` argument containing the property name:

```
class Foo {
   def propertyMissing(String name) { name }
}

assert new Foo().boo == 'boo'
```

The `propertyMissing(String)` method is only called when no getter method for the given property can be found by the Groovy runtime.

For setter methods a second `propertyMissing` definition can be added that takes an additional value

argument:

```
class Foo {
   def storage = [:]
   def propertyMissing(String name, value) { storage[name] = value }
   def propertyMissing(String name) { storage[name] }
}

def f = new Foo()
f.foo = "bar"

assert f.foo == "bar"
```

As with `methodMissing` it is best practice to dynamically register new properties at runtime to improve the overall lookup performance.

static methodMissing

Static variant of `methodMissing` method can be added via the ExpandoMetaClass or can be implemented at the class level with `$static_methodMissing` method.

```
class Foo {
    static def $static_methodMissing(String name, Object args) {
        return "Missing static method name is $name"
    }
}

assert Foo.bar() == 'Missing static method name is bar'
```

static propertyMissing

Static variant of `propertyMissing` method can be added via the ExpandoMetaClass or can be implemented at the class level with `$static_propertyMissing` method.

```
class Foo {
    static def $static_propertyMissing(String name) {
        return "Missing static property name is $name"
    }
}

assert Foo.foobar == 'Missing static property name is foobar'
```

GroovyInterceptable

The groovy.lang.GroovyInterceptable interface is marker interface that extends `GroovyObject` and is used to notify the Groovy runtime that all methods should be intercepted through the method dispatcher mechanism of the Groovy runtime.

```
package groovy.lang;

public interface GroovyInterceptable extends GroovyObject {
}
```

When a Groovy object implements the `GroovyInterceptable` interface, its `invokeMethod()` is called for any method calls. Below you can see a simple example of an object of this type:

```
class Interception implements GroovyInterceptable {

    def definedMethod() { }

    def invokeMethod(String name, Object args) {
        'invokedMethod'
    }
}
```

The next piece of code is a test which shows that both calls to existing and non-existing methods will return the same value.

```
final class InterceptableTest {

    @Test
    void testCheckInterception() {
        def interception = new Interception()

        assert interception.definedMethod() == 'invokedMethod'
        assert interception.someMethod() == 'invokedMethod'
    }
}
```

NOTE We cannot use default groovy methods like `println` because these methods are injected into all Groovy objects so they will be intercepted too.

If we want to intercept all method calls but do not want to implement the `GroovyInterceptable` interface we can implement `invokeMethod()` on an object's `MetaClass`. This approach works for both POGOs and POJOs, as shown by this example:

```
final class InterceptionThroughMetaClassTest {

    @Test
    void testPOJOMetaClassInterception() {
        String invoking = 'ha'
        invoking.metaClass.invokeMethod = { String name, Object args ->
            'invoked'
        }
```

```
        assert invoking.length() == 'invoked'
        assert invoking.someMethod() == 'invoked'
    }

    @Test
    void testPOGOMetaClassInterception() {
        Entity entity = new Entity('Hello')
        entity.metaClass.invokeMethod = { String name, Object args ->
            'invoked'
        }

        assert entity.build(new Object()) == 'invoked'
        assert entity.someMethod() == 'invoked'
    }
}
```

NOTE Additional information about `MetaClass` can be found in the MetaClasses section.

Categories

There are situations where it is useful if a class *not* under control had additional methods. In order to enable this capability, Groovy implements a feature borrowed from Objective-C, called *Categories*.

Categories are implemented with so-called *category classes*. A category class is special in that it needs to meet certain pre-defined rules for defining extension methods.

There are a few categories that are included in the system for adding functionality to classes that make them more usable within the Groovy environment:

- groovy.time.TimeCategory
- groovy.servlet.ServletCategory
- groovy.xml.dom.DOMCategory

Category classes aren't enabled by default. To use the methods defined in a category class it is necessary to apply the scoped `use` method that is provided by the GDK and available from inside every Groovy object instance:

```
use(TimeCategory) {
    println 1.minute.from.now       ①
    println 10.hours.ago

    def someDate = new Date()       ②
    println someDate - 3.months
}
```

① `TimeCategory` adds methods to `Integer`

② `TimeCategory` adds methods to `Date`

The `use` method takes the category class as its first parameter and a closure code block as second parameter. Inside the `Closure` access to the category methods is available. As can be seen in the example above even JDK classes like `java.lang.Integer` or `java.util.Date` can be enriched with user-defined methods.

A category needs not to be directly exposed to the user code, the following will also do:

```
class JPACategory{
  // Let's enhance JPA EntityManager without getting into the JSR committee
  static void persistAll(EntityManager em , Object[] entities) { //add an interface to
save all
    entities?.each { em.persist(it) }
  }
}

def transactionContext = {
  EntityManager em, Closure c ->
  def tx = em.transaction
  try {
    tx.begin()
    use(JPACategory) {
      c()
    }
    tx.commit()
  } catch (e) {
    tx.rollback()
  } finally {
    //cleanup your resource here
  }
}

// user code, they always forget to close resource in exception, some even forget to
commit, let's not rely on them.
EntityManager em; //probably injected
transactionContext (em) {
 em.persistAll(obj1, obj2, obj3)
 // let's do some logics here to make the example sensible
 em.persistAll(obj2, obj4, obj6)
}
```

When we have a look at the `groovy.time.TimeCategory` class we see that the extension methods are all declared as `static` methods. In fact, this is one of the requirements that must be met by category classes for its methods to be successfully added to a class inside the `use` code block:

```
public class TimeCategory {

    public static Date plus(final Date date, final BaseDuration duration) {
```

```
        return duration.plus(date);
    }

    public static Date minus(final Date date, final BaseDuration duration) {
        final Calendar cal = Calendar.getInstance();

        cal.setTime(date);
        cal.add(Calendar.YEAR, -duration.getYears());
        cal.add(Calendar.MONTH, -duration.getMonths());
        cal.add(Calendar.DAY_OF_YEAR, -duration.getDays());
        cal.add(Calendar.HOUR_OF_DAY, -duration.getHours());
        cal.add(Calendar.MINUTE, -duration.getMinutes());
        cal.add(Calendar.SECOND, -duration.getSeconds());
        cal.add(Calendar.MILLISECOND, -duration.getMillis());

        return cal.getTime();
    }

    // ...
```

Another requirement is the first argument of the static method must define the type the method is attached to once being activated. The other arguments are the normal arguments the method will take as parameters.

Because of the parameter and static method convention, category method definitions may be a bit less intuitive than normal method definitions. As an alternative Groovy comes with a `@Category` annotation that transforms annotated classes into category classes at compile-time.

```
class Distance {
    def number
    String toString() { "${number}m" }
}

@Category(Number)
class NumberCategory {
    Distance getMeters() {
        new Distance(number: this)
    }
}

use(NumberCategory) {
    assert 42.meters.toString() == '42m'
}
```

Applying the `@Category` annotation has the advantage of being able to use instance methods without the target type as a first parameter. The target type class is given as an argument to the annotation instead.

NOTE There is a distinct section on `@Category` in the compile-time metaprogramming

section.

NOTE Because category methods are resolved dynamically at runtime, categories cannot be used in code under `@TypeChecked` or `@CompileStatic`; the type checker reports an error for `use` method calls. If you need statically checked code, consider an extension module instead, which provides methods in the same form as a category class but is compatible with type checking.

Metaclasses

As explained earlier, Metaclasses play a central role in method resolution. For every method invocation from groovy code, Groovy will find the `MetaClass` for the given object and delegate the method resolution to the metaclass via groovy.lang.MetaClass#invokeMethod(java.lang.Class,java.lang.Object,java.lang.String,java.lang.Object,boolean,boolean) which should not be confused with groovy.lang.GroovyObject#invokeMethod(java.lang.String,java.lang.Object) which happens to be a method that the metaclass may eventually call.

The default metaclass `MetaClassImpl`

By default, objects get an instance of `MetaClassImpl` that implements the default method lookup. This method lookup includes looking up of the method in the object class ("regular" method) but also if no method is found this way it will resort to calling `methodMissing` and ultimately groovy.lang.GroovyObject#invokeMethod(java.lang.String,java.lang.Object)

```
class Foo {}

def f = new Foo()

assert f.metaClass =~ /MetaClassImpl/
```

Custom metaclasses

You can change the metaclass of any object or class and replace it with a custom implementation of the `MetaClass` groovy.lang.MetaClass. Usually you will want to extend one of the existing metaclasses such as `MetaClassImpl`, `DelegatingMetaClass`, `ExpandoMetaClass`, or `ProxyMetaClass`; otherwise you will need to implement the complete method lookup logic. Before using a new metaclass instance you should call groovy.lang.MetaClass#initialize(), otherwise the metaclass may or may not behave as expected.

Delegating metaclass

If you only need to decorate an existing metaclass the `DelegatingMetaClass` simplifies that use case. The old metaclass implementation is still accessible via `super` making it easy to apply pretransformations to the inputs, routing to other methods and postprocessing the outputs.

```
class Foo { def bar() { "bar" } }
```

```
class MyFooMetaClass extends DelegatingMetaClass {
  MyFooMetaClass(MetaClass metaClass) { super(metaClass) }
  MyFooMetaClass(Class theClass) { super(theClass) }

  Object invokeMethod(Object object, String methodName, Object[] args) {
     def result = super.invokeMethod(object,methodName.toLowerCase(), args)
     result.toUpperCase();
  }
}


def mc =  new MyFooMetaClass(Foo.metaClass)
mc.initialize()

Foo.metaClass = mc
def f = new Foo()

assert f.BAR() == "BAR" // the new metaclass routes .BAR() to .bar() and uppercases
the result
```

Magic package

It is possible to change the metaclass at startup time by giving the metaclass a specially crafted (magic) class name and package name. In order to change the metaclass for `java.lang.Integer` it's enough to put a class `groovy.runtime.metaclass.java.lang.IntegerMetaClass` in the classpath. This is useful, for example, when working with frameworks if you want to do metaclass changes before your code is executed by the framework. The general form of the magic package is `groovy.runtime.metaclass.[package].[class]MetaClass`. In the example below the `[package]` is `java.lang` and the `[class]` is `Integer`:

```
// file: IntegerMetaClass.groovy
package groovy.runtime.metaclass.java.lang;

class IntegerMetaClass extends DelegatingMetaClass {
  IntegerMetaClass(MetaClass metaClass) { super(metaClass) }
  IntegerMetaClass(Class theClass) { super(theClass) }
  Object invokeMethod(Object object, String name, Object[] args) {
    if (name =~ /isBiggerThan/) {
      def other = name.split(/isBiggerThan/)[1].toInteger()
      object > other
    } else {
      return super.invokeMethod(object,name, args);
    }
  }
}
```

By compiling the above file with `groovyc IntegerMetaClass.groovy` a `./groovy/runtime/metaclass/java/lang/IntegerMetaClass.class` will be generated. The example below will use this new metaclass:

```
// File testInteger.groovy
def i = 10

assert i.isBiggerThan5()
assert !i.isBiggerThan15()

println i.isBiggerThan5()
```

By running that file with `groovy -cp . testInteger.groovy` the `IntegerMetaClass` will be in the classpath and therefore it will become the metaclass for `java.lang.Integer` intercepting the method calls to `isBiggerThan*()` methods.

Per instance metaclass

You can change the metaclass of individual objects separately, so it's possible to have multiple object of the same class with different metaclasses.

```
class Foo { def bar() { "bar" }}

class FooMetaClass extends DelegatingMetaClass {
  FooMetaClass(MetaClass metaClass) { super(metaClass) }
  Object invokeMethod(Object object, String name, Object[] args) {
      super.invokeMethod(object,name,args).toUpperCase()
  }
}

def f1 = new Foo()
def f2 = new Foo()
f2.metaClass = new FooMetaClass(f2.metaClass)

assert f1.bar() == "bar"
assert f2.bar() == "BAR"
assert f1.metaClass =~ /MetaClassImpl/
assert f2.metaClass =~ /FooMetaClass/
assert f1.class.toString() == "class Foo"
assert f2.class.toString() == "class Foo"
```

ExpandoMetaClass

Groovy comes with a special `MetaClass` the so-called `ExpandoMetaClass`. It is special in that it allows for dynamically adding or changing methods, constructors, properties and even static methods by using a neat closure syntax.

Applying those modifications can be especially useful in mocking or stubbing scenarios as shown in the Testing Guide.

Every `java.lang.Class` is supplied by Groovy with a special `metaClass` property that will give you a reference to an `ExpandoMetaClass` instance. This instance can then be used to add methods or change

the behaviour of already existing ones.

NOTE	By default `ExpandoMetaClass` doesn't do inheritance. To enable this you must call `ExpandoMetaClass#enableGlobally()` before your app starts such as in the main method or servlet bootstrap.

The following sections go into detail on how `ExpandoMetaClass` can be used in various scenarios.

Methods

Once the `ExpandoMetaClass` is accessed by calling the `metaClass` property, methods can be added by using either the left shift `<<` or the `=` operator.

NOTE	Note that the left shift operator is used to *append* a new method. If a public method with the same name and parameter types is declared by the class or interface, including those inherited from superclasses and superinterfaces but excluding those added to the `metaClass` at runtime, an exception will be thrown. If you want to *replace* a method declared by the class or interface you can use the `=` operator.

The operators are applied on a non-existent property of `metaClass` passing an instance of a `Closure` code block.

```
class Book {
   String title
}

Book.metaClass.titleInUpperCase << {-> title.toUpperCase() }

def b = new Book(title:"The Stand")

assert "THE STAND" == b.titleInUpperCase()
```

The example above shows how a new method can be added to a class by accessing the `metaClass` property and using the `<<` or `=` operator to assign a `Closure` code block. The `Closure` parameters are interpreted as method parameters. Parameterless methods can be added by using the `{→ ⋯}` syntax.

Properties

`ExpandoMetaClass` supports two mechanisms for adding or overriding properties.

Firstly, it has support for declaring a *mutable property* by simply assigning a value to a property of `metaClass`:

```
class Book {
   String title
}
```

```
Book.metaClass.author = "Stephen King"
def b = new Book()

assert "Stephen King" == b.author
```

Another way is to add getter and/or setter methods by using the standard mechanisms for adding instance methods.

```
class Book {
  String title
}
Book.metaClass.getAuthor << {-> "Stephen King" }

def b = new Book()

assert "Stephen King" == b.author
```

In the source code example above the property is dictated by the closure and is a read-only property. It is feasible to add an equivalent setter method but then the property value needs to be stored for later usage. This could be done as shown in the following example.

```
class Book {
  String title
}

def properties = Collections.synchronizedMap([:])

Book.metaClass.setAuthor = { String value ->
   properties[System.identityHashCode(delegate) + "author"] = value
}
Book.metaClass.getAuthor = {->
   properties[System.identityHashCode(delegate) + "author"]
}
```

This is not the only technique however. For example in a servlet container one way might be to store the values in the currently executing request as request attributes (as is done in some cases in Grails).

Constructors

Constructors can be added by using a special `constructor` property. Either the `<<` or `=` operator can be used to assign a `Closure` code block. The `Closure` arguments will become the constructor arguments when the code is executed at runtime.

```
class Book {
    String title
}
```

```
Book.metaClass.constructor << { String title -> new Book(title:title) }

def book = new Book('Groovy in Action - 2nd Edition')
assert book.title == 'Groovy in Action - 2nd Edition'
```

NOTE	Be careful when adding constructors however, as it is very easy to get into stack overflow troubles.

Static Methods

Static methods can be added using the same technique as instance methods with the addition of the `static` qualifier before the method name.

```
class Book {
   String title
}

Book.metaClass.static.create << { String title -> new Book(title:title) }

def b = Book.create("The Stand")
```

Borrowing Methods

With `ExpandoMetaClass` it is possible to use Groovy's method pointer syntax to borrow methods from other classes.

```
class Person {
    String name
}
class MortgageLender {
   def borrowMoney() {
      "buy house"
   }
}

def lender = new MortgageLender()

Person.metaClass.buyHouse = lender.&borrowMoney

def p = new Person()

assert "buy house" == p.buyHouse()
```

Dynamic Method Names

Since Groovy allows you to use Strings as property names this in turns allows you to dynamically create method and property names at runtime. To create a method with a dynamic name simply

use the language feature of reference property names as strings.

```
class Person {
   String name = "Fred"
}

def methodName = "Bob"

Person.metaClass."changeNameTo${methodName}" = {-> delegate.name = "Bob" }

def p = new Person()

assert "Fred" == p.name

p.changeNameToBob()

assert "Bob" == p.name
```

The same concept can be applied to static methods and properties.

One application of dynamic method names can be found in the Grails web application framework. The concept of "dynamic codecs" is implemented by using dynamic method names.

`HTMLCodec` *Class*

```
class HTMLCodec {
    static encode = { theTarget ->
        HtmlUtils.htmlEscape(theTarget.toString())
    }

    static decode = { theTarget ->
        HtmlUtils.htmlUnescape(theTarget.toString())
    }
}
```

The example above shows a codec implementation. Grails comes with various codec implementations each defined in a single class. At runtime there will be multiple codec classes in the application classpath. At application startup the framework adds a `encodeXXX` and a `decodeXXX` method to certain metaclasses where `XXX` is the first part of the codec class name (e.g. `encodeHTML`). This mechanism is in the following shown in some Groovy pseudocode:

```
def codecs = classes.findAll { it.name.endsWith('Codec') }

codecs.each { codec ->
    Object.metaClass."encodeAs${codec.name-'Codec'}" = { codec.newInstance().encode
(delegate) }
    Object.metaClass."decodeFrom${codec.name-'Codec'}" = { codec.newInstance().decode
(delegate) }
```

```
}


def html = '<html><body>hello</body></html>'

assert '<html><body>hello</body></html>' == html.encodeAsHTML()
```

Runtime Discovery

At runtime it is often useful to know what other methods or properties exist at the time the method is executed. `ExpandoMetaClass` provides the following methods as of this writing:

- `getMetaMethod`
- `hasMetaMethod`
- `getMetaProperty`
- `hasMetaProperty`

Why can't you just use reflection? Well because Groovy is different, it has the methods that are "real" methods and methods that are available only at runtime. These are sometimes (but not always) represented as MetaMethods. The MetaMethods tell you what methods are available at runtime, thus your code can adapt.

This is of particular use when overriding `invokeMethod`, `getProperty` and/or `setProperty`.

GroovyObject Methods

Another feature of `ExpandoMetaClass` is that it allows to override the methods `invokeMethod`, `getProperty` and `setProperty`, all of them can be found in the `groovy.lang.GroovyObject` class.

The following example shows how to override `invokeMethod`:

```
class Stuff {
   def invokeMe() { "foo" }
}

Stuff.metaClass.invokeMethod = { String name, args ->
   def metaMethod = Stuff.metaClass.getMetaMethod(name, args)
   def result
   if(metaMethod) result = metaMethod.invoke(delegate,args)
   else {
      result = "bar"
   }
   result
}

def stf = new Stuff()

assert "foo" == stf.invokeMe()
```

```
assert "bar" == stf.doStuff()
```

The first step in the `Closure` code is to look up the `MetaMethod` for the given name and arguments. If the method can be found everything is fine and it is delegated to. If not, a dummy value is returned.

NOTE	A `MetaMethod` is a method that is known to exist on the `MetaClass` whether added at runtime or at compile-time.

The same logic can be used to override `setProperty` or `getProperty`.

```
class Person {
   String name = "Fred"
}

Person.metaClass.getProperty = { String name ->
   def metaProperty = Person.metaClass.getMetaProperty(name)
   def result
   if(metaProperty) result = metaProperty.getProperty(delegate)
   else {
      result = "Flintstone"
   }
   result
}

def p = new Person()

assert "Fred" == p.name
assert "Flintstone" == p.other
```

The important thing to note here is that instead of a `MetaMethod` a `MetaProperty` instance is looked up. If that exists the `getProperty` method of the `MetaProperty` is called, passing the delegate.

Overriding Static invokeMethod

`ExpandoMetaClass` even allows for overriding static method with a special `invokeMethod` syntax.

```
class Stuff {
   static invokeMe() { "foo" }
}

Stuff.metaClass.'static'.invokeMethod = { String name, args ->
   def metaMethod = Stuff.metaClass.getStaticMetaMethod(name, args)
   def result
   if(metaMethod) result = metaMethod.invoke(delegate,args)
   else {
      result = "bar"
   }
   result
```

```
}

assert "foo" == Stuff.invokeMe()
assert "bar" == Stuff.doStuff()
```

The logic that is used for overriding the static method is the same as we've seen before for overriding instance methods. The only difference is the access to the `metaClass.static` property and the call to `getStaticMethodName` for retrieving the static `MetaMethod` instance.

Extending Interfaces

It is possible to add methods onto interfaces with `ExpandoMetaClass`. To do this however, it **must** be enabled globally using the `ExpandoMetaClass.enableGlobally()` method before application start-up.

```
List.metaClass.sizeDoubled = {-> delegate.size() * 2 }

def list = []

list << 1
list << 2

assert 4 == list.sizeDoubled()
```

Extension modules

Extending existing classes

An extension module allows you to add new methods to existing classes, including classes which are precompiled, like classes from the JDK. Those new methods, unlike those defined through a metaclass or using a category, are available globally. For example, when you write:

Standard extension method

```
def file = new File(...)
def contents = file.getText('utf-8')
```

The `getText` method doesn't exist on the `File` class. However, Groovy knows it because it is defined in a special class, `ResourceGroovyMethods`:

ResourceGroovyMethods.java

```
public static String getText(File file, String charset) throws IOException {
 return IOGroovyMethods.getText(newReader(file, charset));
}
```

You may notice that the extension method is defined using a static method in a helper class (where various extension methods are defined). The first argument of the `getText` method corresponds to the receiver, while additional parameters correspond to the arguments of the extension method. So

here, we are defining a method called *getText* on the `File` class (because the first argument is of type `File`), which takes a single argument as a parameter (the encoding `String`).

The process of creating an extension module is simple:

- write an extension class like above
- write a module descriptor file

Then you have to make the extension module visible to Groovy, which is as simple as having the extension module classes and descriptor available on classpath. This means that you have the choice:

- either provide the classes and module descriptor directly on classpath
- or bundle your extension module into a jar for reusability

An extension module may add two kind of methods to a class:

- instance methods (to be called on an instance of a class)
- static methods (to be called on the class itself)

Instance methods

To add an instance method to an existing class, you need to create an extension class. For example, let's say you want to add a `maxRetries` method on `Integer` which accepts a closure and executes it at most *n* times until no exception is thrown. To do that, you only need to write the following:

MaxRetriesExtension.groovy

```
class MaxRetriesExtension {                                    ①
    static void maxRetries(Integer self, Closure code) {       ②
        assert self != null && self >= 0
        int retries = self
        Throwable e = null
        while (retries > 0) {
            try {
                code.call()
                break
            } catch (Throwable err) {
                e = err
                retries -= 1
            }
        }
        if (retries == 0 && e) {
            throw e
        }
    }
}
```

① The extension class

② First argument of the static method corresponds to the receiver of the message, that is to say the extended instance

Then, after having declared your extension class, you can call it this way:

```
int i=0
5.maxRetries {
    i++
}
assert i == 1
i=0
try {
    5.maxRetries {
        i++
        throw new RuntimeException("oops")
    }
} catch (RuntimeException e) {
    assert i == 5
}
```

Static methods

It is also possible to add static methods to a class. In that case, the static method needs to be defined in its **own** file. Static and instance extension methods **cannot** be present in the same class.

StaticStringExtension.groovy

```
class StaticStringExtension {                                    ①
    static String greeting(String self) {                        ②
        'Hello, world!'
    }
}
```

① The static extension class

② First argument of the static method corresponds to the class being extended and is **unused**

In which case you can call it directly on the `String` class:

```
assert String.greeting() == 'Hello, world!'
```

Module descriptor

For Groovy to be able to load your extension methods, you must declare your extension helper classes. You must create a file named `org.codehaus.groovy.runtime.ExtensionModule` into the `META-INF/groovy` directory:

org.codehaus.groovy.runtime.ExtensionModule

```
moduleName=Test module for specifications
moduleVersion=1.0-test
extensionClasses=support.MaxRetriesExtension
staticExtensionClasses=support.StaticStringExtension
```

The module descriptor requires 4 keys:

- *moduleName* : the name of your module
- *moduleVersion*: the version of your module. Note that version number is only used to check that you don't load the same module in two different versions.
- *extensionClasses*: the list of extension helper classes for instance methods. You can provide several classes, given that they are comma separated.
- *staticExtensionClasses*: the list of extension helper classes for static methods. You can provide several classes, given that they are comma separated.

Note that it is not required for a module to define both static helpers and instance helpers, and that you may add several classes to a single module. You can also extend different classes in a single module without problem. It is even possible to use different classes in a single extension class, but it is recommended to group extension methods into classes by feature set.

Extension modules and classpath

It's worth noting that you can't use an extension which is compiled at the same time as code using it. That means that to use an extension, it **has** to be available on classpath, as compiled classes, before the code using it gets compiled. Usually, this means that you can't have the *test* classes in the same source unit as the extension class itself. Since in general, test sources are separated from normal sources and executed in another step of the build, this is not an issue.

Compatibility with type checking

Unlike categories, extension modules are compatible with type checking: if they are found on classpath, then the type checker is aware of the extension methods and will not complain when you call them. It is also compatible with static compilation.

Compile-time metaprogramming

Compile-time metaprogramming in Groovy allows code generation at compile-time. Those transformations are altering the Abstract Syntax Tree (AST) of a program, which is why in Groovy we call it AST transformations. AST transformations allow you to hook into the compilation process, modify the AST and continue the compilation process to generate regular bytecode. Compared to runtime metaprogramming, this has the advantage of making the changes visible in the class file itself (that is to say, in the bytecode). Making it visible in the bytecode is important for example if you want the transformations to be part of the class contract (implementing interfaces, extending abstract classes, …) or even if you need your class to be callable from Java (or other JVM languages). For example, an AST transformation can add methods to a class. If you do it with runtime metaprogramming, the new method would only be visible from Groovy. If you do the same using

compile-time metaprogramming, the method would be visible from Java too. Last but not least, performance would likely be better with compile-time metaprogramming (because no initialization phase is required).

In this section, we will start with explaining the various compile-time transformations that are bundled with the Groovy distribution. In a subsequent section, we will describe how you can implement your own AST transformations and what are the disadvantages of this technique.

Available AST transformations

Groovy comes with various AST transformations covering different needs: reducing boilerplate (code generation), implementing design patterns (delegation, ...), logging, declarative concurrency, cloning, safer scripting, tweaking the compilation, implementing Swing patterns, testing and eventually managing dependencies. If none of those AST transformations cover your needs, you can still implement your own, as show in section Developing your own AST transformations.

AST transformations can be separated into two categories:

- global AST transformations are applied transparently, globally, as soon as they are found on compile classpath
- local AST transformations are applied by annotating the source code with markers. Unlike global AST transformations, local AST transformations may support parameters.

Groovy doesn't ship with any global AST transformation, but you can find a list of local AST transformations available for you to use in your code here:

Code generation transformations

This category of transformation includes AST transformations which help removing boilerplate code. This is typically code that you have to write but that does not carry any useful information. By autogenerating this boilerplate code, the code you have to write is left clean and concise and the chance of introducing an error by getting such boilerplate code incorrect is reduced.

`@groovy.transform.ToString`

The `@ToString` AST transformation generates a human-readable `toString` representation of the class. For example, annotating the `Person` class like below will automatically generate the `toString` method for you:

```
import groovy.transform.ToString

@ToString
class Person {
    String firstName
    String lastName
}
```

With this definition, then the following assertion passes, meaning that a `toString` method taking the field values from the class and printing them out has been generated:

```
def p = new Person(firstName: 'Jack', lastName: 'Nicholson')
assert p.toString() == 'Person(Jack, Nicholson)'
```

The `@ToString` annotation accepts several parameters which are summarized in the following table:

Attribute	Default value	Description	Example

excludes	Empty list	List of properties to exclude from toString	`@ToString(excludes=['firstName'])`
			`class Person {`
			`  String firstName`
			`  String lastName`
			`}`
			
			`def p = new Person(firstName: 'Jack', lastName: 'Nicholson')`
			`assert p.toString() == 'Person(Nicholson)'`

includes	Undefined marker list (indicates all fields)	List of fields to include in toString	`@ToString(includes=['lastName'])`
			`class Person {`
			`  String firstName`
			`  String lastName`
			`}`
			
			`def p = new Person(firstName: 'Jack', lastName: 'Nicholson')`
			`assert p.toString() == 'Person(Nicholson)'`

includeSuper	False	Should superclass be included in toString	`@ToString`
			`class Id { long id }`
			
			`@ToString(includeSuper=true)`
			`class Person extends Id {`
			`  String firstName`
			`  String lastName`
			`}`
			
			`def p = new Person(id:1, firstName: 'Jack', lastName: 'Nicholson')`
			`assert p.toString() == 'Person(Jack, Nicholson, Id(1))'`

Attribute	Default value	Description	Example

includeNames	false	Whether to include names of properties in generated toString.	`@ToString(includeNames=true)`
			`class Person {`
			`  String firstName`
			`  String lastName`
			`}`
			
			`def p = new Person(firstName: 'Jack', lastName: 'Nicholson')`
			`assert p.toString() == 'Person(firstName:Jack, lastName:Nicholson)'`

includeFields	False	Should fields be included in toString, in addition to properties	`@ToString(includeFields=true)`
			`class Person {`
			`  String firstName`
			`  String lastName`
			`  private int age`
			`  void test() {`
			`  age = 42`
			`  }`
			`}`
			
			`def p = new Person(firstName: 'Jack', lastName: 'Nicholson')`
			`p.test()`
			`assert p.toString() == 'Person(Jack, Nicholson, 42)'`

includeSuperProperties	False	Should super properties be included in toString	`class Person {`
			`  String name`
			`}`
			
			`@ToString(includeSuperProperties = true, includeNames = true)`
			`class BandMember extends Person {`
			`  String bandName`
			`}`
			
			`def bono = new BandMember(`
			`name:'Bono', bandName: 'U2'`
			`).toString()`
			
			`assert bono.toString() == 'BandMember(bandName:U2, name:Bono)'`

Attribute	Default value	Description	Example
includeSuperFields	False	Should visible super fields be included in toString	`class Person { protected String name } @ToString(includeSuperFields = true, includeNames = true) @MapConstructor(includeSuperFields = true) class BandMember extends Person { String bandName } def bono = new BandMember(name:'Bono', bandName: 'U2').toString() assert bono.toString() == 'BandMember(bandName:U2, name:Bono)'`
ignoreNulls	False	Should properties/fields with null value be displayed	`@ToString(ignoreNulls=true) class Person { String firstName String lastName } def p = new Person(firstName: 'Jack') assert p.toString() == 'Person(Jack)'`
includePackage	True	Use fully qualified class name instead of simple name in toString	`@ToString(includePackage=true) class Person { String firstName String lastName } def p = new Person(firstName: 'Jack', lastName:'Nicholson') assert p.toString() == 'acme.Person(Jack, Nicholson)'`

Attribute	Default value	Description	Example
allProperties	True	Include all JavaBean properties in toString	`@ToString(includeNames=true) class Person { String firstName String getLastName() { 'Nicholson' } } def p = new Person(firstName: 'Jack') assert p.toString() == 'acme.Person(firstName:Jack, lastName:Nicholson)'`
cache	False	Cache the toString string. Should only be set to true if the class is immutable.	`@ToString(cache=true) class Person { String firstName String lastName } def p = new Person(firstName: 'Jack', lastName:'Nicholson') def s1 = p.toString() def s2 = p.toString() assert s1 == s2 assert s1 == 'Person(Jack, Nicholson)' assert s1.is(s2) // same instance`
allNames	False	Should fields and/or properties with internal names be included in the generated toString	`@ToString(allNames=true) class Person { String $firstName } def p = new Person($firstName: "Jack") assert p.toString() == 'acme.Person(Jack)'`

`@groovy.transform.EqualsAndHashCode`

The `@EqualsAndHashCode` AST transformation aims at generating `equals` and `hashCode` methods for you. The generated hashcode follows the best practices as described in *Effective Java* by *Josh Bloch*:

```
import groovy.transform.EqualsAndHashCode
```

```
@EqualsAndHashCode
class Person {
    String firstName
    String lastName
}

def p1 = new Person(firstName: 'Jack', lastName: 'Nicholson')
def p2 = new Person(firstName: 'Jack', lastName: 'Nicholson')

assert p1==p2
assert p1.hashCode() == p2.hashCode()
```

There are several options available to tweak the behavior of `@EqualsAndHashCode`:

Attribute	Default value	Description	Example

excludes	Empty list	List of properties to exclude from equals/hashCode	`import groovy.transform.EqualsAndHashCode`
			
			`@EqualsAndHashCode(excludes=['firstName'])`
			`class Person {`
			`  String firstName`
			`  String lastName`
			`}`
			
			`def p1 = new Person(firstName: 'Jack', lastName: 'Nicholson')`
			`def p2 = new Person(firstName: 'Bob', lastName: 'Nicholson')`
			
			`assert p1==p2`
			`assert p1.hashCode() == p2.hashCode()`

Attribute	Default value	Description	Example

includes	Undefined marker list (indicating all fields)	List of fields to include in equals/hashCode	`import groovy.transform.EqualsAndHashCode`
			
			`@EqualsAndHashCode(includes=['lastName'])`
			`class Person {`
			`  String firstName`
			`  String lastName`
			`}`
			
			`def p1 = new Person(firstName: 'Jack', lastName: 'Nicholson')`
			`def p2 = new Person(firstName: 'Bob', lastName: 'Nicholson')`
			
			`assert p1==p2`
			`assert p1.hashCode() == p2.hashCode()`

Attribute	Default value	Description	Example

cache	False	Cache the hashCode computation. Should only be set to true if the class is immutable.	<pre>import groovy.transform.EqualsAndHashCode
			import groovy.transform.Immutable
			
			@Immutable
			class SlowHashCode {
			    static final SLEEP_PERIOD = 500
			
			    int hashCode() {
			        sleep SLEEP_PERIOD
			        127
			    }
			}
			
			@EqualsAndHashCode(cache=true)
			@Immutable
			class Person {
			    SlowHashCode slowHashCode = new SlowHashCode()
			}
			
			def p = new Person()
			p.hashCode()
			
			def start = System.currentTimeMillis()
			p.hashCode()
			assert System.currentTimeMillis() - start < SlowHashCode.SLEEP_PERIOD</pre>

Attribute	Default value	Description	Example

callSuper	False	Whether to include super in equals and hashCode calculations	<pre>import groovy.transform.EqualsAndHashCode
			
			@EqualsAndHashCode
			class Living {
			    String race
			}
			
			@EqualsAndHashCode(callSuper=true)
			class Person extends Living {
			    String firstName
			    String lastName
			}
			
			def p1 = new Person(race:'Human', firstName: 'Jack', lastName: 'Nicholson')
			def p2 = new Person(race: 'Human being', firstName: 'Jack', lastName: 'Nicholson')
			
			assert p1!=p2
			assert p1.hashCode() != p2.hashCode()</pre>

Attribute	Default value	Description	Example

includeFields	False	Should fields be included in equals/hashCode, in addition to properties	`import groovy.transform.EqualsAndHashCode`
			
			`@EqualsAndHashCode(includeFields=true)`
			`class Person {`
			`  private String firstName`
			
			`  Person(String firstName) {`
			`  this.firstName = firstName`
			`  }`
			`}`
			
			`def p1 = new Person('Jack')`
			`def p2 = new Person('Jack')`
			`def p3 = new Person('Bob')`
			
			`assert p1 == p2`
			`assert p1 != p3`

useCanEqual	True	Should equals call canEqual helper method.	See http://www.artima.com/lejava/articles/equality.html

allProperties	False	Should JavaBean properties be included in equals and hashCode calculations	`@EqualsAndHashCode(allProperties=true, excludes='first, last')`
			`class Person {`
			`  String first, last`
			`  String getInitials() { first[0] + last[0] }`
			`}`
			
			`def p1 = new Person(first: 'Jack', last: 'Smith')`
			`def p2 = new Person(first: 'Jack', last: 'Spratt')`
			`def p3 = new Person(first: 'Bob', last: 'Smith')`
			
			`assert p1 == p2`
			`assert p1.hashCode() == p2.hashCode()`
			`assert p1 != p3`
			`assert p1.hashCode() != p3.hashCode()`

Attribute	Default value	Description	Example

allNames	False	Should fields and/or properties with internal names be included in equals and hashCode calculations	`import groovy.transform.EqualsAndHashCode`
			
			`@EqualsAndHashCode(allNames=true)`
			`class Person {`
			`  String $firstName`
			`}`
			
			`def p1 = new Person($firstName: 'Jack')`
			`def p2 = new Person($firstName: 'Bob')`
			
			`assert p1 != p2`
			`assert p1.hashCode() != p2.hashCode()`

`@groovy.transform.TupleConstructor`

The `@TupleConstructor` annotation aims at eliminating boilerplate code by generating constructors for you. A tuple constructor is created having a parameter for each property (and possibly each field). Each parameter has a default value (using the initial value of the property if present or otherwise Java's default value according to the properties type).

Implementation Details

Normally you don't need to understand the implementation details of the generated constructor(s); you just use them in the normal way. However, if you want to add multiple constructors, understand Java integration options or meet requirements of some dependency injection frameworks, then some details are useful.

As previously mentioned, the generated constructor has default values applied. In later compilation phases, the Groovy compiler's standard default value processing behavior is then applied. The end result is that multiple constructors are placed within the bytecode of your class. This provides a well understood semantics and is also useful for Java integration purposes. As an example, the following code will generate 3 constructors:

```
import groovy.transform.TupleConstructor

@TupleConstructor
class Person {
    String firstName
    String lastName
}
```

```
// traditional map-style constructor
def p1 = new Person(firstName: 'Jack', lastName: 'Nicholson')
// generated tuple constructor
def p2 = new Person('Jack', 'Nicholson')
// generated tuple constructor with default value for second property
def p3 = new Person('Jack')
```

The first constructor is a no-arg constructor which allows the traditional map-style construction so long as you don't have final properties. Groovy calls the no-arg constructor and then the relevant setters under the covers. It is worth noting that if the first property (or field) has type LinkedHashMap or if there is a single Map, AbstractMap or HashMap property (or field), then the map-style named arguments won't be available.

The other constructors are generated by taking the properties in the order they are defined. Groovy will generate as many constructors as there are properties (or fields, depending on the options).

Setting the `defaults` attribute (see the available configuration options table) to `false`, disables the normal default values behavior which means:

- Exactly one constructor will be produced
- Attempting to use an initial value will produce an error
- Map-style named arguments won't be available

This attribute is normally only used in situations where another Java framework is expecting exactly one constructor, e.g. injection frameworks or JUnit parameterized runners.

Immutability support

If the `@PropertyOptions` annotation is also found on the class with the `@TupleConstructor` annotation, then the generated constructor may contain custom property handling logic. The `propertyHandler` attribute on the `@PropertyOptions` annotation could for instance be set to `ImmutablePropertyHandler` which will result in the addition of the necessary logic for immutable classes (defensive copy in, cloning, etc.). This normally would happen automatically behind the scenes when you use the `@Immutable` meta-annotation. Some of the annotation attributes might not be supported by all property handlers.

Customization options

The `@TupleConstructor` AST transformation accepts several annotation attributes:

Attribute	Default value	Description	Example

excludes	Empty list	List of properties to exclude from tuple constructor generation	<pre>import groovy.transform.TupleConstructor
			
			@TupleConstructor(excludes=['lastName'])
			class Person {
			    String firstName
			    String lastName
			}
			
			def p1 = new Person(firstName: 'Jack', lastName: 'Nicholson')
			def p2 = new Person('Jack')
			try {
			    // will fail because the second property is excluded
			    def p3 = new Person('Jack', 'Nicholson')
			} catch (e) {
			    assert e.message.contains('Could not find matching constructor')
			}</pre>

Attribute	Default value	Description	Example

includes	Undefined list (indicates all fields)	List of fields to include in tuple constructor generation	`import groovy.transform.TupleConstructor`
			
			`@TupleConstructor(includes=['firstName'])`
			`class Person {`
			`  String firstName`
			`  String lastName`
			`}`
			
			`def p1 = new Person(firstName: 'Jack', lastName: 'Nicholson')`
			`def p2 = new Person('Jack')`
			`try {`
			`  // will fail because the second property is not included`
			`  def p3 = new Person('Jack', 'Nicholson')`
			`} catch (e) {`
			`  assert e.message.contains('Could not find matching constructor')`
			`}`

includeProperties	True	Should properties be included in tuple constructor generation	`import groovy.transform.TupleConstructor`
			
			`@TupleConstructor(includeProperties=false)`
			`class Person {`
			`  String firstName`
			`  String lastName`
			`}`
			
			`def p1 = new Person(firstName: 'Jack', lastName: 'Nicholson')`
			
			`try {`
			`  def p2 = new Person('Jack', 'Nicholson')`
			`} catch(e) {`
			`  // will fail because properties are not included`
			`}`

Attribute	Default value	Description	Example

includeFields	False	Should fields be included in tuple constructor generation, in addition to properties	<pre>import groovy.transform.TupleConstructor
			
			@TupleConstructor(includeFields=true)
			class Person {
			    String firstName
			    String lastName
			    private String occupation
			    public String toString() {
			        "$firstName $lastName: $occupation"
			    }
			}
			
			def p1 = new Person(firstName: 'Jack', lastName: 'Nicholson', occupation: 'Actor')
			def p2 = new Person('Jack', 'Nicholson', 'Actor')
			
			assert p1.firstName == p2.firstName
			assert p1.lastName == p2.lastName
			assert p1.toString() == 'Jack Nicholson: Actor'
			assert p1.toString() == p2.toString()</pre>

Attribute	Default value	Description	Example
includeSuperProperties	True	Should properties from super classes be included in tuple constructor generation
```
import groovy.transform.TupleConstructor

class Base {
    String occupation
}

@TupleConstructor(includeSuperProperties=true)
class Person extends Base {
    String firstName
    String lastName
    public String toString() {
        "$firstName $lastName: $occupation"
    }
}

def p1 = new Person(firstName: 'Jack', lastName: 'Nicholson')

def p2 = new Person('Actor', 'Jack', 'Nicholson')

assert p1.firstName == p2.firstName
assert p1.lastName == p2.lastName
assert p1.toString() == 'Jack Nicholson: null'
assert p2.toString() == 'Jack Nicholson: Actor'
```
 |

Attribute	Default value	Description	Example

includeSuperFields	False	Should fields from super classes be included in tuple constructor generation	<pre>import groovy.transform.TupleConstructor
			
			class Base {
			    protected String occupation
			    public String occupation() { this.occupation }
			}
			
			@TupleConstructor(includeSuperFields=true)
			class Person extends Base {
			    String firstName
			    String lastName
			    public String toString() {
			        "$firstName $lastName: ${occupation()}"
			    }
			}
			
			def p1 = new Person(firstName: 'Jack', lastName: 'Nicholson', occupation: 'Actor')
			
			def p2 = new Person('Actor', 'Jack', 'Nicholson')
			
			assert p1.firstName == p2.firstName
			assert p1.lastName == p2.lastName
			assert p1.toString() == 'Jack Nicholson: Actor'
			assert p2.toString() == p1.toString()</pre>

Attribute	Default value	Description	Example

callSuper	False	Should super properties be called within a call to the parent constructor rather than set as properties	<pre>import groovy.transform.TupleConstructor
			
			class Base {
			    String occupation
			    Base() {}
			    Base(String job) { occupation = job?.toLowerCase() }
			}
			
			@TupleConstructor(includeSuperProperties = true, callSuper=true)
			class Person extends Base {
			    String firstName
			    String lastName
			    public String toString() {
			        "$firstName $lastName: $occupation"
			    }
			}
			
			def p1 = new Person(firstName: 'Jack', lastName: 'Nicholson')
			
			def p2 = new Person('ACTOR', 'Jack', 'Nicholson')
			
			assert p1.firstName == p2.firstName
			assert p1.lastName == p2.lastName
			assert p1.toString() == 'Jack Nicholson: null'
			assert p2.toString() == 'Jack Nicholson: actor'</pre>

Attribute	Default value	Description	Example

force	False	By default, the transformation will do nothing if a constructor is already defined. Setting this attribute to true, the constructor will be generated and it's your responsibility to ensure that no duplicate constructor is defined.	`import groovy.transform.*`
			
			`@ToString @TupleConstructor(force=true)`
			`final class Person {`
			`  String name`
			`  // explicit constructor would normally disable tuple constructor`
			`  Person(String first, String last) { this("$first $last") }`
			`}`
			
			`assert new Person('john smith').toString() == 'Person(john smith)'`
			`assert new Person('john', 'smith').toString() == 'Person(john smith)'`

defaults	True	Indicates that default value processing is enabled for constructor parameters. Set to false to obtain exactly one constructor but with initial value support and named-arguments disabled.	`@ToString`
			`@TupleConstructor(defaults=false)`
			`class Musician {`
			`  String name`
			`  String instrument`
			`  int born`
			`}`
			
			`assert new Musician('Jimi', 'Guitar', 1942).toString() == 'Musician(Jimi, Guitar, 1942)'`
			`assert Musician.constructors.size() == 1`

Attribute	Default value	Description	Example

useSetters	False	By default, the transformation will directly set the backing field of each property from its corresponding constructor parameter. Setting this attribute to true, the constructor will instead call setters if they exist. It's usually deemed bad style from within a constructor to call setters that can be overridden. It's your responsibility to avoid such bad style.	`import groovy.transform.*`
			
			`@ToString @TupleConstructor (useSetters=true) final class Foo {`
			`  String bar`
			`  void setBar(String bar) {`
			`  this.bar = bar ?.toUpperCase() // null-safe`
			`  }`
			`}`
			
			`assert new Foo('cat').toString() == 'Foo(CAT)'`
			`assert new Foo(bar: 'cat' ).toString() == 'Foo(CAT)'`

allNames	False	Should fields and/or properties with internal names be included within the constructor	`import groovy.transform.TupleConstructor`
			
			`@TupleConstructor(allNames=true)`
			`class Person {`
			`  String $firstName`
			`}`
			
			`def p = new Person('Jack')`
			
			`assert p.$firstName == 'Jack'`

allProperties	False	Should JavaBean properties be included within the constructor	`@TupleConstructor(allProperties=t rue)`
			`class Person {`
			`  String first`
			`  private String last`
			`  void setLast(String last) {`
			`  this.last = last`
			`  }`
			`  String getName() { "$first $last" }`
			`}`
			
			`assert new Person('john', 'smith').name == 'john smith'`

Attribute	Default value	Description	Example

pre	empty	A closure containing statements to be inserted at the start of the generated constructor(s)	`import groovy.transform.TupleConstructor`
			
			`@TupleConstructor(pre={ first = first?.toLowerCase() })`
			`class Person {`
			`  String first`
			`}`
			
			`def p = new Person('Jack')`
			
			`assert p.first == 'jack'`

post	empty	A closure containing statements to be inserted at the end of the generated constructor(s)	`import groovy.transform.TupleConstructor`
			`import static groovy.test.GroovyAssert.shouldFail`
			
			`@TupleConstructor(post={ assert first })`
			`class Person {`
			`  String first`
			`}`
			
			`def jack = new Person('Jack')`
			`shouldFail {`
			`  def unknown = new Person()`
			`}`

Setting the `defaults` annotation attribute to `false` and the `force` annotation attribute to `true` allows multiple tuple constructors to be created by using different customization options for the different cases (provided each case has a different type signature) as shown in the following example:

```
class Named {
  String name
}

@ToString(includeSuperProperties=true, ignoreNulls=true, includeNames=true,
includeFields=true)
@TupleConstructor(force=true, defaults=false)
@TupleConstructor(force=true, defaults=false, includeFields=true)
@TupleConstructor(force=true, defaults=false, includeSuperProperties=true)
class Book extends Named {
  Integer published
```

```
  private Boolean fiction
  Book() {}
}

assert new Book("Regina", 2015).toString() == 'Book(published:2015, name:Regina)'
assert new Book(2015, false).toString() == 'Book(published:2015, fiction:false)'
assert new Book(2015).toString() == 'Book(published:2015)'
assert new Book().toString() == 'Book()'
assert Book.constructors.size() == 4
```

Similarly, here is another example using different options for `includes`:

```
@ToString(includeSuperProperties=true, ignoreNulls=true, includeNames=true,
includeFields=true)
@TupleConstructor(force=true, defaults=false, includes='name,year')
@TupleConstructor(force=true, defaults=false, includes='year,fiction')
@TupleConstructor(force=true, defaults=false, includes='name,fiction')
class Book {
    String name
    Integer year
    Boolean fiction
}

assert new Book("Regina", 2015).toString() == 'Book(name:Regina, year:2015)'
assert new Book(2015, false).toString() == 'Book(year:2015, fiction:false)'
assert new Book("Regina", false).toString() == 'Book(name:Regina, fiction:false)'
assert Book.constructors.size() == 3
```

`@groovy.transform.MapConstructor`

The `@MapConstructor` annotation aims at eliminating boilerplate code by generating a map constructor for you. A map constructor is created such that each property in the class is set based on the value in the supplied map having the key with the name of the property. Usage is as shown in this example:

```
import groovy.transform.*

@ToString
@MapConstructor
class Person {
    String firstName
    String lastName
}

def p1 = new Person(firstName: 'Jack', lastName: 'Nicholson')
assert p1.toString() == 'Person(Jack, Nicholson)'
```

The generated constructor will be roughly like this:

```
public Person(Map args) {
    if (args.containsKey('firstName')) {
        this.firstName = args.get('firstName')
    }
    if (args.containsKey('lastName')) {
        this.lastName = args.get('lastName')
    }
}
```

@groovy.transform.Canonical

The @Canonical meta-annotation combines the @ToString, @EqualsAndHashCode and @TupleConstructor annotations:

```
import groovy.transform.Canonical

@Canonical
class Person {
    String firstName
    String lastName
}
def p1 = new Person(firstName: 'Jack', lastName: 'Nicholson')
assert p1.toString() == 'Person(Jack, Nicholson)' // Effect of @ToString

def p2 = new Person('Jack','Nicholson') // Effect of @TupleConstructor
assert p2.toString() == 'Person(Jack, Nicholson)'

assert p1==p2 // Effect of @EqualsAndHashCode
assert p1.hashCode()==p2.hashCode() // Effect of @EqualsAndHashCode
```

A similar immutable class can be generated using the @Immutable meta-annotation instead. The @Canonical meta-annotation supports the configuration options found in the annotations it aggregates. See those annotations for more details.

```
import groovy.transform.Canonical

@Canonical(excludes=['lastName'])
class Person {
    String firstName
    String lastName
}
def p1 = new Person(firstName: 'Jack', lastName: 'Nicholson')
assert p1.toString() == 'Person(Jack)' // Effect of @ToString(excludes=['lastName'])

def p2 = new Person('Jack') // Effect of @TupleConstructor(excludes=['lastName'])
assert p2.toString() == 'Person(Jack)'

assert p1==p2 // Effect of @EqualsAndHashCode(excludes=['lastName'])
```

```
assert p1.hashCode()==p2.hashCode() // Effect of
@EqualsAndHashCode(excludes=['lastName'])
```

The `@Canonical` meta-annotation can be used in conjunction with an explicit use one or more of its component annotations, like this:

```
import groovy.transform.Canonical

@Canonical(excludes=['lastName'])
class Person {
    String firstName
    String lastName
}
def p1 = new Person(firstName: 'Jack', lastName: 'Nicholson')
assert p1.toString() == 'Person(Jack)' // Effect of @ToString(excludes=['lastName'])

def p2 = new Person('Jack') // Effect of @TupleConstructor(excludes=['lastName'])
assert p2.toString() == 'Person(Jack)'

assert p1==p2 // Effect of @EqualsAndHashCode(excludes=['lastName'])
assert p1.hashCode()==p2.hashCode() // Effect of
@EqualsAndHashCode(excludes=['lastName'])
```

Any applicable annotation attributes from `@Canonical` are passed along to the explicit annotation but attributes already existing in the explicit annotation take precedence.

`@groovy.transform.InheritConstructors`

The `@InheritConstructor` AST transformation aims at generating constructors matching super constructors for you. This is in particular useful when overriding exception classes:

```
import groovy.transform.InheritConstructors

@InheritConstructors
class CustomException extends Exception {}

// all those are generated constructors
new CustomException()
new CustomException("A custom message")
new CustomException("A custom message", new RuntimeException())
new CustomException(new RuntimeException())

// Java 7 only
// new CustomException("A custom message", new RuntimeException(), false, true)
```

The `@InheritConstructor` AST transformation supports the following configuration options:

Attribute	Default value	Description	Example

constructorAnnotations	False	Whether to carry over annotations from the constructor during copying	`@Retention(RetentionPolicy.RUNTIME)`
			`@Target([ElementType.CONSTRUCTOR])`
			`public @interface ConsAnno {}`
			
			`class Base {`
			`  @ConsAnno Base() {}`
			`}`
			
			`@InheritConstructors(constructorAnnotations=true)`
			`class Child extends Base {}`
			
			`assert Child.constructors[0].annotations[0].annotationType().name == 'groovy.transform.Generated'`
			`assert Child.constructors[0].annotations[1].annotationType().name == 'ConsAnno'`

parameterAnnotations	False	Whether to carry over annotations from the constructor parameters when copying the constructor	`@Retention(RetentionPolicy.RUNTIME)`
			`@Target([ElementType.PARAMETER])`
			`public @interface ParamAnno {}`
			
			`class Base {`
			`  Base(@ParamAnno String name) {}`
			`}`
			
			`@InheritConstructors(parameterAnnotations=true)`
			`class Child extends Base {}`
			
			`assert Child.constructors[0].parameterAnnotations[0][0].annotationType().name == 'ParamAnno'`

`@groovy.lang.Category`

The `@Category` AST transformation simplifies the creation of Groovy categories. Historically, a Groovy category was written like this:

```
class TripleCategory {
    public static Integer triple(Integer self) {
        3*self
    }
}
use (TripleCategory) {
    assert 9 == 3.triple()
}
```

The `@Category` transformation lets you write the same using an instance-style class, rather than a static class style. This removes the need for having the first argument of each method being the receiver. The category can be written like this:

```
@Category(Integer)
class TripleCategory {
    public Integer triple() { 3*this }
}
use (TripleCategory) {
    assert 9 == 3.triple()
}
```

Note that the mixed in class can be referenced using `this` instead. It's also worth noting that using instance fields in a category class is inherently unsafe: categories are not stateful (like traits).

`@groovy.transform.IndexedProperty`

The `@IndexedProperty` annotation aims at generating indexed getters/setters for properties of list/array types. This is in particular useful if you want to use a Groovy class from Java. While Groovy supports GPath to access properties, this is not available from Java. The `@IndexedProperty` annotation will generate indexed properties of the following form:

```
class SomeBean {
    @IndexedProperty String[] someArray = new String[2]
    @IndexedProperty List someList = []
}

def bean = new SomeBean()
bean.setSomeArray(0, 'value')
bean.setSomeList(0, 123)

assert bean.someArray[0] == 'value'
assert bean.someList == [123]
```

`@groovy.lang.Lazy`

The `@Lazy` AST transformation implements lazy initialization of fields. For example, the following code:

```
class SomeBean {
    @Lazy LinkedList myField
}
```

will produce the following code:

```
List $myField
List getMyField() {
    if ($myField!=null) { return $myField }
    else {
        $myField = new LinkedList()
        return $myField
    }
}
```

The default value which is used to initialize the field is the default constructor of the declaration type. It is possible to define a default value by using a closure on the right hand side of the property assignment, as in the following example:

```
class SomeBean {
    @Lazy LinkedList myField = { ['a','b','c']}()
}
```

In that case, the generated code looks like the following:

```
List $myField
List getMyField() {
    if ($myField!=null) { return $myField }
    else {
        $myField = { ['a','b','c']}()
        return $myField
    }
}
```

If the field is declared volatile then initialization will be synchronized using the double-checked locking pattern.

Using the `soft=true` parameter, the helper field will use a `SoftReference` instead, providing a simple way to implement caching. In that case, if the garbage collector decides to collect the reference, initialization will occur the next time the field is accessed.

`@groovy.lang.Newify`

The `@Newify` AST transformation is used to bring alternative syntaxes to construct objects:

- Using the `Python` style:

```
@Newify([Tree,Leaf])
class TreeBuilder {
    Tree tree = Tree(Leaf('A'),Leaf('B'),Tree(Leaf('C')))
}
```

- or using the Ruby style:

```
@Newify([Tree,Leaf])
class TreeBuilder {
    Tree tree = Tree.new(Leaf.new('A'),Leaf.new('B'),Tree.new(Leaf.new('C')))
}
```

The Ruby version can be disabled by setting the auto flag to false.

@groovy.transform.Sortable

The @Sortable AST transformation is used to help write classes that are Comparable and easily sorted typically by numerous properties. It is easy to use as shown in the following example where we annotate the Person class:

```
import groovy.transform.Sortable

@Sortable class Person {
    String first
    String last
    Integer born
}
```

The generated class has the following properties:

- it implements the Comparable interface
- it contains a compareTo method with an implementation based on the natural ordering of the first, last and born properties
- it has three methods returning comparators: comparatorByFirst, comparatorByLast and comparatorByBorn.

The generated compareTo method will look like this:

```
public int compareTo(java.lang.Object obj) {
    if (this.is(obj)) {
        return 0
    }
    if (!(obj instanceof Person)) {
        return -1
    }
    java.lang.Integer value = this.first <=> obj.first
```

```
    if (value != 0) {
        return value
    }
    value = this.last <=> obj.last
    if (value != 0) {
        return value
    }
    value = this.born <=> obj.born
    if (value != 0) {
        return value
    }
    return 0
}
```

As an example of the generated comparators, the `comparatorByFirst` comparator will have a `compare` method that looks like this:

```
public int compare(java.lang.Object arg0, java.lang.Object arg1) {
    if (arg0 == arg1) {
        return 0
    }
    if (arg0 != null && arg1 == null) {
        return -1
    }
    if (arg0 == null && arg1 != null) {
        return 1
    }
    return arg0.first <=> arg1.first
}
```

The `Person` class can be used wherever a `Comparable` is expected and the generated comparators wherever a `Comparator` is expected as shown by these examples:

```
def people = [
    new Person(first: 'Johnny', last: 'Depp', born: 1963),
    new Person(first: 'Keira', last: 'Knightley', born: 1985),
    new Person(first: 'Geoffrey', last: 'Rush', born: 1951),
    new Person(first: 'Orlando', last: 'Bloom', born: 1977)
]

assert people[0] > people[2]
assert people.sort()*.last == ['Rush', 'Depp', 'Knightley', 'Bloom']
assert people.sort(false, Person.comparatorByFirst())*.first == ['Geoffrey', 'Johnny',
'Keira', 'Orlando']
assert people.sort(false, Person.comparatorByLast())*.last == ['Bloom', 'Depp',
'Knightley', 'Rush']
assert people.sort(false, Person.comparatorByBorn())*.last == ['Rush', 'Depp',
'Bloom', 'Knightley']
```

Normally, all properties are used in the generated `compareTo` method in the priority order in which they are defined. You can include or exclude certain properties from the generated `compareTo` method by giving a list of property names in the `includes` or `excludes` annotation attributes. If using `includes`, the order of the property names given will determine the priority of properties when comparing. To illustrate, consider the following `Person` class definition:

```
@Sortable(includes='first,born') class Person {
    String last
    int born
    String first
}
```

It will have two comparator methods `comparatorByFirst` and `comparatorByBorn` and the generated `compareTo` method will look like this:

```
public int compareTo(java.lang.Object obj) {
    if (this.is(obj)) {
        return 0
    }
    if (!(obj instanceof Person)) {
        return -1
    }
    java.lang.Integer value = this.first <=> obj.first
    if (value != 0) {
        return value
    }
    value = this.born <=> obj.born
    if (value != 0) {
        return value
    }
    return 0
}
```

This `Person` class can be used as follows:

```
def people = [
    new Person(first: 'Ben', last: 'Affleck', born: 1972),
    new Person(first: 'Ben', last: 'Stiller', born: 1965)
]

assert people.sort()*.last == ['Stiller', 'Affleck']
```

The behavior of the `@Sortable` AST transformation can be further changed using the following additional parameters:

Attribute	Default value	Description	Example

allProperties	True	Should JavaBean properties (ordered after native properties) be used	<pre>import groovy.transform.*
			
			@Canonical(includeFields = true)
			@Sortable(allProperties = true,
			includes = 'nameSize')
			class Player {
			  String name
			  int getNameSize() { name.size()
			}
			}
			
			def finalists = [
			  new Player('Serena'),
			  new Player('Venus'),
			  new Player('CoCo'),
			  new Player('Mirjana')
			]
			
			assert finalists.sort()*.name ==
			['CoCo', 'Venus', 'Serena',
			'Mirjana']</pre>

allNames	False	Should properties with "internal" names be used	<pre>import groovy.transform.*
			
			@Canonical(allNames = true)
			@Sortable(allNames = false)
			class Player {
			  String $country
			  String name
			}
			
			def finalists = [
			  new Player('USA', 'Serena'),
			  new Player('USA', 'Venus'),
			  new Player('USA', 'CoCo'),
			  new Player('Croatian',
			'Mirjana')
			]
			
			assert finalists.sort()*.name ==
			['Mirjana', 'CoCo', 'Serena',
			'Venus']</pre>

Attribute	Default value	Description	Example

includeSuperProperties	False	Should super properties also be used (ordered first)	<pre>class Person {
			  String name
			}
			
			@Canonical(includeSuperProperties = true)
			@Sortable(includeSuperProperties = true)
			class Citizen extends Person {
			  String country
			}
			
			def people = [
			  new Citizen('Bob', 'Italy'),
			  new Citizen('Cathy', 'Hungary'),
			  new Citizen('Cathy', 'Egypt'),
			  new Citizen('Bob', 'Germany'),
			  new Citizen('Alan', 'France')
			]
			
			assert people.sort()*.name == ['Alan', 'Bob', 'Bob', 'Cathy', 'Cathy']
			assert people.sort()*.country == ['France', 'Germany', 'Italy', 'Egypt', 'Hungary']</pre>

`@groovy.transform.builder.Builder`

The `@Builder` AST transformation is used to help write classes that can be created using *fluent* api calls. The transform supports multiple building strategies to cover a range of cases and there are a number of configuration options to customize the building process. If you're an AST hacker, you can also define your own strategy class. The following table lists the available strategies that are bundled with Groovy and the configuration options each strategy supports.

Strategy	Description	builderClassName	builderMethodName	buildMethodName	prefix	includes/ excludes	includeSuperProperties	allNames
`SimpleStrategy`	chained setters	n/a	n/a	n/a	yes, default "set"	yes	n/a	yes, default `false`

`ExternalStrategy`	explicit builder class, class being built untouched	n/a	n/a	yes, default "build"	yes, default ""	yes	yes, default `false`	yes, default `false`
`DefaultStrategy`	creates a nested helper class	yes, default *<TypeName>*Builder	yes, default "builder"	yes, default "build"	yes, default ""	yes	yes, default `false`	yes, default `false`
`InitializerStrategy`	creates a nested helper class providing type-safe fluent creation	yes, default *<TypeName>*Initializer	yes, default "createInitializer"	yes, default "create" but usually only used internally	yes, default ""	yes	yes, default `false`	yes, default `false`

SimpleStrategy

To use the `SimpleStrategy`, annotate your Groovy class using the `@Builder` annotation, and specify the strategy as shown in this example:

```
import groovy.transform.builder.*

@Builder(builderStrategy=SimpleStrategy)
class Person {
    String first
    String last
    Integer born
}
```

Then, just call the setters in a chained fashion as shown here:

```
def p1 = new Person().setFirst('Johnny').setLast('Depp').setBorn(1963)
assert "$p1.first $p1.last" == 'Johnny Depp'
```

For each property, a generated setter will be created which looks like this:

```
public Person setFirst(java.lang.String first) {
    this.first = first
    return this
```

```
}
```

You can specify a prefix as shown in this example:

```
import groovy.transform.builder.*

@Builder(builderStrategy=SimpleStrategy, prefix="")
class Person {
    String first
    String last
    Integer born
}
```

And calling the chained setters would look like this:

```
def p = new Person().first('Johnny').last('Depp').born(1963)
assert "$p.first $p.last" == 'Johnny Depp'
```

You can use the `SimpleStrategy` in conjunction with `@TupleConstructor`. If your `@Builder` annotation doesn't have explicit `includes` or `excludes` annotation attributes but your `@TupleConstructor` annotation does, the ones from `@TupleConstructor` will be re-used for `@Builder`. The same applies for any annotation aliases which combine `@TupleConstructor` such as `@Canonical`.

The annotation attribute `useSetters` can be used if you have a setter which you want called as part of the construction process. See the JavaDoc for details.

The annotation attributes `builderClassName`, `buildMethodName`, `builderMethodName`, `forClass` and `includeSuperProperties` are not supported for this strategy.

NOTE	Groovy already has built-in building mechanisms. Don't rush to using `@Builder` if the built-in mechanisms meet your needs. Some examples:

```
def p2 = new Person(first: 'Keira', last: 'Knightley', born: 1985)
def p3 = new Person().tap {
    first = 'Geoffrey'
    last = 'Rush'
    born = 1951
}
```

ExternalStrategy

To use the `ExternalStrategy`, create and annotate a Groovy builder class using the `@Builder` annotation, specify the class the builder is for using `forClass` and indicate use of the `ExternalStrategy`. Suppose you have the following class you would like a builder for:

```
class Person {
```

```
    String first
    String last
    int born
}
```

you explicitly create and use your builder class as follows:

```
import groovy.transform.builder.*

@Builder(builderStrategy=ExternalStrategy, forClass=Person)
class PersonBuilder { }

def p = new PersonBuilder().first('Johnny').last('Depp').born(1963).build()
assert "$p.first $p.last" == 'Johnny Depp'
```

Note that the (normally empty) builder class you provide will be filled in with appropriate setters and a build method. The generated build method will look something like:

```
public Person build() {
    Person _thePerson = new Person()
    _thePerson.first = first
    _thePerson.last = last
    _thePerson.born = born
    return _thePerson
}
```

The class you are creating the builder for can be any Java or Groovy class following the normal JavaBean conventions, e.g. a no-arg constructor and setters for the properties. Here is an example using a Java class:

```
import groovy.transform.builder.*

@Builder(builderStrategy=ExternalStrategy, forClass=javax.swing.DefaultButtonModel)
class ButtonModelBuilder {}

def model = new ButtonModelBuilder().enabled(true).pressed(true).armed(true).rollover
(true).selected(true).build()
assert model.isArmed()
assert model.isPressed()
assert model.isEnabled()
assert model.isSelected()
assert model.isRollover()
```

The generated builder can be customised using the `prefix`, `includes`, `excludes` and `buildMethodName` annotation attributes. Here is an example illustrating various customisations:

```
import groovy.transform.builder.*
import groovy.transform.Canonical

@Canonical
class Person {
    String first
    String last
    int born
}

@Builder(builderStrategy=ExternalStrategy, forClass=Person, includes=['first',
'last'], buildMethodName='create', prefix='with')
class PersonBuilder { }

def p = new PersonBuilder().withFirst('Johnny').withLast('Depp').create()
assert "$p.first $p.last" == 'Johnny Depp'
```

The `builderMethodName` and `builderClassName` annotation attributes for `@Builder` aren't applicable for this strategy.

You can use the `ExternalStrategy` in conjunction with `@TupleConstructor`. If your `@Builder` annotation doesn't have explicit `includes` or `excludes` annotation attributes but the `@TupleConstructor` annotation of the class you are creating the builder for does, the ones from `@TupleConstructor` will be re-used for `@Builder`. The same applies for any annotation aliases which combine `@TupleConstructor` such as `@Canonical`.

DefaultStrategy

To use the `DefaultStrategy`, annotate your Groovy class using the `@Builder` annotation as shown in this example:

```
import groovy.transform.builder.Builder

@Builder
class Person {
    String firstName
    String lastName
    int age
}

def person = Person.builder().firstName("Robert").lastName("Lewandowski").age(21
).build()
assert person.firstName == "Robert"
assert person.lastName == "Lewandowski"
assert person.age == 21
```

If you want, you can customize various aspects of the building process using the `builderClassName`, `buildMethodName`, `builderMethodName`, `prefix`, `includes` and `excludes` annotation attributes, some of which are used in the example here:

```
import groovy.transform.builder.Builder

@Builder(buildMethodName='make', builderMethodName='maker', prefix='with', excludes
='age')
class Person {
    String firstName
    String lastName
    int age
}

def p = Person.maker().withFirstName("Robert").withLastName("Lewandowski").make()
assert "$p.firstName $p.lastName" == "Robert Lewandowski"
```

This strategy also supports annotating static methods and constructors. In this case, the static method or constructor parameters become the properties to use for building purposes and in the case of static methods, the return type of the method becomes the target class being built. If you have more than one `@Builder` annotation used within a class (at either the class, method or constructor positions) then it is up to you to ensure that the generated helper classes and factory methods have unique names (i.e. no more than one can use the default name values). Here is an example highlighting method and constructor usage (and also illustrating the renaming required for unique names).

```
import groovy.transform.builder.*
import groovy.transform.*

@ToString
@Builder
class Person {
  String first, last
  int born

  Person(){}

  @Builder(builderClassName='MovieBuilder', builderMethodName='byRoleBuilder')
  Person(String roleName) {
     if (roleName == 'Jack Sparrow') {
         this.first = 'Johnny'; this.last = 'Depp'; this.born = 1963
     }
  }

  @Builder(builderClassName='NameBuilder', builderMethodName='nameBuilder', prefix
='having', buildMethodName='fullName')
  static String join(String first, String last) {
      first + ' ' + last
  }

  @Builder(builderClassName='SplitBuilder', builderMethodName='splitBuilder')
  static Person split(String name, int year) {
      def parts = name.split(' ')
```

```
        new Person(first: parts[0], last: parts[1], born: year)
    }
}

assert Person.splitBuilder().name("Johnny Depp").year(1963).build().toString() ==
'Person(Johnny, Depp, 1963)'
assert Person.byRoleBuilder().roleName("Jack Sparrow").build().toString() ==
'Person(Johnny, Depp, 1963)'
assert Person.nameBuilder().havingFirst('Johnny').havingLast('Depp').fullName() ==
'Johnny Depp'
assert Person.builder().first("Johnny").last('Depp').born(1963).build().toString() ==
'Person(Johnny, Depp, 1963)'
```

The `forClass` annotation attribute is not supported for this strategy.

InitializerStrategy

To use the `InitializerStrategy`, annotate your Groovy class using the `@Builder` annotation, and specify the strategy as shown in this example:

```
import groovy.transform.builder.*
import groovy.transform.*

@ToString
@Builder(builderStrategy=InitializerStrategy)
class Person {
    String firstName
    String lastName
    int age
}
```

Your class will be locked down to have a single public constructor taking a "fully set" initializer. It will also have a factory method to create the initializer. These are used as follows:

```
@CompileStatic
def firstLastAge() {
    assert new Person(Person.createInitializer().firstName("John").lastName("Smith"
).age(21)).toString() == 'Person(John, Smith, 21)'
}
firstLastAge()
```

Any attempt to use the initializer which doesn't involve setting all the properties (though order is not important) will result in a compilation error. If you don't need this level of strictness, you don't need to use `@CompileStatic`.

You can use the `InitializerStrategy` in conjunction with `@Canonical` and `@Immutable`. If your `@Builder` annotation doesn't have explicit `includes` or `excludes` annotation attributes but your `@Canonical` annotation does, the ones from `@Canonical` will be re-used for `@Builder`. Here is an example using

`@Builder` with `@Immutable`:

```
import groovy.transform.builder.*
import groovy.transform.*
import static groovy.transform.options.Visibility.PRIVATE

@Builder(builderStrategy=InitializerStrategy)
@Immutable
@VisibilityOptions(PRIVATE)
class Person {
    String first
    String last
    int born
}

def publicCons = Person.constructors
assert publicCons.size() == 1

@CompileStatic
def createFirstLastBorn() {
  def p = new Person(Person.createInitializer().first('Johnny').last('Depp').born(
1963))
  assert "$p.first $p.last $p.born" == 'Johnny Depp 1963'
}

createFirstLastBorn()
```

The annotation attribute `useSetters` can be used if you have a setter which you want called as part of the construction process. See the JavaDoc for details.

This strategy also supports annotating static methods and constructors. In this case, the static method or constructor parameters become the properties to use for building purposes and in the case of static methods, the return type of the method becomes the target class being built. If you have more than one `@Builder` annotation used within a class (at either the class, method or constructor positions) then it is up to you to ensure that the generated helper classes and factory methods have unique names (i.e. no more than one can use the default name values). For an example of method and constructor usage but using the `DefaultStrategy` strategy, consult that strategy's documentation.

The annotation attribute `forClass` is not supported for this strategy.

`@groovy.transform.AutoImplement`

The `@AutoImplement` AST transformation supplies dummy implementations for any found abstract methods from superclasses or interfaces. The dummy implementation is the same for all abstract methods found and can be:

- essentially empty (exactly true for void methods and for methods with a return type, returns the default value for that type)

- a statement that throws a specified exception (with optional message)
- some user supplied code

The first example illustrates the default case. Our class is annotated with `@AutoImplement`, has a superclass and a single interface as can be seen here:

```
import groovy.transform.AutoImplement

@AutoImplement
class MyNames extends AbstractList<String> implements Closeable { }
```

A `void close()` method from the `Closeable` interface is supplied and left empty. Implementations are also supplied for the three abstract methods from the super class. The `get`, `addAll` and `size` methods have return types of `String`, `boolean` and `int` respectively with default values `null`, `false` and `0`. We can use our class (and check the expected return type for one of the methods) using the following code:

```
assert new MyNames().size() == 0
```

It is also worthwhile examining the equivalent generated code:

```
class MyNames implements Closeable extends AbstractList<String> {

    String get(int param0) {
        return null
    }

    boolean addAll(Collection<? extends String> param0) {
        return false
    }

    void close() throws Exception {
    }

    int size() {
        return 0
    }

}
```

The second example illustrates the simplest exception case. Our class is annotated with `@AutoImplement`, has a superclass and an annotation attribute indicates that an `IOException` should be thrown if any of our "dummy" methods are called. Here is the class definition:

```
@AutoImplement(exception=IOException)
```

```
class MyWriter extends Writer { }
```

We can use the class (and check the expected exception is thrown for one of the methods) using the following code:

```
shouldFail(IOException) {
  new MyWriter().flush()
}
```

It is also worthwhile examining the equivalent generated code where three void methods have been provided all of which throw the supplied exception:

```
class MyWriter extends Writer {

    void flush() throws IOException {
        throw new IOException()
    }

    void write(char[] param0, int param1, int param2) throws IOException {
        throw new IOException()
    }

    void close() throws Exception {
        throw new IOException()
    }

}
```

The third example illustrates the exception case with a supplied message. Our class is annotated with `@AutoImplement`, implements an interface, and has annotation attributes to indicate that an `UnsupportedOperationException` with `Not supported by MyIterator` as the message should be thrown for any supplied methods. Here is the class definition:

```
@AutoImplement(exception=UnsupportedOperationException, message='Not supported by
MyIterator')
class MyIterator implements Iterator<String> { }
```

We can use the class (and check the expected exception is thrown and has the correct message for one of the methods) using the following code:

```
def ex = shouldFail(UnsupportedOperationException) {
     new MyIterator().hasNext()
}
assert ex.message == 'Not supported by MyIterator'
```

It is also worthwhile examining the equivalent generated code where three void methods have been provided all of which throw the supplied exception:

```
class MyIterator implements Iterator<String> {

    boolean hasNext() {
        throw new UnsupportedOperationException('Not supported by MyIterator')
    }

    String next() {
        throw new UnsupportedOperationException('Not supported by MyIterator')
    }

}
```

The fourth example illustrates the case of user supplied code. Our class is annotated with `@AutoImplement`, implements an interface, has an explicitly overridden `hasNext` method, and has an annotation attribute containing the supplied code for any supplied methods. Here is the class definition:

```
@AutoImplement(code = { throw new UnsupportedOperationException('Should never be
called but was called on ' + new Date()) })
class EmptyIterator implements Iterator<String> {
    boolean hasNext() { false }
}
```

We can use the class (and check the expected exception is thrown and has a message of the expected form) using the following code:

```
def ex = shouldFail(UnsupportedOperationException) {
     new EmptyIterator().next()
}
assert ex.message.startsWith('Should never be called but was called on ')
```

It is also worthwhile examining the equivalent generated code where the `next` method has been supplied:

```
class EmptyIterator implements java.util.Iterator<String> {

    boolean hasNext() {
        false
    }

    String next() {
        throw new UnsupportedOperationException('Should never be called but was called
on ' + new Date())
```

```
    }

}
```

`@groovy.transform.NullCheck`

The `@NullCheck` AST transformation adds null-check guard statements to constructors and methods which cause those methods to fail early when supplied with null arguments. It can be seen as a form of defensive programming. The annotation can be added to individual methods or constructors, or to the class in which case it will apply to all methods/constructors.

```
@NullCheck
String longerOf(String first, String second) {
    first.size() >= second.size() ? first : second
}

assert longerOf('cat', 'canary') == 'canary'
def ex = shouldFail(IllegalArgumentException) {
    longerOf('cat', null)
}
assert ex.message == 'second cannot be null'
```

Class design annotations

This category of annotations are aimed at simplifying the implementation of well-known design patterns (delegation, singleton, ...) by using a declarative style.

`@groovy.transform.BaseScript`

`@BaseScript` is used within scripts to indicate that the script should extend from a custom script base class rather than `groovy.lang.Script`. See the documentation for domain specific languages for further details.

`@groovy.lang.Delegate`

The `@Delegate` AST transformation aims at implementing the delegation design pattern. In the following class:

```
class Event {
    @Delegate Date when
    String title
}
```

The `when` property is annotated with `@Delegate`, meaning that the `Event` class will delegate calls to `Date` methods to the `when` property. In this case, the generated code looks like this:

```
class Event {
    Date when
```

```
    String title
    boolean before(Date other) {
        when.before(other)
    }
    // ...
}
```

Then you can call the `before` method, for example, directly on the `Event` class:

```
def ev = new Event(title:'Groovy keynote', when: Date.parse('yyyy/MM/dd',
'2013/09/10'))
def now = new Date()
assert ev.before(now)
```

Instead of annotating a property (or field), you can also annotate a method. In this case, the method can be thought of as a getter or factory method for the delegate. As an example, here is a class which (rather unusually) has a pool of delegates which are accessed in a round-robin fashion:

```
class Test {
    private int robinCount = 0
    private List<List> items = [[0], [1], [2]]

    @Delegate
    List getRoundRobinList() {
        items[robinCount++ % items.size()]
    }

    void checkItems(List<List> testValue) {
        assert items == testValue
    }
}
```

Here is an example usage of that class:

```
def t = new Test()
t << 'fee'
t << 'fi'
t << 'fo'
t << 'fum'
t.checkItems([[0, 'fee', 'fum'], [1, 'fi'], [2, 'fo']])
```

Using a standard list in this round-robin fashion would violate many expected properties of lists, so don't expect the above class to do anything useful beyond this trivial example.

The behavior of the `@Delegate` AST transformation can be changed using the following parameters:

Attribute	Default value	Description	Example

interfaces	True	Should the interfaces implemented by the field be implemented by the class too	`interface Greeter { void sayHello() }`
			`class MyGreeter implements Greeter { void sayHello() { println 'Hello!'} }`
			
			`class DelegatingGreeter { // no explicit interface`
			`    @Delegate MyGreeter greeter = new MyGreeter()`
			`}`
			`def greeter = new DelegatingGreeter()`
			`assert greeter instanceof Greeter // interface was added transparently`

deprecated	false	If true, also delegates methods annotated with @Deprecated	`class WithDeprecation {`
			`    @Deprecated`
			`    void foo() {}`
			`}`
			`class WithoutDeprecation {`
			`    @Deprecated`
			`    void bar() {}`
			`}`
			`class Delegating {`
			`    @Delegate(deprecated=true) WithDeprecation with = new WithDeprecation()`
			`    @Delegate WithoutDeprecation without = new WithoutDeprecation()`
			`}`
			`def d = new Delegating()`
			`d.foo() // passes thanks to deprecated=true`
			`d.bar() // fails because of @Deprecated`

Attribute	Default value	Description	Example

methodAnnotations	False	Whether to carry over annotations from the methods of the delegate to your delegating method.	<pre>class WithAnnotations {
			    @Transactional
			    void method() {
			    }
			}
			class
			DelegatingWithoutAnnotations {
			    @Delegate WithAnnotations
			delegate
			}
			class DelegatingWithAnnotations {
			    @Delegate(methodAnnotations =
			true) WithAnnotations delegate
			}
			def d1 = new
			DelegatingWithoutAnnotations()
			def d2 = new
			DelegatingWithAnnotations()
			assert d1.class.
			getDeclaredMethod('method').annot
			ations.length==1
			assert d2.class.
			getDeclaredMethod('method').annot
			ations.length==2</pre>

Attribute	Default value	Description	Example
parameterAnnotations	False	Whether to carry over annotations from the method parameters of the delegate to your delegating method.	`class WithAnnotations { void method(@NotNull String str) { } } class DelegatingWithoutAnnotations { @Delegate WithAnnotations delegate } class DelegatingWithAnnotations { @Delegate(parameterAnnotations = true) WithAnnotations delegate } def d1 = new DelegatingWithoutAnnotations() def d2 = new DelegatingWithAnnotations() assert d1.class.getDeclaredMethod('method',String).parameterAnnotations[0].length==0 assert d2.class.getDeclaredMethod('method',String).parameterAnnotations[0].length==1`
excludes	Empty array	A list of methods to be excluded from delegation. For more fine-grained control, see also `excludeTypes`.	`class Worker { void task1() {} void task2() {} } class Delegating { @Delegate(excludes=['task2']) Worker worker = new Worker() } def d = new Delegating() d.task1() // passes d.task2() // fails because method is excluded`

Attribute	Default value	Description	Example
includes	Undefined marker array (indicates all methods)	A list of methods to be included in delegation. For more fine-grained control, see also `includeTypes`.	`class Worker { void task1() {} void task2() {} } class Delegating { @Delegate(includes=['task1']) Worker worker = new Worker() } def d = new Delegating() d.task1() // passes d.task2() // fails because method is not included`
excludeTypes	Empty array	A list of interfaces containing method signatures to be excluded from delegation	`interface AppendStringSelector { StringBuilder append(String str) } class UpperStringBuilder { @Delegate(excludeTypes =AppendStringSelector) StringBuilder sb1 = new StringBuilder() @Delegate(includeTypes =AppendStringSelector) StringBuilder sb2 = new StringBuilder() String toString() { sb1 .toString() + sb2.toString ().toUpperCase() } } def usb = new UpperStringBuilder() usb.append(3.5d) usb.append('hello') usb.append(true) assert usb.toString() == '3.5trueHELLO'`

Attribute	Default value	Description	Example
includeTypes	Undefined marker array (indicates no list be default)	A list of interfaces containing method signatures to be included in delegation	`interface AppendBooleanSelector { StringBuilder append(boolean b) } interface AppendFloatSelector { StringBuilder append(float b) } class NumberBooleanBuilder { @Delegate(includeTypes =AppendBooleanSelector, interfaces=false) StringBuilder nums = new StringBuilder() @Delegate(includeTypes =[AppendFloatSelector], interfaces=false) StringBuilder bools = new StringBuilder() String result() { "${nums.toString()} ~ ${bools.toString()}" } } def b = new NumberBooleanBuilder() b.append(true) b.append(3.14f) b.append(false) b.append(0.0f) assert b.result() == "truefalse ~ 3.140.0" b.append(3.5d) // would fail because we didn't include append(double)`
allNames	False	Should the delegate pattern be also applied to methods with internal names	`class Worker { void task$() {} } class Delegating { @Delegate(allNames=true) Worker worker = new Worker() } def d = new Delegating() d.task$() //passes`

`@groovy.transform.Immutable`

The `@Immutable` meta-annotation combines the following annotations:

- @ToString
- @EqualsAndHashCode
- @TupleConstructor
- @MapConstructor
- @Final
- @ImmutableBase
- @ImmutableOptions
- @PropertyOptions
- @KnownImmutable

The `@Immutable` meta-annotation simplifies the creation of immutable classes. Immutable classes are useful since they are often easier to reason about and are inherently thread-safe. See Effective Java, Minimize Mutability for all the details about how to achieve immutable classes in Java. The `@Immutable` meta-annotation does most of the things described in *Effective Java* for you automatically. To use the meta-annotation, all you have to do is annotate the class like in the following example:

```
import groovy.transform.Immutable

@Immutable
class Point {
    int x
    int y
}
```

One of the requirements for immutable classes is that there is no way to modify any state information within the class. One requirement to achieve this is to use immutable classes for each property or alternatively perform special coding such as defensive copy in and defensive copy out for any mutable properties within the constructors and property getters. Between `@ImmutableBase`, `@MapConstructor` and `@TupleConstructor` properties are either identified as immutable or the special coding for numerous known cases is handled automatically. Various mechanisms are provided for you to extend the handled property types which are allowed. See `@ImmutableOptions` and `@KnownImmutable` for details.

The results of applying `@Immutable` to a class are pretty similar to those of applying the @Canonical meta-annotation but the generated class will have extra logic to handle immutability. You will observe this by, for instance, trying to modify a property which will result in a `ReadOnlyPropertyException` being thrown since the backing field for the property will have been automatically made final.

The `@Immutable` meta-annotation supports the configuration options found in the annotations it aggregates. See those annotations for more details.

`@groovy.transform.ImmutableBase`

Immutable classes generated with `@ImmutableBase` are automatically made final. Also, the type of each property is checked and various checks are made on the class, for example, public instance fields currently aren't allowed. It also generates a `copyWith` constructor if desired.

The following annotation attribute is supported:

Attribute	Default value	Description	Example

copyWith	false	A boolean whether to generate a `copyWith( Map )` method.	`import groovy.transform.Immutable`
			
			`@Immutable( copyWith=true )`
			`class User {`
			`  String  name`
			`  Integer age`
			`}`
			
			`def bob   = new User( 'bob', 43 )`
			`def alice = bob.copyWith(`
			`name:'alice' )`
			`assert alice.name == 'alice'`
			`assert alice.age  == 43`

`@groovy.transform.PropertyOptions`

This annotation allows you to specify a custom property handler to be used by transformations during class construction. It is ignored by the main Groovy compiler but is referenced by other transformations like `@TupleConstructor`, `@MapConstructor`, and `@ImmutableBase`. It is frequently used behind the scenes by the `@Immutable` meta-annotation.

`@groovy.transform.VisibilityOptions`

This annotation allows you to specify a custom visibility for a construct generated by another transformation. It is ignored by the main Groovy compiler but is referenced by other transformations like `@TupleConstructor`, `@MapConstructor`, and `@NamedVariant`.

`@groovy.transform.ImmutableOptions`

Groovy's immutability support relies on a predefined list of known immutable classes (like `java.net.URI` or `java.lang.String` and fails if you use a type which is not in that list, you are allowed to add to the list of known immutable types thanks to the following annotation attributes of the `@ImmutableOptions` annotation:

Attribute	Default value	Description	Example
knownImmutableClasses	Empty list	A list of classes which are deemed immutable.	`import groovy.transform.Immutable import groovy.transform.TupleConstructor @TupleConstructor final class Point { final int x final int y public String toString() { "($x,$y)" } } @Immutable(knownImmutableClasses=[Point]) class Triangle { Point a,b,c }`
knownImmutables	Empty list	A list of property names which are deemed immutable.	`import groovy.transform.Immutable import groovy.transform.TupleConstructor @TupleConstructor final class Point { final int x final int y public String toString() { "($x,$y)" } } @Immutable(knownImmutables=['a','b','c']) class Triangle { Point a,b,c }`

If you deem a type as immutable and it isn't one of the ones automatically handled, then it is up to you to correctly code that class to ensure immutability.

`@groovy.transform.KnownImmutable`

The `@KnownImmutable` annotation isn't actually one that triggers any AST transformations. It is simply a marker annotation. You can annotate your classes with the annotation (including Java classes) and they will be recognized as acceptable types for members within an immutable class. This saves you having to explicitly use the `knownImmutables` or `knownImmutableClasses` annotation attributes from `@ImmutableOptions`.

@groovy.transform.Memoized

The @Memoized AST transformations simplifies the implementation of caching by allowing the result of method calls to be cached just by annotating the method with @Memoized. Let's imagine the following method:

```
long longComputation(int seed) {
    // slow computation
    Thread.sleep(100*seed)
    System.nanoTime()
}
```

This emulates a long computation, based on the actual parameters of the method. Without @Memoized, each method call would take several seconds plus it would return a random result:

```
def x = longComputation(1)
def y = longComputation(1)
assert x!=y
```

Adding @Memoized changes the semantics of the method by adding caching, based on the parameters:

```
@Memoized
long longComputation(int seed) {
    // slow computation
    Thread.sleep(100*seed)
    System.nanoTime()
}

def x = longComputation(1) // returns after 100 milliseconds
def y = longComputation(1) // returns immediately
def z = longComputation(2) // returns after 200 milliseconds
assert x==y
assert x!=z
```

The size of the cache can be configured using two optional parameters:

- *protectedCacheSize*: the number of results which are guaranteed not to be cleared after garbage collection
- *maxCacheSize*: the maximum number of results that can be kept in memory

By default, the size of the cache is unlimited and no cache result is protected from garbage collection. Setting a *protectedCacheSize>0* would create an unlimited cache with some results protected. Setting *maxCacheSize>0* would create a limited cache but without any protection from garbage protection. Setting both would create a limited, protected cache.

@groovy.transform.TailRecursive

The @TailRecursive annotation can be used to automatically transform a recursive call at the end of a method into an equivalent iterative version of the same code. This avoids stack overflow due to too many recursive calls. Below is an example of use when calculating factorial:

```
import groovy.transform.CompileStatic
import groovy.transform.TailRecursive

@CompileStatic
class Factorial {

    @TailRecursive
    static BigInteger factorial( BigInteger i, BigInteger product = 1) {
        if( i == 1) {
            return product
        }
        return factorial(i-1, product*i)
    }
}

assert Factorial.factorial(1) == 1
assert Factorial.factorial(3) == 6
assert Factorial.factorial(5) == 120
assert Factorial.factorial(50000).toString().size() == 213237 // Big number and no
Stack Overflow
```

Currently, the annotation will only work for self-recursive method calls, i.e. a single recursive call to the exact same method again. Consider using Closures and trampoline() if you have a scenario involving simple mutual recursion. Also note that only non-void methods are currently handled (void calls will result in a compilation error).

CAUTION	Currently, some forms of method overloading can trick the compiler, and some non-tail recursive calls are erroneously treated as tail recursive.

@groovy.lang.Singleton

The @Singleton annotation can be used to implement the singleton design pattern on a class. The singleton instance is defined eagerly by default, using class initialization, or lazily, in which case the field is initialized using double-checked locking.

```
@Singleton
class GreetingService {
    String greeting(String name) { "Hello, $name!" }
}
assert GreetingService.instance.greeting('Bob') == 'Hello, Bob!'
```

By default, the singleton is created eagerly when the class is initialized and available through the

`instance` property. It is possible to change the name of the singleton using the `property` parameter:

```
@Singleton(property='theOne')
class GreetingService {
    String greeting(String name) { "Hello, $name!" }
}

assert GreetingService.theOne.greeting('Bob') == 'Hello, Bob!'
```

And it is also possible to make initialization lazy using the `lazy` parameter:

```
class Collaborator {
    public static boolean init = false
}
@Singleton(lazy=true,strict=false)
class GreetingService {
    static void init() {}
    GreetingService() {
        Collaborator.init = true
    }
    String greeting(String name) { "Hello, $name!" }
}
GreetingService.init() // make sure class is initialized
assert Collaborator.init == false
GreetingService.instance
assert Collaborator.init == true
assert GreetingService.instance.greeting('Bob') == 'Hello, Bob!'
```

In this example, we also set the `strict` parameter to false, which allows us to define our own constructor.

`@groovy.lang.Mixin`

Deprecated. Consider using traits instead.

Logging improvements

Groovy provides a family of AST transformations that help with integration of the most widely used logging frameworks. There is a transform and associated annotation for each of the common frameworks. These transforms provide a streamlined declarative approach to using the logging framework. In each case, the transform will:

- add a static final `log` field to the annotated class corresponding to the logger
- wrap all calls to `log.level()` into the appropriate `log.isLevelEnabled` guard, depending on the underlying framework

Those transformations support two parameters:

- `value` (default `log`) corresponds to the name of the logger field

- `category` (defaults to the class name) is the name of the logger category

It's worth noting that annotating a class with one of those annotations doesn't prevent you from using the logging framework using the normal long-hand approach.

`@groovy.util.logging.Log`

The first logging AST transformation available is the `@Log` annotation which relies on the JDK logging framework. Writing:

```
@groovy.util.logging.Log
class Greeter {
    void greet() {
        log.info 'Called greeter'
        println 'Hello, world!'
    }
}
```

is equivalent to writing:

```
import java.util.logging.Level
import java.util.logging.Logger

class Greeter {
    private static final Logger log = Logger.getLogger(Greeter.name)
    void greet() {
        if (log.isLoggable(Level.INFO)) {
            log.info 'Called greeter'
        }
        println 'Hello, world!'
    }
}
```

`@groovy.util.logging.Commons`

Groovy supports the Apache Commons Logging framework using the `@Commons` annotation. Writing:

```
@groovy.util.logging.Commons
class Greeter {
    void greet() {
        log.debug 'Called greeter'
        println 'Hello, world!'
    }
}
```

is equivalent to writing:

```
import org.apache.commons.logging.LogFactory
import org.apache.commons.logging.Log

class Greeter {
    private static final Log log = LogFactory.getLog(Greeter)
    void greet() {
        if (log.isDebugEnabled()) {
            log.debug 'Called greeter'
        }
        println 'Hello, world!'
    }
}
```

You still need to add the appropriate commons-logging jar to your classpath.

`@groovy.util.logging.Log4j`

Groovy supports the Apache Log4j 1.x framework using the `@Log4j` annotation. Writing:

```
@groovy.util.logging.Log4j
class Greeter {
    void greet() {
        log.debug 'Called greeter'
        println 'Hello, world!'
    }
}
```

is equivalent to writing:

```
import org.apache.log4j.Logger

class Greeter {
    private static final Logger log = Logger.getLogger(Greeter)
    void greet() {
        if (log.isDebugEnabled()) {
            log.debug 'Called greeter'
        }
        println 'Hello, world!'
    }
}
```

You still need to add the appropriate log4j jar to your classpath. This annotation can also be used with the compatible reload4j log4j drop-in replacement, just use the jar from that project instead of a log4j jar.

`@groovy.util.logging.Log4j2`

Groovy supports the Apache Log4j 2.x framework using the `@Log4j2` annotation. Writing:

```
@groovy.util.logging.Log4j2
class Greeter {
    void greet() {
        log.debug 'Called greeter'
        println 'Hello, world!'
    }
}
```

is equivalent to writing:

```
import org.apache.logging.log4j.LogManager
import org.apache.logging.log4j.Logger

class Greeter {
    private static final Logger log = LogManager.getLogger(Greeter)
    void greet() {
        if (log.isDebugEnabled()) {
            log.debug 'Called greeter'
        }
        println 'Hello, world!'
    }
}
```

You still need to add the appropriate log4j2 jar to your classpath.

`@groovy.util.logging.Slf4j`

Groovy supports the Simple Logging Facade for Java (SLF4J) framework using the `@Slf4j` annotation. Writing:

```
@groovy.util.logging.Slf4j
class Greeter {
    void greet() {
        log.debug 'Called greeter'
        println 'Hello, world!'
    }
}
```

is equivalent to writing:

```
import org.slf4j.LoggerFactory
import org.slf4j.Logger

class Greeter {
    private static final Logger log = LoggerFactory.getLogger(Greeter)
    void greet() {
        if (log.isDebugEnabled()) {
```

```
            log.debug 'Called greeter'
        }
        println 'Hello, world!'
    }
}
```

You still need to add the appropriate slf4j jar(s) to your classpath.

`@groovy.util.logging.PlatformLog`

Groovy supports the Java Platform Logging API and Service framework using the `@PlatformLog` annotation. Writing:

```
@groovy.util.logging.PlatformLog
class Greeter {
    void greet() {
        log.info 'Called greeter'
        println 'Hello, world!'
    }
}
```

is equivalent to writing:

```
import java.lang.System.Logger
import java.lang.System.LoggerFinder
import static java.lang.System.Logger.Level.INFO

class Greeter {
    private static final transient Logger log =
        LoggerFinder.loggerFinder.getLogger(Greeter.class.name, Greeter.class.module)
    void greet() {
        log.log INFO, 'Called greeter'
        println 'Hello, world!'
    }
}
```

You need to be using JDK 9+ to use this capability.

Declarative concurrency

The Groovy language provides a set of annotations aimed at simplifying common concurrency patterns in a declarative approach.

`@groovy.transform.Synchronized`

The `@Synchronized` AST transformations works in a similar way to the `synchronized` keyword but locks on different objects for safer concurrency. It can be applied on any method or static method:

```
import groovy.transform.Synchronized

import java.util.concurrent.Executors
import java.util.concurrent.TimeUnit

class Counter {
    int cpt
    @Synchronized
    int incrementAndGet() {
        cpt++
    }
    int get() {
        cpt
    }
}
```

Writing this is equivalent to creating a lock object and wrapping the whole method into a synchronized block:

```
class Counter {
    int cpt
    private final Object $lock = new Object()

    int incrementAndGet() {
        synchronized($lock) {
            cpt++
        }
    }
    int get() {
        cpt
    }

}
```

By default, `@Synchronized` creates a field named `$lock` (or `$LOCK` for a static method) but you can make it use any field you want by specifying the value attribute, like in the following example:

```
import groovy.transform.Synchronized

import java.util.concurrent.Executors
import java.util.concurrent.TimeUnit

class Counter {
    int cpt
    private final Object myLock = new Object()

    @Synchronized('myLock')
    int incrementAndGet() {
```

```
        cpt++
    }
    int get() {
        cpt
    }
}
```

`@groovy.transform.WithReadLock` and `@groovy.transform.WithWriteLock`

The `@WithReadLock` AST transformation works in conjunction with the `@WithWriteLock` transformation to provide read/write synchronization using the `ReentrantReadWriteLock` facility that the JDK provides. The annotation can be added to a method or a static method. It will transparently create a `$reentrantLock` final field (or `$REENTRANTLOCK` for a static method) and proper synchronization code will be added. For example, the following code:

```
import groovy.transform.WithReadLock
import groovy.transform.WithWriteLock

class Counters {
    public final Map<String,Integer> map = [:].withDefault { 0 }

    @WithReadLock
    int get(String id) {
        map.get(id)
    }

    @WithWriteLock
    void add(String id, int num) {
        Thread.sleep(200) // emulate long computation
        map.put(id, map.get(id)+num)
    }
}
```

is equivalent to this:

```
import groovy.transform.WithReadLock as WithReadLock
import groovy.transform.WithWriteLock as WithWriteLock

public class Counters {

    private final Map<String, Integer> map
    private final java.util.concurrent.locks.ReentrantReadWriteLock $reentrantlock

    public int get(java.lang.String id) {
        $reentrantlock.readLock().lock()
        try {
            map.get(id)
        }
```

```
        finally {
            $reentrantlock.readLock().unlock()
        }
    }

    public void add(java.lang.String id, int num) {
        $reentrantlock.writeLock().lock()
        try {
            java.lang.Thread.sleep(200)
            map.put(id, map.get(id) + num )
        }
        finally {
            $reentrantlock.writeLock().unlock()
        }
    }
}
```

Both `@WithReadLock` and `@WithWriteLock` support specifying an alternative lock object. In that case, the referenced field must be declared by the user, like in the following alternative:

```
import groovy.transform.WithReadLock
import groovy.transform.WithWriteLock

import java.util.concurrent.locks.ReentrantReadWriteLock

class Counters {
    public final Map<String,Integer> map = [:].withDefault { 0 }
    private final ReentrantReadWriteLock customLock = new ReentrantReadWriteLock()

    @WithReadLock('customLock')
    int get(String id) {
        map.get(id)
    }

    @WithWriteLock('customLock')
    void add(String id, int num) {
        Thread.sleep(200) // emulate long computation
        map.put(id, map.get(id)+num)
    }
}
```

For details

- See Javadoc for groovy.transform.WithReadLock
- See Javadoc for groovy.transform.WithWriteLock

Easier cloning and externalizing

Groovy provides two annotations aimed at facilitating the implementation of `Cloneable` and

`Externalizable` interfaces, respectively named `@AutoClone` and `@AutoExternalize`.

`@groovy.transform.AutoClone`

The `@AutoClone` annotation is aimed at implementing the `@java.lang.Cloneable` interface using various strategies, thanks to the `style` parameter:

- the default `AutoCloneStyle.CLONE` strategy calls `super.clone()` first then `clone()` on each cloneable property
- the `AutoCloneStyle.SIMPLE` strategy uses a regular constructor call and copies properties from the source to the clone
- the `AutoCloneStyle.COPY_CONSTRUCTOR` strategy creates and uses a copy constructor
- the `AutoCloneStyle.SERIALIZATION` strategy uses serialization (or externalization) to clone the object

Each of those strategies have pros and cons which are discussed in the Javadoc for groovy.transform.AutoClone and groovy.transform.AutoCloneStyle .

For example, the following example:

```
import groovy.transform.AutoClone

@AutoClone
class Book {
    String isbn
    String title
    List<String> authors
    Date publicationDate
}
```

is equivalent to this:

```
class Book implements Cloneable {
    String isbn
    String title
    List<String> authors
    Date publicationDate

    public Book clone() throws CloneNotSupportedException {
        Book result = super.clone()
        result.authors = authors instanceof Cloneable ? (List) authors.clone() :
authors
        result.publicationDate = publicationDate.clone()
        result
    }
}
```

Note that the String properties aren't explicitly handled because Strings are immutable and the `clone()` method from `Object` will copy the String references. The same would apply to primitive fields and most of the concrete subclasses of `java.lang.Number`.

In addition to cloning styles, `@AutoClone` supports multiple options:

Attribute	Default value	Description	Example

excludes	Empty list	A list of property or field names that need to be excluded from cloning. A string consisting of a comma-separated field/property names is also allowed. See groovy.transform.AutoClone#excludes for details	`import groovy.transform.AutoClone`
			`import groovy.transform.AutoCloneStyle`
			
			`@AutoClone(style=AutoCloneStyle.SIMPLE,excludes='authors')`
			`class Book {`
			`  String isbn`
			`  String title`
			`  List authors`
			`  Date publicationDate`
			`}`

includeFields	false	By default, only properties are cloned. Setting this flag to true will also clone fields.	`import groovy.transform.AutoClone`
			`import groovy.transform.AutoCloneStyle`
			
			`@AutoClone(style=AutoCloneStyle.SIMPLE,includeFields=true)`
			`class Book {`
			`  String isbn`
			`  String title`
			`  List authors`
			`  protected Date publicationDate`
			`}`

`@groovy.transform.AutoExternalize`

The `@AutoExternalize` AST transformation will assist in the creation of `java.io.Externalizable` classes. It will automatically add the interface to the class and generate the `writeExternal` and `readExternal` methods. For example, this code:

```
import groovy.transform.AutoExternalize

@AutoExternalize
class Book {
    String isbn
    String title
```

```
    float price
}
```

will be converted into:

```
class Book implements java.io.Externalizable {
    String isbn
    String title
    float price

    void writeExternal(ObjectOutput out) throws IOException {
        out.writeObject(isbn)
        out.writeObject(title)
        out.writeFloat( price )
    }

    public void readExternal(ObjectInput oin) {
        isbn = (String) oin.readObject()
        title = (String) oin.readObject()
        price = oin.readFloat()
    }

}
```

The `@AutoExternalize` annotation supports two parameters which will let you slightly customize its behavior:

Attribute	Default value	Description	Example

excludes	Empty list	A list of property or field names that need to be excluded from externalizing. A string consisting of a comma-separated field/property names is also allowed. See groovy.transform.AutoExternalize#excludes for details	`import groovy.transform.AutoExternalize`
			
			`@AutoExternalize(excludes='price')`
			`class Book {`
			`  String isbn`
			`  String title`
			`  float price`
			`}`

Attribute	Default value	Description	Example

includeFields	false	By default, only properties are externalized. Setting this flag to true will also clone fields.	`import groovy.transform.AutoExternalize`
			
			`@AutoExternalize(includeFields=true)`
			`class Book {`
			`  String isbn`
			`  String title`
			`  protected float price`
			`}`

Safer scripting

The Groovy language makes it easy to execute user scripts at runtime (for example using groovy.lang.GroovyShell), but how do you make sure that a script won't eat all CPU (infinite loops) or that concurrent scripts won't slowly consume all available threads of a thread pool? Groovy provides several annotations which are aimed towards safer scripting, generating code which will for example allow you to interrupt execution automatically.

`@groovy.transform.ThreadInterrupt`

One complicated situation in the JVM world is when a thread can't be stopped. The `Thread#stop` method exists but is deprecated (and isn't reliable) so your only chance lies in `Thread#interrupt`. Calling the latter will set the `interrupt` flag on the thread, but it will **not** stop the execution of the thread. This is problematic because it's the responsibility of the code executing in the thread to check the interrupt flag and properly exit. This makes sense when you, as a developer, know that the code you are executing is meant to be run in an independent thread, but in general, you don't know it. It's even worse with user scripts, who might not even know what a thread is (think of DSLs).

`@ThreadInterrupt` simplifies this by adding thread interruption checks at critical places in the code:

- loops (for, while)
- first instruction of a method
- first instruction of a closure body

Let's imagine the following user script:

```
while (true) {
    i++
}
```

This is an obvious infinite loop. If this code executes in its own thread, interrupting wouldn't help: if you `join` on the thread, then the calling code would be able to continue, but the thread would still

be alive, running in background without any ability for you to stop it, slowly causing thread starvation.

One possibility to work around this is to set up your shell this way:

```
def config = new CompilerConfiguration()
config.addCompilationCustomizers(
        new ASTTransformationCustomizer(ThreadInterrupt)
)
def binding = new Binding(i:0)
def shell = new GroovyShell(binding,config)
```

The shell is then configured to automatically apply the `@ThreadInterrupt` AST transformations on all scripts. This allows you to execute user scripts this way:

```
def t = Thread.start {
    shell.evaluate(userCode)
}
t.join(1000) // give at most 1000ms for the script to complete
if (t.alive) {
    t.interrupt()
}
```

The transformation automatically modified user code like this:

```
while (true) {
    if (Thread.currentThread().interrupted) {
        throw new InterruptedException('The current thread has been interrupted.')
    }
    i++
}
```

The check which is introduced inside the loop guarantees that if the `interrupt` flag is set on the current thread, an exception will be thrown, interrupting the execution of the thread.

`@ThreadInterrupt` supports multiple options that will let you further customize the behavior of the transformation:

Attribute	Default value	Description	Example

thrown	`java.lang.InterruptedException`	Specifies the type of exception which is thrown if the thread is interrupted.	`class BadException extends Exception { BadException(String message) { super(message) } }`
			
			`def config = new CompilerConfiguration() config.addCompilationCustomizers( new ASTTransformationCustomizer(thrown:BadException, ThreadInterrupt) ) def binding = new Binding(i:0) def shell = new GroovyShell(this.class.classLoader,binding,config )`
			
			`def userCode = """ try { while (true) { i++ } } catch (BadException e) { i = -1 } """`
			
			`def t = Thread.start { shell.evaluate(userCode) } t.join(1000) // give at most 1s for the script to complete assert binding.i > 0 if (t.alive) { t.interrupt() } Thread.sleep(500) assert binding.i == -1''`

checkOnMethodStart	true	Should an interruption check be inserted at the beginning of each method body. See groovy.transform.ThreadInterrupt for details.	`@ThreadInterrupt(checkOnMethodStart=false)`

Attribute	Default value	Description	Example

applyToAllClasses	true	Should the transformation be applied on all classes of the same source unit (in the same source file). See groovy.transform.ThreadInterrupt for details.	`@ThreadInterrupt(applyToAllClasses=false)`
			`class A { ... } // interrupt checks added`
			`class B { ... } // no interrupt checks`

applyToAllMembers	true	Should the transformation be applied on all members of class. See groovy.transform.ThreadInterrupt for details.	`class A {`
			`  @ThreadInterrupt(applyToAllMembers=false)`
			`  void method1() { ... } // interrupt checked added`
			`  void method2() { ... } // no interrupt checks`
			`}`

`@groovy.transform.TimedInterrupt`

The `@TimedInterrupt` AST transformation tries to solve a slightly different problem from `@groovy.transform.ThreadInterrupt`: instead of checking the `interrupt` flag of the thread, it will automatically throw an exception if the thread has been running for too long.

NOTE	This annotation does **not** spawn a monitoring thread. Instead, it works in a similar manner as `@ThreadInterrupt` by placing checks at appropriate places in the code. This means that if you have a thread blocked by I/O, it will **not** be interrupted.

Imagine the following user code:

```
def fib(int n) { n<2?n:fib(n-1)+fib(n-2) }

result = fib(600)
```

The implementation of the famous Fibonacci number computation here is far from optimized. If it is called with a high `n` value, it can take minutes to answer. With `@TimedInterrupt`, you can choose how long a script is allowed to run. The following setup code will allow the user script to run for 1 second at max:

```
def config = new CompilerConfiguration()
config.addCompilationCustomizers(
        new ASTTransformationCustomizer(value:1, TimedInterrupt)
)
def binding = new Binding(result:0)
```

```
def shell = new GroovyShell(this.class.classLoader, binding,config)
```

This code is equivalent to annotating a class with `@TimedInterrupt` like this:

```
@TimedInterrupt(value=1, unit=TimeUnit.SECONDS)
class MyClass {
    def fib(int n) {
        n<2?n:fib(n-1)+fib(n-2)
    }
}
```

`@TimedInterrupt` supports multiple options that will let you further customize the behavior of the transformation:

Attribute	Default value	Description	Example

value	Long.MAX_VALUE	Used in combination with `unit` to specify after how long execution times out.	`@TimedInterrupt(value=500L, unit=TimeUnit.MILLISECONDS, applyToAllClasses = false)`
			`class Slow {`
			`  def fib(n) { n<2?n:fib(n-1)+fib(n-2) }`
			`}`
			`def result`
			`def t = Thread.start {`
			`  result = new Slow().fib(500)`
			`}`
			`t.join(5000)`
			`assert result == null`
			`assert !t.alive`

unit	TimeUnit.SECONDS	Used in combination with `value` to specify after how long execution times out.	`@TimedInterrupt(value=500L, unit=TimeUnit.MILLISECONDS, applyToAllClasses = false)`
			`class Slow {`
			`  def fib(n) { n<2?n:fib(n-1)+fib(n-2) }`
			`}`
			`def result`
			`def t = Thread.start {`
			`  result = new Slow().fib(500)`
			`}`
			`t.join(5000)`
			`assert result == null`
			`assert !t.alive`

Attribute	Default value	Description	Example

thrown	`java.util.concurrent.TimeoutException`	Specifies the type of exception which is thrown if timeout is reached.	`@TimedInterrupt(thrown=TooLongException, applyToAllClasses = false, value=1L)`
			`class Slow {`
			`  def fib(n) { Thread.sleep(100); n<2?n:fib(n-1)+fib(n-2) }`
			`}`
			`def result`
			`def t = Thread.start {`
			`  try {`
			`  result = new Slow().fib(50)`
			`  } catch (TooLongException e) {`
			`  result = -1`
			`  }`
			`}`
			`t.join(5000)`
			`assert result == -1`

checkOnMethodStart	true	Should an interruption check be inserted at the beginning of each method body. See groovy.transform.TimedInterrupt for details.	`@TimedInterrupt(checkOnMethodStart=false)`

applyToAllClasses	true	Should the transformation be applied on all classes of the same source unit (in the same source file). See groovy.transform.TimedInterrupt for details.	`@TimedInterrupt(applyToAllClasses=false)`
			`class A { ... } // interrupt checks added`
			`class B { ... } // no interrupt checks`

applyToAllMembers	true	Should the transformation be applied on all members of class. See groovy.transform.TimedInterrupt for details.	`class A {`
			`  @TimedInterrupt(applyToAllMembers=false)`
			`  void method1() { ... } // interrupt checked added`
			`  void method2() { ... } // no interrupt checks`
			`}`

WARNING `@TimedInterrupt` is currently not compatible with static methods!

`@groovy.transform.ConditionalInterrupt`

The last annotation for safer scripting is the base annotation when you want to interrupt a script using a custom strategy. In particular, this is the annotation of choice if you want to use resource management (limit the number of calls to an API, ...). In the following example, user code is using an infinite loop, but `@ConditionalInterrupt` will allow us to check a quota manager and interrupt automatically the script:

```
@ConditionalInterrupt({Quotas.disallow('user')})
class UserCode {
    void doSomething() {
        int i=0
        while (true) {
            println "Consuming resources ${++i}"
        }
    }
}
```

The quota checking is very basic here, but it can be any code:

```
class Quotas {
    static def quotas = [:].withDefault { 10 }
    static boolean disallow(String userName) {
        println "Checking quota for $userName"
        (quotas[userName]--)<0
    }
}
```

We can make sure `@ConditionalInterrupt` works properly using this test code:

```
assert Quotas.quotas['user'] == 10
def t = Thread.start {
    new UserCode().doSomething()
}
t.join(5000)
assert !t.alive
assert Quotas.quotas['user'] < 0
```

Of course, in practice, it is unlikely that `@ConditionalInterrupt` will be itself added by hand on user code. It can be injected in a similar manner as the example shown in the ThreadInterrupt section, using the org.codehaus.groovy.control.customizers.ASTTransformationCustomizer :

```
def config = new CompilerConfiguration()
def checkExpression = new ClosureExpression(
```

```
        Parameter.EMPTY_ARRAY,
        new ExpressionStatement(
                new MethodCallExpression(new ClassExpression(ClassHelper.make(
Quotas)), 'disallow', new ConstantExpression('user'))
        )
)
config.addCompilationCustomizers(
        new ASTTransformationCustomizer(value: checkExpression, ConditionalInterrupt)
)

def shell = new GroovyShell(this.class.classLoader,new Binding(),config)

def userCode = """
        int i=0
        while (true) {
            println "Consuming resources \\${++i}"
        }
"""

assert Quotas.quotas['user'] == 10
def t = Thread.start {
    shell.evaluate(userCode)
}
t.join(5000)
assert !t.alive
assert Quotas.quotas['user'] < 0
```

`@ConditionalInterrupt` supports multiple options that will let you further customize the behavior of the transformation:

Attribute	Default value	Description	Example
value		The closure which will be called to check if execution is allowed. If the closure returns false, execution is allowed. If it returns true, then an exception will be thrown.	`@ConditionalInterrupt({ ... })`

Attribute	Default value	Description	Example
thrown	`java.lang.InterruptedException`	Specifies the type of exception which is thrown if execution should be aborted.	`config.addCompilationCustomizers(new ASTTransformationCustomizer(thrown: QuotaExceededException,value: checkExpression, ConditionalInterrupt)) assert Quotas.quotas['user'] == 10 def t = Thread.start { try { shell.evaluate(userCode) } catch (QuotaExceededException) { Quotas.quotas['user'] = 'Quota exceeded' } } t.join(5000) assert !t.alive assert Quotas.quotas['user'] == 'Quota exceeded'`
checkOnMethodStart	true	Should an interruption check be inserted at the beginning of each method body. See groovy.transform.ConditionalInterrupt for details.	`@ConditionalInterrupt(checkOnMethodStart=false)`
applyToAllClasses	true	Should the transformation be applied on all classes of the same source unit (in the same source file). See groovy.transform.ConditionalInterrupt for details.	`@ConditionalInterrupt(applyToAllClasses=false) class A { ... } // interrupt checks added class B { ... } // no interrupt checks`

Attribute	Default value	Description	Example
applyToAllMembers	true	Should the transformation be applied on all members of class. See groovy.transform.ConditionalInterrupt for details.	`class A { @ConditionalInterrupt(applyToAllMembers=false) void method1() { ... } // interrupt checked added void method2() { ... } // no interrupt checks }`

`@groovy.transform.SafeRegex`

Regular expression evaluation is another way for untrusted input to consume unbounded CPU. Java's regex engine has no native timeout, so a pattern subject to *catastrophic backtracking* can effectively hang a thread when fed adversarial input — a Regular Expression Denial of Service (ReDoS). Thread interruption doesn't help here: the regex engine never checks the interrupt flag, so even `@ThreadInterrupt` or `@TimedInterrupt` cannot stop a runaway match — the hang occurs inside a single JDK library call, not in user code where those transforms insert their checks.

`@SafeRegex` addresses this for the regex operators. Within the annotated scope, find (`=~`) and match (`==~`) expressions are rewritten at compile time to deadline-guarded groovy.util.regex.RegexGuard calls which throw a groovy.util.regex.RegexTimeoutException if an evaluation exceeds the configured timeout. Matching semantics are otherwise unchanged:

```
@SafeRegex(millis = 200)
class Handler {
    boolean check(String input) {
        input ==~ /(.*,){16}X/
    }
}
```

The `check` method is rewritten to the equivalent of:

```
class Handler {
    boolean check(String input) {
        groovy.util.regex.RegexGuard.matchRegex(input, /(.*,){16}X/, 200)
    }
}
```

Well-behaved evaluations are unaffected, while a runaway one is cut off shortly after the deadline:

```
def handler = new Handler()
assert !handler.check('1,2,3')          // well-behaved input: unchanged semantics
```

```
try {
    handler.check('1,' * 40)            // adversarial input: catastrophic
backtracking
    assert false, 'should have timed out'
} catch (RegexTimeoutException expected) {
}
```

The annotation may be placed on a class, method, or constructor. It may also be placed on a field or local variable declaration, in which case only regex operators lexically within the initializer expression are guarded (a guarded field initializer runs, as usual, in the constructor, or in the static initializer for a static field, where a timeout surfaces as an `ExceptionInInitializerError` whose cause is the `RegexTimeoutException`).

`@SafeRegex` supports one option:

Attribute	Default value	Description	Example

millis	1000	The timeout in milliseconds applied to each guarded regex evaluation within the annotated scope. Must be a positive constant.	`@SafeRegex(millis = 200)`
			`boolean check(String input) {`
			`  input ==~ /(.*,){16}X/`
			`}`

Only regex operators lexically visible within the annotated scope are rewritten. Regex evaluation via method calls such as `String#matches`, `replaceAll` or `split`, or occurring in code called from the annotated scope, is not guarded by this transform; for those, use groovy.util.regex.RegexGuard explicitly — its `guard(timeout) { … }` form guards all regex evaluations reached through Groovy's runtime for the dynamic extent of a closure. See the operator documentation for details.

Compiler directives

This category of AST transformations groups annotations which have a direct impact on the semantics of the code, rather than focusing on code generation. With that regards, they can be seen as compiler directives that either change the behavior of a program at compile time or runtime.

`@groovy.transform.Field`

The `@Field` annotation only makes sense in the context of a script and aims at solving a common scoping error with scripts. The following example will for example fail at runtime:

```
def x

String line() {
    "="*x
}

x=3
assert "===" == line()
```

```
x=5
assert "=====" == line()
```

The error that is thrown may be difficult to interpret: groovy.lang.MissingPropertyException: No such property: x. The reason is that scripts are compiled to classes and the script body is itself compiled as a single *run()* method. Methods which are defined in the scripts are independent, so the code above is equivalent to this:

```
class MyScript extends Script {

    String line() {
        "="*x
    }

    public def run() {
        def x
        x=3
        assert "===" == line()
        x=5
        assert "=====" == line()
    }
}
```

So `def x` is effectively interpreted as a local variable, outside of the scope of the `line` method. The `@Field` AST transformation aims at fixing this by changing the scope of the variable to a field of the enclosing script:

```
@Field def x

String line() {
    "="*x
}

x=3
assert "===" == line()
x=5
assert "=====" == line()
```

The resulting, equivalent, code is now:

```
class MyScript extends Script {

    def x

    String line() {
        "="*x
    }
```

```
    public def run() {
        x=3
        assert "===" == line()
        x=5
        assert "=====" == line()
    }
}
```

@groovy.transform.PackageScope

By default, Groovy visibility rules imply that if you create a field without specifying a modifier, then the field is interpreted as a property:

```
class Person {
    String name // this is a property
}
```

Should you want to create a package private field instead of a property (private field+getter/setter), then annotate your field with `@PackageScope`:

```
class Person {
    @PackageScope String name // not a property anymore
}
```

The `@PackageScope` annotation can also be used for classes, methods and constructors. In addition, by specifying a list of `PackageScopeTarget` values as the annotation attribute at the class level, all members within that class that don't have an explicit modifier and match the provided `PackageScopeTarget` will remain package protected. For example to apply to fields within a class use the following annotation:

```
import static groovy.transform.PackageScopeTarget.FIELDS
@PackageScope(FIELDS)
class Person {
  String name     // not a property, package protected
  Date dob        // not a property, package protected
  private int age // explicit modifier, so won't be touched
}
```

The `@PackageScope` annotation is seldom used as part of normal Groovy conventions but is sometimes useful for factory methods that should be visible internally within a package or for methods or constructors provided for testing purposes, or when integrating with third-party libraries which require such visibility conventions.

@groovy.transform.Final

@Final is essentially an alias for the final modifier. The intention is that you would almost never use the @Final annotation directly (just use final). However, when creating meta-annotations that should apply the final modifier to the node being annotated, you can mix in @Final, e.g..

```
@AnnotationCollector([Singleton,Final]) @interface MySingleton {}

@MySingleton
class GreetingService {
    String greeting(String name) { "Hello, $name!" }
}
assert GreetingService.instance.greeting('Bob') == 'Hello, Bob!'
assert Modifier.isFinal(GreetingService.modifiers)
```

@groovy.transform.AutoFinal

The @AutoFinal annotation instructs the compiler to automatically insert the final modifier in numerous places within the annotated node. If applied on a method (or constructor), the parameters for that method (or constructor) will be marked as final. If applied on a class definition, the same treatment will occur for all declared methods and constructors within that class.

It is often considered bad practice to reassign parameters of a method or constructor with its body. By adding the final modifier to all parameter declarations you can avoid this practice entirely. Some programmers feel that adding final everywhere increases the amount of boilerplate code and makes the method signatures somewhat noisy. An alternative might instead be to use a code review process or apply a codenarc rule to give warnings if that practice is observed but these alternatives might lead to delayed feedback during quality checking rather than within the IDE or during compilation. The @AutoFinal annotation aims to maximise compiler/IDE feedback while retaining succinct code with minimum boilerplate noise.

The following example illustrates applying the annotation at the class level:

```
import groovy.transform.AutoFinal

@AutoFinal
class Person {
    private String first, last

    Person(String first, String last) {
        this.first = first
        this.last = last
    }

    String fullName(String separator) {
        "$first$separator$last"
    }

    String greeting(String salutation) {
```

```
        "$salutation, $first"
    }
}
```

In this example, the two parameters for the constructor and the single parameter for both the `fullname` and `greeting` methods will be final. Attempts to modify those parameters within the constructor or method bodies will be flagged by the compiler.

The following example illustrates applying the annotation at the method level:

```
class Calc {
    @AutoFinal
    int add(int a, int b) { a + b }

    int mult(int a, int b) { a * b }
}
```

Here, the `add` method will have final parameters but the `mult` method will remain unchanged.

`@groovy.transform.AnnotationCollector`

`@AnnotationCollector` allows the creation of meta-annotations, which are described in a dedicated section.

`@groovy.transform.TypeChecked`

`@TypeChecked` activates compile-time type checking on your Groovy code. See section on type checking for details.

`@groovy.transform.CompileStatic`

`@CompileStatic` activates static compilation on your Groovy code. See section on type checking for details.

`@groovy.transform.CompileDynamic`

`@CompileDynamic` disables static compilation on parts of your Groovy code. See section on type checking for details.

`@groovy.lang.DelegatesTo`

`@DelegatesTo` is not, technically speaking, an AST transformation. It is aimed at documenting code and helping the compiler in case you are using type checking or static compilation. The annotation is described thoroughly in the DSL section of this guide.

`@groovy.transform.SelfType`

`@SelfType` is not an AST transformation but rather a marker interface used with traits. See the traits documentation for further details.

Swing patterns

@groovy.beans.Bindable

`@Bindable` is an AST transformation that transforms a regular property into a bound property (according to the JavaBeans specification). The `@Bindable` annotation can be placed on a property or a class. To convert all properties of a class into bound properties, on can annotate the class like in this example:

```
import groovy.beans.Bindable

@Bindable
class Person {
    String name
    int age
}
```

This is equivalent to writing this:

```
import java.beans.PropertyChangeListener
import java.beans.PropertyChangeSupport

class Person {
    final private PropertyChangeSupport this$propertyChangeSupport

    String name
    int age

    public void addPropertyChangeListener(PropertyChangeListener listener) {
        this$propertyChangeSupport.addPropertyChangeListener(listener)
    }

    public void addPropertyChangeListener(String name, PropertyChangeListener
listener) {
        this$propertyChangeSupport.addPropertyChangeListener(name, listener)
    }

    public void removePropertyChangeListener(PropertyChangeListener listener) {
        this$propertyChangeSupport.removePropertyChangeListener(listener)
    }

    public void removePropertyChangeListener(String name, PropertyChangeListener
listener) {
        this$propertyChangeSupport.removePropertyChangeListener(name, listener)
    }

    public void firePropertyChange(String name, Object oldValue, Object newValue) {
        this$propertyChangeSupport.firePropertyChange(name, oldValue, newValue)
    }
```

```
    public PropertyChangeListener[] getPropertyChangeListeners() {
        return this$propertyChangeSupport.getPropertyChangeListeners()
    }

    public PropertyChangeListener[] getPropertyChangeListeners(String name) {
        return this$propertyChangeSupport.getPropertyChangeListeners(name)
    }
}
```

`@Bindable` therefore removes a lot of boilerplate from your class, dramatically increasing readability. If the annotation is put on a single property, only that property is bound:

```
import groovy.beans.Bindable

class Person {
    String name
    @Bindable int age
}
```

`@groovy.beans.ListenerList`

The `@ListenerList` AST transformation generates code for adding, removing and getting the list of listeners to a class, just by annotating a collection property:

```
import java.awt.event.ActionListener
import groovy.beans.ListenerList

class Component {
    @ListenerList
    List<ActionListener> listeners;
}
```

The transform will generate the appropriate add/remove methods based on the generic type of the list. In addition, it will also create `fireXXX` methods based on the public methods declared on the class:

```
import java.awt.event.ActionEvent
import java.awt.event.ActionListener as ActionListener
import groovy.beans.ListenerList as ListenerList

public class Component {

    @ListenerList
    private List<ActionListener> listeners

    public void addActionListener(ActionListener listener) {
        if ( listener == null) {
```

```
            return
        }
        if ( listeners == null) {
            listeners = []
        }
        listeners.add(listener)
    }

    public void removeActionListener(ActionListener listener) {
        if ( listener == null) {
            return
        }
        if ( listeners == null) {
            listeners = []
        }
        listeners.remove(listener)
    }

    public ActionListener[] getActionListeners() {
        Object __result = []
        if ( listeners != null) {
            __result.addAll(listeners)
        }
        return (( __result ) as ActionListener[])
    }

    public void fireActionPerformed(ActionEvent param0) {
        if ( listeners != null) {
            ArrayList<ActionListener> __list = new ArrayList<ActionListener>(
listeners)
            for (def listener : __list ) {
                listener.actionPerformed(param0)
            }
        }
    }
}
```

`@Bindable` supports multiple options that will let you further customize the behavior of the transformation:

Attribute	Default value	Description	Example
name	Generic type name	By default, the suffix which will be appended to add/remove/... methods is the simple class name of the generic type of the list.	`class Component { @ListenerList(name='item') List<ActionListener> listeners; }`

Attribute	Default value	Description	Example
synchronize	false	If set to true, generated methods will be synchronized	`class Component { @ListenerList(synchronize = true) List<ActionListener> listeners; }`

@groovy.beans.Vetoable

The `@Vetoable` annotation works in a similar manner to `@Bindable` but generates constrained property according to the JavaBeans specification, instead of bound properties. The annotation can be placed on a class, meaning that all properties will be converted to constrained properties, or on a single property. For example, annotating this class with `@Vetoable`:

```
import groovy.beans.Vetoable

import java.beans.PropertyVetoException
import java.beans.VetoableChangeListener

@Vetoable
class Person {
    String name
    int age
}
```

is equivalent to writing this:

```
public class Person {

    private String name
    private int age
    final private java.beans.VetoableChangeSupport this$vetoableChangeSupport

    public void addVetoableChangeListener(VetoableChangeListener listener) {
        this$vetoableChangeSupport.addVetoableChangeListener(listener)
    }

    public void addVetoableChangeListener(String name, VetoableChangeListener
listener) {
        this$vetoableChangeSupport.addVetoableChangeListener(name, listener)
    }

    public void removeVetoableChangeListener(VetoableChangeListener listener) {
        this$vetoableChangeSupport.removeVetoableChangeListener(listener)
    }
```

```
    public void removeVetoableChangeListener(String name, VetoableChangeListener
listener) {
        this$vetoableChangeSupport.removeVetoableChangeListener(name, listener)
    }

    public void fireVetoableChange(String name, Object oldValue, Object newValue)
throws PropertyVetoException {
        this$vetoableChangeSupport.fireVetoableChange(name, oldValue, newValue)
    }

    public VetoableChangeListener[] getVetoableChangeListeners() {
        return this$vetoableChangeSupport.getVetoableChangeListeners()
    }

    public VetoableChangeListener[] getVetoableChangeListeners(String name) {
        return this$vetoableChangeSupport.getVetoableChangeListeners(name)
    }

    public void setName(String value) throws PropertyVetoException {
        this.fireVetoableChange('name', name, value)
        name = value
    }

    public void setAge(int value) throws PropertyVetoException {
        this.fireVetoableChange('age', age, value)
        age = value
    }
}
```

If the annotation is put on a single property, only that property is made vetoable:

```
import groovy.beans.Vetoable

class Person {
    String name
    @Vetoable int age
}
```

Test assistance

`@groovy.test.NotYetImplemented`

`@NotYetImplemented` is used to invert the result of a JUnit 3/4 test case. It is in particular useful if a feature is not yet implemented but the test is. In that case, it is expected that the test fails. Marking it with `@NotYetImplemented` will inverse the result of the test, like in this example:

```
import groovy.test.NotYetImplemented
```

```
class Maths {
    static int fib(int n) {
        // todo: implement later
    }
}

class MathsTest {
    @NotYetImplemented
    void testFib() {
        def dataTable = [
                1:1,
                2:1,
                3:2,
                4:3,
                5:5,
                6:8,
                7:13
        ]
        dataTable.each { i, r ->
            assert Maths.fib(i) == r
        }
    }
}
```

Another advantage of using this technique is that you can write test cases for bugs before knowing how to fix them. If some time in the future, a modification in the code fixes a bug by side effect, you'll be notified because a test which was expected to fail passed.

`@groovy.transform.ASTTest`

`@ASTTest` is a special AST transformation meant to help debugging other AST transformations or the Groovy compiler itself. It will let the developer "explore" the AST during compilation and perform assertions on the AST rather than on the result of compilation. This means that this AST transformations gives access to the AST before the bytecode is produced. `@ASTTest` can be placed on any annotable node and requires two parameters:

- *phase*: sets at which phase at which `@ASTTest` will be triggered. The test code will work on the AST tree at the end of this phase.
- *value*: the code which will be executed once the phase is reached, on the annotated node

> **TIP** Compile phase has to be chosen from one of org.codehaus.groovy.control.CompilePhase . However, since it is not possible to annotate a node twice with the same annotation, you will not be able to use `@ASTTest` on the same node at two distinct compile phases.

`value` is a closure expression which has access to a special variable `node` corresponding to the annotated node, and a helper `lookup` method which will be discussed here. For example, you can annotate a class node like this:

```
import groovy.transform.ASTTest
import org.codehaus.groovy.ast.ClassNode

@ASTTest(phase=CONVERSION, value={   ①
    assert node instanceof ClassNode ②
    assert node.name == 'Person'     ③
})
class Person {
}
```

① we're checking the state of the Abstract Syntax Tree after the CONVERSION phase

② node refers to the AST node which is annotated by @ASTTest

③ it can be used to perform assertions at compile time

One interesting feature of `@ASTTest` is that if an assertion fails, then **compilation will fail**. Now imagine that we want to check the behavior of an AST transformation at compile time. We will take `@PackageScope` here, and we will want to verify that a property annotated with `@PackageScope` becomes a package private field. For this, we have to know at which phase the transform runs, which can be found in org.codehaus.groovy.transform.PackageScopeASTTransformation : semantic analysis. Then a test can be written like this:

```
import groovy.transform.ASTTest
import groovy.transform.PackageScope

@ASTTest(phase=SEMANTIC_ANALYSIS, value={
    def nameNode = node.properties.find { it.name == 'name' }
    def ageNode = node.properties.find { it.name == 'age' }
    assert nameNode
    assert ageNode == null // shouldn't be a property anymore
    def ageField = node.getDeclaredField 'age'
    assert ageField.modifiers == 0
})
class Person {
    String name
    @PackageScope int age
}
```

The `@ASTTest` annotation can only be placed wherever the grammar allows it. Sometimes, you would like to test the contents of an AST node which is not annotable. In this case, `@ASTTest` provides a convenient `lookup` method which will search the AST for nodes which are labelled with a special token:

```
def list = lookup('anchor') ①
Statement stmt = list[0] ②
```

① returns the list of AST nodes which label is 'anchor'

② it is always necessary to choose which element to process since lookup always returns a list

Imagine, for example, that you want to test the declared type of a for loop variable. Then you can do it like this:

```
import groovy.transform.ASTTest
import groovy.transform.PackageScope
import org.codehaus.groovy.ast.ClassHelper
import org.codehaus.groovy.ast.expr.DeclarationExpression
import org.codehaus.groovy.ast.stmt.ForStatement

class Something {
    @ASTTest(phase=SEMANTIC_ANALYSIS, value={
        def forLoop = lookup('anchor')[0]
        assert forLoop instanceof ForStatement
        def decl = forLoop.collectionExpression.expressions[0]
        assert decl instanceof DeclarationExpression
        assert decl.variableExpression.name == 'i'
        assert decl.variableExpression.originType == ClassHelper.int_TYPE
    })
    void someMethod() {
        int x = 1;
        int y = 10;
        anchor: for (int i=0; i<x+y; i++) {
            println "$i"
        }
    }
}
```

`@ASTTest` also exposes those variables inside the test closure:

- `node` corresponds to the annotated node, as usual
- `compilationUnit` gives access to the current `org.codehaus.groovy.control.CompilationUnit`
- `compilePhase` returns the current compile phase (`org.codehaus.groovy.control.CompilePhase`)

The latter is interesting if you don't specify the `phase` attribute. In that case, the closure will be executed after each compile phase after (and including) `SEMANTIC_ANALYSIS`. The context of the transformation is kept after each phase, giving you a chance to check what changed between two phases.

As an example, here is how you could dump the list of AST transformations registered on a class node:

```
import groovy.transform.ASTTest
import groovy.transform.CompileStatic
import groovy.transform.Immutable
import org.codehaus.groovy.ast.ClassNode
import org.codehaus.groovy.control.CompilePhase
```

```
@ASTTest(value={
    System.err.println "Compile phase: $compilePhase"
    ClassNode cn = node
    System.err.println "Global AST xforms:
${compilationUnit?.ASTTransformationsContext?.globalTransformNames}"
    CompilePhase.values().each {
        def transforms = cn.getTransforms(it)
        if (transforms) {
            System.err.println "Ast xforms for phase $it:"
            transforms.each { map ->
                System.err.println(map)
            }
        }
    }
})
@CompileStatic
@Immutable
class Foo {
}
```

And here is how you can memorize variables for testing between two phases:

```
import groovy.transform.ASTTest
import groovy.transform.ToString
import org.codehaus.groovy.ast.ClassNode
import org.codehaus.groovy.control.CompilePhase

@ASTTest(value={
    if (compilePhase == CompilePhase.INSTRUCTION_SELECTION) {           ①
        println "toString() was added at phase: ${added}"
        assert added == CompilePhase.CANONICALIZATION                   ②
    } else {
        if (node.getDeclaredMethods('toString') && added == null) {     ③
            added = compilePhase                                        ④
        }
    }
})
@ToString
class Foo {
    String name
}
```

① if the current compile phase is instruction selection

② then we want to make sure `toString` was added at `CANONICALIZATION`

③ otherwise, if `toString` exists and that the variable from the context, `added` is null

④ then it means that this compile phase is the one where `toString` was added

Grape handling

`@groovy.lang.Grab`

`@groovy.lang.GrabConfig`

`@groovy.lang.GrabExclude`

`@groovy.lang.GrabResolver`

`@groovy.lang.Grapes`

`Grape` is a dependency management engine embedded into Groovy, relying on several annotations which are described thoroughly in this section of the guide.

Developing AST transformations

There are two kinds of transformations: global and local transformations.

- Global transformations are applied by the compiler on the code being compiled, wherever the transformation applies. Compiled classes that implement global transformations are in a JAR added to the classpath of the compiler and contain service locator file `META-INF/services/org.codehaus.groovy.transform.ASTTransformation` with a line with the name of the transformation class. The transformation class must have a no-arg constructor and implement the `org.codehaus.groovy.transform.ASTTransformation` interface. It will be run against **every source in the compilation**, so if you want fast compilation times, don't create transformations which scan all the AST in an expansive and time-consuming manner.
- Local transformations are transformations applied locally by annotating code elements you want to transform. For this, we reuse the annotation notation, and those annotations should implement `org.codehaus.groovy.transform.ASTTransformation`. The compiler will discover them and apply the transformation on these code elements.

Compilation phases guide

Groovy AST transformations must be performed in one of the nine defined compilation phases (org.codehaus.groovy.control.CompilePhase).

Global transformations may be applied in any phase, but local transformations may only be applied in the semantic analysis phase or later. Briefly, the compiler phases are:

- *Initialization*: source files are opened and environment configured
- *Parsing*: the grammar is used to produce tree of tokens representing the source code
- *Conversion*: An abstract syntax tree (AST) is created from token trees.
- *Semantic Analysis*: Performs consistency and validity checks that the grammar can't check for, and resolves classes.
- *Canonicalization*: Complete building the AST
- *Instruction Selection*: Prepare for class generation, for example static type checking is performed in this phase
- *Class Generation*: creates the bytecode of the class in memory

- *Output*: write the binary output to the file system
- *Finalization*: Perform any last cleanup

Generally speaking, there is more type information available later in the phases. If your transformation is concerned with reading the AST, then a later phase where information is more plentiful might be a good choice. If your transformation is concerned with writing AST, then an earlier phase where the tree is more sparse might be more convenient.

Transformations and static compilation

The phase advice above is about convenience. If classes using your transformation may also use `@TypeChecked` or `@CompileStatic` — which for a transformation shipped by a library or framework is out of your hands — the phase you pick additionally decides whether the type checker sees the code you generate at all.

Static type checking runs during instruction selection. Code that your transformation adds in an earlier phase is visited by the type checker exactly like code the user wrote: it is checked, and under static compilation every method call and property access in it is annotated with the resolution metadata the bytecode writers depend on, chiefly `StaticTypesMarker.DIRECT_METHOD_CALL_TARGET` (the resolved method) and `StaticTypesMarker.INFERRED_TYPE` (see org.codehaus.groovy.transform.stc.StaticTypesMarker). Code injected during or after instruction selection is never visited by the type checker. Dynamic Groovy and `@TypeChecked` classes do not care, because their bytecode dispatches dynamically either way — but under `@CompileStatic` the static writers have nothing to link against. The result is code that works dynamically and type-checked yet breaks statically compiled, typically with "Target method for method call expression hasn't been set", a `VerifyError`, or a "Cannot find matching method" error that only appears under `@CompileStatic`. If users of your transformation report errors like these, check what phase the generating code runs in before suspecting their code.

The safe pattern is therefore to generate code before instruction selection — usually semantic analysis for local transformations — and let the type checker annotate it for you. If your transformation must run late, the options are to mark the generated methods with `@CompileDynamic` (or generate calls through the dynamic runtime deliberately), or to attach the required metadata yourself; the latter means hand-resolving targets the way the type checker would, which is expert territory for the same reasons described in Transforming the AST in an extension.

> **TIP** The mirror image of this problem — a type checking **extension** that vouches for dynamic code under `@TypeChecked`, leaving the static compiler nothing to link against — is covered in the type checking extensions documentation: prefer `makeDynamic` (see mixed mode compilation) or resolve a real method target instead of simply setting `handled`.

Local transformations

Local AST transformations are relative to the context they are applied to. In most cases, the context is defined by an annotation that will define the scope of the transform. For example, annotating a field would mean that the transformation *applies to* the field, while annotating the class would mean that the transformation *applies to* the whole class.

As a naive and simple example, consider wanting to write a `@WithLogging` transformation that would add console messages at the start and end of a method invocation. So the following "Hello World" example would actually print "Hello World" along with a start and stop message:

Poor man's aspect oriented programming

```
@WithLogging
def greet() {
    println "Hello World"
}

greet()
```

A local AST transformation is an easy way to do this. It requires two things:

- a definition of the `@WithLogging` annotation
- an implementation of org.codehaus.groovy.transform.ASTTransformation that adds the logging expressions to the method

An `ASTTransformation` is a callback that gives you access to the org.codehaus.groovy.control.SourceUnit, through which you can get a reference to the org.codehaus.groovy.ast.ModuleNode (AST).

The AST (Abstract Syntax Tree) is a tree structure consisting mostly of org.codehaus.groovy.ast.expr.Expression (expressions) or org.codehaus.groovy.ast.expr.Statement (statements). An easy way to learn about the AST is to explore it in a debugger. Once you have the AST, you can analyze it to find out information about the code or rewrite it to add new functionality.

The local transformation annotation is the simple part. Here is the `@WithLogging` one:

```
import org.codehaus.groovy.transform.GroovyASTTransformationClass

import java.lang.annotation.ElementType
import java.lang.annotation.Retention
import java.lang.annotation.RetentionPolicy
import java.lang.annotation.Target

@Retention(RetentionPolicy.SOURCE)
@Target([ElementType.METHOD])
@GroovyASTTransformationClass(["gep.WithLoggingASTTransformation"])
public @interface WithLogging {
}
```

The annotation retention can be `SOURCE` because you won't need the annotation past that. The element type here is `METHOD`, the `@WithLogging` because the annotation applies to methods.

But the most important part is the `@GroovyASTTransformationClass` annotation. This links the `@WithLogging` annotation to the `ASTTransformation` class you will write.

gep.WithLoggingASTTransformation is the fully qualified class name of the ASTTransformation we are going to write. This line wires the annotation to the transformation.

With this in place, the Groovy compiler is going to invoke gep.WithLoggingASTTransformation every time an @WithLogging is found in a source unit. Any breakpoint set within LoggingASTTransformation will now be hit within the IDE when running the sample script.

The ASTTransformation class is a little more complex. Here is the very simple, and very naive, transformation to add a method start and stop message for @WithLogging:

```
@CompileStatic                                                          ①
@GroovyASTTransformation(phase=CompilePhase.SEMANTIC_ANALYSIS)          ②
class WithLoggingASTTransformation implements ASTTransformation {      ③

    @Override
    void visit(ASTNode[] nodes, SourceUnit sourceUnit) {                ④
        MethodNode method = (MethodNode) nodes[1]                       ⑤

        def startMessage = createPrintlnAst("Starting $method.name")    ⑥
        def endMessage = createPrintlnAst("Ending $method.name")        ⑦

        def existingStatements = ((BlockStatement)method.code).statements   ⑧
        existingStatements.add(0, startMessage)                         ⑨
        existingStatements.add(endMessage)                              ⑩

    }

    private static Statement createPrintlnAst(String message) {         ⑪
        new ExpressionStatement(
            new MethodCallExpression(
                new VariableExpression("this"),
                new ConstantExpression("println"),
                new ArgumentListExpression(
                    new ConstantExpression(message)
                )
            )
        )
    }
}
```

① even if not mandatory, if you write an AST transformation in Groovy, it is highly recommended to use CompileStatic because it will improve performance of the compiler.

② annotate with org.codehaus.groovy.transform.GroovyASTTransformation to tell at which compilation phase the transform needs to run. Here, it's at the *semantic analysis* phase.

③ implement the ASTTransformation interface

④ which only has a single visit method

⑤ the nodes parameter is a 2 AST node array, for which the first one is the annotation node (@WithLogging) and the second one is the annotated node (the method node)

⑥ create a statement that will print a message when we enter the method

⑦ create a statement that will print a message when we exit the method

⑧ get the method body, which in this case is a `BlockStatement`

⑨ add the enter method message before the first statement of existing code

⑩ append the exit method message after the last statement of existing code

⑪ creates an `ExpressionStatement` wrapping a `MethodCallExpression` corresponding to `this.println("message")`

It is important to notice that for the brevity of this example, we didn't make the necessary checks, such as checking that the annotated node is really a `MethodNode`, or that the method body is an instance of `BlockStatement`. This exercise is left to the reader.

Note the creation of the new println statements in the `createPrintlnAst(String)` method. Creating AST for code is not always simple. In this case we need to construct a new method call, passing in the receiver/variable, the name of the method, and an argument list. When creating AST, it might be helpful to write the code you're trying to create in a Groovy file and then inspect the AST of that code in the debugger to learn what to create. Then write a function like `createPrintlnAst` using what you learned through the debugger.

In the end:

```
@WithLogging
def greet() {
    println "Hello World"
}

greet()
```

Produces:

```
Starting greet
Hello World
Ending greet
```

NOTE

It is important to note that an AST transformation participates directly in the compilation process. A common error by beginners is to have the AST transformation code in the same source tree as a class that uses the transformation. Being in the same source tree in general means that they are compiled at the same time. Since the transformation itself is going to be compiled in phases and that each compile phase processes all files of the same source unit before going to the next one, there's a direct consequence: the transformation will not be compiled before the class that uses it! In conclusion, AST transformations need to be precompiled before you can use them. In general, it is as easy as having them in a separate source tree.

Relocating code the user wrote

Some annotations take a closure as a member, and that closure is code written by the user of your transform. Transforms routinely move such code somewhere else — `@TupleConstructor(pre={…})` inlines it into the constructor it generates, for example — and several then replace the annotation member with an empty closure so that the type checker does not also try to check it in place.

That is fine, provided the relocated code ends up somewhere in the AST and keeps the source position it had in the original source. Anything inspecting the AST at compile time — IDE tooling, static analysis, other transforms, a org.codehaus.groovy.control.customizers.CompilationCustomizer — tells code the user wrote apart from code the compiler generated by its source position, because generated nodes carry none. Moving user code somewhere only your own transform can find it, such as node metadata, makes it invisible to all of them.

If your transform cannot do that, because it evaluates the user's code during compilation or takes it out of the AST altogether, then provide a documented system property which turns the transform off, so a build compiling source it does not control can decline the behaviour. `@Grab` does this with `groovy.grape.enable`, and `@ASTTest` with `groovy.asttest.enable`; both default to `true`.

Global transformations

Global AST transformation are similar to local one with a major difference: they do not need an annotation, meaning that they are applied *globally*, that is to say on each class being compiled. It is therefore very important to limit their use to last resort, because it can have a significant impact on the compiler performance.

Following the example of the local AST transformation, imagine that we would like to trace all methods, and not only those which are annotated with `@WithLogging`. Basically, we need this code to behave the same as the one annotated with `@WithLogging` before:

```
def greet() {
    println "Hello World"
}

greet()
```

To make this work, there are two steps:

1. create the `org.codehaus.groovy.transform.ASTTransformation` descriptor inside the `META-INF/services` directory
2. create the `ASTTransformation` implementation

The descriptor file is required and must be found on classpath. It will contain a single line:

META-INF/services/org.codehaus.groovy.transform.ASTTransformation

```
gep.WithLoggingASTTransformation
```

The code for the transformation looks similar to the local case, but instead of using the `ASTNode[]` parameter, we need to use the `SourceUnit` instead:

gep/WithLoggingASTTransformation.groovy

```
@CompileStatic                                                                  ①
@GroovyASTTransformation(phase=CompilePhase.SEMANTIC_ANALYSIS)                  ②
class WithLoggingASTTransformation implements ASTTransformation {               ③

    @Override
    void visit(ASTNode[] nodes, SourceUnit sourceUnit) {                        ④
        def methods = sourceUnit.AST.methods                                    ⑤
        methods.each { method ->                                                ⑥
            def startMessage = createPrintlnAst("Starting $method.name")        ⑦
            def endMessage = createPrintlnAst("Ending $method.name")            ⑧

            def existingStatements = ((BlockStatement)method.code).statements   ⑨
            existingStatements.add(0, startMessage)                             ⑩
            existingStatements.add(endMessage)                                  ⑪
        }
    }

    private static Statement createPrintlnAst(String message) {                 ⑫
        new ExpressionStatement(
            new MethodCallExpression(
                new VariableExpression("this"),
                new ConstantExpression("println"),
                new ArgumentListExpression(
                    new ConstantExpression(message)
                )
            )
        )
    }
}
```

① even if not mandatory, if you write an AST transformation in Groovy, it is highly recommended to use `CompileStatic` because it will improve performance of the compiler.

② annotate with org.codehaus.groovy.transform.GroovyASTTransformation to tell at which compilation phase the transform needs to run. Here, it's at the *semantic analysis* phase.

③ implement the `ASTTransformation` interface

④ which only has a single `visit` method

⑤ the `sourceUnit` parameter gives access to the source being compiled, so we get the AST of the current source and retrieve the list of methods from this file

⑥ we iterate on each method from the source file

⑦ create a statement that will print a message when we enter the method

⑧ create a statement that will print a message when we exit the method

⑨ get the method body, which in this case is a `BlockStatement`

⑩ add the enter method message before the first statement of existing code

⑪ append the exit method message after the last statement of existing code

⑫ creates an `ExpressionStatement` wrapping a `MethodCallExpression` corresponding to `this.println("message")`

AST API guide

AbstractASTTransformation

While you have seen that you can directly implement the `ASTTransformation` interface, in almost all cases you will not do this but extend the org.codehaus.groovy.transform.AbstractASTTransformation class. This class provides several utility methods that make AST transformations easier to write. Almost all AST transformations included in Groovy extend this class.

ClassCodeExpressionTransformer

It is a common use case to be able to transform an expression into another. Groovy provides a class which makes it very easy to do this: org.codehaus.groovy.ast.ClassCodeExpressionTransformer

To illustrate this, let's create a `@Shout` transformation that will transform all `String` constants in method call arguments into their uppercase version. For example:

```
@Shout
def greet() {
    println "Hello World"
}

greet()
```

should print:

```
HELLO WORLD
```

Then the code for the transformation can use the `ClassCodeExpressionTransformer` to make this easier:

```
@CompileStatic
@GroovyASTTransformation(phase=CompilePhase.SEMANTIC_ANALYSIS)
class ShoutASTTransformation implements ASTTransformation {

    @Override
    void visit(ASTNode[] nodes, SourceUnit sourceUnit) {
        ClassCodeExpressionTransformer trn = new ClassCodeExpressionTransformer() {
①
```

```
            private boolean inArgList = false
            @Override
            protected SourceUnit getSourceUnit() {
                sourceUnit
②
            }

            @Override
            Expression transform(final Expression exp) {
                if (exp instanceof ArgumentListExpression) {
                    inArgList = true
                } else if (inArgList &&
                    exp instanceof ConstantExpression && exp.value instanceof String)
{
                    return new ConstantExpression(exp.value.toUpperCase())
③
                }
                def trn = super.transform(exp)
                inArgList = false
                trn
            }
        }
        trn.visitMethod((MethodNode)nodes[1])
④
    }
}
```

① Internally the transformation creates a `ClassCodeExpressionTransformer`

② The transformer needs to return the source unit

③ if a constant expression of type string is detected inside an argument list, transform it into its upper case version

④ call the transformer on the method being annotated

AST Nodes

WARNING	Writing an AST transformation requires a deep knowledge of the internal Groovy API. In particular it requires knowledge about the AST classes. Since those classes are internal, there are chances that the API will change in the future, meaning that your transformations *could* break. Despite that warning, the AST has been very stable over time and such a thing rarely happens.

Classes of the Abstract Syntax Tree belong to the `org.codehaus.groovy.ast` package. It is recommended to the reader to use the Groovy Console, in particular the AST browser tool, to gain knowledge about those classes. Another resource for learning is the AST Builder test suite.

Macros

Introduction

Until version 2.5.0, when developing AST transformations, developers should have a deep knowledge about how the AST (Abstract Syntax Tree) was built by the compiler in order to know how to add new expressions or statements during compile time.

Although the use of `org.codehaus.groovy.ast.tool.GeneralUtils` static methods could mitigate the burden of creating expressions and statements, it's still a low-level way of writing those AST nodes directly. We needed something to abstract us from writing the AST directly and that's exactly what Groovy macros were made for. They allow you to directly add code during compilation, without having to translate the code you had in mind to the `org.codehaus.groovy.ast.*` node related classes.

Statements and expressions

Let's see an example, lets create a local AST transformation: `@AddMessageMethod`. When applied to a given class it will add a new method called `getMessage` to that class. The method will return "42". The annotation is pretty straight forward:

```
@Retention(RetentionPolicy.SOURCE)
@Target([ElementType.TYPE])
@GroovyASTTransformationClass(["metaprogramming.AddMethodASTTransformation"])
@interface AddMethod { }
```

What would the AST transformation look like without the use of a macro ? Something like this:

```
@GroovyASTTransformation(phase = CompilePhase.INSTRUCTION_SELECTION)
class AddMethodASTTransformation extends AbstractASTTransformation {
    @Override
    void visit(ASTNode[] nodes, SourceUnit source) {
        ClassNode classNode = (ClassNode) nodes[1]

        ReturnStatement code =
                new ReturnStatement(                                    ①
                        new ConstantExpression("42"))                   ②

        MethodNode methodNode =
                new MethodNode(
                        "getMessage",
                        ACC_PUBLIC,
                        ClassHelper.make(String),
                        [] as Parameter[],
                        [] as ClassNode[],
                        code)                                           ③

        classNode.addMethod(methodNode)                                 ④
    }
}
```

① Create a return statement

② Create a constant expression "42"

③ Adding the code to the new method

④ Adding the new method to the annotated class

If you're not used to the AST API, that definitely doesn't look like the code you had in mind. Now look how the previous code simplifies with the use of macros.

```
@GroovyASTTransformation(phase = CompilePhase.INSTRUCTION_SELECTION)
class AddMethodWithMacrosASTTransformation extends AbstractASTTransformation {
    @Override
    void visit(ASTNode[] nodes, SourceUnit source) {
        ClassNode classNode = (ClassNode) nodes[1]

        ReturnStatement simplestCode = macro { return "42" }   ①

        MethodNode methodNode =
                new MethodNode(
                        "getMessage",
                        ACC_PUBLIC,
                        ClassHelper.make(String),
                        [] as Parameter[],
                        [] as ClassNode[],
                        simplestCode)                          ②

        classNode.addMethod(methodNode)                        ③
    }
}
```

① Much simpler. You wanted to add a return statement that returned "42" and that's exactly what you can read inside the `macro` utility method. Your plain code will be translated for you to a `org.codehaus.groovy.ast.stmt.ReturnStatement`

② Adding the return statement to the new method

③ Adding the new code to the annotated class

Although the `macro` method is used in this example to create a **statement** the `macro` method could also be used to create **expressions** as well, it depends on which `macro` signature you use:

- `macro(Closure)`: Create a given statement with the code inside the closure.
- `macro(Boolean,Closure)`: if **true** wrap expressions inside the closure inside a statement, if **false** then return an expression
- `macro(CompilePhase, Closure)`: Create a given statement with the code inside the closure in a specific compile phase
- `macro(CompilePhase, Boolean, Closure)`: Create a statement or an expression (true == statement, false == expression) in a specific compilation phase.

NOTE	All these signatures can be found at `org.codehaus.groovy.macro.runtime.MacroGroovyMethods`

Sometimes we could be only interested in creating a given expression, not the whole statement, in order to do that we should use any of the `macro` invocations with a boolean parameter:

```
@GroovyASTTransformation(phase = CompilePhase.INSTRUCTION_SELECTION)
class AddGetTwoASTTransformation extends AbstractASTTransformation {

    BinaryExpression onePlusOne() {
        return macro(false) { 1 + 1 }                                    ①
    }

    @Override
    void visit(ASTNode[] nodes, SourceUnit source) {
        ClassNode classNode = nodes[1]
        BinaryExpression expression = onePlusOne()                       ②
        ReturnStatement returnStatement = GeneralUtils.returnS(expression) ③

        MethodNode methodNode =
                new MethodNode("getTwo",
                        ACC_PUBLIC,
                        ClassHelper.Integer_TYPE,
                        [] as Parameter[],
                        [] as ClassNode[],
                        returnStatement                                  ④
                )

        classNode.addMethod(methodNode)                                  ⑤
    }
}
```

① We're telling macro not to wrap the expression in a statement, we're only interested in the expression

② Assigning the expression

③ Creating a `ReturnStatement` using a method from `GeneralUtils` and returning the expression

④ Adding the code to the new method

⑤ Adding the method to the class

Variable substitution

Macros are great but we can't create anything useful or reusable if our macros couldn't receive parameters or resolve surrounding variables.

In the following example we're creating an AST transformation `@MD5` that when applied to a given String field will add a method returning the MD5 value of that field.

```
@Retention(RetentionPolicy.SOURCE)
@Target([ElementType.FIELD])
@GroovyASTTransformationClass(["metaprogramming.MD5ASTTransformation"])
@interface MD5 { }
```

And the transformation:

```
@GroovyASTTransformation(phase = CompilePhase.CANONICALIZATION)
class MD5ASTTransformation extends AbstractASTTransformation {

    @Override
    void visit(ASTNode[] nodes, SourceUnit source) {
        FieldNode fieldNode = nodes[1]
        ClassNode classNode = fieldNode.declaringClass
        String capitalizedName = fieldNode.name.capitalize()
        MethodNode methodNode = new MethodNode(
                "get${capitalizedName}MD5",
                ACC_PUBLIC,
                ClassHelper.STRING_TYPE,
                [] as Parameter[],
                [] as ClassNode[],
                buildMD5MethodCode(fieldNode))

        classNode.addMethod(methodNode)
    }

    BlockStatement buildMD5MethodCode(FieldNode fieldNode) {
        VariableExpression fieldVar = GeneralUtils.varX(fieldNode.name) ①

        return macro(CompilePhase.SEMANTIC_ANALYSIS, true) {            ②
            return java.security.MessageDigest
                    .getInstance('MD5')
                    .digest($v { fieldVar }.getBytes())                 ③
                    .encodeHex()
                    .toString()
        }
    }
}
```

① We need a reference to a variable expression

② If using a class outside the standard packages we should add any needed imports or use the qualified name. When using the qualified name of a given static method you need to make sure it's resolved in the proper compile phase. In this particular case we're instructing the macro to resolve it at the SEMANTIC_ANALYSIS phase, which is the first compile phase with type information.

③ In order to substitute any `expression` inside the macro we need to use the `$v` method. `$v` receives a closure as an argument, and the closure is only allowed to substitute expressions, meaning

classes inheriting `org.codehaus.groovy.ast.expr.Expression`.

MacroClass

As we mentioned earlier, the `macro` method is only capable of producing `statements` and `expressions`. But what if we want to produce other types of nodes, such as a method, a field and so on?

`org.codehaus.groovy.macro.transform.MacroClass` can be used to create **classes** (ClassNode instances) in our transformations the same way we created statements and expressions with the `macro` method before.

The next example is a local transformation `@Statistics`. When applied to a given class, it will add two methods **getMethodCount()** and **getFieldCount()** which return how many methods and fields within the class respectively. Here is the marker annotation.

```
@Retention(RetentionPolicy.SOURCE)
@Target([ElementType.TYPE])
@GroovyASTTransformationClass(["metaprogramming.StatisticsASTTransformation"])
@interface Statistics {}
```

And the AST transformation:

```
@CompileStatic
@GroovyASTTransformation(phase = CompilePhase.INSTRUCTION_SELECTION)
class StatisticsASTTransformation extends AbstractASTTransformation {

    @Override
    void visit(ASTNode[] nodes, SourceUnit source) {
        ClassNode classNode = (ClassNode) nodes[1]
        ClassNode templateClass = buildTemplateClass(classNode)  ①

        templateClass.methods.each { MethodNode node ->          ②
            classNode.addMethod(node)
        }
    }

    @CompileDynamic
    ClassNode buildTemplateClass(ClassNode reference) {          ③
        def methodCount = constX(reference.methods.size())       ④
        def fieldCount = constX(reference.fields.size())         ⑤

        return new MacroClass() {
            class Statistics {
                java.lang.Integer getMethodCount() {             ⑥
                    return $v { methodCount }
                }

                java.lang.Integer getFieldCount() {              ⑦
                    return $v { fieldCount }
```

```
                }
            }
        }
    }
}
```

① Creating a template class

② Adding template class methods to the annotated class

③ Passing the reference class

④ Extracting reference class method count value expression

⑤ Extracting reference class field count value expression

⑥ Building the **getMethodCount()** method using reference's method count value expression

⑦ Building the **getFieldCount()** method using reference's field count value expression

Basically we've created the **Statistics** class as a template to avoid writing low level AST API, then we copied methods created in the template class to their final destination.

NOTE	Types inside the `MacroClass` implementation should be resolved inside, that's why we had to write `java.lang.Integer` instead of simply writing `Integer`.

IMPORTANT	Notice that we're using `@CompileDynamic`. That's because the way we use `MacroClass` is like we were actually implementing it. So if you were using `@CompileStatic` it will complain because an implementation of an abstract class can't be another different class.

@Macro methods

You have seen that by using `macro` you can save yourself a lot of work but you might wonder where that method came from. You didn't declare it or static import it. You can think of it as a special global method (or if you prefer, a method on every `Object`). This is much like how the `println` extension method is defined. But unlike `println` which becomes a method selected for execution later in the compilation process, `macro` expansion is done early in the compilation process. The declaration of `macro` as one of the available methods for this early expansion is done by annotating a `macro` method definition with the `@Macro` annotation and making that method available using a similar mechanism for extension modules. Such methods are known as *macro* methods and the good news is you can define your own.

To define your own macro method, create a class in a similar way to an extension module and add a method such as:

```
public class ExampleMacroMethods {

    @Macro
    public static Expression safe(MacroContext macroContext, MethodCallExpression
callExpression) {
        return ternaryX(
```

```
                notNullX(callExpression.getObjectExpression()),
                callExpression,
                constX(null)
        );
    }
    ...
}
```

Now you would register this as an extension module using a `org.codehaus.groovy.runtime.ExtensionModule` file within the `META-INF/groovy` directory.

Now, assuming that the class and meta info file are on your classpath, you can use the macro method in the following way:

```
def nullObject = null
assert null == safe(safe(nullObject.hashcode()).toString())
```

Disabling macro methods

The `groovy.macro.disable` system property suppresses selected `@Macro` methods from extension lookup before macro expansion runs. Its value is a comma-separated list of bare method names; any macro method whose name appears in the list is filtered out of the macro extension cache. For example, compiling with `-Dgroovy.macro.disable=DO,SVI` skips both the `DO` and `SVI` macros that ship with the `groovy-macro-library` module.

A few details to be aware of:

- **Bare-name matching only.** Every overload of a disabled name is filtered together, and the same name in different modules collides globally. There is no per-signature form (unlike `groovy.extension.disable`, which accepts `name(type1,type2,⋯)` for regular extension methods).
- **Read once at compiler startup.** The property is captured when the macro extension cache is first initialised, so it must be set as a `-D` argument to the JVM that runs the Groovy compiler.
- **No whitespace trimming.** Entries are split on commas only, so write `-Dgroovy.macro.disable=DO,SVI`, not `-Dgroovy.macro.disable=DO, SVI`.

Testing AST transformations

Separating source trees

This section is about good practices in regard to testing AST transformations. Previous sections highlighted the fact that to be able to execute an AST transformation, it has to be precompiled. It might sound obvious but a lot of people get caught on this, trying to use an AST transformation in the same source tree as where it is defined.

The first tip for testing AST transformation is therefore to separate test sources from the sources of the transform. Again, this is nothing but best practices, but you must make sure that your build too does actually compile them separately. This is the case by default with both Apache Maven and Gradle.

Debugging AST transformations

It is very handy to be able to put a breakpoint in an AST transformation, so that you can debug your code in the IDE. However, you might be surprised to see that your IDE doesn't stop on the breakpoint. The reason is actually simple: if your IDE uses the Groovy compiler to compile the unit tests for your AST transformation, then the compilation is triggered from the IDE, but the process which will compile the files doesn't have debugging options. It's only when the test case is executed that the debugging options are set on the virtual machine. In short: it is too late, the class has been compiled already, and your transformation is already applied.

A very easy workaround is to use the `GroovyTestCase` class which provides an `assertScript` method. This means that instead of writing this in a test case:

```
static class Subject {
    @MyTransformToDebug
    void methodToBeTested() {}
}

@Test
void testMyTransform() {
    def c = new Subject()
    c.methodToBeTested()
}
```

You should write:

```
@Test
void testMyTransformWithBreakpoint() {
    assertScript '''
        import metaprogramming.MyTransformToDebug

        class Subject {
            @MyTransformToDebug
            void methodToBeTested() {}
        }
        def c = new Subject()
        c.methodToBeTested()
    '''
}
```

The difference is that when you use `assertScript`, the code in the `assertScript` block is compiled **when the unit test is executed**. That is to say that this time, the `Subject` class will be compiled with debugging active, and the breakpoint is going to be hit.

ASTMatcher

Sometimes you may want to make assertions over AST nodes; perhaps to filter the nodes, or to make sure a given transformation has built the expected AST node.

Filtering nodes

For instance if you would like to apply a given transformation only to a specific set of AST nodes, you could use **ASTMatcher** to filter these nodes. The following example shows how to transform a given expression to another. Using **ASTMatcher** it looks for a specific expression `1 + 1` and it transforms it to `3`. That's why we called it the `@Joking` example.

First we create the `@Joking` annotation that only can be applied to methods:

```
@Retention(RetentionPolicy.SOURCE)
@Target([ElementType.METHOD])
@GroovyASTTransformationClass(["metaprogramming.JokingASTTransformation"])
@interface Joking { }
```

Then the transformation, that only applies an instance of `org.codehaus.groovy.ast.ClassCodeExpressionTransformer` to all the expressions within the method code block.

```
@CompileStatic
@GroovyASTTransformation(phase = CompilePhase.INSTRUCTION_SELECTION)
class JokingASTTransformation extends AbstractASTTransformation {
    @Override
    void visit(ASTNode[] nodes, SourceUnit source) {
        MethodNode methodNode = (MethodNode) nodes[1]

        methodNode
            .getCode()
            .visit(new ConvertOnePlusOneToThree(source))  ①
    }
}
```

① Get the method's code statement and apply the expression transformer

And this is when the **ASTMatcher** is used to apply the transformation only to those expressions matching the expression `1 + 1`.

```
class ConvertOnePlusOneToThree extends ClassCodeExpressionTransformer {
    SourceUnit sourceUnit

    ConvertOnePlusOneToThree(SourceUnit sourceUnit) {
        this.sourceUnit = sourceUnit
    }

    @Override
    Expression transform(Expression exp) {
        Expression ref = macro { 1 + 1 }     ①

        if (ASTMatcher.matches(ref, exp)) {  ②
```

```
            return macro { 3 }              ③
        }

        return super.transform(exp)
    }
}
```

① Builds the expression used as reference pattern

② Checks the current expression evaluated matches the reference expression

③ If it matches then replaces the current expression with the expression built with `macro`

Then you could test the implementation as follows:

```
package metaprogramming

class Something {
    @Joking
    Integer getResult() {
        return 1 + 1
    }
}

assert new Something().result == 3
```

Unit testing AST transforms

Normally we test AST transformations just checking that the final use of the transformation does what we expect. But it would be great if we could have an easy way to check, for example, that the nodes the transformation adds are what we expected from the beginning.

The following transformation adds a new method `giveMeTwo` to an annotated class.

```
@GroovyASTTransformation(phase = CompilePhase.INSTRUCTION_SELECTION)
class TwiceASTTransformation extends AbstractASTTransformation {

    static final String VAR_X = 'x'

    @Override
    void visit(ASTNode[] nodes, SourceUnit source) {
        ClassNode classNode = (ClassNode) nodes[1]
        MethodNode giveMeTwo = getTemplateClass(sumExpression)
            .getDeclaredMethods('giveMeTwo')
            .first()

        classNode.addMethod(giveMeTwo)                  ①
    }

    BinaryExpression getSumExpression() {               ②
```

```
        return macro {
            $v{ varX(VAR_X) } +
            $v{ varX(VAR_X) }
        }
    }

    ClassNode getTemplateClass(Expression expression) { ③
        return new MacroClass() {
            class Template {
                java.lang.Integer giveMeTwo(java.lang.Integer x) {
                    return $v { expression }
                }
            }
        }
    }
}
```

① Adding the method to the annotated class

② Building a binary expression. The binary expression uses the same variable expression in both sides of the `+` token (check `varX` method at **org.codehaus.groovy.ast.tool.GeneralUtils**).

③ Builds a new **ClassNode** with a method called `giveMeTwo` which returns the result of an expression passed as parameter.

Now instead of creating a test executing the transformation over a given sample code. I would like to check that the construction of the binary expression is done properly:

```
@Test
void testTestingSumExpression() {
    use(ASTMatcher) {                   ①
        TwiceASTTransformation sample = new TwiceASTTransformation()
        Expression referenceNode = macro {
            a + a                       ②
        }.withConstraints {             ③
            placeholder 'a'             ④
        }

        assert sample
            .sumExpression
            .matches(referenceNode)     ⑤
    }
}
```

① Using ASTMatcher as a category

② Build a template node

③ Apply some constraints to that template node

④ Tells compiler that `a` is a placeholder.

⑤ Asserts reference node and current node are equal

Of course you can/should always check the actual execution:

```
@Test
void testASTBehavior() {
    assertScript '''
    package metaprogramming

    @Twice
    class AAA {

    }

    assert new AAA().giveMeTwo(1) == 2
    '''
}
```

ASTTest

Last but not least, testing an AST transformation is also about testing the state of the AST **during compilation**. Groovy provides a tool named `@ASTTest` for this: it is an annotation that will let you add assertions on an abstract syntax tree. Please check the documentation for ASTTest for more details.

External references

If you are interested in a step-by-step tutorial about writing AST transformations, you can follow this workshop.

Dependency management with Grape

Quick start

Add a Dependency

Grape is a JAR dependency manager embedded into Groovy. Grape lets you quickly add maven repository dependencies to your classpath, making scripting even easier. The simplest use is as simple as adding an annotation to your script:

```
@Grab(group='org.springframework', module='spring-orm', version='5.2.8.RELEASE')
import org.springframework.jdbc.core.JdbcTemplate
```

`@Grab` also supports a shorthand notation:

```
@Grab('org.springframework:spring-orm:5.2.8.RELEASE')
import org.springframework.jdbc.core.JdbcTemplate
```

Note that we are using an annotated import here, which is the recommended way. You can also search for dependencies on mvnrepository.com and it will provide you the `@Grab` annotation form of the `pom.xml` entry.

Specify Additional Repositories

Not all dependencies are in maven central. You can add new ones like this:

```
@GrabResolver(name='restlet', root='http://maven.restlet.org/')
@Grab(group='org.restlet', module='org.restlet', version='1.1.6')
```

Maven Classifiers

Some maven dependencies need classifiers in order to be able to resolve. You can fix that like this:

```
@Grab(group='net.sf.json-lib', module='json-lib', version='2.2.3', classifier='jdk15')
```

Excluding Transitive Dependencies

Sometimes you will want to exclude transitive dependencies as you might be already using a slightly different but compatible version of some artifact. You can do this as follows:

```
@Grab('net.sourceforge.htmlunit:htmlunit:2.8')
@GrabExclude('xml-apis:xml-apis')
```

JDBC Drivers

Because of the way JDBC drivers are loaded, you'll need to configure Grape to attach JDBC driver dependencies to the system class loader. I.e:

```
@GrabConfig(systemClassLoader=true)
@Grab(group='mysql', module='mysql-connector-java', version='5.1.6')
```

Using Grape From the Groovy Shell

From groovysh use the method call variant:

```
groovy.grape.Grape.grab(group:'org.springframework', module:'spring', version:'2.5.6')
```

Proxy settings

If you are behind a firewall and/or need to use Groovy/Grape through a proxy server, you can specify those settings on the command like via the `http.proxyHost` and `http.proxyPort` system properties:

```
groovy -Dhttp.proxyHost=yourproxy -Dhttp.proxyPort=8080 yourscript.groovy
```

Or you can make this system-wide by adding these properties to your JAVA_OPTS environment variable:

```
JAVA_OPTS = -Dhttp.proxyHost=yourproxy -Dhttp.proxyPort=8080
```

Logging

The Grape facade (`groovy.grape.Grape`) uses JDK Platform Logging and can be configured through `~/.groovy/logging.properties` (see the groovy command for details).

Logging of the underlying engine depends on which Grape implementation is active.

GrapeIvy logging

GrapeIvy routes Ivy's internal logging through JDK Platform Logging under the logger name `groovy.grape.ivy`. To control the verbosity, set the level in `~/.groovy/logging.properties`:

```
# Show Ivy info messages (INFO, WARNING, ERROR)
groovy.grape.ivy.level=INFO

# Show verbose Ivy output (adds dependency resolution details)
groovy.grape.ivy.level=FINE

# Show all Ivy debug output
groovy.grape.ivy.level=FINEST
```

The `grape` command's `-q`, `-w`, `-i`, `-V`, and `-d` flags also adjust the Ivy logging level for that invocation.

If you want to see download progress specifically, set the system property `groovy.grape.report.downloads` to `true` (e.g. add `-Dgroovy.grape.report.downloads=true` to invocation or JAVA_OPTS) and Grape will print the following infos to System.error:

- Starting resolve of a dependency
- Starting download of an artifact
- Retrying download of an artifact
- Download size and time for downloaded artifacts

GrapeMaven logging

When using the GrapeMaven engine, set `-Dgroovy.grape.debug=true` to enable verbose Maven Resolver logging to `System.err`. The `groovy.grape.report.downloads` property has no effect with this engine.

Detail

Grape (The *Groovy Adaptable Packaging Engine* or *Groovy Advanced Packaging Engine*) is the infrastructure enabling the grab() calls in Groovy, a repository-driven module system that allows a developer to write a script with an essentially arbitrary library requirement, and ship just the script. Grape will, at runtime, download as needed and link the named libraries and all dependencies forming a transitive closure when the script is run from existing repositories such as Maven Central.

Groovy ships with two Grape engine implementations: `GrapeIvy`, based on Apache Ivy, and `GrapeMaven`, based on Maven Resolver (Aether). They provide the same core `@Grab` / `@GrabResolver` / `@GrabExclude` experience but differ in some areas of advanced configuration and behaviour; see Grape Implementations for details.

Grape follows the Ivy conventions for module version identification, with naming change.

- `group` - Which module group the module comes from. Translates directly to a Maven groupId or an Ivy Organization. Any group matching `/groovy[x][\..*]^/` is reserved and may have special meaning to the groovy endorsed modules.
- `module` - The name of the module to load. Translated directly to a Maven artifactId or an Ivy artifact.
- `version` - The version of the module to use. Either a literal version `1.1-RC3' or a range `[2.2.1,)' meaning 2.2.1 or any greater version).
- `classifier` - The optional classifier to use (for example, *jdk15*)

The downloaded modules will be stored according to the active engine's cache conventions (see Grape Implementations for per-engine cache directories).

Grape Implementations

Groovy ships with two built-in Grape engine implementations. Both support the same core `@Grab`, `@GrabResolver`, and `@GrabExclude` annotations and method calls. They differ in their underlying resolution mechanism, cache layout, and some advanced configuration options.

Built-in implementations

GrapeIvy (default)

`groovy.grape.ivy.GrapeIvy` uses Apache Ivy for dependency resolution. It is the default engine when both implementations are on the classpath.

- **Backend**: Apache Ivy
- **Local cache**: `~/.groovy/grapes` (Ivy's standard cache format)
- **Configuration file**: `~/.groovy/grapeConfig.xml` (full Ivy XML settings — see Customize Ivy settings (GrapeIvy only))
- **`@GrabConfig(autoDownload=false)`**: Supported. When `false`, resolution is restricted to the local cache; no network calls are made.

- **`conf:` parameter**: Mapped directly to Ivy configurations; multiple configurations such as `conf:['default','optional']` are supported.
- **Version wildcards**: `'*'` resolves to Ivy's `latest.default`.

GrapeMaven

`groovy.grape.maven.GrapeMaven` uses Maven Resolver (Aether) for dependency resolution.

- **Backend**: Maven Resolver (Aether)
- **Local cache**: `~/.groovy/grapesM2` (Maven local-repository format)
- **Configuration**: Remote repositories configured via `@GrabResolver` or the `Grape.addResolver()` API; no configuration file equivalent.
- **`@GrabConfig(autoDownload=…)`**: The `autoDownload` flag is not honoured; GrapeMaven always attempts remote resolution when an artifact is not already cached.
- **`conf:` parameter**: Treated as a Maven scope; `'default'` is mapped to `'compile'`. Multiple configurations in a single grab are not supported.
- **Version wildcards**: `'*'` resolves to Maven's `LATEST`.

Selecting an implementation

When both `groovy-grape-ivy` and `groovy-grape-maven` are on the classpath, `GrapeIvy` is preferred by default. To switch to GrapeMaven for a script, pass the system property:

```
groovy -Dgroovy.grape.impl=groovy.grape.maven.GrapeMaven yourscript.groovy
```

Or set it globally via `JAVA_OPTS`:

```
JAVA_OPTS=-Dgroovy.grape.impl=groovy.grape.maven.GrapeMaven
```

Migration considerations

Most existing `@Grab` scripts work unchanged with both engines, but there are some compatibility notes to consider:

- **`conf:` parameter**: GrapeIvy supports multiple Ivy configurations (e.g., `conf:['default','optional']`), while GrapeMaven treats `conf:` as a singular Maven scope (e.g., `conf:'compile'`). Scripts using multiple configurations must remain on GrapeIvy or be reworked.
- **Custom Ivy configuration**: If you have customized `~/.groovy/grapeConfig.xml` with non-standard Ivy settings (especially if you disabled the default `m2compatible=true`), those settings do not apply to GrapeMaven. Review your configuration before switching, or reconfigure custom repositories using `@GrabResolver` annotations.
- **`@GrabConfig(autoDownload=false)`**: GrapeMaven does not honour `autoDownload=false` as a per-grab switch. The closest equivalent is to point `@GrabResolver` (or `Grape.addResolver`) to a local-only repository/cache location. If you rely on `autoDownload=false` semantics broadly, remain on

GrapeIvy.

Custom implementations

You can provide your own `GrapeEngine` implementation by including a jar on the classpath that declares it as a Java SPI (Service Provider Interface). Groovy will discover and load it via `java.util.ServiceLoader`.

To register a custom implementation:

1. Create a class implementing `groovy.grape.GrapeEngine`.
2. Add a service provider configuration file at `META-INF/services/groovy.grape.GrapeEngine` in your jar with a single line naming the fully-qualified class name.
3. Place the jar on the classpath.

You can then select it with:

```
groovy -Dgroovy.grape.impl=my.custom.GrapeImpl yourscript.groovy
```

If your implementation is the only one on the classpath, it will be automatically discovered and used by default. If multiple implementations are present (including the built-ins), you must specify which one to use via the system property.

Usage

Annotation

One or more `groovy.lang.Grab` annotations can be added at any place that annotations are accepted to tell the compiler that this code relies on the specific library. This will have the effect of adding the library to the classloader of the groovy compiler. This annotation is detected and evaluated before any other resolution of classes in the script, so imported classes can be properly resolved by a `@Grab` annotation.

Supported annotation targets include imports (recommended), classes, methods, constructors, fields, parameters, and local variables.

Here is an example placing `@Grab` on a class:

```
import com.jidesoft.swing.JideSplitButton
@Grab(group='com.jidesoft', module='jide-oss', version='[2.2.1,2.3.0)')
public class TestClassAnnotation {
    public static String testMethod () {
        return JideSplitButton.class.name
    }
}
```

Here is an example placing `@Grab` on a local variable:

```
@Grab('commons-lang:commons-lang:2.4')
var input = 'hello world'
println org.apache.commons.lang.WordUtils.capitalize(input)
```

An appropriate `grab(…)` call will be added to the static initializer of the class of the containing class (or script class in the case of an annotated script element).

Multiple Grape Annotations

In early versions of Groovy, if you wanted to use a Grab annotation multiple times on the same node you had to use the `@Grapes` annotation, e.g.:

```
@Grapes([
   @Grab(group='commons-primitives', module='commons-primitives', version='1.0'),
   @Grab(group='org.ccil.cowan.tagsoup', module='tagsoup', version='0.9.7')])
class Example {
// ...
}
```

Otherwise you'd encounter the following error:

```
Cannot specify duplicate annotation on the same member
```

But in recent versions, @Grapes is purely optional.

Technical notes:

- Originally, Groovy stored the Grab annotations for access at runtime and duplicates aren't allowed in the bytecode. In current versions, @Grab has only SOURCE retention, so the multiple occurrences aren't an issue.
- Future versions of Grape may support using the Grapes annotation to provide a level of structuring, e.g. allowing a GrabExclude or GrabResolver annotation to apply to only a subset of the Grab annotations.

Method call

Typically a call to grab will occur early in the script or in class initialization. This is to ensure that the libraries are made available to the ClassLoader before the groovy code relies on the code. A couple of typical calls may appear as follows:

```
import groovy.grape.Grape
// random maven library
Grape.grab(group:'com.jidesoft', module:'jide-oss', version:'[2.2.0,)')
Grape.grab([group:'org.apache.ivy', module:'ivy', version:'2.0.0-beta1', conf:
['default', 'optional']],
```

```
    [group:'org.apache.ant', module:'ant', version:'1.7.0'])
```

- Multiple calls to grab in the same context with the same parameters should be idempotent. However, if the same code is called with a different `ClassLoader` context then resolution may be re-run.
- If the `args` map passed into the `grab` call has an attribute `noExceptions` that evaluates true no exceptions will be thrown.
- `grab` requires that a `RootLoader` or `GroovyClassLoader` be specified or be in the `ClassLoader` chain of the calling class. By default failure to have such a `ClassLoader` available will result in module resolution and an exception being thrown
  - The ClassLoader passed in via the `classLoader:` argument and its parent classloaders.
  - The ClassLoader of the object passed in as the `referenceObject:` argument, and its parent classloaders.
  - The ClassLoader of the class issuing the call to `grab`

grab(HashMap) Parameters

- `group:` - <String> - Which module group the module comes from. Translates directly to a Maven groupId. Any group matching `/groovy(|\..|x|x\..)/` is reserved and may have special meaning to the groovy endorsed modules.
- `module:` - <String> - The name of the module to load. Translated directly to a Maven artifactId.
- `version:` - <String> and possibly <Range> - The version of the module to use. Either a literal version `1.1-RC3' or an Ivy Range `[2.2.1,)' meaning 2.2.1 or any greater version).
- `classifier:` - <String> - The Maven classifier to resolve by.
- `conf:` - <String>, default `default' - The configuration or scope of the module to download. The default conf is `default:` which maps to the maven `runtime` and `master` scopes.
- `force:`- <boolean>, defaults true - Used to indicate that this revision must be used in case of conflicts, independently of
- conflicts manager
- `changing:` - <boolean>, default false - Whether the artifact can change without its version designation changing.
- `transitive:` - <boolean>, default true - Whether to resolve other dependencies this module has or not.

There are two principal variants of `grab`, one with a single Map and one with an arguments Map and multiple dependencies map. A call to the single map grab is the same as calling grab with the same map passed in twice, so grab arguments and dependencies can be mixed in the same map, and grab can be called as a single method with named parameters.

There are synonyms for these parameters. Submitting more than one is a runtime exception.

- `group:`, `groupId:`, `organisation:`, `organization:`, `org:`
- `module:`, `artifactId:`, `artifact:`

- `version:`, `revision:`, `rev:`
- `conf:`, `scope:`, `configuration:`

Arguments Map arguments

- `classLoader:` - <GroovyClassLoader> or <RootClassLoader> - The ClassLoader to add resolved Jars to
- `refObject:` - <Object> - The closest parent ClassLoader for the object's class will be treated as though it were passed in as `classLoader:`
- `validate:` - <boolean>, default false - Should poms or ivy files be validated (true), or should we trust the cache (false).
- `noExceptions:` - <boolean>, default false - If ClassLoader resolution or repository querying fails, should we throw an exception (false) or fail silently (true).

Command Line Tool

Grape added a command line executable `grape' that allows for the inspection and management of the local grape cache.

```
grape install [-hv] <group> <module> [<version>] [<classifier>]
```

This installs the specified groovy module or maven artifact. If a version is specified that specific version will be installed, otherwise the most recent version will be used (as if `*' we passed in).

```
grape list
```

Lists locally installed modules (with their full maven name in the case of groovy modules) and versions.

```
grape resolve [-adhisv] (<groupId> <artifactId> <version>)+
```

This returns the file locations of the jars representing the artifacts for the specified module(s) and the respective transitive dependencies. You may optionally pass in -ant, -dos, or -shell to get the dependencies expressed in a format applicable for an ant script, windows batch file, or unix shell script respectively. -ivy may be passed to see the dependencies expressed in an ivy like format.

```
grape uninstall [-hv] <group> <module> <version>
```

This uninstalls a particular grape: it non-transitively removes the respective jar file from the grape cache.

Advanced configuration

Repository Directory

The local cache directory used by each engine is:

- **GrapeIvy**: `~/.groovy/grapes`
- **GrapeMaven**: `~/.groovy/grapesM2`

If you need to change the root directory that Grape uses for downloading libraries you can specify the `grape.root` system property to change the default (which is `~/.groovy`). Both engines respect this property.

```
groovy -Dgrape.root=/repo/grapes yourscript.groovy
```

Customize Ivy settings (GrapeIvy only)

When using GrapeIvy, you can customize the Ivy settings that Grape uses by creating a `~/.groovy/grapeConfig.xml` file. If no such file exists, here are the default settings used by Grape.

For more information on how to customize these settings, please refer to the Ivy documentation.

This configuration file is not used by the GrapeMaven engine; with GrapeMaven, additional remote repositories are registered via `@GrabResolver` annotations or programmatically via the `Grape.addResolver` API.

More Examples

Using Apache Commons Collections:

```
// create and use a primitive array list
@Grab(group='commons-primitives', module='commons-primitives', version='1.0')
import org.apache.commons.collections.primitives.ArrayIntList

def createEmptyInts() { new ArrayIntList() }

def ints = createEmptyInts()
ints.add(0, 42)
assert ints.size() == 1
assert ints.get(0) == 42
```

Using TagSoup:

```
// find the PDF links of the Java specifications
@Grab(group='org.ccil.cowan.tagsoup', module='tagsoup', version='1.2.1')
def getHtml() {
    def parser = new XmlParser(new org.ccil.cowan.tagsoup.Parser())
    parser.parse("https://docs.oracle.com/javase/specs/")
}
html.body.'**'.a.@href.grep(~/.*\.pdf/).each{ println it }
```

Using Google Collections:

```
import com.google.common.collect.HashBiMap
@Grab(group='com.google.code.google-collections', module='google-collect', version
='snapshot-20080530')
def getFruit() { [grape:'purple', lemon:'yellow', orange:'orange'] as HashBiMap }
assert fruit.lemon == 'yellow'
assert fruit.inverse().yellow == 'lemon'
```

Launching a Jetty server to serve Groovy templates:

```
@Grab('org.eclipse.jetty.aggregate:jetty-server:8.1.19.v20160209')
@Grab('org.eclipse.jetty.aggregate:jetty-servlet:8.1.19.v20160209')
@Grab('javax.servlet:javax.servlet-api:3.0.1')
import org.eclipse.jetty.server.Server
import org.eclipse.jetty.servlet.ServletContextHandler
import groovy.servlet.TemplateServlet

def runServer(duration) {
    def server = new Server(8080)
    def context = new ServletContextHandler(server, "/", ServletContextHandler
.SESSIONS)
    context.resourceBase = "."
    context.addServlet(TemplateServlet, "*.gsp")
    server.start()
    sleep duration
    server.stop()
}

runServer(10000)
```

Grape will download Jetty and its dependencies on first launch of this script, and cache them. We create a new Jetty Server on port 8080, then expose Groovy's TemplateServlet at the root of the context — Groovy comes with its own powerful template engine mechanism. We start the server and let it run for a certain duration. Each time someone will hit http://localhost:8080/somepage.gsp, it will display the somepage.gsp template to the user — those template pages should be situated in the same directory as this server script.

Testing Guide

Introduction

The Groovy programming language comes with great support for writing tests. In addition to the language features and test integration with state-of-the-art testing libraries and frameworks, the Groovy ecosystem has born a rich set of testing libraries and frameworks.

This chapter will start with language specific testing features and continue with a closer look at

JUnit integration, Spock for specifications, and Geb for functional tests. Finally, we'll do an overview of other testing libraries known to be working with Groovy.

Language Features

Besides integrated support for JUnit, the Groovy programming language comes with features that have proven to be very valuable for test-driven development. This section gives insight on them.

Power Assertions

Writing tests means formulating assumptions by using assertions. In Java this can be done by using the `assert` keyword that has been added in J2SE 1.4. In Java, `assert` statements can be enabled via the JVM parameters `-ea` (or `-enableassertions`) and `-da` (or `-disableassertions`). Assertion statements in Java are disabled by default.

Groovy comes with a rather *powerful variant* of `assert` also known as *power assertion statement*. Groovy's power `assert` differs from the Java version in its output given the boolean expression validates to `false`:

```
def x = 1
assert x == 2

// Output:             ①
//
// Assertion failed:
// assert x == 2
//        | |
//        1 false
```

① This section shows the std-err output

The `java.lang.AssertionError` that is thrown whenever the assertion can not be validated successfully, contains an extended version of the original exception message. The power assertion output shows evaluation results from the outer to the inner expression.

The power assertion statements true power unleashes in complex Boolean statements, or statements with collections or other `toString`-enabled classes:

```
def x = [1,2,3,4,5]
assert (x << 6) == [6,7,8,9,10]

// Output:
//
// Assertion failed:
// assert (x << 6) == [6,7,8,9,10]
//         | |     |
//         | |     false
//         | [1, 2, 3, 4, 5, 6]
```

```
//        [1, 2, 3, 4, 5, 6]
```

Another important difference from Java is that in Groovy assertions are *enabled by default*. It has been a language design decision to remove the possibility to deactivate assertions. Or, as Bertrand Meyer stated, `it makes no sense to take off your swim ring if you put your feet into real water`.

One thing to be aware of are methods with side effects inside Boolean expressions in power assertion statements. As the internal error message construction mechanism does only store references to instances under target, it happens that the error message text is invalid at rendering time in case of side-effecting methods involved:

```
assert [[1,2,3,3,3,3,4]].first().unique() == [1,2,3]

// Output:
//
// Assertion failed:
// assert [[1,2,3,3,3,3,4]].first().unique() == [1,2,3]
//                          |       |        |
//                          |       |        false
//                          |       [1, 2, 3, 4]
//                          [1, 2, 3, 4]          ①
```

① The error message shows the actual state of the collection, not the state before the `unique` method was applied

NOTE If you choose to provide a custom assertion error message this can be done by using the Java syntax `assert expression1 : expression2` where `expression1` is the Boolean expression and `expression2` is the custom error message. Be aware though that this will disable the power assert and will fully fall back to custom error messages on assertion errors.

Mocking and Stubbing

Groovy has excellent built-in support for a range of mocking and stubbing alternatives. When using Java, dynamic mocking frameworks are very popular. A key reason for this is that it is hard work creating custom hand-crafted mocks using Java. Such frameworks can be used easily with Groovy if you choose but creating custom mocks is much easier in Groovy. You can often get away with simple maps or closures to build your custom mocks.

The following sections show ways to create mocks and stubs with Groovy language features only.

Map Coercion

By using maps or expandos, we can incorporate desired behaviour of a collaborator very easily as shown here:

```
class TranslationService {
    String convert(String key) {
```

```
        return "test"
    }
}

def service = [convert: { String key -> 'some text' }] as TranslationService
assert 'some text' == service.convert('key.text')
```

The `as` operator can be used to coerce a map to a particular class. The given map keys are interpreted as method names and the values, being `groovy.lang.Closure` blocks, are interpreted as method code blocks.

NOTE	Be aware that map coercion can get into the way if you deal with custom `java.util.Map` descendant classes in combination with the `as` operator. The map coercion mechanism is targeted directly at certain collection classes, it doesn't take custom classes into account.

Closure Coercion

The 'as' operator can be used with closures in a neat way which is great for developer testing in simple scenarios. We haven't found this technique to be so powerful that we want to do away with dynamic mocking, but it can be very useful in simple cases none-the-less.

Classes or interfaces holding a single method, including SAM (single abstract method) classes, can be used to coerce a closure block to be an object of the given type. Be aware that for doing this, Groovy internally create a proxy object descending for the given class. So the object will not be a direct instance of the given class. This important if, for example, the generated proxy object's metaclass is altered afterwards.

Let's have an example on coercing a closure to be of a specific type:

```
def service = { String key -> 'some text' } as TranslationService
assert 'some text' == service.convert('key.text')
```

Groovy supports a feature called implicit SAM coercion. This means that the `as` operator is not necessary in situations where the runtime can infer the target SAM type. This type of coercion might be useful in tests to mock entire SAM classes:

```
abstract class BaseService {
    abstract void doSomething()
}

BaseService service = { -> println 'doing something' }
service.doSomething()
```

MockFor and StubFor

The Groovy mocking and stubbing classes can be found in the `groovy.mock.interceptor` package.

The `MockFor` class supports (typically unit) testing of classes in isolation by allowing a *strictly ordered* expectation of the behavior of collaborators to be defined. A typical test scenario involves a class under test and one or more collaborators. In such a scenario it is often desirable to just test the business logic of the class under test. One strategy for doing that is to replace the collaborator instances with simplified mock objects to help isolate out the logic in the test target. MockFor allows such mocks to be created using meta-programming. The desired behavior of collaborators is defined as a behavior specification. The behavior is enforced and checked automatically.

Let's assume our target classes looked like this:

```
class Person {
    String first, last
}

class Family {
    Person father, mother
    def nameOfMother() { "$mother.first $mother.last" }
}
```

With `MockFor`, a mock expectation is always sequence dependent and its use automatically ends with a call to `verify`:

```
def mock = new MockFor(Person)      ①
mock.demand.getFirst{ 'dummy' }
mock.demand.getLast{ 'name' }
mock.use {                          ②
    def mary = new Person(first:'Mary', last:'Smith')
    def f = new Family(mother:mary)
    assert f.nameOfMother() == 'dummy name'
}
mock.expect.verify()                ③
```

① a new mock is created by a new instance of `MockFor`

② a `Closure` is passed to `use` which enables the mocking functionality

③ a call to `verify` checks whether the sequence and number of method calls is as expected

The `StubFor` class supports (typically unit) testing of classes in isolation by allowing a *loosely-ordered* expectation of the behavior of collaborators to be defined. A typical test scenario involves a class under test and one or more collaborators. In such a scenario it is often desirable to just test the business logic of the CUT. One strategy for doing that is to replace the collaborator instances with simplified stub objects to help isolate out the logic in the target class. `StubFor` allows such stubs to be created using meta-programming. The desired behavior of collaborators is defined as a behavior specification.

In contrast to `MockFor` the stub expectation checked with `verify` is sequence independent and its use is optional:

```
def stub = new StubFor(Person)      ①
stub.demand.with {                  ②
    getLast{ 'name' }
    getFirst{ 'dummy' }
}
stub.use {                          ③
    def john = new Person(first:'John', last:'Smith')
    def f = new Family(father:john)
    assert f.father.first == 'dummy'
    assert f.father.last == 'name'
}
stub.expect.verify()                ④
```

① a new stub is created by a new instance of `StubFor`

② the `with` method is used for delegating all calls inside the closure to the `StubFor` instance

③ a `Closure` is passed to `use` which enables the stubbing functionality

④ a call to `verify` (optional) checks whether the number of method calls is as expected

`MockFor` and `StubFor` can not be used to test statically compiled classes e.g. for Java classes or Groovy classes that make use of `@CompileStatic`. To stub and/or mock these classes you can use Spock or one of the Java mocking libraries.

Expando Meta-Class (EMC)

Groovy includes a special `MetaClass` the so-called `ExpandoMetaClass` (EMC). It allows to dynamically add methods, constructors, properties and static methods using a neat closure syntax.

Every `java.lang.Class` is supplied with a special `metaClass` property that will give a reference to an `ExpandoMetaClass` instance. The expando metaclass is not restricted to custom classes, it can be used for JDK classes like for example `java.lang.String` as well:

```
String.metaClass.swapCase = {->
    def sb = new StringBuffer()
    delegate.each {
        sb << (Character.isUpperCase(it as char) ? Character.toLowerCase(it as char) :
            Character.toUpperCase(it as char))
    }
    sb.toString()
}

def s = "heLLo, worLD!"
assert s.swapCase() == 'HEllO, WORld!'
```

The `ExpandoMetaClass` is a rather good candidate for mocking functionality as it allows for more advanced stuff like mocking static methods

```
class Book {
```

```
    String title
}

Book.metaClass.static.create << { String title -> new Book(title:title) }

def b = Book.create("The Stand")
assert b.title == 'The Stand'
```

or even constructors

```
Book.metaClass.constructor << { String title -> new Book(title:title) }

def b = new Book("The Stand")
assert b.title == 'The Stand'
```

NOTE

Mocking constructors might seem like a hack that's better not even to be considered but even there might be valid use cases. An example can be found in Grails where domain class constructors are added at run-time with the help of `ExpandoMetaClass`. This lets the domain object register itself in the Spring application context and allows for injection of services or other beans controlled by the dependency-injection container.

If you want to change the `metaClass` property on a per test method level you need to remove the changes that were done to the metaclass, otherwise those changes would be persistent across test method calls. Changes are removed by replacing the metaclass in the `GroovyMetaClassRegistry`:

```
GroovySystem.metaClassRegistry.removeMetaClass(String)
```

Another alternative is to register a `MetaClassRegistryChangeEventListener`, track the changed classes and remove the changes in the cleanup method of your chosen testing runtime. A good example can be found in the Grails web development framework.

Besides using the `ExpandoMetaClass` on a class-level, there is also support for using the metaclass on a per-object level:

```
def b = new Book(title: "The Stand")
b.metaClass.getTitle {-> 'My Title' }

assert b.title == 'My Title'
```

In this case the metaclass change is related to the instance only. Depending on the test scenario this might be a better fit than the global metaclass change.

GDK Methods

The following section gives a brief overview on GDK methods that can be leveraged in test case

scenarios, for example for test data generation.

Iterable#combinations

The `combinations` method that is added on `java.lang.Iterable` compliant classes can be used to get a list of combinations from a list containing two or more sub-lists:

```
@Test
void testCombinations() {
    def combinations = [[2, 3],[4, 5, 6]].combinations()
    assert combinations == [[2, 4], [3, 4], [2, 5], [3, 5], [2, 6], [3, 6]]
}
```

The method could be used in test case scenarios to generate all possible argument combinations for a specific method call.

Iterable#eachCombination

The `eachCombination` method that is added on `java.lang.Iterable` can be used to apply a function (or in this case a `groovy.lang.Closure`) to each if the combinations that has been built by the `combinations` method:

`eachCombination` is a GDK method that is added to all classes conforming to the `java.lang.Iterable` interface. It applies a function on each combination of the input lists:

```
@Test
void testEachCombination() {
    [[2, 3],[4, 5, 6]].eachCombination { println it[0] + it[1] }
}
```

The method could be used in the testing context to call methods with each of the generated combinations.

Tool Support

Test Code Coverage

Code coverage is a useful measure of the effectiveness of (unit) tests. A program with high code coverage has a lower chance to hold critical bugs than a program with no or low coverage. To get code coverage metrics, the generated byte-code usually needs to be instrumented before the tests are executed. One tool with Groovy support for this task is Cobertura.

The following code listing shows an example on how to enable Cobertura test coverage reports in a Gradle build script from a Groovy project:

```
def pluginVersion = '<plugin version>'
def groovyVersion = '<groovy version>'
def junitVersion = '<junit version>'
```

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.eriwen:gradle-cobertura-plugin:${pluginVersion}'
    }
}

apply plugin: 'groovy'
apply plugin: 'cobertura'

repositories {
    mavenCentral()
}

dependencies {
    compile "org.codehaus.groovy:groovy-all:${groovyVersion}"
    testCompile "junit:junit:${junitVersion}"
}

cobertura {
    format = 'html'
    includes = ['**/*.java', '**/*.groovy']
    excludes = ['com/thirdparty/**/*.*']
}
```

Several output formats can be chosen for Cobertura coverage reports and test code coverage reports can be added to continuous integration build tasks.

JavaShell

The `org.apache.groovy.util.JavaShell` class (covered in detail in the integration guide) is also useful in tests when you need real Java classes alongside your Groovy code. It avoids two recurring frictions:

- you can declare Java fixture classes *inside* the test that uses them, instead of adding `src/test/java/…` files that exist only to support one Groovy test;
- you can capture genuine `javac`-emitted bytecode for use cases such as comparing what the Groovy compiler produces against what Java produces for an equivalent source — useful when asserting joint-compilation behaviour, annotation-processing output, or `@CompileStatic` codegen equivalence.

A short fixture example:

```
import org.apache.groovy.util.JavaShell

def js = new JavaShell()
```

```
js.compile('fixtures.PojoFixture', """
    package fixtures;
    public class PojoFixture {
        private final String name;
        public PojoFixture(String name) { this.name = name; }
        public String getName() { return name; }
    }
""")                                                                    ①

// share JavaShell\'s class loader so Groovy code under test sees the fixture
def shell = new GroovyShell(js.classLoader)                             ②
shell.evaluate """
    def bean = fixtures.PojoFixture.newInstance('Groovy')
    assert bean.name == 'Groovy'
"""                                                                     ③
```

① compile a Java POJO in-memory; no `.java` file is added to the test source tree

② attach `JavaShell`'s class loader to a `GroovyShell` so the test's Groovy code can resolve the freshly-compiled class

③ exercise the fixture from Groovy

For a bytecode-equivalence assertion, `compileAllTo` writes the generated `.class` files to a temporary directory from which the bytes can be read and fed into a bytecode-diff tool of choice (ASM, `javap` output, etc.). The corresponding Groovy-side bytecode can be obtained through the `GroovyClassLoader` / `CompilationUnit` paths used elsewhere in Groovy's own test suite.

```
import org.apache.groovy.util.JavaShell
import java.nio.file.Files

def out = Files.createTempDirectory('bc-')
try {
    new JavaShell().compileAllTo('bc.Calc', """
        package bc;
        public class Calc { public static int add(int a, int b) { return a + b; } }
    """, out)                                                           ①
    byte[] javaBytes = Files.readAllBytes(out.resolve('bc/Calc.class'))  ②
    // Feed javaBytes (and bytes for the equivalent Groovy source) into ASM,
    // javap, or your bytecode-diff tool of choice for an equivalence assertion. ③
    assert javaBytes.length > 0
    assert javaBytes[0..3] == [(byte)0xCA, (byte)0xFE, (byte)0xBA, (byte)0xBE]
} finally {
    out.toFile().deleteDir()
}
```

① compile and write the Java class to a temp directory

② read the class bytes back from disk

③ compare against equivalent Groovy bytes obtained however best suits your test

Testing with JUnit

Groovy simplifies JUnit testing in the following ways:

- You use the same overall practices as you would when testing with Java but you can adopt much of Groovy's concise syntax in your tests making them succinct. You can even use the capabilities for writing testing domain specific languages (DSLs) if you feel so inclined.
- There are numerous helper classes that simplify many testing activities. The details differ in some cases depending on the version of JUnit you are using. We'll cover those details shortly.
- Groovy's PowerAssert mechanism is wonderful to use in your tests
- Groovy deems that tests are so important you should be able to run them as easily as scripts or classes. This is why Groovy includes an automatic test runner when using the `groovy` command or the GroovyConsole. This gives you some additional options over and above running your tests

In the following sections we will have a closer look at JUnit 3, 4 and 5 Groovy integration.

JUnit 3

Maybe one of the most prominent Groovy classes supporting JUnit 3 tests is the `GroovyTestCase` class. Being derived from `junit.framework.TestCase` it offers a bunch of additional methods that make testing in Groovy a breeze.

NOTE

Although `GroovyTestCase` inherits from `TestCase` doesn't mean you can't use JUnit 4 features in your project. In fact, the most recent Groovy versions come with a bundled JUnit 4 and that comes with a backwards compatible `TestCase` implementation. There have been some discussion on the Groovy mailing-list on whether to use `GroovyTestCase` or JUnit 4 with the result that it is mostly a matter of taste, but with `GroovyTestCase` you get a bunch of methods for free that make certain types of tests easier to write.

In this section, we will have a look at some of the methods provided by `GroovyTestCase`. A full list of these can be found in the JavaDoc documentation for groovy.test.GroovyTestCase , don't forget it is inherited from `junit.framework.TestCase` which inherits all the `assert*` methods.

Assertion Methods

`GroovyTestCase` is inherited from `junit.framework.TestCase` therefore it inherits a large number of assertion methods being available to be called in every test method:

```
class MyTestCase {

    @Test
    void testAssertions() {
        assertTrue(1 == 1)
        assertEquals("test", "test")

        def x = "42"
```

```
        assertNotNull "x must not be null", x
        assertNull null

        assertSame x, x
    }

}
```

As can be seen above, in contrast to Java it is possible to leave out the parenthesis in most situations which leads to even more readability of JUnit assertion method call expressions.

An interesting assertion method that is added by `GroovyTestCase` is `assertScript`. It ensures that the given Groovy code string succeeds without any exception:

```
@Test
void testScriptAssertions() {
    assertScript '''
        def x = 1
        def y = 2

        assert x + y == 3
    '''
}
```

shouldFail Methods

`shouldFail` can be used to check whether the given code block fails or not. In case it fails, the assertion does hold, otherwise the assertion fails:

```
@Test
void testInvalidIndexAccess1() {
    def numbers = [1,2,3,4]
    shouldFail {
        numbers.get(4)
    }
}
```

The example above uses the basic `shouldFail` method interface that takes a `groovy.lang.Closure` as a single argument. The `Closure` instance holds the code that is supposed to be breaking during runtime.

If we wanted to assert `shouldFail` on a specific `java.lang.Exception` type we could have done so by using the `shouldFail` implementation that takes the `Exception` class as first argument and the `Closure` as second argument:

```
@Test
void testInvalidIndexAccess2() {
```

```
    def numbers = [1,2,3,4]
    shouldFail IndexOutOfBoundsException, {
        numbers.get(4)
    }
}
```

If anything other than `IndexOutOfBoundsException` (or a descendant class of it) is thrown, the test case will fail.

A pretty nice feature of `shouldFail` hasn't been visible so far: it returns the exception message. This is really useful if you want to assert on the exception error message:

```
@Test
void testInvalidIndexAccess3() {
    def numbers = [1,2,3,4]
    def msg = shouldFail IndexOutOfBoundsException, {
        numbers.get(4)
    }
    assert msg.message.contains('Index: 4, Size: 4') ||
        msg.message.contains('Index 4 out-of-bounds for length 4') ||
        msg.message.contains('Index 4 out of bounds for length 4')
}
```

notYetImplemented Method

The `notYetImplemented` method has been greatly influenced by HtmlUnit. It allows to write a test method but mark it as not yet implemented. As long as the test method fails and is marked with `notYetImplemented` the test goes green:

```
@NotYetImplemented                  ①
@Test
void testNotYetImplemented1() {
    assert 1 == 2                   ②
}
```

① a call to `notYetImplemented` is necessary for `GroovyTestCase` to get the current method stack.

② as long as the test evaluates to `false` the test execution will be successful.

An alternative to the `notYetImplemented` method is the `@NotYetImplemented` annotation. It allows for annotating a method as not yet implemented, with the exact same behavior as `GroovyTestCase#notYetImplemented` but without the need for the `notYetImplemented` method call:

```
@NotYetImplemented
@Test
void testNotYetImplemented2() {
    assert 1 == 2
}
```

JUnit 4

Groovy can be used to write JUnit 4 test cases without any restrictions. The `groovy.test.GroovyAssert` holds various static methods that can be used as replacement for the `GroovyTestCase` methods in JUnit 4 tests:

```
import org.junit.jupiter.api.Test

import static groovy.test.GroovyAssert.shouldFail

final class JUnit4ExampleTests {

    @Test
    void indexOutOfBoundsAccess() {
        def numbers = [1,2,3,4]
        shouldFail {
            numbers.get(4)
        }
    }

}
```

As can be seen in the example above, the static methods found in `GroovyAssert` are imported at the beginning of the class definition thus `shouldFail` can be used the same way it can be used in a `GroovyTestCase`.

NOTE `groovy.test.GroovyAssert` descends from `org.junit.Assert` that means it inherits all JUnit assertion methods. However, with the introduction of the power assertion statement, it turned out to be *good practice to rely on assertion statements* instead of using the JUnit assertion methods with the improved message being the main reason.

It is worth mentioning that `GroovyAssert.shouldFail` is not absolutely identical to `GroovyTestCase.shouldFail`. While `GroovyTestCase.shouldFail` returns the exception message, `GroovyAssert.shouldFail` returns the exception itself. It takes a few more keystrokes to get the message, but in return you can access other properties and methods of the exception:

```
@Test
void shouldFailReturn() {
    def e = shouldFail {
        throw new RuntimeException('foo',
                                   new RuntimeException('bar'))
    }
    assert e instanceof RuntimeException
    assert e.message == 'foo'
    assert e.cause.message == 'bar'
}
```

JUnit 5

Much of the approach and helper classes described under JUnit4 apply when using JUnit5 however JUnit5 uses some slightly different class annotations when writing your tests. See the JUnit5 documentation for more details.

Create your test classes as per normal JUnit5 guidelines as shown in this example:

```
class MyTest {
  @Test
  void streamSum() {
    assertTrue(Stream.of(1, 2, 3)
      .mapToInt(i -> i)
      .sum() > 5, () -> "Sum should be greater than 5")
  }

  @RepeatedTest(value=2, name = "{displayName}
{currentRepetition}/{totalRepetitions}")
  void streamSumRepeated() {
    assert Stream.of(1, 2, 3).mapToInt(i -> i).sum() == 6
  }

  private boolean isPalindrome(s) { s == s.reverse()  }

  @ParameterizedTest                                                    ①
  @ValueSource(strings = [ "racecar", "radar", "able was I ere I saw elba" ])
  void palindromes(String candidate) {
    assert isPalindrome(candidate)
  }

  @TestFactory
  def dynamicTestCollection() {[
    dynamicTest("Add test") { -> assert 1 + 1 == 2 },
    dynamicTest("Multiply Test", () -> { assert 2 * 3 == 6 })
  ]}
}
```

① This test requires the additional `org.junit.jupiter:junit-jupiter-params` dependency if not already in your project.

You can run the tests in your IDE or build tool if it supports and is configured for JUnit5. If you run the above test in the GroovyConsole or via the `groovy` command, you will see a short text summary of the result of running the test:

```
JUnit5 launcher: passed=8, failed=0, skipped=0, time=246ms
```

More detailed information is available at the `FINE` logging level. You can configure your logging to display such information or do it programmatically as follows:

```
@BeforeAll
static void init() {
  def logger = Logger.getLogger(LoggingListener.name)
  logger.level = Level.FINE
  logger.addHandler(new ConsoleHandler(level: Level.FINE))
}
```

Testing with Spock

Spock is a testing and specification framework for Java and Groovy applications. What makes it stand out from the crowd is its beautiful and highly expressive specification DSL. In practice, Spock specifications are written as Groovy classes. Although written in Groovy they can be used to test Java classes. Spock can be used for unit, integration or BDD (behavior-driven-development) testing, it doesn't put itself into a specific category of testing frameworks or libraries.

NOTE	Beside these awesome features Spock is a good example on how to leverage advanced Groovy programming language features in third party libraries, for example, by using Groovy AST transformations.

NOTE	This section should not serve as detailed guide on how to use Spock, it should rather give an impression what Spock is about and how it can be leveraged for unit, integration, functional or any other type of testing.

The next section we will have a first look at the anatomy of a Spock specification. It should give a pretty good feeling on what Spock is up to.

Specifications

Spock lets you write specifications that describe features (properties, aspects) exhibited by a system of interest. The "system" can be anything between a single class and an entire application, a more advanced term for it is *system under specification*. The *feature description* starts from a specific snapshot of the system and its collaborators, this snapshot is called the *feature's fixture*.

Spock specification classes are derived from `spock.lang.Specification`. A concrete specification class might consist of fields, fixture methods, features methods and helper methods.

Let's have a look at a simple specification with a single feature method for an imaginary `Stack` class:

```
class StackSpec extends Specification {

    def "adding an element leads to size increase"() {  ①
        setup: "a new stack instance is created"        ②
            def stack = new Stack()

        when:                                           ③
            stack.push 42
```

```
        then:                                                ④
            stack.size() == 1
    }
}
```

① Feature method, is by convention named with a String literal.

② Setup block, here is where any setup work for this feature needs to be done.

③ When block describes a stimulus, a certain action under target by this feature specification.

④ Then block any expressions that can be used to validate the result of the code that was triggered by the when block.

Spock feature specifications are defined as methods inside a `spock.lang.Specification` class. They describe the feature by using a String literal instead of a method name.

A feature method holds multiple blocks, in our example we used `setup`, `when` and `then`. The `setup` block is special in that it is optional and allows to configure local variables visible inside the feature method. The `when` block defines the stimulus and is a companion of the `then` block which describes the response to the stimulus.

Note that the `setup` method in the `StackSpec` above additionally has a description String. Description Strings are optional and can be added after block labels (like `setup`, `when`, `then`).

More Spock

Spock provides much more features like data tables or advanced mocking capabilities. Feel free to consult the Spock GitHub page for more documentation and download information.

Functional Tests with Geb

Geb is a functional web testing and scraper library that integrates with JUnit and Spock. It is based upon the Selenium web drivers and, like Spock, provides a Groovy DSL to write functional tests for web applications.

Geb has great features that make it a good fit for a functional testing library:

- DOM access via a JQuery-like `$` function
- implements the *page pattern*
- support for modularization of certain web components (e.g. menu-bars, etc.) with *modules*
- integration with JavaScript via the JS variable

NOTE	This section should not serve as detailed guide on how to use Geb, it should rather give an impression what Geb is about and how it can be leveraged functional testing.

The next section will give an example on how Geb can be used to write a functional test for a simple web page with a single search field.

A Geb Script

Although Geb can be used standalone in a Groovy script, in many scenarios it's used in combination with other testing frameworks. Geb comes with various base classes that can be used in JUnit 3, 4, TestNG or Spock tests. The base classes are part of additional Geb modules that need to be added as a dependency.

For example, the following `@Grab` dependencies can be used to run Geb with the Selenium Firefox driver in JUnit4 tests. The module that is needed for JUnit 3/4 support is `geb-junit4`:

```
@Grab('org.gebish:geb-core:0.9.2')
@Grab('org.gebish:geb-junit4:0.9.2')
@Grab('org.seleniumhq.selenium:selenium-firefox-driver:2.26.0')
@Grab('org.seleniumhq.selenium:selenium-support:2.26.0')
```

The central class in Geb is the `geb.Browser` class. As its name implies it is used to browse pages and access DOM elements:

```
import geb.Browser
import org.openqa.selenium.firefox.FirefoxDriver

def browser = new Browser(driver: new FirefoxDriver(), baseUrl:
'http://myhost:8080/myapp')  ①
browser.drive {
    go "/login"                          ②

    $("#username").text = 'John'         ③
    $("#password").text = 'Doe'

    $("#loginButton").click()

    assert title == "My Application - Dashboard"
}
```

① A new `Browser` instance is created. In this case it uses the Selenium `FirefoxDriver` and sets the `baseUrl`.

② `go` is used to navigate to a URL or relative URI

③ `$` together with CSS selectors is used to access the `username` and `password` DOM fields.

The `Browser` class comes with a `drive` method that delegates all method/property calls to the current `browser` instance. The `Browser` configuration must not be done inline, it can also be externalized in a `GebConfig.groovy` configuration file for example. In practice, the usage of the `Browser` class is mostly hidden by Geb test base classes. They delegate all missing properties and method calls to the current `browser` instance that exists in the background:

```
class SearchTests extends geb.junit4.GebTest {
```

```
    @Test
    void executeSearch() {
        go 'http://somehost/mayapp/search'              ①
        $('#searchField').text = 'John Doe'             ②
        $('#searchButton').click()                      ③

        assert $('.searchResult a').first().text() == 'Mr. John Doe' ④
    }
}
```

① `Browser#go` takes a relative or absolute link and calls the page.

② `Browser#$` is used to access DOM content. Any CSS selectors supported by the underlying Selenium drivers are allowed

③ `click` is used to click a button.

④ `$` is used to get the first link out of the `searchResult` block

The example above shows a simple Geb web test with the JUnit 4 base class `geb.junit4.GebTest`. Note that in this case the `Browser` configuration is externalized. `GebTest` delegates methods like `go` and `$` to the underlying `browser` instance.

More Geb

In the previous section we only scratched the surface of the available Geb features. More information on Geb can be found at the project homepage.

Tune parsing performance of Parrot parser

The Parrot parser is based on antlr4 and introduced since Groovy 3.0.0. It provides the following options to tune parsing performance:

Option	Description	Default	Version	Example
groovy.parallel.parse	Parsing groovy source files in parallel.	`false` util Groovy 4.0.0	3.0.5+	-Dgroovy.parallel.parse=true

Option	Description	Default	Version	Example
groovy.antlr4.cache.threshold	antlr4 relies on DFA cache heavily for better performance, so antlr4 will not clear DFA cache, thus OutOfMemoryError will probably occur. Groovy trades off parsing performance and memory usage, when the count of Groovy source files parsed hits the cache threshold, the DFA cache will be cleared. **Note:** `0` means managing the DFA cache automatically since Groovy 5.0.0, `-1` means never clearing DFA cache, so requiring bigger JVM heap size. Or set a greater value, e.g. 200 to clear DFA cache if threshold hits. **Note:** the threshold specified is the count of groovy source files	`64`; `0` since Groovy 5.0.0	3.0.5+	-Dgroovy.antlr4.cache.threshold=200

Option	Description	Default	Version	Example
groovy.antlr4.sll.threshold	Parrot parser will try SLL mode and then try LL mode if SLL failed. But the more tokens to parse, the more likely SLL will fail. If SLL threshold hits, SLL will be skipped. Setting the threshold to `0` means never trying SLL mode, which is not recommended at most cases because SLL is the fastest mode though SLL is less powerful than LL. **Note:** the threshold specified is the token count	`-1` (disabled by default)	3.0.9+	-Dgroovy.antlr4.sll.threshold=1000
groovy.antlr4.clear.lexer.dfa.cache	Clear the DFA cache for lexer. The DFA cache for lexer is always small and important for parsing performance, so it's strongly recommended to leave it as it is until OutOfMemoryError will truly occur	`false`	3.0.9+	-Dgroovy.antlr4.clear.lexer.dfa.cache=true

Processing JSON

Groovy comes with integrated support for converting between Groovy objects and JSON. The classes dedicated to JSON serialization and parsing are found in the `groovy.json` package.

JsonSlurper

`JsonSlurper` is a class that parses JSON text or reader content into Groovy data structures (objects) such as maps, lists and primitive types like `Integer`, `Double`, `Boolean` and `String`.

The class comes with a bunch of overloaded `parse` methods plus some special methods such as `parseText`, `parseFile` and others. For the next example we will use the `parseText` method. It parses a JSON `String` and recursively converts it to a list or map of objects. The other `parse*` methods are similar in that they return a JSON `String` but for different parameter types.

```
def jsonSlurper = new JsonSlurper()
def object = jsonSlurper.parseText('{ "name": "John Doe" } /* some comment */')

assert object instanceof Map
assert object.name == 'John Doe'
```

Notice the result is a plain map and can be handled like a normal Groovy object instance. `JsonSlurper` parses the given JSON as defined by the ECMA-404 JSON Interchange Standard plus support for JavaScript comments and dates.

In addition to maps `JsonSlurper` supports JSON arrays which are converted to lists.

```
def jsonSlurper = new JsonSlurper()
def object = jsonSlurper.parseText('{ "myList": [4, 8, 15, 16, 23, 42] }')

assert object instanceof Map
assert object.myList instanceof List
assert object.myList == [4, 8, 15, 16, 23, 42]
```

The JSON standard supports the following primitive data types: string, number, object, `true`, `false` and `null`. `JsonSlurper` converts these JSON types into corresponding Groovy types.

```
def jsonSlurper = new JsonSlurper()
def object = jsonSlurper.parseText '''
    { "simple": 123,
      "fraction": 123.66,
      "exponential": 123e12
    }'''

assert object instanceof Map
assert object.simple.class == Integer
assert object.fraction.class == BigDecimal
assert object.exponential.class == BigDecimal
```

As `JsonSlurper` is returning pure Groovy object instances without any special JSON classes in the back, its usage is transparent. In fact, `JsonSlurper` results conform to GPath expressions. GPath is a powerful expression language that is supported by multiple slurpers for different data formats

(`XmlSlurper` for XML being one example).

NOTE	For more details please have a look at the section on GPath expressions.

The following table gives an overview of the JSON types and the corresponding Groovy data types:

JSON	Groovy
string	`java.lang.String`
number	`java.lang.BigDecimal` or `java.lang.Integer`
object	`java.util.LinkedHashMap`
array	`java.util.ArrayList`
true	`true`
false	`false`
null	`null`
date	`java.util.Date` based on the `yyyy-MM-dd'T'HH:mm:ssZ` date format

NOTE	Whenever a value in JSON is `null`, `JsonSlurper` supplements it with the Groovy `null` value. This is in contrast to other JSON parsers that represent a `null` value with a library-provided singleton object.

Parser Variants

`JsonSlurper` comes with a couple of parser implementations. Each parser fits different requirements, it could well be that for certain scenarios the `JsonSlurper` default parser is not the best bet for all situations. Here is an overview of the shipped parser implementations:

- The `JsonParserCharArray` parser basically takes a JSON string and operates on the underlying character array. During value conversion it copies character sub-arrays (a mechanism known as "chopping") and operates on them.
- The `JsonFastParser` is a special variant of the `JsonParserCharArray` and is the fastest parser. However, it is not the default parser for a reason. `JsonFastParser` is a so-called index-overlay parser. During parsing of the given JSON `String` it tries as hard as possible to avoid creating new char arrays or `String` instances. It keeps pointers to the underlying original character array only. In addition, it defers object creation as late as possible. If parsed maps are put into long-term caches care must be taken as the map objects might not be created and still consist of pointer to the original char buffer only. However, `JsonFastParser` comes with a special chop mode which dices up the char buffer early to keep a small copy of the original buffer. Recommendation is to use the `JsonFastParser` for JSON buffers under 2MB and keeping the long-term cache restriction in mind.
- The `JsonParserLax` is a special variant of the `JsonParserCharArray` parser. It has similar performance characteristics as `JsonFastParser` but differs in that it isn't exclusively relying on the ECMA-404 JSON grammar. For example it allows for comments, no quote strings etc.
- The `JsonParserUsingCharacterSource` is a special parser for very large files. It uses a technique called "character windowing" to parse large JSON files (large means files over 2MB size in this

case) with constant performance characteristics.

The default parser implementation for `JsonSlurper` is `JsonParserCharArray`. The `JsonParserType` enumeration contains constants for the parser implementations described above:

Implementation	Constant
`JsonParserCharArray`	`JsonParserType#CHAR_BUFFER`
`JsonFastParser`	`JsonParserType#INDEX_OVERLAY`
`JsonParserLax`	`JsonParserType#LAX`
`JsonParserUsingCharacterSource`	`JsonParserType#CHARACTER_SOURCE`

Changing the parser implementation is as easy as setting the `JsonParserType` with a call to `JsonSlurper#setType()`.

```
def jsonSlurper = new JsonSlurper(type: JsonParserType.INDEX_OVERLAY)
def object = jsonSlurper.parseText('{ "myList": [4, 8, 15, 16, 23, 42] }')

assert object instanceof Map
assert object.myList instanceof List
assert object.myList == [4, 8, 15, 16, 23, 42]
```

Date-like strings

`JsonParserType#INDEX_OVERLAY` and `JsonParserType#LAX` return a `java.util.Date` for a string in a full ISO-8601 or JSON-date form, rather than the `String` the other two parser types return for the same document. It is on by default and `JsonSlurper#setCheckDates(false)` turns it off:

```
def json = '{"when":"2026-09-04T10:00:00.000Z"}'

assert new JsonSlurper(type: JsonParserType.INDEX_OVERLAY)
        .parseText(json).when instanceof Date

assert new JsonSlurper(type: JsonParserType.INDEX_OVERLAY)
        .setCheckDates(false)
        .parseText(json).when instanceof String

assert new JsonSlurper(type: JsonParserType.CHAR_BUFFER)
        .parseText(json).when instanceof String        // never converts
```

A date carrying no time, such as `"2026-09-04"`, is left as a `String` by all four.

`JsonSlurper#setDateHandling(JsonDateHandling)` says which type to produce, rather than only whether to convert. A `Date` is an instant, as is an `Instant`, so neither keeps an offset the document carried; `OFFSET_DATE_TIME` does:

```
def offsetJson = '{"when":"2026-09-04T10:00:00+10:00"}'
```

```
def slurp = { JsonDateHandling h ->
    new JsonSlurper(type: JsonParserType.INDEX_OVERLAY)
            .setDateHandling(h)
            .parseText(offsetJson).when
}

assert slurp(JsonDateHandling.STRING) instanceof String
assert slurp(JsonDateHandling.UTIL_DATE) instanceof Date              // the default
assert slurp(JsonDateHandling.INSTANT) instanceof Instant

// only OFFSET_DATE_TIME keeps the offset the document carried
def odt = slurp(JsonDateHandling.OFFSET_DATE_TIME)
assert odt instanceof OffsetDateTime
assert odt.offset == ZoneOffset.ofHours(10)
```

`setCheckDates(boolean)` remains, mapping onto `UTIL_DATE` and `STRING`, and is deprecated. Once `setDateHandling` has been called on a slurper the boolean has no further effect on it, whichever order the two are used in, so a choice made through the wider option is not undone by the narrower one.

JsonOutput

`JsonOutput` is responsible for serializing Groovy objects into JSON strings. It can be seen as companion object to JsonSlurper, being a JSON parser.

`JsonOutput` comes with overloaded, static `toJson` methods. Each `toJson` implementation takes a different parameter type. The static methods can either be used directly or by importing the methods with a static import statement.

The result of a `toJson` call is a `String` containing the JSON code.

```
def json = JsonOutput.toJson([name: 'John Doe', age: 42])

assert json == '{"name":"John Doe","age":42}'
```

`JsonOutput` does not only support primitive, maps or list data types to be serialized to JSON, it goes further and even has support for serializing POGOs, that is, plain-old Groovy objects.

```
class Person { String name }

def json = JsonOutput.toJson([ new Person(name: 'John'), new Person(name: 'Max') ])

assert json == '[{"name":"John"},{"name":"Max"}]'
```

Customizing Output

If you need control over the serialized output you can use a `JsonGenerator`. The

`JsonGenerator.Options` builder can be used to create a customized generator. One or more options can be set on this builder in order to alter the resulting output. When you are done setting the options simply call the `build()` method in order to get a fully configured instance that will generate output based on the options selected.

```
class Person {
    String name
    String title
    int age
    String password
    Date dob
    URL favoriteUrl
}

Person person = new Person(name: 'John', title: null, age: 21, password: 'secret',
                           dob: Date.parse('yyyy-MM-dd', '1984-12-15'),
                           favoriteUrl: new URL('http://groovy-lang.org/'))

def generator = new JsonGenerator.Options()
    .excludeNulls()
    .dateFormat('yyyy@MM')
    .excludeFieldsByName('age', 'password')
    .excludeFieldsByType(URL)
    .build()

assert generator.toJson(person) == '{"name":"John","dob":"1984@12"}'
```

A closure can be used to transform a type. These closure converters are registered for a given type and will be called any time that type or a subtype is encountered. The first parameter to the closure is an object matching the type for which the converter is registered and this parameter is required. The closure may take an optional second `String` parameter and this will be set to the key name if one is available.

```
class Person {
    String name
    URL favoriteUrl
}

Person person = new Person(name: 'John', favoriteUrl: new URL('http://groovy-
lang.org/json.html#_jsonoutput'))

def generator = new JsonGenerator.Options()
    .addConverter(URL) { URL u, String key ->
        if (key == 'favoriteUrl') {
            u.getHost()
        } else {
            u
        }
    }
```

```
    .build()

assert generator.toJson(person) == '{"name":"John","favoriteUrl":"groovy-lang.org"}'

// No key available when generating a JSON Array
def list = [new URL('http://groovy-lang.org/json.html#_jsonoutput')]
assert generator.toJson(list) == '["http://groovy-lang.org/json.html#_jsonoutput"]'

// First parameter to the converter must match the type for which it is registered
shouldFail(IllegalArgumentException) {
    new JsonGenerator.Options()
        .addConverter(Date) { Calendar cal -> }
}
```

Formatted Output

As we saw in previous examples, the JSON output is not pretty printed per default. However, the `prettyPrint` method in `JsonOutput` comes to rescue for this task.

```
def json = JsonOutput.toJson([name: 'John Doe', age: 42])

assert json == '{"name":"John Doe","age":42}'

assert JsonOutput.prettyPrint(json) == '''\
{
    "name": "John Doe",
    "age": 42
}'''.stripIndent()
```

`prettyPrint` takes a `String` as single parameter; therefore, it can be applied on arbitrary JSON `String` instances, not only the result of `JsonOutput.toJson`.

Typed JSON Parsing

Since `JsonSlurper` parses JSON values into their natural JVM types (`Integer`, `BigDecimal`, `Boolean`, `List`, `Map`), Groovy's built-in `as` coercion works well for converting parsed JSON directly into typed objects — no additional dependencies are required:

```
static class ServerConfig {
    String host
    int port
    boolean debug
}

static class AppConfig {
    String name
    ServerConfig server
}
```

```
enum Color { RED, GREEN, BLUE }

static class Item { String name; Color color }
```

```
def json = '{"host":"localhost","port":8080,"debug":true}'
def config = new JsonSlurper().parseText(json) as ServerConfig

assert config instanceof ServerConfig
assert config.host == 'localhost'
assert config.port == 8080
assert config.debug == true
```

Nested objects are automatically coerced:

```
def json = '{"name":"myapp","server":{"host":"localhost","port":9090,"debug":false}}'
def config = new JsonSlurper().parseText(json) as AppConfig

assert config.name == 'myapp'
assert config.server instanceof ServerConfig
assert config.server.host == 'localhost'
assert config.server.port == 9090
```

Enum values are also handled:

```
def item = new JsonSlurper().parseText('{"name":"widget","color":"GREEN"}') as Item
assert item.color == Color.GREEN
```

This covers most common use cases. For advanced scenarios — such as typed element collections (`List<Item>`), date/time parsing, or annotation-driven property mapping (`@JsonProperty`, `@JsonFormat`) — use `jackson-databind` directly:

```
@Grab('com.fasterxml.jackson.core:jackson-databind')
import com.fasterxml.jackson.databind.ObjectMapper

def config = new ObjectMapper().readValue(jsonString, ServerConfig)
```

Builders

Another way to create JSON from Groovy is to use `JsonBuilder` or `StreamingJsonBuilder`. Both builders provide a DSL which allows to formulate an object graph which is then converted to JSON.

NOTE	For more details on builders, have a look at the builders chapter which covers both JsonBuilder and StreamingJsonBuilder.

Interacting with a SQL database

Groovy's `groovy-sql` module provides a higher-level abstraction over Java's JDBC technology. JDBC itself provides a lower-level but fairly comprehensive API which provides uniform access to a whole variety of supported relational database systems. We'll use HSQLDB in our examples here but you can alternatively use Oracle, SQL Server, MySQL and a host of others. The most frequently used class within the `groovy-sql` module is the `groovy.sql.Sql` class which raises the JDBC abstractions up one level. We'll cover that first.

Connecting to the database

Connecting to a database with Groovy's `Sql` class requires four pieces of information:

- The database uniform resource locator (URL)
- Username
- Password
- The driver class name (which can be derived automatically in some situations)

For our HSQLDB database, the values will be something like that shown in the following table:

Property	**Value**
url	`jdbc:hsqldb:mem:yourdb`
user	sa (or your *username*)
password	*yourPassword*
driver	`org.hsqldb.jdbcDriver`

Consult the documentation for the JDBC driver that you plan to use to determine the correct values for your situation.

The `Sql` class has a `newInstance` factory method which takes these parameters. You would typically use it as follows:

Connecting to HSQLDB

```
import groovy.sql.Sql

def url = 'jdbc:hsqldb:mem:yourDB'
def user = 'sa'
def password = ''
def driver = 'org.hsqldb.jdbcDriver'
def sql = Sql.newInstance(url, user, password, driver)

// use 'sql' instance ...

sql.close()
```

If you don't want to have to handle resource handling yourself (i.e. call `close()` manually) then you can use the `withInstance` variation as shown here:

Connecting to HSQLDB (`withInstance` variation)

```
Sql.withInstance(url, user, password, driver) { sql ->
  // use 'sql' instance ...
}
```

Connecting with a DataSource

It is often preferred to use a DataSource. You may have one available to you from a connection pool. Here we'll use the one provided as part of the HSQLDB driver jar as shown here:

Connecting to HSQLDB with a DataSource

```
import groovy.sql.Sql
import org.hsqldb.jdbc.JDBCDataSource

def dataSource = new JDBCDataSource(
    database: 'jdbc:hsqldb:mem:yourDB', user: 'sa', password: '')
def sql = new Sql(dataSource)

// use then close 'sql' instance ...
```

If you have your own connection pooling, the details will be different, e.g. for Apache Commons DBCP:

Connecting to HSQLDB with a DataSource using Apache Commons DBCP

```
@Grab('org.apache.commons:commons-dbcp2:2.7.0')
import groovy.sql.Sql
import org.apache.commons.dbcp2.BasicDataSource

def ds = new BasicDataSource(driverClassName: "org.hsqldb.jdbcDriver",
    url: 'jdbc:hsqldb:mem:yourDB', username: 'sa', password: '')
def sql = new Sql(ds)
// use then close 'sql' instance ...
```

Connecting using @Grab

The previous examples assume that the necessary database driver jar is already on your classpath. For a self-contained script you can add `@Grab` statements to the top of the script to automatically download the necessary jar as shown here:

Connecting to HSQLDB using @Grab

```
@Grab('org.hsqldb:hsqldb:2.7.4')
@GrabConfig(systemClassLoader=true)
```

```
// create, use, and then close sql instance ...
```

The `@GrabConfig` statement is necessary to make sure the system classloader is used. This ensures that the driver classes and system classes like `java.sql.DriverManager` are in the same classloader.

Executing SQL

You can execute arbitrary SQL commands using the `execute()` method. Let's have a look at using it to create a table.

Creating tables

The simplest way to execute SQL is to call the `execute()` method passing the SQL you wish to execute as a String as shown here:

Creating a table

```
// ... create 'sql' instance
sql.execute '''
  CREATE TABLE Author (
    id          INTEGER GENERATED BY DEFAULT AS IDENTITY,
    firstname   VARCHAR(64),
    lastname    VARCHAR(64)
  );
'''
// close 'sql' instance ...
```

There is a variant of this method which takes a GString and another with a list of parameters. There are also other variants with similar names: `executeInsert` and `executeUpdate`. We'll see examples of these variants in other examples in this section.

Basic CRUD operations

The basic operations on a database are Create, Read, Update and Delete (the so-called CRUD operations). We'll examine each of these in turn.

Creating/Inserting data

You can use the same `execute()` statement we saw earlier but to insert a row by using a SQL insert statement as follows:

Inserting a row

```
sql.execute "INSERT INTO Author (firstname, lastname) VALUES ('Dierk', 'Koenig')"
```

You can use a special `executeInsert` method instead of `execute`. This will return a list of all keys generated. Both the `execute` and `executeInsert` methods allow you to place '?' placeholders into your SQL string and supply a list of parameters. In this case a PreparedStatement is used which avoids any risk of SQL injection. The following example illustrates `executeInsert` using placeholders and

parameters:

Inserting a row using executeInsert with placeholders and parameters

```
def insertSql = 'INSERT INTO Author (firstname, lastname) VALUES (?,?)'
def params = ['Jon', 'Skeet']
def keys = sql.executeInsert insertSql, params
assert keys[0] == [1]
```

In addition, both the `execute` and `executeInsert` methods allow you to use GStrings. Any '$' placeholders within the SQL are assumed to be placeholders. An escaping mechanism exists if you want to supply part of the GString with a variable in a position which isn't where normal placeholders go within SQL. See the GroovyDoc for more details.

WARNING

Do **not** surround an interpolated GString expression with SQL quotes, for example `sql.rows("select * from PERSON where name = '${name}'")`. The surrounding quotes prevent Groovy from binding the value via a JDBC `PreparedStatement` placeholder, which would reopen a SQL injection hole. Simply remove the quotes (`sql.rows("select * from PERSON where name = $name")`) and the value will be bound safely as a parameter.

For secure-by-default behaviour, Groovy **rejects** such quoted dynamic expressions by throwing a `java.sql.SQLException`. The previous (insecure) behaviour of inlining the value via string concatenation can be restored by setting the `groovy.sql.injection.lenient` system property to `true`, but this is strongly discouraged.

Also, `executeInsert` allows you to supply a list of key column names, when multiple keys are returned and you are only interested in some of them. Here is a fragment illustrating key name specification and GStrings:

Inserting a row using executeInsert with a GString and specifying key names

```
def first = 'Guillaume'
def last = 'Laforge'
def myKeyNames = ['ID']
def myKeys = sql.executeInsert """
  INSERT INTO Author (firstname, lastname)
  VALUES (${first}, ${last})
""", myKeyNames
assert myKeys[0] == [ID: 2]
```

Reading rows

Reading rows of data from the database is accomplished using one of several available methods: `query`, `eachRow`, `firstRow` and `rows`.

Use the `query` method if you want to iterate through the `ResultSet` returned by the underlying JDBC API as shown here:

Reading data using `query`

```
def expected = ['Dierk Koenig', 'Jon Skeet', 'Guillaume Laforge']

def rowNum = 0
sql.query('SELECT firstname, lastname FROM Author') { resultSet ->
  while (resultSet.next()) {
    def first = resultSet.getString(1)
    def last = resultSet.getString('lastname')
    assert expected[rowNum++] == "$first $last"
  }
}
```

Use the `eachRow` method if you want a slightly higher-level abstraction which provides a Groovy friendly map-like abstraction for the `ResultSet` as shown here:

Reading data using `eachRow`

```
rowNum = 0
sql.eachRow('SELECT firstname, lastname FROM Author') { row ->
  def first = row[0]
  def last = row.lastname
  assert expected[rowNum++] == "$first $last"
}
```

Note that you can use Groovy list-style and map-style notations when accessing the row of data.

Use the `firstRow` method if you for similar functionality as `eachRow` but returning only one row of data as shown here:

Reading data using `firstRow`

```
def first = sql.firstRow('SELECT lastname, firstname FROM Author')
assert first.values().sort().join(',') == 'Dierk,Koenig'
```

Use the `rows` method if you want to process a list of map-like data structures as shown here:

Reading data using `rows`

```
List authors = sql.rows('SELECT firstname, lastname FROM Author')
assert authors.size() == 3
assert authors.collect { "$it.FIRSTNAME ${it[-1]}" } == expected
```

Note that the map-like abstraction has case-insensitive keys (hence we can use 'FIRSTNAME' or 'firstname' as the key) and also that -ve indices (a standard Groovy feature) works when using an index value (to count column numbers from the right).

You can also use any of the above methods to return scalar values, though typically `firstRow` is all that is required in such cases. An example returning the count of rows is shown here:

Reading scalar values

```
assert sql.firstRow('SELECT COUNT(*) AS num FROM Author').num == 3
```

Updating rows

Updating rows can again be done using the `execute()` method. Just use a SQL update statement as the argument to the method. You can insert an author with just a lastname and then update the row to also have a firstname as follows:

Updating a row

```
sql.execute "INSERT INTO Author (lastname) VALUES ('Thorvaldsson')"
sql.execute "UPDATE Author SET firstname='Erik' where lastname='Thorvaldsson'"
```

There is also a special `executeUpdate` variant which returns the number of rows updated as a result of executing the SQL. For example, you can change the lastname of an author as follows:

Using executeUpdate

```
def updateSql = "UPDATE Author SET lastname='Pragt' where lastname='Thorvaldsson'"
def updateCount = sql.executeUpdate updateSql
assert updateCount == 1

def row = sql.firstRow "SELECT * FROM Author where firstname = 'Erik'"
assert "${row.firstname} ${row.lastname}" == 'Erik Pragt'
```

Deleting rows

The `execute` method is also used for deleting rows as this example shows:

Deleting rows

```
assert sql.firstRow('SELECT COUNT(*) as num FROM Author').num == 3
sql.execute "DELETE FROM Author WHERE lastname = 'Skeet'"
assert sql.firstRow('SELECT COUNT(*) as num FROM Author').num == 2
```

Advanced SQL operations

Working with transactions

The easiest way to perform database operations within a transaction is to include the database operation within a `withTransaction` closure as shown in the following example:

A successful transaction

```
assert sql.firstRow('SELECT COUNT(*) as num FROM Author').num == 0
sql.withTransaction {
```

```
  sql.execute "INSERT INTO Author (firstname, lastname) VALUES ('Dierk', 'Koenig')"
  sql.execute "INSERT INTO Author (firstname, lastname) VALUES ('Jon', 'Skeet')"
}
assert sql.firstRow('SELECT COUNT(*) as num FROM Author').num == 2
```

Here the database starts empty and has two rows after successful completion of the operation. Outside the scope of the transaction, the database is never seen as having just one row.

If something goes wrong, any earlier operations within the `withTransaction` block are rolled back. We can see that in operation in the following example where we use database metadata (more details coming up shortly) to find the maximum allowable size of the `firstname` column and then attempt to enter a firstname one larger than that maximum value as shown here:

A failed transaction will cause a rollback

```
def maxFirstnameLength
def metaClosure = { meta -> maxFirstnameLength = meta.getPrecision(1) }
def rowClosure = {}
def rowCountBefore = sql.firstRow('SELECT COUNT(*) as num FROM Author').num
try {
  sql.withTransaction {
    sql.execute "INSERT INTO Author (firstname) VALUES ('Dierk')"
    sql.eachRow "SELECT firstname FROM Author WHERE firstname = 'Dierk'", metaClosure,
rowClosure
    sql.execute "INSERT INTO Author (firstname) VALUES (?)", 'X' * (maxFirstnameLength
+ 1)
  }
} catch(ignore) { println ignore.message }
def rowCountAfter = sql.firstRow('SELECT COUNT(*) as num FROM Author').num
assert rowCountBefore == rowCountAfter
```

Even though the first sql execute succeeds initially, it will be rolled back and the number of rows will remain the same.

Using batches

When dealing with large volumes of data, particularly when inserting such data, it can be more efficient to chunk the data into batches. This is done using the `withBatch` statement as shown in the following example:

Batching SQL statements

```
sql.withBatch(3) { stmt ->
  stmt.addBatch "INSERT INTO Author (firstname, lastname) VALUES ('Dierk', 'Koenig')"
  stmt.addBatch "INSERT INTO Author (firstname, lastname) VALUES ('Paul', 'King')"
  stmt.addBatch "INSERT INTO Author (firstname, lastname) VALUES ('Guillaume',
'Laforge')"
  stmt.addBatch "INSERT INTO Author (firstname, lastname) VALUES ('Hamlet',
'D''Arcy')"
  stmt.addBatch "INSERT INTO Author (firstname, lastname) VALUES ('Cedric',
```

```
'Champeau')"
  stmt.addBatch "INSERT INTO Author (firstname, lastname) VALUES ('Erik', 'Pragt')"
  stmt.addBatch "INSERT INTO Author (firstname, lastname) VALUES ('Jon', 'Skeet')"
}
```

After executing these statements, there will be 7 new rows in the database. In fact, they will have been added in batches even though you can't easily tell that after that fact. If you want to confirm what is going on under the covers, you can add a little bit of extra logging into your program. Add the following lines before the `withBatch` statement:

Logging additional SQL information

```
import java.util.logging.*

// next line will add fine logging
Logger.getLogger('groovy.sql').level = Level.FINE
// also adjust logging.properties file in JRE_HOME/lib to have:
// java.util.logging.ConsoleHandler.level = FINE
```

With this extra logging turned on, and the changes made as per the above comment for the logging.properties file, you should see output such as:

SQL logging output with batching enable

```
FINE: Successfully executed batch with 3 command(s)
Apr 19, 2015 8:38:42 PM groovy.sql.BatchingStatementWrapper processResult

FINE: Successfully executed batch with 3 command(s)
Apr 19, 2015 8:38:42 PM groovy.sql.BatchingStatementWrapper processResult

FINE: Successfully executed batch with 1 command(s)
Apr 19, 2015 8:38:42 PM groovy.sql.Sql getStatement
```

We should also note, that any combination of SQL statements can be added to the batch. They don't all have to be inserting a new row to the same table.

We noted earlier that to avoid SQL injection, we encourage you to use prepared statements, this is achieved using the variants of methods which take GStrings or a list of extra parameters. Prepared statements can be used in combination with batches as shown in the following example:

Batching prepared statements

```
def qry = 'INSERT INTO Author (firstname, lastname) VALUES (?,?)'
sql.withBatch(3, qry) { ps ->
  ps.addBatch('Dierk', 'Koenig')
  ps.addBatch('Paul', 'King')
  ps.addBatch('Guillaume', 'Laforge')
  ps.addBatch('Hamlet', "D'Arcy")
  ps.addBatch('Cedric', 'Champeau')
```

```
  ps.addBatch('Erik', 'Pragt')
  ps.addBatch('Jon', 'Skeet')
}
```

This provides a much safer option if the data could come from a user such as via a script or a web form. Of course, given that a prepared statement is being used, you are limited to a batch of the same SQL operation (insert in our example) to the one table.

Performing pagination

When presenting large tables of data to a user, it is often convenient to present information a page at a time. Many of Groovy's SQL retrieval methods have extra parameters which can be used to select a particular page of interest. The starting position and page size are specified as integers as shown in the following example using `rows`:

Retrieving pages of data

```
def qry = 'SELECT * FROM Author'
assert sql.rows(qry, 1, 3)*.firstname == ['Dierk', 'Paul', 'Guillaume']
assert sql.rows(qry, 4, 3)*.firstname == ['Hamlet', 'Cedric', 'Erik']
assert sql.rows(qry, 7, 3)*.firstname == ['Jon']
```

Fetching metadata

JDBC metadata can be retrieved in numerous ways. Perhaps the most basic approach is to extract the metadata from any row as shown in the following example which examines the tablename, column names and column type names:

Using row metadata

```
sql.eachRow("SELECT * FROM Author WHERE firstname = 'Dierk'") { row ->
  def md = row.getMetaData()
  assert md.getTableName(1) == 'AUTHOR'
  assert (1..md.columnCount).collect{ md.getColumnName(it) } == ['ID', 'FIRSTNAME',
'LASTNAME']
  assert (1..md.columnCount).collect{ md.getColumnTypeName(it) } == ['INTEGER',
'VARCHAR', 'VARCHAR']
}
```

And another slight variant to the previous example, this time also looking at the column label:

Also using row metadata

```
sql.eachRow("SELECT firstname AS first FROM Author WHERE firstname = 'Dierk'") { row
->
  def md = row.getMetaData()
  assert md.getColumnName(1) == 'FIRSTNAME'
  assert md.getColumnLabel(1) == 'FIRST'
```

```
}
```

Accessing metadata is quite common, so Groovy also provides variants to many of its methods that let you supply a closure that will be called once with the row metadata in addition to the normal row closure which is called for each row. The following example illustrates the two closure variant for `eachRow`:

Using row and metadata closures

```
def metaClosure = { meta -> assert meta.getColumnName(1) == 'FIRSTNAME' }
def rowClosure = { row -> assert row.FIRSTNAME == 'Dierk' }
sql.eachRow("SELECT firstname FROM Author WHERE firstname = 'Dierk'", metaClosure,
rowClosure)
```

Note that our SQL query will only return one row, so we could have equally used `firstRow` for the previous example.

Finally, JDBC also provides metadata per connection (not just for rows). You can also access such metadata from Groovy as shown in this example:

Using connection metadata

```
def md = sql.connection.metaData
assert md.driverName == 'HSQL Database Engine Driver'
assert md.databaseProductVersion == '2.7.4'
assert ['JDBCMajorVersion', 'JDBCMinorVersion'].collect{ md[it] } == [4, 2]
assert md.stringFunctions.tokenize(',').contains('CONCAT')
def rs = md.getTables(null, null, 'AUTH%', null)
assert rs.next()
assert rs.getString('TABLE_NAME').startsWith('AUTHOR')
```

Consult the JavaDoc for your driver to find out what metadata information is available for you to access.

Named and named-ordinal parameters

Groovy supports some additional alternative placeholder syntax variants. The GString variants are typically preferred over these alternatives but the alternatives are useful for Java integration purposes and sometimes in templating scenarios where GStrings might already be in heavy use as part of a template. The named parameter variants are much like the String plus list of parameter variants but instead of having a list of `?` placeholders followed by a list of parameters, you have one or more placeholders having the form `:propName` or `?.propName` and a single map, named arguments or a domain object as the parameter. The map or domain object should have a property named `propName` corresponding to each supplied placeholder.

Here is an example using the colon form:

Named parameters (colon form)

```
sql.execute "INSERT INTO Author (firstname, lastname) VALUES (:first, :last)", first:
'Dierk', last: 'Koenig'
```

And another example using the question mark form:

Named parameters (question mark form)

```
sql.execute "INSERT INTO Author (firstname, lastname) VALUES (?.first, ?.last)",
first: 'Jon', last: 'Skeet'
```

If the information you need to supply is spread across multiple maps or domain objects you can use the question mark form with an additional ordinal index as shown here:

Named-ordinal parameters

```
class Rockstar { String first, last }
def pogo = new Rockstar(first: 'Paul', last: 'McCartney')
def map = [lion: 'King']
sql.execute "INSERT INTO Author (firstname, lastname) VALUES (?1.first, ?2.lion)",
pogo, map
```

Collection parameters for IN clauses

JDBC drivers generally don't support binding a collection directly to a single `?` placeholder in an `IN` clause — a query like `WHERE id IN (?)` with a list parameter either fails or, with GString parameters, binds the whole collection as a single value (which matches no rows).

To pass a variable-length list of values to an `IN` clause, wrap the collection with `Sql.inList(…)`. The marker expands at prepare time to the correct number of placeholders and splices the values into the parameter list. It works across all three parameter styles — positional, GString, and named:

IN clause using positional parameters

```
def names = ['Dierk', 'Jon']
def rows = sql.rows('SELECT lastname FROM Author WHERE firstname IN (?)',
                    [Sql.inList(names)])
assert rows*.lastname.sort() == ['Koenig', 'Skeet']
```

IN clause using GString parameters

```
def picks = ['Dierk', 'Guillaume']
def rows2 = sql.rows "SELECT lastname FROM Author WHERE firstname IN
(${Sql.inList(picks)})"
assert rows2*.lastname.sort() == ['Koenig', 'Laforge']
```

IN clause using named parameters

```
def rows3 = sql.rows('SELECT lastname FROM Author WHERE firstname IN (:firsts)',
                     [firsts: Sql.inList(['Jon', 'Guillaume'])])
assert rows3*.lastname.sort() == ['Laforge', 'Skeet']
```

`Sql.inList(…)` rejects null or empty collections with an `IllegalArgumentException` — callers must guard empty input explicitly. This matches the behaviour of Hibernate and Spring's `NamedParameterJdbcTemplate`, and makes "the form selected nothing" bugs easy to spot rather than silently returning zero rows. A typical guard looks like:

Guarding against empty input

```
def selected = []    // e.g. nothing selected on a form
if (selected) {
    sql.rows('SELECT lastname FROM Author WHERE firstname IN (?)',
             [Sql.inList(selected)])
} else {
    // skip the query, or build different SQL — Sql.inList([]) throws.
}
```

Stored procedures

The exact syntax for creating a stored procedure or function varies slightly between different databases. For the HSQLDB database we are using, we can create a stored function which returns the initials of all authors in a table as follows:

Creating a stored function

```
sql.execute """
  CREATE FUNCTION SELECT_AUTHOR_INITIALS()
  RETURNS TABLE (firstInitial VARCHAR(1), lastInitial VARCHAR(1))
  READS SQL DATA
  RETURN TABLE (
    SELECT LEFT(Author.firstname, 1) as firstInitial, LEFT(Author.lastname, 1) as
lastInitial
    FROM Author
  )
"""
```

We can use a SQL `CALL` statement to invoke the function using Groovy's normal SQL retrieval methods. Here is an example using `eachRow`.

Creating a stored procedure or function

```
def result = []
sql.eachRow('CALL SELECT_AUTHOR_INITIALS()') {
  result << "$it.firstInitial$it.lastInitial"
}
```

```
assert result == ['DK', 'JS', 'GL']
```

Here is the code for creating another stored function, this one taking the lastname as a parameter:

Creating a stored function with a parameter

```
sql.execute """
  CREATE FUNCTION FULL_NAME (p_lastname VARCHAR(64))
  RETURNS VARCHAR(100)
  READS SQL DATA
  BEGIN ATOMIC
    DECLARE ans VARCHAR(100);
    SELECT CONCAT(firstname, ' ', lastname) INTO ans
    FROM Author WHERE lastname = p_lastname;
    RETURN ans;
  END
"""
```

We can use the placeholder syntax to specify where the parameter belongs and note the special placeholder position to indicate the result:

Using a stored function with a parameter

```
def result = sql.firstRow("{? = call FULL_NAME(?)}", ['Koenig'])
assert result[0] == 'Dierk Koenig'
```

Finally, here is a stored procedure with input and output parameters:

Creating a stored procedure with input and output parameters

```
sql.execute """
  CREATE PROCEDURE CONCAT_NAME (OUT fullname VARCHAR(100),
    IN first VARCHAR(50), IN last VARCHAR(50))
  BEGIN ATOMIC
    SET fullname = CONCAT(first, ' ', last);
  END
"""
```

To use the `CONCAT_NAME` stored procedure parameter, we make use of a special `call` method. Any input parameters are simply provided as parameters to the method call. For output parameters, the resulting type must be specified as shown here:

Using a stored procedure with input and output parameters

```
sql.call("{call CONCAT_NAME(?, ?, ?)}", [Sql.VARCHAR, 'Dierk', 'Koenig']) {
  fullname -> assert fullname == 'Dierk Koenig'
}
```

The `call`, `callWithRows`, and `callWithAllRows` methods also accept a `Map` of named parameters. Use `{call NAME(:paramName)}` placeholders in the SQL and supply the values — including any output-parameter type markers — as map entries. Map iteration order doesn't matter; bindings are resolved by name against the `:name` tokens in the SQL:

Using a stored procedure with a map of named parameters

```
sql.call('{call CONCAT_NAME(:fullname, :first, :last)}',
         [fullname: Sql.VARCHAR, first: 'Dierk', last: 'Koenig']) {
  fullname -> assert fullname == 'Dierk Koenig'
}
```

The same overloads support Groovy's named-argument call syntax (map entries written as `name: value` pairs before the SQL string):

Using a stored procedure with Groovy named-argument syntax

```
sql.call(fullname: Sql.VARCHAR, first: 'Dierk', last: 'Koenig',
         '{call CONCAT_NAME(:fullname, :first, :last)}') {
  fullname -> assert fullname == 'Dierk Koenig'
}
```

Creating a stored procedure with an input/output parameter

```
sql.execute """
  CREATE PROCEDURE CHECK_ID_POSITIVE_IN_OUT ( INOUT p_err VARCHAR(64), IN pparam
INTEGER, OUT re VARCHAR(15))
  BEGIN ATOMIC
    IF pparam > 0 THEN
      set p_err = p_err || '_OK';
      set re = 'RET_OK';
    ELSE
      set p_err = p_err || '_ERROR';
      set re = 'RET_ERROR';
    END IF;
  END;
"""
```

Using a stored procedure with an input/output parameter

```
def scall = "{call CHECK_ID_POSITIVE_IN_OUT(?, ?, ?)}"
sql.call scall, [Sql.inout(Sql.VARCHAR("MESSAGE")), 1, Sql.VARCHAR], {
  res, p_err -> assert res == 'MESSAGE_OK' && p_err == 'RET_OK'
}
```

Using DataSets

Groovy provides a groovy.sql.DataSet class which enhances the groovy.sql.Sql class with what can

be thought of as mini ORM functionality. Databases are accessed and queried using POGO fields and operators rather than JDBC-level API calls and RDBMS column names.

So, instead of a query like:

```
def qry = """SELECT * FROM Author
  WHERE (firstname > ?)
  AND (lastname < ?)
  ORDER BY lastname DESC"""
def params = ['Dierk', 'Pragt']
def result = sql.rows(qry, params)
assert result*.firstname == ['Eric', 'Guillaume', 'Paul']
```

You can write code like this:

```
def authorDS = sql.dataSet('Author')
def result = authorDS.findAll{ it.firstname > 'Dierk' }
        .findAll{ it.lastname < 'Pragt' }
        .sort{ it.lastname }
        .reverse()
assert result.rows()*.firstname == ['Eric', 'Guillaume', 'Paul']
```

Here we have a helper "domain" class:

```
class Author {
    String firstname
    String lastname
}
```

Database access and manipulation involves creating or working with instances of the domain class.

Querying collections in SQL-like style

GINQ (Groovy-Integrated Query) lets you query in-memory collections using familiar SQL-like syntax. It also works with parsed XML, JSON, YAML, and other formats that produce collections.

The heart of GINQ is its query-like expressions, which can be very simple:

```
from n in [0, 1, 2]
select n
```

Or involve multiple clauses:

```
from s in sales
where s.amount > 50
```

```
groupby s.customer as customer into g
having g.count() > 1
orderby g.customer
select g.customer, g.count() as items, g.sum(s -> s.amount) as total
```

A GINQ expression can include the following clauses:

- `from` — the data source (required). Accepts any `Iterable`, `Stream`, array, or GINQ result set.
- `join` — combines data from additional sources (`join`, `leftjoin`, `rightjoin`, `fulljoin`, `crossjoin`), matched using an `on` condition.
- `where` — filters rows before grouping, as `s.amount > 50` does above.
- `groupby` — groups rows by one or more expressions. The `into` clause binds the group to a variable (like `g` above) that supports aggregate methods such as `count()`, `sum()`, `min()`, `max()`, `avg()`, and `toList()`. Named keys (via `as`) can be accessed as properties, e.g. `g.customer`.
- `having` — filters groups after aggregation, as `g.count() > 1` does above.
- `orderby` — sorts results in ascending (`in asc`, the default) or descending (`in desc`) order, with `nullslast`/`nullsfirst` control.
- `limit` — restricts output to a given size, with optional offset for pagination.
- `select` — the projection (required). Defines the output columns, with optional `as` aliases.
- `union`, `unionall`, `intersect`, `minus` — set operations that combine two complete `from⋯select` expressions.

GINQ also supports window functions and nested subqueries — see Advanced Topics for details.

How it works: Under the covers GINQ transforms the query into calls within a fluent API. For example, the above query is transformed into:

```
from(sales)
    .where(s -> s.amount > 50)
    .groupByInto(s -> s.customer, g -> g.count() > 1)
    .orderBy(new Order(g -> g.key, true))
    .select((g, q) -> Tuple.tuple(g.key, g.count(), g.sum(s -> s.amount)))
```

You normally never see this transformed code, but if you are debugging or looking at stacktraces, don't be surprised to see fluent API calls instead of an exact match to the terms in the SQL-like syntax.

Basics

This section covers how to embed GINQ in your code and walks through each clause: data sources, projection, filtering, joining, grouping with aggregation, sorting, and pagination.

Integration

Other code in your scripts and classes may use the same names as the SQL-like syntax, such as `from`, `select`, `where`, etc. To avoid conflicts, GINQ expressions must be wrapped in a `GQ` block, which serves as a marker for the GINQ parser to recognize and transform the SQL-like syntax into the underlying fluent API calls.

A GINQ expression is wrapped in a `GQ` block and returns a lazy `Queryable` result:

```
def result = GQ {
    /* GINQ SQL-LIKE EXPRESSION */
}
def stream = result.stream() // get the stream from GINQ result
def list = result.toList() // get the list from GINQ result
```

Since the result is a standard `Queryable`/`Stream`, it composes naturally with other Groovy and JDK library calls:

```
import java.util.stream.Collectors

def numbers = [0, 1, 2]
assert '0#1#2' == GQ {
    from n in numbers
    select n
}.stream()
    .map(e -> String.valueOf(e))
    .collect(Collectors.joining('#'))
```

Two other markers, `GQL` and `@GQ`, are also supported. `GQL` is a shorthand that returns a `List` directly (equivalent to `GQ {⋯}.toList()`):

```
def list = GQL {
    /* GINQ SQL-LIKE EXPRESSION */
}
```

GINQ can also be used as a method annotation — see <<`@GQ` Annotation>> in Advanced Topics:

```
@GQ
def queryMethod() {
    /* GINQ SQL-LIKE EXPRESSION */
}
```

Data Source

The data source for GINQ could be specified by `from` clause, which is equivalent to SQL's `FROM`. Currently GINQ supports `Iterable`, `Stream`, array and GINQ result set as its data source:

`Iterable` Data Source

```
from n in [1, 2, 3] select n
```

`Stream` Data Source

```
from n in [1, 2, 3].stream() select n
```

Array Data Source

```
from n in new int[] {1, 2, 3} select n
```

GINQ Result Set Data Source

```
def vt = GQ {from m in [1, 2, 3] select m}
assert [1, 2, 3] == GQ {
    from n in vt select n
}.toList()
```

Projection

The column names could be renamed with `as` clause:

```
def result = GQ {
    from n in [1, 2, 3]
    select Math.pow(n, 2) as powerOfN
}
assert [[1, 1], [4, 4], [9, 9]] == result.stream().map(r -> [r[0], r.powerOfN]).
toList()
```

NOTE	The renamed column could be referenced by its new name, e.g. `r.powerOfN`. Also, it could be referenced by its index, e.g. `r[0]`

```
assert [[1, 1], [2, 4], [3, 9]] == GQ {
    from v in (
        from n in [1, 2, 3]
        select n, Math.pow(n, 2) as powerOfN
    )
    select v.n, v.powerOfN
}.toList()
```

NOTE	`select P1, P2, ⋯, Pn` is a simplified syntax of `select new NamedRecord(P1, P2, ⋯, Pn)` when and only when `n` >= 2. Also, `NamedRecord` instance will be created if `as`

> clause is used. The values stored in the `NamedRecord` could be referenced by their names.

Construct new objects as column values:

```
@groovy.transform.EqualsAndHashCode
class Person {
    String name
    Person(String name) {
        this.name = name
    }
}
def persons = [new Person('Daniel'), new Person('Paul'), new Person('Eric')]
assert persons == GQ {
    from n in ['Daniel', 'Paul', 'Eric']
    select new Person(n)
}.toList()
```

Distinct

`distinct` is equivalent to SQL's `DISTINCT`

```
def result = GQ {
    from n in [1, 2, 2, 3, 3, 3]
    select distinct(n)
}
assert [1, 2, 3] == result.toList()
```

```
def result = GQ {
    from n in [1, 2, 2, 3, 3, 3]
    select distinct(n, n + 1)
}
assert [[1, 2], [2, 3], [3, 4]] == result.toList()
```

Filtering

`where` is equivalent to SQL's `WHERE`

```
from n in [0, 1, 2, 3, 4, 5]
where n > 0 && n <= 3
select n * 2
```

In

```
from n in [0, 1, 2]
```

```
where n in [1, 2]
select n
```

```
from n in [0, 1, 2]
where n in (
    from m in [1, 2]
    select m
)
select n
```

```
import static groovy.lang.Tuple.tuple
assert [0, 1] == GQ {
    from n in [0, 1, 2]
    where tuple(n, n + 1) in (
        from m in [1, 2]
        select m - 1, m
    )
    select n
}.toList()
```

Not In

```
from n in [0, 1, 2]
where n !in [1, 2]
select n
```

```
from n in [0, 1, 2]
where n !in (
    from m in [1, 2]
    select m
)
select n
```

```
import static groovy.lang.Tuple.tuple
assert [2] == GQ {
    from n in [0, 1, 2]
    where tuple(n, n + 1) !in (
        from m in [1, 2]
        select m - 1, m
    )
    select n
}.toList()
```

Exists

```
from n in [1, 2, 3]
where (
    from m in [2, 3]
    where m == n
    select m
).exists()
select n
```

Not Exists

```
from n in [1, 2, 3]
where !(
    from m in [2, 3]
    where m == n
    select m
).exists()
select n
```

Joining

More data sources for GINQ could be specified by join clauses.

```
from n1 in [1, 2, 3]
join n2 in [1, 3] on n1 == n2
select n1, n2
```

NOTE	`join` is preferred over `innerjoin` and `innerhashjoin` as it has better readability, and it is smart enough to choose the correct concrete join(i.e. `innerjoin` or `innerhashjoin`) by its `on` clause.

```
from n1 in [1, 2, 3]
innerjoin n2 in [1, 3] on n1 == n2
select n1, n2
```

```
from n1 in [1, 2, 3]
leftjoin n2 in [2, 3, 4] on n1 == n2
select n1, n2
```

```
from n1 in [2, 3, 4]
rightjoin n2 in [1, 2, 3] on n1 == n2
select n1, n2
```

```
from n1 in [1, 2, 3]
fulljoin n2 in [2, 3, 4] on n1 == n2
select n1, n2
```

```
from n1 in [1, 2, 3]
crossjoin n2 in [3, 4, 5]
select n1, n2
```

hash join is especially efficient when data sources contain lots of objects

```
from n1 in [1, 2, 3]
innerhashjoin n2 in [1, 3] on n1 == n2
select n1, n2
```

```
from n1 in [1, 2, 3]
lefthashjoin n2 in [2, 3, 4] on n1 == n2
select n1, n2
```

```
from n1 in [2, 3, 4]
righthashjoin n2 in [1, 2, 3] on n1 == n2
select n1, n2
```

```
from n1 in [1, 2, 3]
fullhashjoin n2 in [2, 3, 4] on n1 == n2
select n1, n2
```

NOTE	Only binary expressions(`==`, `&&`) are allowed in the `on` clause of hash join

Grouping

`groupby` is equivalent to SQL's `GROUP BY`, and `having` is equivalent to SQL's `HAVING`. The `into` clause binds the grouped result to a named variable, enabling direct method calls for key access and aggregates. The variable is a `GroupResult` which extends `Queryable`, so all aggregate methods (`count()`, `sum()`, `toList()`, etc.) are available as real method calls:

```
assert [[1, 2], [3, 2], [6, 3]] == GQ {
    from n in [1, 1, 3, 3, 6, 6, 6]
    groupby n into g
    select g.key, g.count()
}.toList()
```

The group variable supports all `Queryable` aggregate methods:

```
assert [[1, 2], [3, 6], [6, 18]] == GQ {
    from n in [1, 1, 3, 3, 6, 6, 6]
    groupby n into g
    select g.key, g.sum(n -> n)
}.toList()
```

For multi-key grouping, individual keys can be named with `as` and accessed as properties or via subscript:

```
def result = GQ {
    from e in employees
    groupby e.dept as department, e.role as role into g
    select g.department, g.role, g.count()
}.toList()
```

Subscript access (`g['department']`) and explicit `g.get('department')` are also supported.

`having` works with `into` — the group variable can be used directly in the condition:

```
assert [[6, 3]] == GQ {
    from n in [1, 1, 3, 3, 6, 6, 6]
    groupby n into g
    having g.count() > 2
    select g.key, g.count()
}.toList()
```

NOTE The `where` clause after `groupby···into` is reserved for future use; use `having` for now.

Aggregate Functions

GINQ provides some built-in aggregate functions:

Function	Argument Type(s)	Return Type	Description
count()		java.lang.Long	number of rows, similar to `count(*)` in SQL
count(*expression*)	any	java.lang.Long	number of rows for which the value of *expression* is not `null`
min(*expression*)	java.lang.Comparable	same as argument type	minimum value of expression across all non-null values

Function	Argument Type(s)	Return Type	Description
max(*expression*)	java.lang.Comparable	same as argument type	maximum value of expression across all non-null values
sum(*expression*)	java.lang.Number	java.math.BigDecimal	sum of expression across all non-null values
avg(*expression*)	java.lang.Number	java.math.BigDecimal	the average (arithmetic mean) of all non-null values
list(*expression*)	any	java.util.List	the aggregated list of all non-null values
median(*expression*)	java.lang.Number	java.math.BigDecimal	value such that the number of non-null values above and below it is the same ("middle" value, not necessarily same as average or mean)
stdev(*expression*)	java.lang.Number	java.math.BigDecimal	the statistical standard deviation of all non-null values
stdevp(*expression*)	java.lang.Number	java.math.BigDecimal	the statistical standard deviation for the population for all non-null values
var(*expression*)	java.lang.Number	java.math.BigDecimal	the statistical variance of all non-null values
varp(*expression*)	java.lang.Number	java.math.BigDecimal	the statistical variance for the population for all non-null values
agg(*expression*)	any	any	customizes the aggregation logic in *expression* and returns single value

```
from n in [1, 1, 3, 3, 6, 6, 6]
groupby n
select n, count()
```

```
from s in ['a', 'b', 'cd', 'ef']
groupby s.size() as length
```

```
select length, min(s)
```

```
from s in ['a', 'b', 'cd', 'ef']
groupby s.size() as length
select length, max(s)
```

```
from n in [1, 1, 3, 3, 6, 6, 6]
groupby n
select n, sum(n)
```

```
from n in [1, 1, 3, 3, 6, 6, 6]
groupby n
select n, avg(n)
```

```
from n in [1, 1, 3, 3, 6, 6, 6]
groupby n
select n, median(n)
```

```
assert [['A', ['APPLE', 'APRICOT']],
        ['B', ['BANANA']],
        ['C', ['CANTALOUPE']]] == GQL {
    from fruit in ['Apple', 'Apricot', 'Banana', 'Cantaloupe']
    groupby fruit[0] as firstChar
    select firstChar, list(fruit.toUpperCase()) as fruit_list
}
```

```
def persons = [new Person('Linda', 100, 'Female'),
               new Person('Daniel', 135, 'Male'),
               new Person('David', 122, 'Male')]
assert [['Male', ['Daniel', 'David']], ['Female', ['Linda']]] == GQL {
    from p in persons
    groupby p.gender
    select p.gender, list(p.name)
}
```

```
from n in [1, 1, 3, 3, 6, 6, 6]
groupby n
select n, agg(_g.stream().map(r -> r.n).reduce(BigDecimal.ZERO, BigDecimal::add))
```

NOTE `_g` is an implicit variable for `agg` aggregate function, it represents the grouped `Queryable` object and its record(e.g. `r`) could reference the data source by alias(e.g. `n`)

```
from fruit in ['Apple', 'Apricot', 'Banana', 'Cantaloupe']
groupby fruit.substring(0, 1) as firstChar
select firstChar, agg(_g.stream().map(r -> r.fruit).toList()) as fruit_list
```

Also, we could apply the aggregate functions for the whole GINQ result, i.e. no `groupby` clause is needed:

```
assert [3] == GQ {
    from n in [1, 2, 3]
    select max(n)
}.toList()
```

```
assert [[1, 3, 2, 2, 6, 3, 3, 6]] == GQ {
    from n in [1, 2, 3]
    select min(n), max(n), avg(n), median(n), sum(n), count(n), count(),
            agg(_g.stream().map(r -> r.n).reduce(BigDecimal.ZERO, BigDecimal::add))
}.toList()
```

```
assert [0.816496580927726] == GQ {
    from n in [1, 2, 3]
    select stdev(n)
}.toList()
```

```
assert [1] == GQ {
    from n in [1, 2, 3]
    select stdevp(n)
}.toList()
```

```
assert [0.6666666666666667] == GQ {
    from n in [1, 2, 3]
    select var(n)
}.toList()
```

```
assert [1] == GQ {
    from n in [1, 2, 3]
    select varp(n)
}.toList()
```

Sorting

`orderby` is equivalent to SQL's `ORDER BY`

```
from n in [1, 5, 2, 6]
orderby n
select n
```

NOTE `in asc` is optional when sorting `in` ascending order

```
from n in [1, 5, 2, 6]
orderby n in asc
select n
```

```
from n in [1, 5, 2, 6]
orderby n in desc
select n
```

```
from s in ['a', 'b', 'ef', 'cd']
orderby s.length() in desc, s in asc
select s
```

```
from s in ['a', 'b', 'ef', 'cd']
orderby s.length() in desc, s
select s
```

```
from n in [1, null, 5, null, 2, 6]
orderby n in asc(nullslast)
select n
```

NOTE `nullslast` is equivalent to SQL's `NULLS LAST` and applied by default. `nullsfirst` is equivalent to SQL's `NULLS FIRST`.

```
from n in [1, null, 5, null, 2, 6]
orderby n in asc(nullsfirst)
select n
```

```
from n in [1, null, 5, null, 2, 6]
orderby n in desc(nullslast)
select n
```

```
from n in [1, null, 5, null, 2, 6]
orderby n in desc(nullsfirst)
```

```
select n
```

Pagination

`limit` is similar to the `limit` clause of MySQL, which could specify the `offset`(first argument) and `size`(second argument) for paginating, or just specify the only one argument as `size`

```
from n in [1, 2, 3, 4, 5]
limit 3
select n
```

```
from n in [1, 2, 3, 4, 5]
limit 1, 3
select n
```

Set Operations

Set operations combine the results of two complete GINQ expressions.

`union` returns distinct rows from both queries (duplicates removed):

```
assert [1, 2, 3, 4, 5] == GQ {
    from n in [1, 2, 3]
    select n
    union
    from m in [3, 4, 5]
    select m
}.toList()
```

`unionall` returns all rows from both queries (duplicates preserved):

```
assert [1, 2, 3, 3, 4, 5] == GQ {
    from n in [1, 2, 3]
    select n
    unionall
    from m in [3, 4, 5]
    select m
}.toList()
```

`intersect` returns only rows that appear in both queries:

```
assert [3, 4] == GQ {
    from n in [1, 2, 3, 4]
    select n
    intersect
```

```
    from m in [3, 4, 5, 6]
    select m
}.toList()
```

`minus` returns rows from the first query that do not appear in the second:

```
assert [1, 2] == GQ {
    from n in [1, 2, 3, 4]
    select n
    minus
    from m in [3, 4, 5, 6]
    select m
}.toList()
```

Each side of a set operation is a complete `from···select` expression and can include `where`, `groupby`, `orderby`, `limit`, and any other clauses.

NOTE	The set operation keyword must appear on its own line. Set operations are also available as methods on any `Queryable` result: `q1.union(q2)`, `q1.intersect(q2)`, `q1.minus(q2)`.

Common Recipes

Recipes for everyday tasks: row numbering, list comprehensions, querying JSON, updating collections, and workarounds for SQL features not yet in GINQ.

Row Number

`_rn` is the implicit variable representing row number for each record in the result set. It starts with `0`

```
from n in [1, 2, 3]
select _rn, n
```

List Comprehension

List comprehension is an elegant way to define and create lists based on existing lists:

```
assert [4, 16, 36, 64, 100] == GQ {from n in 1..<11 where n % 2 == 0 select n ** 2
}.toList()
```

```
assert [4, 16, 36, 64, 100] == GQ {from n in 1..<11 where n % 2 == 0 select n ** 2} as
List
```

```
assert [4, 16, 36, 64, 100] == GQL {from n in 1..<11 where n % 2 == 0 select n ** 2}
```

NOTE `GQL {⋯}` is the abbreviation of `GQ {⋯}.toList()`

GINQ could be used as list comprehension in the loops directly:

```
def result = []
for (def x : GQ {from n in 1..<11 where n % 2 == 0 select n ** 2}) {
    result << x
}
assert [4, 16, 36, 64, 100] == result
```

Query JSON

```
def json = new JsonSlurper().parseText('''
    {
        "fruits": [
            {"name": "Orange", "price": 11},
            {"name": "Apple", "price": 6},
            {"name": "Banana", "price": 4},
            {"name": "Mango", "price": 28},
            {"name": "Durian", "price": 32}
        ]
    }
''')

assert [['Mango', 28], ['Orange', 11], ['Apple', 6], ['Banana', 4]] == GQ {
    from f in json.fruits
    where f.price < 30
    orderby f.price in desc
    select f.name, f.price
}.toList()
```

Query & Update

This is like `update` statement in SQL

```
import groovy.transform.*
@TupleConstructor
@EqualsAndHashCode
@ToString
class Person {
    String name
    String nickname
}
```

```
def linda = new Person('Linda', null)
def david = new Person('David', null)
def persons = [new Person('Daniel', 'ShanFengXiaoZi'), linda, david]
def result = GQ {
    from p in persons
    where p.nickname == null
    select p
}.stream()
    .peek(p -> { p.nickname = 'Unknown' }) // update `nickname`
    .toList()

def expected = [new Person('Linda', 'Unknown'), new Person('David', 'Unknown')]
assert expected == result
assert ['Unknown', 'Unknown'] == [linda, david]*.nickname // ensure the original
objects are updated
```

Alternative for `with` clause

GINQ does not support `with` clause for now, but we could define a temporary variable to workaround:

```
def v = GQ { from n in [1, 2, 3] where n < 3 select n }
def result = GQ {
    from n in v
    where n > 1
    select n
}
assert [2] == result.toList()
```

Alternative for `case-when`

`case-when` of SQL could be replaced with switch expression:

```
assert ['a', 'b', 'c', 'c'] == GQ {
    from n in [1, 2, 3, 4]
    select switch (n) {
        case 1 -> 'a'
        case 2 -> 'b'
        default -> 'c'
    }
}.toList()
```

Advanced Topics

This section covers window functions, nested subqueries, the classic (pre-`into`) groupby style, the `@GQ` annotation for declaring GINQ methods, parallel querying, and options for customizing and optimizing GINQ. It also includes the full syntax reference.

Window Functions

Window can be defined by `partitionby`, `orderby`, `rows` and `range`:

```
over(
    [partitionby <expression> (, <expression>)*]
    [orderby <expression> (, <expression>)*
       [rows <lower>, <upper> | range <lower>, <upper>]]
)
```

- `0` used as bound of `rows` and `range` clause is equivalent to SQL's `CURRENT ROW`, and negative means `PRECEDING`, positive means `FOLLOWING`
- `null` used as the lower bound of `rows` and `range` clause is equivalent to SQL's `UNBOUNDED PRECEDING`
- `null` used as the upper bound of `rows` and `range` clause is equivalent to SQL's `UNBOUNDED FOLLOWING`

Also, GINQ provides some built-in window functions:

Function	Argument Type(s)	Return Type	Description
rowNumber()		java.lang.Long	number of the current row within its partition, counting from `0`
rank()		java.lang.Long	rank of the current row with gaps
denseRank()		java.lang.Long	rank of the current row without gaps
percentRank()		java.math.BigDecimal	relative rank of the current row: (rank - 1) / (total rows - 1)
cumeDist()		java.math.BigDecimal	relative rank of the current row: (number of rows preceding or peer with current row) / (total rows)
ntile(*expression*)	java.lang.Long	java.lang.Long	bucket index ranging from `0` to `expression - 1`, dividing the partition as equally as possible

Function	Argument Type(s)	Return Type	Description
lead(*expression* [, *offset* [, *default*]])	any [, java.lang.Long [, same as *expression* type]]	same as *expression* type	returns *expression* evaluated at the row that is *offset* rows after the current row within the partition; if there is no such row, instead return *default* (which must be of the same type as *expression*). Both *offset* and *default* are evaluated with respect to the current row. If omitted, *offset* defaults to `1` and *default* to `null`
lag(*expression* [, *offset* [, *default*]])	any [, java.lang.Long [, same as *expression* type]]	same as *expression* type	returns *expression* evaluated at the row that is *offset* rows before the current row within the partition; if there is no such row, instead return *default* (which must be of the same type as *expression*). Both *offset* and *default* are evaluated with respect to the current row. If omitted, *offset* defaults to `1` and *default* to `null`
firstValue(*expression*)	any	same type as *expression*	returns *expression* evaluated at the row that is the first row of the window frame
lastValue(*expression*)	any	same type as *expression*	returns *expression* evaluated at the row that is the last row of the window frame
nthValue(*expression*, *n*)	any, java.lang.Long	same type as *expression*	returns *expression* evaluated at the row that is the *nth* row of the window frame

Function	Argument Type(s)	Return Type	Description
count()		java.lang.Long	number of rows, similar to `count(*)` in SQL
count(*expression*)	any	java.lang.Long	number of rows for which the value of *expression* is not `null`
min(*expression*)	java.lang.Comparable	same as argument type	minimum value of expression across all non-null values
max(*expression*)	java.lang.Comparable	same as argument type	maximum value of expression across all non-null values
sum(*expression*)	java.lang.Number	java.math.BigDecimal	sum of expression across all non-null values
avg(*expression*)	java.lang.Number	java.math.BigDecimal	the average (arithmetic mean) of all non-null values
median(*expression*)	java.lang.Number	java.math.BigDecimal	value such that the number of non-null values above and below it is the same ("middle" value, not necessarily same as average or mean)
stdev(*expression*)	java.lang.Number	java.math.BigDecimal	the statistical standard deviation of all non-null values
stdevp(*expression*)	java.lang.Number	java.math.BigDecimal	the statistical standard deviation for the population for all non-null values
var(*expression*)	java.lang.Number	java.math.BigDecimal	the statistical variance of all non-null values
varp(*expression*)	java.lang.Number	java.math.BigDecimal	the statistical variance for the population for all non-null values
agg(*expression*)	any	any	**INCUBATING**: customizes the aggregation logic in *expression* and returns single value

`rowNumber`

```
assert [[2, 1, 1, 1], [1, 0, 0, 2], [null, 3, 3, 3], [3, 2, 2, 0]] == GQ {
    from n in [2, 1, null, 3]
    select n, (rowNumber() over(orderby n)),
              (rowNumber() over(orderby n in asc)),
              (rowNumber() over(orderby n in desc))
}.toList()
```

```
assert [[1, 0, 1, 2, 3], [2, 1, 2, 1, 2], [null, 3, 0, 3, 0], [3, 2, 3, 0, 1]] == GQ {
    from n in [1, 2, null, 3]
    select n, (rowNumber() over(orderby n in asc(nullslast))),
              (rowNumber() over(orderby n in asc(nullsfirst))),
              (rowNumber() over(orderby n in desc(nullslast))),
              (rowNumber() over(orderby n in desc(nullsfirst)))
}.toList()
```

NOTE The parentheses around the window function is required.

`rank`, `denseRank`, `percentRank`, `cumeDist` **and** `ntile`

```
assert [['a', 1, 1], ['b', 2, 2], ['b', 2, 2],
        ['c', 4, 3], ['c', 4, 3], ['d', 6, 4],
        ['e', 7, 5]] == GQ {
    from s in ['a', 'b', 'b', 'c', 'c', 'd', 'e']
    select s,
        (rank() over(orderby s)),
        (denseRank() over(orderby s))
}.toList()
```

```
assert [[60, 0, 0.4], [60, 0, 0.4], [80, 0.5, 0.8], [80, 0.5, 0.8], [100, 1, 1]] == GQ
{
    from n in [60, 60, 80, 80, 100]
    select n,
        (percentRank() over(orderby n)),
        (cumeDist() over(orderby n))
}.toList()
```

```
assert [[1, 0], [2, 0], [3, 0],
        [4, 1], [5, 1],
        [6, 2], [7, 2],[8, 2],
        [9, 3], [10, 3]] == GQ {
    from n in 1..10
    select n, (ntile(4) over(orderby n))
}.toList()
```

`lead` and `lag`

```
assert [[2, 3], [1, 2], [3, null]] == GQ {
    from n in [2, 1, 3]
    select n, (lead(n) over(orderby n))
}.toList()
```

```
assert [[2, 3], [1, 2], [3, null]] == GQ {
    from n in [2, 1, 3]
    select n, (lead(n) over(orderby n in asc))
}.toList()
```

```
assert [['a', 'bc'], ['ab', null], ['b', 'a'], ['bc', 'ab']] == GQ {
    from s in ['a', 'ab', 'b', 'bc']
    select s, (lead(s) over(orderby s.length(), s in desc))
}.toList()
```

```
assert [['a', null], ['ab', null], ['b', 'a'], ['bc', 'ab']] == GQ {
    from s in ['a', 'ab', 'b', 'bc']
    select s, (lead(s) over(partitionby s.length() orderby s.length(), s in desc))
}.toList()
```

```
assert [[2, 1], [1, null], [3, 2]] == GQ {
    from n in [2, 1, 3]
    select n, (lag(n) over(orderby n))
}.toList()
```

```
assert [[2, 3], [1, 2], [3, null]] == GQ {
    from n in [2, 1, 3]
    select n, (lag(n) over(orderby n in desc))
}.toList()
```

```
assert [['a', null], ['b', 'a'], ['aa', null], ['bb', 'aa']] == GQ {
    from s in ['a', 'b', 'aa', 'bb']
    select s, (lag(s) over(partitionby s.length() orderby s))
}.toList()
```

```
assert [[2, 3, 1], [1, 2, null], [3, null, 2]] == GQ {
    from n in [2, 1, 3]
    select n, (lead(n) over(orderby n)), (lag(n) over(orderby n))
}.toList()
```

The offset can be specified other than the default offset 1:

```
assert [[2, null, null], [1, 3, null], [3, null, 1]] == GQ {
    from n in [2, 1, 3]
    select n, (lead(n, 2) over(orderby n)), (lag(n, 2) over(orderby n))
}.toList()
```

The default value can be returned when the index specified by offset is out of window, e.g. 'NONE':

```
assert [[2, 'NONE', 'NONE'], [1, 3, 'NONE'], [3, 'NONE', 1]] == GQ {
    from n in [2, 1, 3]
    select n, (lead(n, 2, 'NONE') over(orderby n)), (lag(n, 2, 'NONE') over(orderby
n))
}.toList()
```

`firstValue`, `lastValue` and `nthValue`

```
assert [[2, 1], [1, 1], [3, 2]] == GQ {
    from n in [2, 1, 3]
    select n, (firstValue(n) over(orderby n rows -1, 1))
}.toList()
```

```
assert [[2, 3], [1, 2], [3, 3]] == GQ {
    from n in [2, 1, 3]
    select n, (lastValue(n) over(orderby n rows -1, 1))
}.toList()
```

```
assert [[2, 2], [1, 1], [3, 3]] == GQ {
    from n in [2, 1, 3]
    select n, (firstValue(n) over(orderby n rows 0, 1))
}.toList()
```

```
assert [[2, 1], [1, null], [3, 1]] == GQ {
    from n in [2, 1, 3]
    select n, (firstValue(n) over(orderby n rows -2, -1))
}.toList()
```

```
assert [[2, 1], [1, null], [3, 2]] == GQ {
    from n in [2, 1, 3]
    select n, (lastValue(n) over(orderby n rows -2, -1))
}.toList()
```

```
assert [[2, 3], [1, 3], [3, null]] == GQ {
    from n in [2, 1, 3]
    select n, (lastValue(n) over(orderby n rows 1, 2))
}.toList()
```

```
assert [[2, 3], [1, 2], [3, null]] == GQ {
    from n in [2, 1, 3]
    select n, (firstValue(n) over(orderby n rows 1, 2))
}.toList()
```

```
assert [[2, 2], [1, 1], [3, 3]] == GQ {
    from n in [2, 1, 3]
    select n, (lastValue(n) over(orderby n rows -1, 0))
}.toList()
```

```
assert [[2, 1], [1, 1], [3, 1]] == GQ {
    from n in [2, 1, 3]
    select n, (firstValue(n) over(orderby n rows null, 1))
}.toList()
```

```
assert [[2, 3], [1, 3], [3, 3]] == GQ {
    from n in [2, 1, 3]
    select n, (lastValue(n) over(orderby n rows -1, null))
}.toList()
```

```
assert [['a', 'a', 'b'], ['aa', 'aa', 'bb'], ['b', 'a', 'b'], ['bb', 'aa', 'bb']] ==
GQ {
    from s in ['a', 'aa', 'b', 'bb']
    select s, (firstValue(s) over(partitionby s.length() orderby s)),
            (lastValue(s) over(partitionby s.length() orderby s))
}.toList()
```

```
assert [[1, 1, 2, 3, null], [2, 1, 2, 3, null], [3, 1, 2, 3, null]] == GQ {
    from n in 1..3
    select n, (nthValue(n, 0) over(orderby n)),
              (nthValue(n, 1) over(orderby n)),
              (nthValue(n, 2) over(orderby n)),
              (nthValue(n, 3) over(orderby n))
}.toList()
```

```
assert [['a', 'a', 'b'], ['b', 'a', 'b'], ['aa', 'aa', 'bb'], ['bb', 'aa', 'bb']] ==
GQ {
    from s in ['a', 'b', 'aa', 'bb']
    select s, (min(s) over(partitionby s.length())), (max(s) over(partitionby s.
length()))
}.toList()
```

```
assert [[1, 2, 2, 2, 1, 1], [1, 2, 2, 2, 1, 1],
        [2, 2, 2, 4, 2, 2], [2, 2, 2, 4, 2, 2],
        [3, 2, 2, 6, 3, 3], [3, 2, 2, 6, 3, 3]] == GQ {
    from n in [1, 1, 2, 2, 3, 3]
    select n, (count() over(partitionby n)),
              (count(n) over(partitionby n)),
              (sum(n) over(partitionby n)),
              (avg(n) over(partitionby n)),
              (median(n) over(partitionby n))
}.toList()
```

```
assert [[2, 6, 3, 1, 3, 4], [1, 6, 3, 1, 3, 4],
        [3, 6, 3, 1, 3, 4], [null, 6, 3, 1, 3, 4]] == GQ {
    from n in [2, 1, 3, null]
    select n, (sum(n) over()),
              (max(n) over()),
              (min(n) over()),
              (count(n) over()),
              (count() over())
}.toList()
```

```
assert [[1, 1, 1], [2, 2, 3], [5, 2, 10], [5, 2, 10]] == GQ {
    from n in [1, 2, 5, 5]
    select n, (count() over(orderby n range -2, 0)),
              (sum(n) over(orderby n range -2, 0))
}.toList()
```

```
assert [[1, 2, 3], [2, 1, 2], [5, 2, 10], [5, 2, 10]] == GQ {
    from n in [1, 2, 5, 5]
    select n, (count() over(orderby n range 0, 1)),
              (sum(n) over(orderby n range 0, 1))
}.toList()
```

```
assert [[1, 2, 3], [2, 2, 3], [5, 2, 10], [5, 2, 10]] == GQ {
    from n in [1, 2, 5, 5]
```

```
    select n, (count() over(orderby n range -1, 1)),
              (sum(n) over(orderby n range -1, 1))
}.toList()
```

```
assert [[1, 1, 2], [2, 0, 0], [5, 0, 0], [5, 0, 0]] == GQ {
    from n in [1, 2, 5, 5]
    select n, (count() over(orderby n in desc range 1, 2)),
              (sum(n) over(orderby n in desc range 1, 2))
}.toList()
```

```
assert [[1, 0, 0], [2, 1, 1], [5, 0, 0], [5, 0, 0]] == GQ {
    from n in [1, 2, 5, 5]
    select n, (count() over(orderby n in desc range -2, -1)),
              (sum(n) over(orderby n in desc range -2, -1))
}.toList()
```

```
assert [[1, 3, 12], [2, 2, 10], [5, 0, 0], [5, 0, 0]] == GQ {
    from n in [1, 2, 5, 5]
    select n, (count() over(orderby n range 1, null)),
              (sum(n) over(orderby n range 1, null))
}.toList()
```

```
assert [[1, 2, 3], [2, 2, 3], [5, 4, 13], [5, 4, 13]] == GQ {
    from n in [1, 2, 5, 5]
    select n, (count() over(orderby n range null, 1)),
              (sum(n) over(orderby n range null, 1))
}.toList()
```

```
assert [[1, 0.816496580927726],
        [2, 0.816496580927726],
        [3, 0.816496580927726]] == GQ {
    from n in [1, 2, 3]
    select n, (stdev(n) over())
}.toList()
```

```
assert [[1, 1], [2, 1], [3, 1]] == GQ {
    from n in [1, 2, 3]
    select n, (stdevp(n) over())
}.toList()
```

```
assert [[1, 0.6666666666666667],
        [2, 0.6666666666666667],
```

```
        [3, 0.6666666666666667]] == GQ {
    from n in [1, 2, 3]
    select n, (var(n) over())
}.toList()
```

```
assert [[1, 1], [2, 1], [3, 1]] == GQ {
    from n in [1, 2, 3]
    select n, (varp(n) over())
}.toList()
```

```
assert [[1, 4], [2, 2], [3, 4]] == GQ {
    from n in [1, 2, 3]
    select n,
           (agg(_g.stream().map(r -> r.n).reduce(BigDecimal.ZERO, BigDecimal::add))
over(partitionby n % 2))
}.toList()
```

Nested GINQ

Nested GINQ in `from` clause

```
from v in (
    from n in [1, 2, 3]
    select n
)
select v
```

Nested GINQ in `where` clause

```
from n in [0, 1, 2]
where n in (
    from m in [1, 2]
    select m
)
select n
```

```
from n in [0, 1, 2]
where (
    from m in [1, 2]
    where m == n
    select m
).exists()
select n
```

Nested GINQ in `select` clause

```
assert [null, 2, 3] == GQ {
    from n in [1, 2, 3]
    select (
        from m in [2, 3, 4]
        where m == n
        limit 1
        select m
    )
}.toList()
```

NOTE	It's recommended to use `limit 1` to restrict the count of sub-query result because `TooManyValuesException` will be thrown if more than one values returned

We could use `as` clause to name the sub-query result

```
assert [[1, null], [2, 2], [3, 3]] == GQ {
    from n in [1, 2, 3]
    select n, (
        from m in [2, 3, 4]
        where m == n
        select m
    ) as sqr
}.toList()
```

Classic groupby style

GINQ also supports an older style without the `into` keyword which looks simpler for some cases but has some limitations — aggregate functions use a special syntax rather than real method calls, and the group cannot be accessed as a composable collection. Each field in any nonaggregate expression in the `select` clause **must** be included in the `groupby` clause:

```
from n in [1, 1, 3, 3, 6, 6, 6]
groupby n
select n, count(n)
```

```
from n in [1, 1, 3, 3, 6, 6, 6]
groupby n
having n >= 3
select n, count(n)
```

```
from n in [1, 1, 3, 3, 6, 6, 6]
groupby n
having count() < 3
```

```
select n, count()
```

The group columns could be renamed with `as` clause:

```
from s in ['ab', 'ac', 'bd', 'acd', 'bcd', 'bef']
groupby s.size() as length, s[0] as firstChar
select length, firstChar, max(s)
```

```
from s in ['ab', 'ac', 'bd', 'acd', 'bcd', 'bef']
groupby s.size() as length, s[0] as firstChar
having length == 3 && firstChar == 'b'
select length, firstChar, max(s)
```

Using the Queryable API directly

The `groupByInto` method is also available on the `Queryable` API directly, which can be useful when building queries programmatically:

```
def nums = [1, 2, 2, 3, 3, 4, 4, 5]
def result = from(nums)
    .groupByInto(e -> e, g -> g.count() > 1)
    .select((g, q) -> Tuple.tuple(g.key, g.count()))
    .toList()
assert [[2, 2], [3, 2], [4, 2]] == result
```

`@GQ` Annotation

GINQ could be written in a method marked with `@GQ`:

```
@GQ
def someGinqMethod() {
    /* GINQ CODE */
}
```

For example,

- Mark the `ginq` method as a GINQ method with `@GQ` annotation:

```
@groovy.ginq.transform.GQ
def ginq(list, b, e) {
    from n in list
    where b < n && n < e
    select n
}
```

```
assert [3, 4] == ginq([1, 2, 3, 4, 5, 6], 2, 5).toList()
```

- Specify the result type as `List`:

```
import groovy.ginq.transform.GQ

@GQ(List)
def ginq(b, e) {
    from n in [1, 2, 3, 4, 5, 6]
    where b < n && n < e
    select n
}

assert [3, 4] == ginq(2, 5)
```

NOTE GINQ supports many result types, e.g. `List`, `Set`, `Collection`, `Iterable`, `Iterator`, `java.util.stream.Stream` and array types.

- Enable parallel querying:

```
import groovy.ginq.transform.GQ

@GQ(parallel=true)
def ginq(x) {
    from n in [1, 2, 3]
    where n < x
    select n
}

assert [1] == ginq(2).toList()
```

Parallel Querying

Parallel querying is especially efficient when querying big data sources. It is disabled by default, but we could enable it by hand:

```
assert [[1, 1], [2, 2], [3, 3]] == GQ(parallel: true) {
    from n1 in 1..1000
    join n2 in 1..10000 on n2 == n1
    where n1 <= 3 && n2 <= 5
    select n1, n2
}.toList()
```

As parallel querying will use a shared thread pool, the following code can release resources after all GINQ statements execution are completed, and it will wait util all tasks of threads are completed.

```
GQ {
    shutdown
}
```

WARNING	Once `shutdown` is issued, parallel querying can not work anymore.

The following code is equivalent to the above code, in other words, `immediate` is optional:

```
GQ {
    shutdown immediate
}
```

Shutdown without waiting tasks to complete:

```
GQ {
    shutdown abort
}
```

Asynchronous Querying

A GINQ query is an ordinary (blocking) expression, so you can run it asynchronously using Groovy's `async`/`await` with no GINQ-specific support: wrap the query in `async { ⋯ }` to launch it, carry on with other work, then `await` the result when you need it.

```
def numbers = [1, 2, 3, 4, 5]

// launch the GINQ query asynchronously
def query = async {
    GQL {
        from n in numbers
        where n % 2 == 0
        select n
    }
}

// ... do other work here ...

// await the result when it is needed
def evens = await query
assert evens == [2, 4]
```

Customize GINQ

NOTE	**INCUBATING**: the custom code-generator (provider) extension point described here — the `astWalker` option and the `GinqAstVisitor` SPI it selects — is still incubating

> and may change in a future release, even though the rest of GINQ is stable.

For advanced users, you could customize GINQ behaviour by specifying your own target code generator. For example, we could specify the qualified class name `org.apache.groovy.ginq.provider.collection.GinqAstWalker` as the target code generator to generate GINQ method calls for querying collections, which is the default behaviour of GINQ:

```
assert [0, 1, 2] == GQ(astWalker:
'org.apache.groovy.ginq.provider.collection.GinqAstWalker') {
    from n in [0, 1, 2]
    select n
}.toList()
```

Optimize GINQ

GINQ optimizer is enabled by default for better performance. It will transform the GINQ AST to achieve better execution plan. We could disable it by hand:

```
assert [[2, 2]] == GQ(optimize: false) {
    from n1 in [1, 2, 3]
    join n2 in [1, 2, 3] on n1 == n2
    where n1 > 1 &&  n2 < 3
    select n1, n2
}.toList()
```

GINQ Syntax Reference

The full GINQ clause structure for reference:

```
GQ, i.e. abbreviation for GINQ
|__ <query>
|   |__ from
|   |   |__ <data_source_alias> in <data_source>
|   |__ [join/innerjoin/leftjoin/rightjoin/fulljoin/crossjoin]*
|   |   |__ <data_source_alias> in <data_source>
|   |   |__ on <condition> ((&& | ||) <condition>)* (NOTE: `crossjoin` does not need
`on` clause)
|   |__ [where]
|   |   |__ <condition> ((&& | ||) <condition>)*
|   |__ [groupby]
|   |   |__ <expression> [as <alias>] (, <expression> [as <alias>])* [into
<group_alias>]
|   |   |__ [having]
|   |       |__ <condition> ((&& | ||) <condition>)*
|   |__ [orderby]
|   |   |__ <expression> [in (asc|desc)] (, <expression> [in (asc|desc)])*
|   |__ [limit]
```

```
|   |   |__ [<offset>,] <size>
|   |__ select
|       |__ <expression> [as <alias>] (, <expression> [as <alias>])*
|__ [union/unionall/intersect/minus]*
    |__ <query>
```

NOTE	`[]` means the related clause is optional, `*` means zero or more times, and `+` means one or more times. Also, the clauses within each query are order sensitive, so the order of clauses should be kept as the above structure. *ONLY ONE* `from` clause is required per query. Multiple data sources are supported through `from` and the related joins. Set operations chain left-to-right, so `Q1 union Q2 minus Q3` is evaluated as `(Q1 union Q2) minus Q3`.

Known Limitations

WARNING	Currently GINQ can not work well when STC is enabled.

GINQ Examples

Complete worked examples showing GINQ in action.

Generate Multiplication Table

```
from v in (
    from a in 1..9
    join b in 1..9 on a <= b
    select a as f, b as s, "$a * $b = ${a * b}".toString() as r
)
groupby v.s
select max(v.f == 1 ? v.r : '') as v1,
       max(v.f == 2 ? v.r : '') as v2,
       max(v.f == 3 ? v.r : '') as v3,
       max(v.f == 4 ? v.r : '') as v4,
       max(v.f == 5 ? v.r : '') as v5,
       max(v.f == 6 ? v.r : '') as v6,
       max(v.f == 7 ? v.r : '') as v7,
       max(v.f == 8 ? v.r : '') as v8,
       max(v.f == 9 ? v.r : '') as v9
```

More examples

link: the latest GINQ examples

NOTE	Some examples in the above link require the latest SNAPSHOT version of Groovy to run.

Processing XML

Parsing XML

XmlParser and XmlSlurper

The most commonly used approach for parsing XML with Groovy is to use one of:

- `groovy.xml.XmlParser`
- `groovy.xml.XmlSlurper`

Both have a similar approach to parsing XML. Both come with a bunch of overloaded parse methods plus some special methods such as `parseText` and others. For the next example we will use the `parseText` method. It parses an XML `String` and recursively converts it to a list or map of objects.

XmlSlurper

```
def text = '''
    <list>
        <technology>
            <name>Groovy</name>
        </technology>
    </list>
'''

def list = new XmlSlurper().parseText(text) ①

assert list instanceof groovy.xml.slurpersupport.GPathResult ②
assert list.technology.name == 'Groovy' ③
```

① Parsing the XML and returning the root node as a GPathResult

② Checking we're using a GPathResult

③ Traversing the tree in a GPath style

XmlParser

```
def text = '''
    <list>
        <technology>
            <name>Groovy</name>
        </technology>
    </list>
'''

def list = new XmlParser().parseText(text) ①

assert list instanceof groovy.util.Node ②
assert list.technology.name.text() == 'Groovy' ③
```

① Parsing the XML and returning the root node as a Node

② Checking we're using a Node

③ Traversing the tree in a GPath style

Let's see the **similarities** between `XmlParser` and `XmlSlurper` first:

- Both are based on SAX so both have a low memory footprint
- Both can update/transform the XML

But they have key **differences**:

- `XmlSlurper` evaluates the structure lazily. So if you update the xml you'll have to evaluate the whole tree again.
- `XmlSlurper` returns `GPathResult` instances when parsing XML
- `XmlParser` returns `Node` objects when parsing XML

When to use one or the other?

> **NOTE** There is a discussion at StackOverflow. The conclusions written here are based partially on this entry.

- **If you want to transform an existing document to another** then `XmlSlurper` will be the choice
- **If you want to update and read at the same time** then `XmlParser` is the choice.

The rationale behind this is that every time you create a node with `XmlSlurper` it won't be available until you parse the document again with another `XmlSlurper` instance.

- **If you just have to read a few nodes** `XmlSlurper` should be your choice, since it will not have to create a complete structure in memory.

In general both classes perform in a similar way. Even the way of using GPath expressions with them is the same (both use `breadthFirst()` and `depthFirst()` expressions). So it depends on the write/read frequency.

Parser configuration with named parameters

Both `XmlParser` and `XmlSlurper` support Groovy's named parameter syntax for configuration, making it easy to set options without remembering positional boolean argument order:

XmlSlurper

```
def slurper = new XmlSlurper(namespaceAware: false, keepIgnorableWhitespace: true)
def result = slurper.parseText('<root><item>value</item></root>')
assert result.item.text() == 'value'
```

XmlParser

```
def parser = new XmlParser(namespaceAware: false, trimWhitespace: true)
```

```
def result = parser.parseText('<root><item>value</item></root>')
assert result.item[0].text() == 'value'
```

Available options for both parsers:

Option	Description	Default
`namespaceAware`	Enable XML namespace handling	`true`
`validating`	Validate documents against a DTD	`false`
`allowDocTypeDeclaration`	Allow DOCTYPE declarations in documents (see XML security)	`false`
`keepIgnorableWhitespace`	Preserve whitespace before elements	`false`

`XmlParser` additionally supports `trimWhitespace` (default `false`) to trim whitespace from element text content.

NOTE Options that affect the underlying SAX parser (`namespaceAware`, `validating`, `allowDocTypeDeclaration`) must be set before the first parse call. Setting them afterwards throws an `IllegalStateException`.

TIP Groovy's XML parsers are secure by default — DOCTYPE declarations, external entities, billion-laughs payloads and external resource resolution are all blocked. See XML security for details and the relax knobs available when you legitimately need DTD processing.

XML security

Groovy's XML parsers and serializer are **secure by default**. This section covers what's blocked, why, and the relax knobs available when you legitimately need DOCTYPE support, external entity resolution, or external resource access.

What's blocked by default

The defaults across all primary XML APIs:

Parser / API	DOCTYPE declarations	External resource access
`groovy.xml.XmlParser`	Rejected (`SAXParseException`)	Blocked via `FEATURE_SECURE_PROCESSING`
`groovy.xml.XmlSlurper`	Rejected	Blocked
`groovy.xml.DOMBuilder.parse(…)` (static)	Rejected	Blocked

Parser / API	DOCTYPE declarations	External resource access
`groovy.xml.DOMBuilder.newInstance()` + DSL	Rejected (factory hardened)	Blocked
`groovy.xml.XmlUtil.serialize(…)`	Rejected (TransformerFactory)	External DTDs / stylesheets blocked

This protects against:

- **XXE (XML eXternal Entity)** — payloads that smuggle in `file://` or `http://` references via internal DTD entity declarations.
- **Billion laughs / XML bomb** — exponential entity expansion attacks (mitigated by `FEATURE_SECURE_PROCESSING's entity-expansion budget).
- **External resource disclosure** — `<xs:import>`, `<xsl:import>`, external DTD subset resolution that would fetch from arbitrary URLs.

Relaxing when you need to

Three knobs cover the common cases. All default to the secure setting; opt in explicitly when your input is trusted.

`allowDocTypeDeclaration`

For `XmlParser`, `XmlSlurper`, `DOMBuilder`, and `XmlUtil.newSAXParser`:

```
// XmlParser / XmlSlurper — settable property or 3-arg constructor
def parser = new XmlParser(false, true, true)   // validating, nsAware, allowDocType
// or
def parser = new XmlParser()
parser.allowDocTypeDeclaration = true           // must be set before first parse

// DOMBuilder — relax knob on static parse and on the new newInstance overload
def doc = DOMBuilder.parse(reader, false, true, true)
def builder = DOMBuilder.newInstance(false, true, true)
```

Setting `allowDocTypeDeclaration=true` permits `<!DOCTYPE>` declarations. External entity references (`<!ENTITY foo SYSTEM "…">`) are still constrained by `FEATURE_SECURE_PROCESSING`; supply an `EntityResolver` if you need to resolve them yourself in a controlled way.

`EntityResolver` / `setEntityBaseUrl`

Both `XmlParser` and `XmlSlurper` accept a custom `EntityResolver`:

```
def slurper = new XmlSlurper(false, true, true)
slurper.setEntityResolver { publicId, systemId ->
    // resolve from a known-safe location, e.g. classpath
    new InputSource(getClass().getResourceAsStream("/dtds/" + new File(systemId).
name))
```

```
}
def root = slurper.parseText(xmlWithInternalDtd)

// XmlSlurper also has a convenience for a fixed base URL:
slurper.setEntityBaseUrl(new File("trusted-dtds/").toURI().toURL())
```

This is the recommended path when you have legitimate internal DTDs that need resolving against a controlled location, rather than turning the parser loose against the network.

`SerializeOptions.allowExternalResources`

For `XmlUtil.serialize` when transforming XSLT documents that legitimately reference external resources via `<xsl:import>` or `<xsl:include>`:

```
def opts = new SerializeOptions()
opts.allowDocTypeDeclaration = true     // permit DOCTYPE in output
opts.allowExternalResources = true      // permit external DTD/stylesheet access
XmlUtil.serialize(transformResult, output, opts)
```

For the overwhelmingly common case of pretty-printing already-parsed Groovy nodes or DOM trees, no changes are needed — the defaults work.

Hardened factories from `FactorySupport`

If you drop down to JAXP directly, `groovy.xml.FactorySupport` returns factories that already apply Groovy's hardening recipe:

```
import groovy.xml.FactorySupport

def dbf = FactorySupport.createDocumentBuilderFactory()                // DOCTYPE
blocked
def dbf2 = FactorySupport.createDocumentBuilderFactory(true)           // DOCTYPE
allowed

def spf = FactorySupport.createSaxParserFactory()                      // DOCTYPE
blocked
def spf2 = FactorySupport.createSaxParserFactory(true)                 // DOCTYPE
allowed

def xif = FactorySupport.createXMLInputFactory()                       // StAX,
hardened
def xif2 = FactorySupport.createXMLInputFactory(true)                  // StAX,
DOCTYPE allowed

def tf = FactorySupport.createTransformerFactory(false, false)         // doctype +
external both blocked

def sf = FactorySupport.createSchemaFactory(W3C_XML_SCHEMA_NS_URI)     // FSP enabled
```

```
def xpf = FactorySupport.createXPathFactory()                    // FSP enabled
```

The zero-arg `createDocumentBuilderFactory()` and `createSaxParserFactory()` return **hardened** factories — convenient aliases for the common secure default. This is a behaviour change in Groovy 6.0.0; prior versions returned bare JDK factories.

Streaming large XML documents

For multi-gigabyte feeds where loading the whole document into memory is impractical, use `XmlUtil.streamElements` to pull each matching subtree as a small DOM `Node`:

```
new File("huge-feed.xml").withReader { reader ->
    XmlUtil.streamElements(reader, "record").forEach { Node record ->
        // each subtree is a small, complete DOM tree
        process(record)
    }
}
```

Matching can be namespace-qualified:

```
XmlUtil.streamElements(reader, "urn:example:atom", "entry").forEach { entry ->
    // ...
}
```

For raw event-level streaming use `XmlUtil.events(reader)` to obtain a `Stream<XMLEvent>` over a hardened `XMLInputFactory`.

The factory underlying these streams is hardened by default; pass `allowDocTypeDeclaration=true` on the four-argument overload of `streamElements` if you need DTD support.

For per-record parallel processing on JDK 21+, sink the stream into a virtual-thread executor:

```
def executor = Executors.newVirtualThreadPerTaskExecutor()
try {
    XmlUtil.streamElements(reader, "record")
        .map { record -> executor.submit { process(record) } }
        .toList()
        *.get()
} finally {
    executor.close()
}
```

Notes for advanced users

- `XmlUtil.setFeatureQuietly` (and the equivalent helpers inside `FactorySupport`) silently ignore features that the underlying JAXP provider does not recognise. If you ship a custom JAXP provider that doesn't implement `FEATURE_SECURE_PROCESSING`, hardening is best-effort and your

factory may fall back to provider defaults.

- For that narrow case — a non-JDK JAXP provider on the classpath, where you want protection that does not depend on the provider honouring the JAXP feature flags — you can optionally add Apache Commons Secure XML to **your** project (Groovy itself does not depend on it). It installs an implementation-independent floor: external references resolve to empty content on any provider. Compose it with Groovy's APIs by injecting a Commons-secured parser into the constructor:

```
import org.apache.commons.xml.secure.SecureSAXParserFactory
import org.apache.commons.xml.secure.SecureDocumentBuilderFactory

// SAX-based (XmlParser / XmlSlurper) — Groovy's GPath on top of Commons' floor
def sp = SecureSAXParserFactory.newInstance().newSAXParser()
def root = new XmlSlurper(sp).parseText(text)

// DOM-based (DOMBuilder)
def db = SecureDocumentBuilderFactory.newInstance().newDocumentBuilder()
def dom = new DOMBuilder(db)
```

  This is defence-in-depth for a small residual: on the stock JDK, Groovy's own defaults already block XXE, entity expansion and external resource access, so no extra dependency is needed. Two caveats: injection applies only to parsers you construct explicitly (the convenience entry points such as `new XmlParser()` and `XmlUtil.*` keep using Groovy's own hardened factories), and it bypasses Groovy's factory configuration — so set `jdk.xml.maxElementDepth` on the injected parser yourself if you also want Groovy's element-depth bound.

- `SchemaFactory` instances returned by `FactorySupport.createSchemaFactory` enable `FEATURE_SECURE_PROCESSING` but do **not** set `ACCESS_EXTERNAL_SCHEMA`. This preserves the ability to resolve `<xs:import schemaLocation="…">` against trusted locations. Tighten further by setting `XMLConstants.ACCESS_EXTERNAL_SCHEMA` to `""` on the returned factory if your schema sources never legitimately import.
- `setFeature` calls within the hardening recipes are wrapped in try/catch blocks. To verify hardening took effect at runtime, read back via `factory.getFeature(…)` or `factory.getProperty(…)`.

DOMCategory

There is another way of parsing XML documents with Groovy with the use of `groovy.xml.dom.DOMCategory` which is a category class which adds GPath style operations to Java's DOM classes.

NOTE	Java has in-built support for DOM processing of XML using classes representing the various parts of XML documents, e.g. `Document`, `Element`, `NodeList`, `Attr` etc. For more information about these classes, refer to the respective JavaDocs.

Having an XML like the following:

```
static def CAR_RECORDS = '''
<records>
  <car name='HSV Maloo' make='Holden' year='2006'>
    <country>Australia</country>
    <record type='speed'>Production Pickup Truck with speed of 271kph</record>
  </car>
  <car name='P50' make='Peel' year='1962'>
    <country>Isle of Man</country>
    <record type='size'>Smallest Street-Legal Car at 99cm wide and 59 kg in
weight</record>
  </car>
  <car name='Royale' make='Bugatti' year='1931'>
    <country>France</country>
    <record type='price'>Most Valuable Car at $15 million</record>
  </car>
</records>
'''
```

You can parse it using `groovy.xml.DOMBuilder` and `groovy.xml.dom.DOMCategory`.

```
def reader = new StringReader(CAR_RECORDS)
def doc = DOMBuilder.parse(reader) ①
def records = doc.documentElement

use(DOMCategory) { ②
    assert records.car.size() == 3
}
```

① Parsing the XML

② Creating `DOMCategory` scope to be able to use helper method calls

GPath

The most common way of querying XML in Groovy is using `GPath`:

GPath is a path expression language integrated into Groovy which allows parts of nested structured data to be identified. In this sense, it has similar aims and scope as XPath does for XML. The two main places where you use GPath expressions is when dealing with nested POJOs or when dealing with XML

It is similar to XPath expressions and you can use it not only with XML but also with POJO classes. As an example, you can specify a path to an object or element of interest:

- `a.b.c` → for XML, yields all the `<c>` elements inside `<b>` inside `<a>`
- `a.b.c` → all POJOs, yields the `<c>` properties for all the `<b>` properties of `<a>` (sort of like a.getB().getC() in JavaBeans)

For XML, you can also specify attributes, e.g.:

- `a["@href"]` → the href attribute of all the a elements
- `a.'@href'` → an alternative way of expressing this
- `a.@href` → an alternative way of expressing this when using XmlSlurper

Let's illustrate this with an example:

```
static final String books = '''
    <response version-api="2.0">
        <value>
            <books>
                <book available="20" id="1">
                    <title>Don Quixote</title>
                    <author id="1">Miguel de Cervantes</author>
                </book>
                <book available="14" id="2">
                    <title>Catcher in the Rye</title>
                   <author id="2">JD Salinger</author>
               </book>
               <book available="13" id="3">
                   <title>Alice in Wonderland</title>
                   <author id="3">Lewis Carroll</author>
               </book>
               <book available="5" id="4">
                   <title>Don Quixote</title>
                   <author id="4">Miguel de Cervantes</author>
               </book>
           </books>
       </value>
    </response>
'''
```

Simply traversing the tree

First thing we could do is to get a value using POJO's notation. Let's get the first book's author's name

Getting node value

```
def response = new XmlSlurper().parseText(books)
def authorResult = response.value.books.book[0].author

assert authorResult.text() == 'Miguel de Cervantes'
```

First we parse the document with `XmlSlurper` and then we have to consider the returning value as the root of the XML document, so in this case is "response".

That's why we start traversing the document from response and then `value.books.book[0].author`. Note that in `XPath` the node arrays starts in [1] instead of [0], but because `GPath` is Java-based it

begins at index 0.

In the end we'll have the instance of the `author` node and because we wanted the text inside that node we should be calling the `text()` method. The `author` node is an instance of `GPathResult` type and `text()` a method giving us the content of that node as a String.

When using `GPath` with an XML parsed with `XmlSlurper` we'll have as a result a `GPathResult` object. `GPathResult` has many other convenient methods to convert the text inside a node to any other type such as:

- `toInteger()`
- `toFloat()`
- `toBigInteger()`
- ...

All these methods try to convert a `String` to the appropriate type.

If we were using an XML parsed with `XmlParser` we could be dealing with instances of type `Node`. But still all the actions applied to `GPathResult` in these examples could be applied to a Node as well. Creators of both parsers took into account `GPath` compatibility.

Next step is to get some values from a given node's attribute. In the following sample we want to get the first book's author's id. We'll be using two different approaches. Let's see the code first:

Getting an attribute's value

```
def response = new XmlSlurper().parseText(books)

def book = response.value.books.book[0] ①
def bookAuthorId1 = book.@id ②
def bookAuthorId2 = book['@id'] ③

assert bookAuthorId1 == '1' ④
assert bookAuthorId1.toInteger() == 1 ⑤
assert bookAuthorId1 == bookAuthorId2
```

① Getting the first book node

② Getting the book's id attribute `@id`

③ Getting the book's id attribute with `map notation` `['@id']`

④ Getting the value as a String

⑤ Getting the value of the attribute as an `Integer`

As you can see there are two types of notations to get attributes, the

- *direct notation* with `@nameoftheattribute`
- *map notation* using `['@nameoftheattribute']`

Both of them are equally valid.

Flexible navigation with children (*), depthFirst (**) and breadthFirst

If you ever have used XPath, you may have used expressions like:

- `/following-sibling::othernode` : Look for a node "othernode" in the same level
- `//` : Look everywhere

More or less we have their counterparts in GPath with the shortcuts `*` (aka `children()`) and `**` (aka `depthFirst()`).

The first example shows a simple use of `*`, which only iterates over the direct children of the node.

*Using **

```
def response = new XmlSlurper().parseText(books)

// .'*' could be replaced by .children()
def catcherInTheRye = response.value.books.'*'.find { node ->
    // node.@id == 2 could be expressed as node['@id'] == 2
    node.name() == 'book' && node.@id == '2'
}

assert catcherInTheRye.title.text() == 'Catcher in the Rye'
```

This test searches for any child nodes of the "books" node matching the given condition. In a bit more detail, the expression says: *Look for any node with a tag name equal to 'book' having an id with a value of '2' directly under the 'books' node.*

This operation roughly corresponds to the `breadthFirst()` method, except that it only stops at **one level** instead of continuing to the inner levels.

What if we would like to look for a given value without having to know exactly where it is. Let's say that the only thing we know is the id of the author "Lewis Carroll" . How are we going to be able to find that book? Using `**` is the solution:

*Using **

```
def response = new XmlSlurper().parseText(books)

// .'**' could be replaced by .depthFirst()
def bookId = response.'**'.find { book ->
    book.author.text() == 'Lewis Carroll'
}.@id

assert bookId == 3
```

`**` is the same as looking for something *everywhere in the tree from this point down*. In this case, we've used the method `find(Closure cl)` to find just the first occurrence.

What if we want to collect all book's titles? That's easy, just use `findAll`:

```
def response = new XmlSlurper().parseText(books)

def titles = response.'**'.findAll { node -> node.name() == 'title' }*.text()

assert titles.size() == 4
```

In the last two examples, ** is used as a shortcut for the `depthFirst()` method. It goes as far down the tree as it can while navigating down the tree from a given node. The `breadthFirst()` method finishes off all nodes on a given level before traversing down to the next level.

The following example shows the difference between these two methods:

depthFirst() vs .breadthFirst

```
def response = new XmlSlurper().parseText(books)
def nodeName = { node -> node.name() }
def withId2or3 = { node -> node.@id in [2, 3] }

assert ['book', 'author', 'book', 'author'] ==
        response.value.books.depthFirst().findAll(withId2or3).collect(nodeName)
assert ['book', 'book', 'author', 'author'] ==
        response.value.books.breadthFirst().findAll(withId2or3).collect(nodeName)
```

In this example, we search for any nodes with an id attribute with value 2 or 3. There are both `book` and `author` nodes that match that criteria. The different traversal orders will find the same nodes in each case but in different orders corresponding to how the tree was traversed.

It is worth mentioning again that there are some useful methods converting a node's value to an integer, float, etc. Those methods could be convenient when doing comparisons like this:

helpers

```
def response = new XmlSlurper().parseText(books)

def titles = response.value.books.book.findAll { book ->
    /* You can use toInteger() over the GPathResult object */
    book.@id.toInteger() > 2
}*.title

assert titles.size() == 2
```

In this case the number 2 has been hardcoded but imagine that value could have come from any other source (database... etc.).

Creating XML

The most commonly used approach for creating XML with Groovy is to use a builder, i.e. one of:

- `groovy.xml.MarkupBuilder`
- `groovy.xml.StreamingMarkupBuilder`

MarkupBuilder

Here is an example of using Groovy's MarkupBuilder to create a new XML file:

Creating Xml with MarkupBuilder

```
def writer = new StringWriter()
def xml = new MarkupBuilder(writer) ①

xml.records() { ②
    car(name: 'HSV Maloo', make: 'Holden', year: 2006) {
        country('Australia')
        record(type: 'speed', 'Production Pickup Truck with speed of 271kph')
    }
    car(name: 'Royale', make: 'Bugatti', year: 1931) {
        country('France')
        record(type: 'price', 'Most Valuable Car at $15 million')
    }
}

def records = new XmlSlurper().parseText(writer.toString()) ③

assert records.car.first().@name == 'HSV Maloo'
assert records.car.last().@name == 'Royale'
```

① Create an instance of `MarkupBuilder`

② Start creating the XML tree

③ Create an instance of `XmlSlurper` to traverse and test the generated XML

Let's take a look a little bit closer:

Creating XML elements

```
def xmlString = "<movie>the godfather</movie>" ①

def xmlWriter = new StringWriter() ②
def xmlMarkup = new MarkupBuilder(xmlWriter)

xmlMarkup.movie("the godfather") ③

assert xmlString == xmlWriter.toString() ④
```

① We're creating a reference string to compare against

② The `xmlWriter` instance is used by `MarkupBuilder` to convert the xml representation to a String instance eventually

③ The `xmlMarkup.movie(…)` call will create an XML node with a tag called `movie` and with content `the godfather`.

Creating XML elements with attributes

```
def xmlString = "<movie id='2'>the godfather</movie>"

def xmlWriter = new StringWriter()
def xmlMarkup = new MarkupBuilder(xmlWriter)

xmlMarkup.movie(id: "2", "the godfather") ①

assert xmlString == xmlWriter.toString()
```

① This time in order to create both attributes and node content you can create as many map entries as you like and finally add a value to set the node's content

NOTE	The value could be any `Object`, the value will be serialized to its `String` representation.

Creating XML nested elements

```
def xmlWriter = new StringWriter()
def xmlMarkup = new MarkupBuilder(xmlWriter)

xmlMarkup.movie(id: 2) { ①
    name("the godfather")
}

def movie = new XmlSlurper().parseText(xmlWriter.toString())

assert movie.@id == 2
assert movie.name.text() == 'the godfather'
```

① A closure represents the children elements of a given node. Notice this time instead of using a String for the attribute we're using a number.

Sometimes you may want to use a specific namespace in your xml documents:

Namespace aware

```
def xmlWriter = new StringWriter()
def xmlMarkup = new MarkupBuilder(xmlWriter)

xmlMarkup
        .'x:movies'('xmlns:x': 'http://www.groovy-lang.org') { ①
    'x:movie'(id: 1, 'the godfather')
    'x:movie'(id: 2, 'ronin')
}
```

```
def movies =
        new XmlSlurper() ②
                .parseText(xmlWriter.toString())
                .declareNamespace(x: 'http://www.groovy-lang.org')

assert movies.'x:movie'.last().@id == 2
assert movies.'x:movie'.last().text() == 'ronin'
```

① Creating a node with a given namespace `xmlns:x`

② Creating a `XmlSlurper` registering the namespace to be able to test the XML we just created

What about having some more meaningful example. We may want to generate more elements, to have some logic when creating our XML:

Mix code

```
def xmlWriter = new StringWriter()
def xmlMarkup = new MarkupBuilder(xmlWriter)

xmlMarkup
        .'x:movies'('xmlns:x': 'http://www.groovy-lang.org') {
    (1..3).each { n -> ①
        'x:movie'(id: n, "the godfather $n")
        if (n % 2 == 0) { ②
            'x:movie'(id: n, "the godfather $n (Extended)")
        }
    }
}

def movies =
        new XmlSlurper()
                .parseText(xmlWriter.toString())
                .declareNamespace(x: 'http://www.groovy-lang.org')

assert movies.'x:movie'.size() == 4
assert movies.'x:movie'*.text().every { name -> name.startsWith('the') }
```

① Generating elements from a range

② Using a conditional for creating a given element

Of course the instance of a builder can be passed as a parameter to refactor/modularize your code:

Mix code

```
def xmlWriter = new StringWriter()
def xmlMarkup = new MarkupBuilder(xmlWriter)

①
Closure<MarkupBuilder> buildMovieList = { MarkupBuilder builder ->
    (1..3).each { n ->
```

```
        builder.'x:movie'(id: n, "the godfather $n")
        if (n % 2 == 0) {
            builder.'x:movie'(id: n, "the godfather $n (Extended)")
        }
    }

    return builder
}

xmlMarkup.'x:movies'('xmlns:x': 'http://www.groovy-lang.org') {
    buildMovieList(xmlMarkup) ②
}

def movies =
        new XmlSlurper()
                .parseText(xmlWriter.toString())
                .declareNamespace(x: 'http://www.groovy-lang.org')

assert movies.'x:movie'.size() == 4
assert movies.'x:movie'*.text().every { name -> name.startsWith('the') }
```

① In this case we've created a Closure to handle the creation of a list of movies

② Just using the `buildMovieList` function when necessary

StreamingMarkupBuilder

The class `groovy.xml.StreamingMarkupBuilder` is a builder class for creating XML markup. This implementation uses a `groovy.xml.streamingmarkupsupport.StreamingMarkupWriter` to handle output.

Using StreamingMarkupBuilder

```
def xml = new StreamingMarkupBuilder().bind { ①
    records {
        car(name: 'HSV Maloo', make: 'Holden', year: 2006) { ②
            country('Australia')
            record(type: 'speed', 'Production Pickup Truck with speed of 271kph')
        }
        car(name: 'P50', make: 'Peel', year: 1962) {
            country('Isle of Man')
            record(type: 'size', 'Smallest Street-Legal Car at 99cm wide and 59 kg in
weight')
        }
        car(name: 'Royale', make: 'Bugatti', year: 1931) {
            country('France')
            record(type: 'price', 'Most Valuable Car at $15 million')
        }
    }
}

def records = new XmlSlurper().parseText(xml.toString()) ③
```

```
assert records.car.size() == 3
assert records.car.find { it.@name == 'P50' }.country.text() == 'Isle of Man'
```

① Note that `StreamingMarkupBuilder.bind` returns a `Writable` instance that may be used to stream the markup to a Writer

② We're capturing the output in a String to parse it again and check the structure of the generated XML with `XmlSlurper`.

MarkupBuilderHelper

The `groovy.xml.MarkupBuilderHelper` is, as its name reflects, a helper for `groovy.xml.MarkupBuilder`.

This helper normally can be accessed from within an instance of class `groovy.xml.MarkupBuilder` or an instance of `groovy.xml.StreamingMarkupBuilder`.

This helper could be handy in situations when you may want to:

- Produce a comment in the output
- Produce an XML processing instruction in the output
- Produce an XML declaration in the output
- Print data in the body of the current tag, escaping XML entities
- Print data in the body of the current tag

In both `MarkupBuilder` and `StreamingMarkupBuilder` this helper is accessed by the property `mkp`:

Using MarkupBuilder's 'mkp'

```
def xmlWriter = new StringWriter()
def xmlMarkup = new MarkupBuilder(xmlWriter).rules {
    mkp.comment('THIS IS THE MAIN RULE') ①
    rule(sentence: mkp.yield('3 > n')) ②
}

③
assert xmlWriter.toString().contains('3 &gt; n')
assert xmlWriter.toString().contains('<!-- THIS IS THE MAIN RULE -->')
```

① Using `mkp` to create a comment in the XML

② Using `mkp` to generate an escaped value

③ Checking both assumptions were true

Here is another example to show the use of `mkp` property accessible from within the `bind` method scope when using `StreamingMarkupBuilder`:

Using StreamingMarkupBuilder's 'mkp'

```
def xml = new StreamingMarkupBuilder().bind {
```

```
    records {
        car(name: mkp.yield('3 < 5')) ①
        car(name: mkp.yieldUnescaped('1 < 3')) ②
    }
}

assert xml.toString().contains('3 &lt; 5')
assert xml.toString().contains('1 < 3')
```

① If we want to generate an escaped value for the name attribute with `mkp.yield`

② Checking the values later on with `XmlSlurper`

DomToGroovy

Suppose we have an existing XML document and we want a head start on writing the markup rather than typing it all in? We just need to use `org.apache.groovy.xml.tools.DomToGroovy` as shown in the following example:

`DomToGroovy` is an authoring aid. It produces builder code to read and edit, so the generated source is a starting point rather than a finished script. The example below evaluates it directly to show that the round trip works; do that only for documents you control, since evaluating generated source runs whatever the document produced.

Building MarkupBuilder from DomToGroovy

```
def songs = """
    <songs>
      <song>
        <title>Here I go</title>
        <band>Whitesnake</band>
      </song>
    </songs>
"""

def builder =
        javax.xml.parsers.DocumentBuilderFactory.newInstance().newDocumentBuilder()

def inputStream = new ByteArrayInputStream(songs.bytes)
def document = builder.parse(inputStream)
def output = new StringWriter()
def converter = new DomToGroovy(new PrintWriter(output)) ①

converter.print(document) ②

String xmlRecovered =
        new GroovyShell()
                .evaluate("""
           def writer = new StringWriter()
           def builder = new groovy.xml.MarkupBuilder(writer)
           builder.${output}
```

```
            return writer.toString()
        """) ③

assert new XmlSlurper().parseText(xmlRecovered).song.title.text() == 'Here I go' ④
```

① Creating `DomToGroovy` instance

② Converts the XML to `MarkupBuilder` calls which are available in the output `StringWriter`

③ Using `output` variable to create the whole MarkupBuilder

④ Back to XML string

Manipulating XML

In this chapter you'll see the different ways of adding / modifying / removing nodes using `XmlSlurper` or `XmlParser`. The xml we are going to be handling is the following:

```
def xml = """
<response version-api="2.0">
    <value>
        <books>
            <book id="2">
                <title>Don Quixote</title>
                <author id="1">Miguel de Cervantes</author>
            </book>
        </books>
    </value>
</response>
"""
```

Adding nodes

The main difference between `XmlSlurper` and `XmlParser` is that when former creates the nodes they won't be available until the document's been evaluated again, so you should parse the transformed document again in order to be able to see the new nodes. So keep that in mind when choosing any of both approaches.

If you needed to see a node right after creating it then `XmlParser` should be your choice, but if you're planning to do many changes to the XML and send the result to another process maybe `XmlSlurper` would be more efficient.

You can't create a new node directly using the `XmlSlurper` instance, but you can with `XmlParser`. The way of creating a new node from XmlParser is through its method `createNode(..)`

```
def parser = new XmlParser()
def response = parser.parseText(xml)
def numberOfResults = parser.createNode(
        response,
```

```
        new QName("numberOfResults"),
        [:]
)

numberOfResults.value = "1"
assert response.numberOfResults.text() == "1"
```

The `createNode()` method receives the following parameters:

- parent node (could be null)
- The qualified name for the tag (In this case we only use the local part without any namespace). We're using an instance of `groovy.namespace.QName`
- A map with the tag's attributes (None in this particular case)

Anyway you won't normally be creating a node from the parser instance but from the parsed XML instance. That is from a `Node` or a `GPathResult` instance.

Take a look at the next example. We are parsing the xml with `XmlParser` and then creating a new node from the parsed document's instance (Notice the method here is slightly different in the way it receives the parameters):

```
def parser = new XmlParser()
def response = parser.parseText(xml)

response.appendNode(
        new QName("numberOfResults"),
        [:],
        "1"
)

response.numberOfResults.text() == "1"
```

When using `XmlSlurper`, `GPathResult` instances don't have `createNode()` method.

Modifying / Removing nodes

We know how to parse the document, add new nodes, now I want to change a given node's content. Let's start using `XmlParser` and `Node`. This example changes the first book information to actually another book.

```
def response = new XmlParser().parseText(xml)

/* Use the same syntax as groovy.xml.MarkupBuilder */
response.value.books.book[0].replaceNode { ①
    book(id: "3") {
        title("To Kill a Mockingbird")
        author(id: "3", "Harper Lee")
    }
```

```
}

def newNode = response.value.books.book[0]

assert newNode.name() == "book"
assert newNode.@id == "3"
assert newNode.title.text() == "To Kill a Mockingbird"
assert newNode.author.text() == "Harper Lee"
assert newNode.author.@id.first() == "3"
```

When using `replaceNode()` the closure we pass as parameter should follow the same rules as if we were using `groovy.xml.MarkupBuilder`:

Here's the same example using `XmlSlurper`:

```
def response = new XmlSlurper().parseText(books)

/* Use the same syntax as groovy.xml.MarkupBuilder */
response.value.books.book[0].replaceNode {
    book(id: "3") {
        title("To Kill a Mockingbird")
        author(id: "3", "Harper Lee")
    }
}

assert response.value.books.book[0].title.text() == "Don Quixote"

/* That mkp is a special namespace used to escape away from the normal building mode
   of the builder and get access to helper markup methods
   'yield', 'pi', 'comment', 'out', 'namespaces', 'xmlDeclaration' and
   'yieldUnescaped' */
def result = new StreamingMarkupBuilder().bind { mkp.yield response }.toString()
def changedResponse = new XmlSlurper().parseText(result)

assert changedResponse.value.books.book[0].title.text() == "To Kill a Mockingbird"
```

Notice how using `XmlSlurper` we have to parse the transformed document again in order to find the created nodes. In this particular example this could be a little bit annoying.

Finally both parsers also use the same approach for adding a new attribute to a given node. This time again the difference is whether you want the new nodes to be available right away or not. First `XmlParser`:

```
def parser = new XmlParser()
def response = parser.parseText(xml)

response.@numberOfResults = "1"
```

```
assert response.@numberOfResults == "1"
```

And `XmlSlurper`:

```
def response = new XmlSlurper().parseText(books)
response.@numberOfResults = "2"

assert response.@numberOfResults == "2"
```

When using `XmlSlurper`, adding a new attribute does **not** require you to perform a new evaluation.

Printing XML

XmlUtil

Sometimes it's useful to get not only the value of a given node but the node itself (for instance to add this node to another XML).

For that you can use `groovy.xml.XmlUtil` class. It has several static methods to serialize the xml fragment from several types of sources (Node, GPathResult, String...)

Getting a node as a string

```
def response = new XmlParser().parseText(xml)
def nodeToSerialize = response.'**'.find { it.name() == 'author' }
def nodeAsText = XmlUtil.serialize(nodeToSerialize)

assert nodeAsText ==
        XmlUtil.serialize('<?xml version="1.0" encoding="UTF-8"?><author id="1">Miguel
de Cervantes</author>')
```

Serialization can be customized using `SerializeOptions` to control encoding, indentation, and DOCTYPE handling:

Customizing serialization

```
def response = new XmlParser().parseText(xml)

// Custom encoding
def latin1 = XmlUtil.serialize(response, new SerializeOptions(encoding: 'ISO-8859-1'))
assert latin1.contains('encoding="ISO-8859-1"')

// Custom indent (4 spaces instead of default 2)
def wideIndent = XmlUtil.serialize(response, new SerializeOptions(indent: 4))
assert wideIndent.contains('    <value>')

// Multiple options
def custom = XmlUtil.serialize(response, new SerializeOptions(encoding: 'ISO-8859-1',
indent: 4))
```

```
assert custom.contains('encoding="ISO-8859-1"')
assert custom.contains('    <value>')
```

Typed XML Conversion

Groovy 6 adds support for converting XML nodes into typed objects, complementing the dynamic GPath-style navigation that `XmlParser` and `XmlSlurper` already provide.

Converting Nodes to Maps

The `toMap()` method on `Node` converts an XML node tree into a nested `Map<String, Object>`. This requires no additional dependencies.

```
def node = new XmlParser().parseText('''
    <server>
        <host>localhost</host>
        <port>8080</port>
        <debug>true</debug>
    </server>'''.stripIndent())

def map = node.toMap()
assert map == [host: 'localhost', port: '8080', debug: 'true']
```

Attributes and child elements are merged into a single map. Nested elements become nested maps:

```
def node = new XmlParser().parseText('''
    <app>
        <name>myapp</name>
        <server>
            <host>localhost</host>
            <port>8080</port>
        </server>
    </app>'''.stripIndent())

def map = node.toMap()
assert map == [name: 'myapp', server: [host: 'localhost', port: '8080']]
```

Attributes are also included:

```
def node = new XmlParser().parseText('<server host="localhost" port="8080"/>')

def map = node.toMap()
assert map == [host: 'localhost', port: '8080']
```

Repeated same-name elements become lists:

```
def node = new XmlParser().parseText('''
    <config>
        <alias>web1</alias>
        <alias>web2</alias>
        <alias>web3</alias>
    </config>'''.stripIndent())

def map = node.toMap()
assert map == [alias: ['web1', 'web2', 'web3']]
```

If a child element name collides with an attribute name, the child element wins.

When an element has both attributes and text content, the text is stored under the `_text` key (available as the `Node.TEXT_KEY` constant):

```
def node = new XmlParser().parseText('<price currency="USD">9.99</price>')

def map = node.toMap()
assert map == [currency: 'USD', _text: '9.99']
assert map[Node.TEXT_KEY] == '9.99'
```

Type coercion with `as`

Because `toMap()` produces a standard Groovy `Map`, you can use Groovy's built-in `as` coercion to convert a node directly to a typed object. No additional dependencies are required:

```
static class SimpleConfig {
    String host
    String port
}
```

```
def node = new XmlParser().parseText('''
    <server>
        <host>localhost</host>
        <port>8080</port>
    </server>'''.stripIndent())

def config = node as SimpleConfig
assert config.host == 'localhost'
assert config.port == '8080'
```

You can also coerce to `Map` directly:

```
def node = new XmlParser().parseText('<server><host>localhost</host></server>')
```

```
Map map = node as Map
assert map.host == 'localhost'
```

NOTE	The `as` coercion uses Groovy's standard named-parameter constructor, which works well when target properties are `String`-typed. For richer type conversion (e.g. `String` to `int`, `boolean`, `BigDecimal`), use `parseTextAs` described below.

Typed parsing with `parseTextAs`

The `parseTextAs` and `parseAs` methods on `XmlParser` parse XML directly into typed objects using Jackson's `ObjectMapper.convertValue`, which handles type conversion automatically. This requires `jackson-databind` on the classpath — a clear error is thrown if it is absent.

```
static class ServerConfig {
    String host
    int port
    boolean debug
}
```

```
def config = new XmlParser().parseTextAs(ServerConfig, '''
    <server>
        <host>localhost</host>
        <port>8080</port>
        <debug>true</debug>
    </server>'''.stripIndent())

assert config instanceof ServerConfig
assert config.host == 'localhost'
assert config.port == 8080
assert config.debug == true
```

Nested objects are supported:

```
static class AppConfig {
    String name
    ServerConfig server
}
```

```
def config = new XmlParser().parseTextAs(AppConfig, '''
    <app>
        <name>myapp</name>
        <server>
            <host>localhost</host>
            <port>9090</port>
            <debug>false</debug>
```

```
        </server>
    </app>'''.stripIndent())

assert config.name == 'myapp'
assert config.server.host == 'localhost'
assert config.server.port == 9090
```

Standard Jackson annotations such as `@JsonProperty` and `@JsonFormat` are supported for property name mapping and type conversion.

Overloaded variants of `parseAs` accept a `Reader`, `InputStream`, `File`, or `Path` in addition to the `String`-based `parseTextAs`.

Full Jackson XML support

The `parseTextAs` method uses `jackson-databind` (not `jackson-dataformat-xml`), so XML-specific Jackson annotations like `@JacksonXmlText` and `@JacksonXmlElementWrapper` are not available through this path.

If you need the full Jackson XML experience, use `jackson-dataformat-xml` directly:

```
@Grab('com.fasterxml.jackson.dataformat:jackson-dataformat-xml')
import com.fasterxml.jackson.dataformat.xml.XmlMapper

def config = new XmlMapper().readValue(xmlString, ServerConfig)
```

Or parse first with `XmlParser` for inspection or validation, then re-serialize and hand off to Jackson:

```
def node = new XmlParser().parseText(xmlString)
// ... inspect or validate node ...
def config = new XmlMapper().readValue(XmlUtil.serialize(node), ServerConfig)
```

Processing YAML

Groovy has an optional `groovy-yaml` module which provides support for converting between Groovy objects and YAML. The classes dedicated to YAML serialization and parsing are found in the `groovy.yaml` package.

YamlSlurper

`YamlSlurper` is a class that parses YAML text or reader content into Groovy data structures (objects) such as maps, lists and primitive types like `Integer`, `Double`, `Boolean` and `String`.

The class comes with a bunch of overloaded `parse` methods plus some special methods such as `parseText` and others. For the next example we will use the `parseText` method. It parses a YAML `String` and recursively converts it to a list or map of objects. The other `parse*` methods are similar in that they return a YAML `String` but for different parameter types.

```
        def ys = new YamlSlurper()
        def yaml = ys.parseText '''
language: groovy
sudo: required
dist: trusty

matrix:
  include:
    - jdk: openjdk10
    - jdk: oraclejdk9
    - jdk: oraclejdk8

before_script:
  - |
    unset _JAVA_OPTIONS

        '''

        assert 'groovy' == yaml.language
        assert 'required' == yaml.sudo
        assert 'trusty' == yaml.dist
        assert ['openjdk10', 'oraclejdk9', 'oraclejdk8'] ==  yaml.matrix.include.jdk
        assert ['unset _JAVA_OPTIONS'] == yaml.before_script*.trim()
```

Notice the result is a plain map and can be handled like a normal Groovy object instance. `YamlSlurper` parses the given YAML as defined by the YAML Ain't Markup Language (YAML™).

As `YamlSlurper` is returning pure Groovy object instances without any special YAML classes in the back, its usage is transparent. In fact, `YamlSlurper` results conform to GPath expressions. GPath is a powerful expression language that is supported by multiple slurpers for different data formats (`XmlSlurper` for XML being one example).

NOTE For more details please have a look at the section on GPath expressions.

The following table gives an overview of the YAML types and the corresponding Groovy data types:

YAML	Groovy
string	`java.lang.String`
number	`java.lang.BigDecimal` or `java.lang.Integer`
object	`java.util.LinkedHashMap`
array	`java.util.ArrayList`
true	`true`
false	`false`
null	`null`
date/time	`java.lang.String` (see the note below)

NOTE	Whenever a value in YAML is `null`, `YamlSlurper` supplements it with the Groovy `null` value. This is in contrast to other YAML parsers that represent a `null` value with a library-provided singleton object.

NOTE	For the untyped `parse`/`parseText` API, YAML date and time values are returned as `String`. `YamlSlurper` routes the untyped path through a YAML-to-JSON conversion, and JSON has no native temporal types. Use the typed `parseAs`/`parseTextAs` API (see Typed date and time values) for `java.time.LocalDate`, `java.time.LocalTime`, `java.time.LocalDateTime`, and `java.time.OffsetDateTime` fidelity.

Typed parsing

`YamlSlurper` can parse YAML directly into typed objects using Jackson databinding. Standard Jackson annotations such as `@JsonProperty` and `@JsonFormat` are supported for property mapping and type conversion:

```
static class ServerConfig {
    String host
    int port
}
```

```
        def config = new YamlSlurper().parseTextAs(ServerConfig, '''\
host: localhost
port: 8080
''')
        assert config.host == 'localhost'
        assert config.port == 8080
```

NOTE	For simple cases, Groovy's `as` coercion also works with the untyped result: `def config = new YamlSlurper().parseText(yaml) as ServerConfig`. The `parseTextAs` method uses Jackson databinding, which supports richer annotation-driven mapping via `@JsonProperty`, `@JsonFormat`, etc.

Typed date and time values

When a target type declares `java.time.*` fields, the typed `parseAs`/`parseTextAs` API and `YamlBuilder.toYaml` round-trip YAML temporal values with full fidelity, including the original zone offset for `OffsetDateTime`:

```
static class Event {
    java.time.OffsetDateTime created
    java.time.LocalDate effective
    java.time.LocalTime windowStart
    java.time.LocalDateTime updated
    String name
```

```
}
```

```
def original = new Event(
        created: java.time.OffsetDateTime.parse('1979-05-27T07:32:00-08:00'),
        effective: java.time.LocalDate.of(1979, 5, 27),
        windowStart: java.time.LocalTime.of(7, 32, 0),
        updated: java.time.LocalDateTime.of(1979, 5, 27, 7, 32, 0),
        name: 'demo')

def yaml = YamlBuilder.toYaml(original)
def parsed = new YamlSlurper().parseTextAs(Event, yaml)

assert parsed.created == original.created           // OffsetDateTime, non-UTC offset
preserved
assert parsed.effective == original.effective       // LocalDate
assert parsed.windowStart == original.windowStart   // LocalTime
assert parsed.updated == original.updated           // LocalDateTime
assert parsed.name == original.name
```

Anchors and aliases

`YamlSlurper` does not resolve YAML anchors and aliases. An alias arrives as a plain string holding the anchor's name, and a merge key (`<<`) is left as a literal entry rather than merging anything:

```
def yaml = '''\
    defaults: &d
      host: example.com
      port: 443
    prod: *d
    '''.stripIndent()

def loaded = new YamlSlurper().parseText(yaml)

assert loaded.defaults == [host: 'example.com', port: 443]
assert loaded.prod == 'd'     // the anchor's name, not the mapping it names
```

```
def merged = new YamlSlurper().parseText('''\
    defaults: &d
      a: 1
      b: 9
    prod:
      <<: *d
      b: 2
    '''.stripIndent())

assert merged.prod == ['<<': 'd', b: 2]   // no merge occurs
```

This comes from the Jackson YAML backend the slurper is built on, which reads the document as a stream of events and does not replay an anchored subtree when an alias refers to it. It is a long-standing, deliberate limitation (jackson-dataformats-text#98, closed as not planned), so a document that relies on anchors should either be flattened before parsing, or read as shown below.

For the typed `parseAs` API only, Jackson's own object-identity mechanism can stand in for aliases. Annotate the referenced class with `@JsonIdentityInfo` and an anchored mapping round-trips as one shared instance:

```
@JsonIdentityInfo(generator = ObjectIdGenerators.StringIdGenerator, property = '@id')
static class Address {
    String street
    String city
}

static class Shipment {
    Address billTo
    Address shipTo
}
```

```
def shipment = new YamlSlurper().parseTextAs(Shipment, '''\
    billTo: &id001
      street: 123 Tornado Alley
      city: East Centerville
    shipTo: *id001
    '''.stripIndent())

assert shipment.shipTo.street == '123 Tornado Alley'
assert shipment.billTo.is(shipment.shipTo)   // one shared instance
```

This works for mutable, setter-based classes only. A type deserialized through a constructor — `@JsonCreator`, and immutable classes generally — fails with `No Object Id found` even for a plain backward reference, so do not reach for this with `@Immutable` targets. It is Jackson's object-identity feature standing in for aliases, not alias support.

Resolving aliases with SnakeYAML directly

The SnakeYAML library resolves anchors, aliases, and merge keys in full. This is not part of `groovy-yaml` — it is a second YAML parser used directly, shown here because the need is common. Today SnakeYAML happens to reach the classpath transitively via the Jackson YAML backend; that arrangement is not yours to rely on (and ends with Jackson 3), so declare `org.yaml:snakeyaml` as a dependency if you use this.

```
def yaml = new Yaml(new SafeConstructor(new LoaderOptions()))

def loaded = yaml.load('''\
```

```
    defaults: &d
      a: 1
      b: 9
    prod:
      <<: *d
      b: 2
    again: *d
    '''.stripIndent())

assert loaded.prod == [a: 1, b: 2]       // merge keys apply
assert loaded.again == [a: 1, b: 9]      // aliases resolve
assert loaded.again.is(loaded.defaults)  // to the same object
```

`SafeConstructor` matters: it produces only maps, lists, and scalars, and refuses the tagged-type instantiation that makes careless YAML loading dangerous. Differences from `YamlSlurper` to be aware of:

- it follows YAML 1.1 typing, so an untagged timestamp becomes a `java.util.Date` where `YamlSlurper` returns a `String`;
- input is capped at 3MB of code points by default, and a document may use at most 50 aliases of non-scalar nodes — both raised via `LoaderOptions` if needed, and the alias cap is what keeps an alias-expansion document from being a denial of service;
- a self-referential anchor constructs a genuinely cyclic object graph, which naive traversal — `toString()`, `JsonOutput.toJson` — will not survive.

Builders

Another way to create YAML from Groovy is to use `YamlBuilder`. The builder provide a DSL which allows to formulate an object graph which is then converted to YAML.

```
        def builder = new YamlBuilder()
        builder.records {
            car {
                name 'HSV Maloo'
                make 'Holden'
                year 2006
                country 'Australia'
                homepage new URL('http://example.org')
                record {
                    type 'speed'
                    description 'production pickup truck with speed of 271kph'
                }
            }
        }

        assert builder.toString() == '''---
records:
  car:
    name: "HSV Maloo"
```

```
    make: "Holden"
    year: 2006
    country: "Australia"
    homepage: "http://example.org"
    record:
      type: "speed"
      description: "production pickup truck with speed of 271kph"
'''
```

Typed writing

`YamlBuilder.toYaml(object)` serializes a typed object directly to YAML using Jackson databinding:

```
static class ServerConfig {
    String host
    int port
}

@Test
void testToYaml() {
    def config = new ServerConfig(host: 'localhost', port: 8080)
    def yaml = YamlBuilder.toYaml(config)
    assert yaml.contains('host:')
    assert yaml.contains('localhost')
    assert yaml.contains('port:')
    assert yaml.contains('8080')
}
```

Processing TOML

Groovy has an optional `groovy-toml` module which provides support for converting between Groovy objects and TOML. The classes dedicated to TOML serialization and parsing are found in the `groovy.toml` package.

TomlSlurper

`TomlSlurper` is a class that parses TOML text or reader content into Groovy data structures (objects) such as maps, lists and primitive types like `Integer`, `Double`, `Boolean` and `String`.

The class comes with a bunch of overloaded `parse` methods plus some special methods such as `parseText` and others. For the next example we will use the `parseText` method. It parses a TOML `String` and recursively converts it to a list or map of objects. The other `parse*` methods are similar in that they return a TOML `String` but for different parameter types.

```
        def ts = new TomlSlurper()
        def toml = ts.parseText '''
language = "groovy"
```

```
sudo = "required"
dist = "trusty"
before_script = [ "unset _JAVA_OPTIONS\\n\\n    \\n" ]

[[matrix.include]]
jdk = "openjdk10"

[[matrix.include]]
jdk = "oraclejdk9"

[[matrix.include]]
jdk = "oraclejdk8"
'''

        assert 'groovy' == toml.language
        assert 'required' == toml.sudo
        assert 'trusty' == toml.dist
        assert ['openjdk10', 'oraclejdk9', 'oraclejdk8'] ==  toml.matrix.include.jdk
        assert ['unset _JAVA_OPTIONS'] == toml.before_script*.trim()
```

Notice the result is a plain map and can be handled like a normal Groovy object instance. `TomlSlurper` parses the given TOML as defined by the Tom's Obvious, Minimal Language.

As `TomlSlurper` is returning pure Groovy object instances without any special TOML classes in the back, its usage is transparent. In fact, `TomlSlurper` results conform to GPath expressions. GPath is a powerful expression language that is supported by multiple slurpers for different data formats (`XmlSlurper` for XML being one example).

NOTE For more details please have a look at the section on GPath expressions.

The following table gives an overview of the TOML types and the corresponding Groovy data types:

TOML	Groovy
string	`java.lang.String`
number	`java.lang.BigDecimal` or `java.lang.Integer`
object	`java.util.LinkedHashMap`
array	`java.util.ArrayList`
true	`true`
false	`false`
null	`null`
offset/local date-time, local date, local time	`java.lang.String` (see the note below)

NOTE Whenever a value in TOML is `null`, `TomlSlurper` supplements it with the Groovy `null` value. This is in contrast to other TOML parsers that represent a `null` value with a

library-provided singleton object.

NOTE For the untyped `parse`/`parseText` API, TOML date and time values are returned as `String`. This is a limitation of the underlying Jackson TOML support, which only surfaces `java.time.*` types when a target type is supplied. Use the typed `parseAs`/`parseTextAs` API (see Typed date and time values) for `java.time.LocalDate`, `java.time.LocalTime`, `java.time.LocalDateTime`, and `java.time.OffsetDateTime` fidelity.

Typed parsing

`TomlSlurper` can parse TOML directly into typed objects using Jackson databinding. Standard Jackson annotations such as `@JsonProperty` and `@JsonFormat` are supported for property mapping and type conversion:

```
static class ServerConfig {
    String host
    int port
}
```

```
        def config = new TomlSlurper().parseTextAs(ServerConfig, '''
host = "localhost"
port = 8080
''')
        assert config.host == 'localhost'
        assert config.port == 8080
```

NOTE For simple cases, Groovy's `as` coercion also works with the untyped result: `def config = new TomlSlurper().parseText(toml) as ServerConfig`. The `parseTextAs` method uses Jackson databinding, which supports richer annotation-driven mapping via `@JsonProperty`, `@JsonFormat`, etc.

Typed date and time values

When a target type declares `java.time.*` fields, the typed `parseAs`/`parseTextAs` API and `TomlBuilder.toToml` round-trip TOML temporal values with full fidelity, including the original zone offset for `OffsetDateTime`:

```
static class Event {
    java.time.OffsetDateTime created
    java.time.LocalDate effective
    java.time.LocalTime windowStart
    java.time.LocalDateTime updated
    String name
}
```

```
def original = new Event(
        created: java.time.OffsetDateTime.parse('1979-05-27T07:32:00-08:00'),
        effective: java.time.LocalDate.of(1979, 5, 27),
        windowStart: java.time.LocalTime.of(7, 32, 0),
        updated: java.time.LocalDateTime.of(1979, 5, 27, 7, 32, 0),
        name: 'demo')

def toml = groovy.toml.TomlBuilder.toToml(original)
def parsed = new TomlSlurper().parseTextAs(Event, toml)

assert parsed.created == original.created           // OffsetDateTime, non-UTC offset
preserved
assert parsed.effective == original.effective       // LocalDate
assert parsed.windowStart == original.windowStart   // LocalTime
assert parsed.updated == original.updated           // LocalDateTime
assert parsed.name == original.name
```

Builders

Another way to create TOML from Groovy is to use `TomlBuilder`. The builder provide a DSL which allows to formulate an object graph which is then converted to TOML.

```
        def builder = new TomlBuilder()
        builder.records {
            car {
                name 'HSV Maloo'
                make 'Holden'
                year 2006
                country 'Australia'
                homepage new URL('http://example.org')
                record {
                    type 'speed'
                    description 'production pickup truck with speed of 271kph'
                }
            }
        }

        assert builder.toString() == '''\
records.car.name = 'HSV Maloo'
records.car.make = 'Holden'
records.car.year = 2006
records.car.country = 'Australia'
records.car.homepage = 'http://example.org'
records.car.record.type = 'speed'
records.car.record.description = 'production pickup truck with speed of 271kph'
'''
```

Typed writing

`TomlBuilder.toToml(object)` serializes a typed object directly to TOML using Jackson databinding:

```
static class ServerConfig {
    String host
    int port
}

@Test
void testToToml() {
    def config = new ServerConfig(host: 'localhost', port: 8080)
    def toml = TomlBuilder.toToml(config)
    assert toml.contains('host')
    assert toml.contains('localhost')
    assert toml.contains('port')
    assert toml.contains('8080')
}
```

Processing CSV

Groovy has an optional `groovy-csv` module which provides support for reading and writing CSV (RFC 4180) data. The classes are found in the `groovy.csv` package.

CsvSlurper

`CsvSlurper` parses CSV text into a list of maps, where each row becomes a map keyed by the column headers from the first row. Values are returned as strings.

```
def csv = new CsvSlurper().parseText('name,age\nAlice,30\nBob,25')
assert csv.size() == 2
assert csv[0].name == 'Alice'
assert csv[0].age == '30'
assert csv[1].name == 'Bob'
```

Rows support dynamic property access using the header names:

```
def csv = new CsvSlurper().parseText('''\
    name,city,country
    Alice,London,UK
    Bob,Paris,France'''.stripIndent())
assert csv[0].city == 'London'
assert csv[1].country == 'France'
```

Configuration

The separator character and quote character can be customised:

```
def csv = new CsvSlurper().setSeparator((char) '\t').parseText('name\tage\nAlice\t30')
assert csv[0].name == 'Alice'
assert csv[0].age == '30'
```

Quoted fields follow RFC 4180 — fields containing the separator, newlines, or the quote character are enclosed in quotes, with embedded quotes doubled:

```
def csv = new CsvSlurper().parseText('name,note\nAlice,"hello, world"\nBob,"say
""hi"""')
assert csv[0].note == 'hello, world'
assert csv[1].note == 'say "hi"'
```

CSV without a header row

When the CSV has no header row, call `setUseHeader(false)`. Each row is returned as a map keyed by generated column names `c1`, `c2`, and so on:

```
def csv = new CsvSlurper().setUseHeader(false).parseText('1,2\n3,4')
assert csv.size() == 2
assert csv[0].c1 == '1'
assert csv[1].c2 == '4'
```

Alternatively, supply explicit column names with `setColumns`:

```
def csv = new CsvSlurper().setUseHeader(false).setColumns('x', 'y').parseText
('1,2\n3,4')
assert csv[0].x == '1'
assert csv[1].y == '4'
```

When explicit column names are supplied and `useHeader` remains true, the first row is still consumed as a header row but the supplied names take precedence over the names it contains.

Typed parsing

`CsvSlurper` can parse CSV directly into typed objects using Jackson databinding. Standard Jackson annotations such as `@JsonProperty` and `@JsonFormat` are supported for column name mapping and type conversion. This is particularly useful for CSV since all values are strings — Jackson handles the conversion to numeric, date, and other types automatically:

```
static class Sale {
    String customer
    BigDecimal amount
```

```
}
```

```
def sales = new CsvSlurper().parseAs(Sale,
'customer,amount\nAcme,1500.00\nGlobex,250.50')
assert sales.size() == 2
assert sales[0].customer == 'Acme'
assert sales[0].amount == 1500.00
assert sales[1].customer == 'Globex'
```

Headerless CSV can also be parsed into typed objects. Positional columns are derived from the target type's properties; use `@JsonPropertyOrder` to control the expected column order:

```
@JsonPropertyOrder(['customer', 'amount'])
static class Order {
    String customer
    BigDecimal amount
}
```

```
def orders = new CsvSlurper().setUseHeader(false)
        .parseAs(Order, 'Acme,1500.00\nGlobex,250.50')
assert orders.size() == 2
assert orders[0].customer == 'Acme'
assert orders[1].amount == 250.50
```

NOTE CSV has no native types — every cell is text, so the untyped `parse`/`parseText` API always returns `String` values, including for date/time-looking columns. Use the typed `parseAs` API with `java.time.*` fields (see Typed date and time values) for `java.time.LocalDate`, `java.time.LocalTime`, `java.time.LocalDateTime`, and `java.time.OffsetDateTime` fidelity.

CsvBuilder

`CsvBuilder` converts collections of maps or typed objects to CSV. The keys of the first map are used as column headers.

```
def data = [
    [name: 'Alice', age: 30],
    [name: 'Bob', age: 25]
]
def csv = CsvBuilder.toCsv(data)
assert csv.contains('name,age')
assert csv.contains('Alice,30')
assert csv.contains('Bob,25')
```

Typed writing

`CsvBuilder` can also write typed objects. Jackson annotations are supported for column naming and formatting:

```
static class Product {
    String name
    BigDecimal price
}
```

```
def products = [new Product(name: 'Widget', price: 9.99),
                new Product(name: 'Gadget', price: 24.50)]
def csv = CsvBuilder.toCsv(products, Product)
assert csv.contains('name,price')
assert csv.contains('Widget,9.99')
assert csv.contains('Gadget,24.5')
```

Columns for records appear in component declaration order:

```
record Measurement(String name, BigDecimal value, String unit) {}
```

```
def data = [new Measurement('temp', 21.5, 'C')]
def csv = CsvBuilder.toCsv(data, Measurement)
assert csv.startsWith('name,value,unit')
assert csv.contains('temp,21.5,C')
```

For other classes the column order is alphabetical. Use `@JsonPropertyOrder` on the type to set an explicit column order.

Round-trip

CSV written by `CsvBuilder` can be read back with `CsvSlurper`:

```
def original = [[name: 'Alice', age: '30'], [name: 'Bob', age: '25']]
def csv = CsvBuilder.toCsv(original)
def parsed = new CsvSlurper().parseText(csv)
assert parsed[0].name == 'Alice'
assert parsed[1].age == '25'
```

Typed date and time values

When a target type declares `java.time.*` fields, `CsvBuilder.toCsv` and the typed `CsvSlurper.parseAs` API round-trip temporal values with full fidelity:

```
static class Event {
    java.time.LocalDate day
    java.time.LocalTime windowStart
    java.time.LocalDateTime updated
    java.time.OffsetDateTime created
    String label
}
```

```
def original = [new Event(
        day: java.time.LocalDate.of(1979, 5, 27),
        windowStart: java.time.LocalTime.of(7, 32, 0),
        updated: java.time.LocalDateTime.of(1979, 5, 27, 7, 32, 0),
        created: java.time.OffsetDateTime.parse('1979-05-27T07:32:00-08:00'),
        label: 'first')]

def csv = CsvBuilder.toCsv(original, Event)
def parsed = new CsvSlurper().parseAs(Event, csv)

assert parsed[0].day == original[0].day                  // LocalDate
assert parsed[0].windowStart == original[0].windowStart  // LocalTime
assert parsed[0].updated == original[0].updated          // LocalDateTime
assert parsed[0].created == original[0].created          // OffsetDateTime, non-UTC
offset preserved
assert parsed[0].label == original[0].label
```

Processing Markdown

Groovy has an optional `groovy-markdown` module which provides support for parsing CommonMark Markdown. The classes are found in the `groovy.markdown` package.

MarkdownSlurper

`MarkdownSlurper` parses Markdown text into a `MarkdownDocument` backed by nested lists and maps. Each node is a `Map` with a `type` key plus type-specific fields:

```
def doc = new MarkdownSlurper().parseText('# Hello World')
def h = doc.headings[0]
assert h.level == 1
assert h.text == 'Hello World'
```

The raw structure supports all standard Groovy list/map operations. The document also exposes convenience properties — `headings`, `codeBlocks`, `links`, `tables` — that recursively walk the tree.

Extracting code blocks

A common pattern when consuming LLM output is to pull fenced code blocks by language:

```
def md = '''
    Some intro text.

    ```groovy
    println 'hi'
    ```

    ```sql
    SELECT 1
    ```
    '''.stripIndent()
def doc = new MarkdownSlurper().parseText(md)
def groovySnippets = doc.codeBlocks.findAll { it.lang == 'groovy' }*.text
assert groovySnippets == ["println 'hi'\n"]
```

`codeBlocks` walks the entire tree, so blocks nested inside block quotes or list items are included.

Links

```
def doc = new MarkdownSlurper().parseText('See [docs](https://example.com/docs).')
def link = doc.links[0]
assert link.href == 'https://example.com/docs'
assert link.text == 'docs'
```

Sections

`section(headingText)` returns the nodes between the given heading and the next heading of equal or higher level — useful for parsing structured agent replies:

```
def md = '''
    # Title

    ## Summary

    The work is on track.

    ## Next Steps

    - Item one
    - Item two
    '''.stripIndent()
def doc = new MarkdownSlurper().parseText(md)
def summary = doc.section('Summary')
assert summary[0].type == 'paragraph'
assert summary[0].children[0].value == 'The work is on track.'

def next = doc.section('Next Steps')
assert next[0].type == 'list'
```

```
assert next[0].items*.text == ['Item one', 'Item two']
```

Tables (optional)

GFM-style tables are supported when the `org.commonmark:commonmark-ext-gfm-tables` jar is on the classpath. Call `enableTables(true)` on the slurper:

```
def md = '''
    | name  | age |
    |-------|-----|
    | Alice | 30  |
    | Bob   | 25  |
    '''.stripIndent()
def doc = new MarkdownSlurper().enableTables(true).parseText(md)
def rows = doc.tables[0].rows
assert rows[0].name == 'Alice'
assert rows[0].age == '30'
assert rows[1].name == 'Bob'
```

Each row is returned as a `Map` keyed by header.

Node types

type	fields
`heading`	`level`, `text`, `children`
`paragraph`	`children`
`code_block`	`lang`, `text`
`list`	`ordered`, `items` (`start` for ordered lists)
`list_item`	`text`, `children`
`block_quote`	`children`
`link`	`href`, `title`, `text`, `children`
`image`	`src`, `title`, `alt`
`text`	`value`
`inline_code`	`text`
`emphasis`, `strong`	`children`
`html_block`, `html_inline`	`text`
`thematic_break`	(no fields)
`hard_line_break`, `soft_line_break`	(no fields)
`table`	`headers`, `alignments`, `rows`

Untrusted input

Prefer parsing Markdown from trusted sources. Like the sibling `JsonSlurper` and `XmlSlurper`, `MarkdownSlurper` is a convenience parser, not a security boundary, so the safest choice is not to hand it attacker-controlled input. If you must, bound the input by size yourself before parsing and treat the parsed result defensively.

As a backstop, `MarkdownSlurper` caps how deeply a document may nest (both block containers and inline emphasis), so that a small but pathologically nested input — thousands of nested block quotes, or a long run of emphasis markers — is reported as a `MarkdownRuntimeException` rather than being allowed to exhaust the stack with a `StackOverflowError`. The limit defaults to `1000` and can be set per instance or globally via the `groovy.markdown.maxNestingDepth` system property; a value of `0` or less disables the check.

```
def slurper = new MarkdownSlurper()
slurper.maxNestingDepth = 100   // cap block nesting (default 1000; 0 disables the
check)

// a hostile, deeply nested document is rejected, not allowed to crash the parser
def rejected = false
try {
    slurper.parseText(('>' * 5000) + ' text')
} catch (MarkdownRuntimeException ignored) {
    rejected = true
}
assert rejected
```

This cap is a robustness backstop, not a licence to treat the parser as hardened.

Built-in auxiliary type checkers

Introduction

Groovy's static nature includes an extensible type-checking mechanism. This mechanism allows users to selectively strengthen or weaken type checking as needed to cater for scenarios where the standard type checking isn't sufficient.

This chapter catalogues the type-checking extensions that ship with Groovy. For the API used to write your own, see the Type checking extensions chapter.

Groovy offers a suite of built-in type checkers that provide additional compile-time verification for specific concerns:

Checker	What it does

RegexChecker	Validates regex patterns at compile time — no more `PatternSyntaxException` at runtime
	`assert 'Apple' =~ /(a|A)./  // ✔ okay`
	`assert 'apple' =~ /(a|A./  // ✖ compile error: unclosed group`

FormatStringChecker	Validates `printf`/`format` specifiers against argument types — catches mismatches at compile time
	`String.format('%d: %s', 42, 'ok')  // ✔ okay`
	`String.format('%d: %s', 'x', 42)  // ✖ IllegalFormatConversion: d != String`

NullChecker	Detects potential null dereferences using annotations and optional flow analysis
	`if (name != null) name.size()  // ✔ null guard`
	`name?.size()  // ✔ safe navigation`
	`name.size()  // ✖ potential null dereference`

ModifiesChecker	Verifies methods only modify declared fields — humans and AI know what changed and what didn't
	`@Modifies({ this.balance })`
	`void deposit(n) { balance += n }  // ✔ balance is declared`
	`@Modifies({ })`
	`void withdraw(n) { balance -= n }  // ✖ balance not in @Modifies`

PurityChecker	Enforces that `@Pure` methods have no side effects, with configurable strictness
	`@Pure int twice() { value * 2 }  // ✔ reads only`
	`@Pure int bad()  { count++ }  // ✖ field mutation`

CombinerChecker	Verifies a combiner passed to a parallel reduction carries the associativity contract those methods silently require
	`list.injectParallel(0, Maths.&add)  // ✔ @Associative method`
	`list.injectParallel(0) { a, b -> a - b }  // ✖ non-associative: undefined result`

Checker	What it does

MonadicChecker	Teaches `@CompileStatic`/`@TypeChecked` about the `DO` macro's desugared output so monadic comprehensions over `Optional`, `Stream`, `Awaitable`, and `@Monadic` types type-check correctly
	`@CompileStatic(extensions = 'groovy.typecheckers.MonadicChecker')`
	`def m = DO { x <- Optional.of(1); yield x + 1 }`

MonadicShapeChecker	Lint for hand-written `flatMap`/`map` chains over the standard carriers — catches `bind` returning a non-carrier or `map` returning the same carrier
	`Optional.of(1).flatMap { x -> x + 1 }  // ✖ bind returned Integer, not Optional`
	`Optional.of(1).map { x -> Optional.of(x) }  // ✖ map returned Optional, use flatMap`

These checkers work with annotations from multiple libraries (JSpecify, JetBrains, Checker Framework, and others) — matching by simple name so you can use whichever annotation library your project already depends on.



`RegexChecker` (since Groovy 4.0.0) and `FormatStringChecker` (since Groovy 5.0.0) are stable. The remaining checkers are new in Groovy 6.0.0 and remain incubating; their options and semantics may evolve in a subsequent 6.x release based on usage feedback.

Why these checkers matter

In a world which seems to be having an ever-increasing focus on AI, here are numerous reasons you might want to make liberal use of checkers and their annotations:

- The checker declarations and associated annotations, along with other Groovy annotations used with AST transforms, e.g. `@Invariant` from `groovy-contracts`, form a declarative specification of program behavior. This specification replaces "read the body" reasoning allowing humans and AI to understand system behavior at scale.
- When an AI agent generates or modifies code, no human may fully understand the result. Type checkers act as a compile-time safety net on what those programs can do.
- If we can treat our systems as composable black boxes each with well specified behavior we can more easily start to apply compositional reasoning to our systems.
- Groovy's dynamic features (metaprogramming, runtime dispatch, invokeMethod) are powerful but can make static reasoning hard. AI models can hallucinate about what dynamic code does. Type checkers let teams selectively fill in the missing gaps in static reasoning, that AI would otherwise struggle with.
- As AI agents increasingly write, review, and deploy code autonomously, we need machine-

checkable invariants which form trust boundaries in agentic workflows.

As an example, let's explore how combining some checkers and design-by-contract annotations allow both humans and AI to reason about method behavior without reading method implementations. Consider this annotated class:

```
@Invariant({ balance >= 0 })
class Account {
    BigDecimal balance = 0
    List<String> log = []

    @Requires({ amount > 0 })
    @Ensures({ balance == old.balance + amount })
    @Modifies({ [this.balance, this.log] })
    void deposit(BigDecimal amount) {
        balance += amount
        log.add("deposit $amount")
    }

    @Requires({ amount > 0 && amount <= balance })
    @Ensures({ balance == old.balance - amount })
    @Modifies({ [this.balance, this.log] })
    void withdraw(BigDecimal amount) {
        balance -= amount
        log.add("withdraw $amount")
    }

    @Pure
    BigDecimal available() { balance }
}
```

When analyzing a sequence of calls:

```
account.deposit(100)
account.withdraw(30)
def bal = account.available()
```

With annotations, each call is a self-contained specification — 3 linear reasoning steps:

1. `deposit(100)`: `@Requires` met (100 > 0), `@Ensures` gives `balance == old + 100`, `@Modifies` proves only `balance` and `log` changed
2. `withdraw(30)`: `@Requires` met (30 > 0, 30 ⇐ 100), `@Ensures` gives `balance == 100 - 30 = 70`, `@Modifies` proves `withdraw` didn't undo the deposit
3. `available()`: `@Pure` proves no side effects — just returns `balance` (70)

This is human and AI reasoning that scales!

Without annotations, the analyzer must read every method body, verify what each one modifies (2

fields × 3 calls = 6 "did this change?" questions), re-verify earlier state after later calls, and check whether `available()` has hidden side effects. This grows as *O(fields × calls × call_depth)*.

Using the checkers

In the examples which follow, we explicitly show declaring the type checker to use. Groovy's compiler customization mechanisms allow you to simplify application of such checkers, e.g. make it apply globally using a compiler configuration script, as just one example.

Checking Regular Expressions

Stable since Groovy 4.0.0.

A regular expression (regex) defines a search pattern for text. The pattern can be as simple as a single character, a fixed string, or a complex expression containing special characters describing a pattern. The JDK has special regex classes and Groovy adds some special syntactic sugar and functionality on top of the JDK classes.

Here is an example the checks that the String 'Foo' matches a regular expression:

```
assert 'foo'.matches(/[Ff].{2}\b/)
```

The regex pattern is made up of three parts:

- `[Ff]` is a character class matching upper or lowercase 'f',
- `.{2}` means exactly two occurrences of the `.` character class which matches any character
- `\b` matches a word boundary

There are more options we could add here such as grouping, lookahead and look-behind, to name just a few. With so many options and special characters, it isn't hard to create an invalid regular expression pattern. Such invalid patterns are typically found at runtime with a runtime exception.

The goal of the `RegexChecker` is to find such errors at compile-time. If compilation of the expressions succeeds, they are guaranteed not to fail at runtime.

Typical usage

The previous example can be type checked as follows:

```
@TypeChecked(extensions='groovy.typecheckers.RegexChecker')
static main(args) {
    assert 'foo'.matches(/[Ff].{2}\b/)
}
```

If you have many examples where such checking is desirable, you could consider automatically adding this extension to your compilation process.

Covered methods

The methods which are checked include:

`java.util.regex.Pattern`

```
static Pattern compile(String regex)
static Pattern compile(String regex, int flags)
static boolean matches(String regex, CharSequence input)
```

`java.util.regex.Matcher`

```
String group(int group)
matcher[matchCount][groupCount]  // getAt shorthand
```

Operators

```
~/regex/                 // pattern operator
'string' =~ /regex/      // find operator
'string' ==~ /regex/     // (exact) match operator
```

`String` methods taking a regex argument (`matches` *since Groovy 4.0.0, the rest since 6.0.0)*

```
'string'.matches(regex)             'string'.split(regex)
'string'.find(regex)                'string'.findAll(regex)
'string'.findGroups(regex)          'string'.findAllGroups(regex)
'string'.eachMatch(regex) { }       'string'.splitEachLine(regex) { }
'string'.replaceAll(regex, repl)    'string'.replaceFirst(regex, repl)
```

`groovy.util.regex.RegexGuard` *(since Groovy 6.0.0)*

```
static boolean matches(Object pattern, Object input, long millis)
static Matcher matcher(Object pattern, Object input, long millis)
static boolean matchRegex(Object input, Object pattern, long millis)
static Matcher findRegex(Object input, Object pattern, long millis)
```

findGroups destructuring

`findGroups` returns the full match followed by each capture group, so it pairs naturally with multiple assignment. Where the regex can be determined, `RegexChecker` verifies that the number of variables does not exceed the full match plus one per group — more could never all be bound, even on a successful match:

```
def semver = /(\d+)\.(\d+)\.(\d+)(?:-(.+))?/

def (all, major, minor, patch, qualifier) = '6.0.0-beta-2'.findGroups(semver)  // OK
```

```
def (a, b, c, d, e, f) = '6.0.0-beta-2'.findGroups(semver)  // one variable too many
```

```
[Static type checking] - Too many variables 6 for findGroups: a regex with 4 groups
yields at most 5 elements (the full match plus each group)
```

Using fewer variables than available elements is legitimate and not flagged.

Guarded regex evaluation

Deadline-guarded evaluations via `groovy.util.regex.RegexGuard` are checked in the same way as the regex operators: the pattern argument is validated where a constant regex string can be determined, and matcher group counts are tracked for the matcher-returning methods. This also means code within a `@groovy.transform.SafeRegex` scope keeps its full checking — the transform rewrites the find and match operators to `RegexGuard` calls (the operator-shaped `matchRegex` and `findRegex` entry points, which take the input first), and `RegexChecker` recognizes the rewritten form, including the smarter matcher type inference:

```
@TypeChecked(extensions='groovy.typecheckers.RegexChecker')
@SafeRegex(millis = 200)
static main(args) {
    var m = 'foobar' =~ /(...)(...)/     // guarded by @SafeRegex, still checked
    assert m[0][1] == 'foo'              // group inference still applies
    assert m[0][2] == 'bar'

    assert RegexGuard.matches(/(\d{4})-(\d{1,2})-(\d{1,2})/, '2020-12-31', 200)
}
```

The RegexChecker will check any method calls listed above. It might find such methods when they involve explicit static method calls or when the receiver's type can be inferred.

The regular expression must be a *constant-like* String. Any group-count values must be a *constant-like* integer. By *constant-like*, we mean, either an explicit String literal, or a field with an initializer expression or a local variable with an initializer expression. These possibilities are shown in the following code, which contains 3 regular expressions:

```
static final String TWO_GROUPS_OF_THREE = /(...)(...)/  ①

@TypeChecked(extensions='groovy.typecheckers.RegexChecker')
static main(args) {
    var m = 'foobar' =~ TWO_GROUPS_OF_THREE  //  ②
    assert m.find()
    assert m.group(1) == 'foo'
    assert m.group(2) == 'bar'  ③

    var pets = ['cat', 'dog', 'goldfish']
    var shortNamed = ~/\w{3}/  ④
```

```
    assert pets.grep(shortNamed) == ['cat', 'dog']  ⑤

    assert Pattern.matches(/(\d{4})-(\d{1,2})-(\d{1,2})/, '2020-12-31')  ⑥
}
```

① A constant String regex

② Using the constant regex with the find operator

③ Checking the matcher's second group

④ A local variable containing a regex pattern

⑤ Using the pattern with grep

⑥ An API call passing a String literal regex

Errors detected

Luckily, in the above example, we made no errors in our regex definitions, and the code compiles and executes successfully.

If however, we did make certain errors in those definitions, then we would expect the potential of runtime errors. Using `RegexChecker`, we can find certain kinds of errors during compilation. Let's look at some examples.

Suppose at □ we looked for the third group instead of the second:

```
assert m.group(3) == 'bar'
```

We would see the following error at compile-time:

```
[Static type checking] - Invalid group count 3 for regex with 2 groups
```

Alternatively, suppose at □ we accidentally left off the closing curly brace:

```
var shortNamed = ~/\w{3/
```

We would see the following error at compile-time:

```
[Static type checking] - Bad regex: Unclosed counted closure near index 4
```

Alternatively, suppose at □ we accidentally left off the closing round bracket for the last (day of month) group:

```
assert Pattern.matches(/(\d{4})-(\d{1,2})-(\d{1,2}/, '2020-12-31')
```

We would see the following error at compile-time:

```
[Static type checking] - Bad regex: Unclosed group near index 26
```

Over-and-above these examples, detected errors include:

- Bad class syntax
- Bad named capturing group
- Dangling meta character
- Empty character family
- Illegal character range
- Illegal repetition
- Illegal/unsupported escape sequence
- Look-behind group does not have an obvious maximum length
- Unexpected character
- Unclosed character family
- Unclosed character class
- Unclosed counted closure
- Unclosed group
- Unknown character property name
- Unknown Unicode property
- Unknown group type
- Unknown inline modifier
- Unknown character name
- Unmatched closing ')'

Checking Format Strings

Stable since Groovy 5.0.0.

The `format` methods in `java.util.Formatter`, and other similar methods, support formatted printing in the style of C's `printf` method with a format string and zero or more arguments.

Let's consider an example which produces a string comprised of three terms:

- a floating-point representation (`%f`) of PI (with 2 decimal places of precision),
- the hex representation (`%X`) of 15 (as two uppercase digits with a leading 0),
- and the Boolean (`%B`) `True` (in uppercase).

The assertion checks our expectations:

```
assert String.format('%4.2f %02X %B', Math.PI, 15, true) == '3.14 0F TRUE'
```

This is a powerful method supporting numerous conversions and flags. If the developer supplies incorrect conversions or flags, they will receive one of numerous possible runtime errors. As examples, consider the following mistakes and resulting runtime exceptions:

- supplying a String as the parameter for either of the first two arguments results in an `IllegalFormatConversionException`,
- leaving out the last argument results in a `MissingFormatArgumentException`,
- supplying the *leading zero* flag for the Boolean parameter results in a `FlagsConversionMismatchException`.

The goal of the `FormatStringChecker` is to eliminate a large number of such runtime errors. If the API call passes type checking, it will be guaranteed to succeed at runtime.

Typical usage

Here is an example of correctly using some of these methods with type checking in place:

```
@TypeChecked(extensions='groovy.typecheckers.FormatStringChecker')
static main(args) {
    assert String.format('%x', 1023) == '3ff'
    assert String.format(Locale.US, '%x', 1024) == '400'
    assert sprintf('%s%s', 'foo', 'bar') == 'foobar'
    assert new Formatter().format('xy%s', 'zzy').toString() == 'xyzzy'
    def baos = new ByteArrayOutputStream()
    new PrintStream(baos).printf('%-4c|%6b', 'x' as char, true)
    assert baos.toString() == 'x   |  true'
}
```

Over and above these methods, if you have your own method of this style, you can annotate it with `@FormatMethod`, and it will also be checked.

```
@FormatMethod
static String log(String formatString, Object... args) {
    sprintf(formatString, args)
}

@TypeChecked(extensions='groovy.typecheckers.FormatStringChecker')
static main(args) {
    assert log('%x', 1023) == '3ff'
    assert log('%x', 1024) == '400'
}
```

You can use the `FormatMethod` annotation provided in the `groovy-typecheckers` module or the similarly named annotation from the Java-based checker framework.

Covered methods

The format string methods have the following characteristics:

- The first argument must be of type `java.util.Locale` or `java.lang.String`.
- If the first argument is of type `Locale`, the second argument must be of type `String`.
- The `String` argument is treated as a format string containing zero or more embedded format specifiers as well as plain text. The format specifiers determine how the remaining arguments will be used within the resulting output.

The `FormatStringChecker` ensures that the format string is valid and the remaining arguments are compatible with the embedded format specifiers.

The methods which are checked include:

`java.lang.String`

```
static String format(String format, Object... args)
static String format(Locale l, String format, Object... args)
String formatted(Object... args)  // JDK15+
```

`java.util.Formatter`

```
String format(String format, Object... args)
String format(Locale l, String format, Object... args)
```

`java.io.PrintStream`

```
PrintStream printf(String format, Object... args)
PrintStream printf(Locale l, String format, Object... args)
```

DGM methods

```
printf(String format, Object... args)
printf(Locale l, String format, Object... args)
sprintf(String format, Object... args)
sprintf(Locale l, String format, Object... args)
```

The format string checker looks for a constant literal format string. For arguments, it also looks for constant literals or otherwise makes checks based on inferred type. When looking for constants, the checker will find inline literals, local variables with a constant initializer, and fields with a constant initializer. Here are some examples:

```
static final String FIELD_PATTERN = '%x'
static final int FIELD_PARAM = 1023

@TypeChecked(extensions='groovy.typecheckers.FormatStringChecker')
```

```
static main(args) {
    assert sprintf('%x', 1024) == '400'  ①

    assert sprintf(FIELD_PATTERN, FIELD_PARAM) == '3ff'  ②

    var local_var_pattern = '%x'
    var local_var_param = 1022
    assert sprintf(local_var_pattern, local_var_param) == '3fe'  ③

    assert sprintf('%x%s', theParam(), 'a' + 'b') == '3fdab'  ④
}

static int theParam() { 1021 }
```

① Constant literals

② Fields with an initializer

③ Local variables with an initializer

④ Checking of the parameter will be based on inferred type

Errors detected

Here are some examples of the kinds of errors detected.

- The arguments much match the type of the format classifier, e.g. we can't use the character classifier with a boolean argument:

  ```
  sprintf '%c', true
  ```

  We would see the following error at compile-time:

  ```
  [Static type checking] - IllegalFormatConversion: c != java.lang.Boolean
  ```

- We shouldn't specify precision for non-scientific arguments, e.g. we shouldn't expect 7 decimal places of precision for an integral value:

  ```
  String.format('%1.7d', 3)
  ```

  We would see the following error at compile-time:

  ```
  [Static type checking] - IllegalFormatPrecision: 7
  ```

- We should specify only known format specifier conversions, e.g. v isn't known:

```
System.out.printf('%v', 7)
```

We would see the following error at compile-time:

```
[Static type checking] - UnknownFormatConversion: Conversion = 'v'
```

The complete list of errors detected include:

- duplicate format flags
- illegal format flags
- illegal format precision values
- unknown format conversions
- illegal format conversions
- format flags conversion mismatches
- missing arguments

Checking Null Safety (Incubating)

Null pointer dereferences are one of the most common sources of runtime errors. The `NullChecker` performs compile-time null-safety analysis, detecting potential null dereferences and null-safety violations before code is executed.

Languages like Kotlin and Swift distinguish nullable and non-nullable types throughout the type system, eliminating null errors by construction. Groovy takes a more pragmatic approach: flow analysis can catch many null issues without annotations (in `strict` mode), and `@Nullable`/`@NonNull` annotations provide additional precision where needed. This lets you adopt null safety incrementally — annotate the critical boundaries first, and let flow analysis handle the rest.

Two modes are provided:

- `NullChecker` — **Annotation-based**: Checks code annotated with `@Nullable` and `@NonNull` (or equivalent) annotations.
- `NullChecker(strict: true)` — **Annotation-based** + **flow-sensitive**: Includes all checks from `NullChecker` plus flow-sensitive tracking through assignments and control flow, detecting potential null dereferences even in unannotated code.

For code bases with a mix of strictness requirements, apply `NullChecker` to relaxed code and `NullChecker(strict: true)` to strict code via per-class or per-method `@TypeChecked` annotations.

Typical usage

Here is an example showing a method that safely handles a `@Nullable` parameter:

```
@TypeChecked(extensions='groovy.typecheckers.NullChecker')
```

```
int safeLength(@Nullable String text) {
    if (text != null) {
        return text.length()    // ok: null guard
    }
    return -1
}

assert safeLength('hello') == 5
assert safeLength(null) == -1
```

The checker verifies that `text` is only dereferenced inside the null guard. Without the `if (text != null)` guard, the checker would report a compile-time error.

Supported annotations

The checker recognizes nullability annotations by their simple name, so annotations from any package are supported. The following annotation names are recognized:

Nullable	`@Nullable`, `@CheckForNull`, `@MonotonicNonNull`
Non-null	`@NonNull`, `@NotNull`, `@Nonnull`
Non-null by default (class-level)	`@NonNullByDefault`, `@NullMarked`, `@ParametersAreNonnullByDefault`
Opt-out (class-level)	`@NullUnmarked`

The annotation classes must be on the compile classpath so the Groovy compiler can resolve them. Both declaration-targeted and type-use (`ElementType.TYPE_USE`) annotations are recognized, including type-use annotations found in already compiled Java or Groovy classes on the classpath, where they reside in type-annotation bytecode attributes rather than on the member declarations. Since the checker matches by simple name, you are free to choose any annotation library:

- JSpecify: `@Nullable` / `@NullMarked` / `@NullUnmarked`
- `javax.annotation.Nullable` / `javax.annotation.Nonnull` (JSR-305)
- `org.jetbrains.annotations.Nullable` / `org.jetbrains.annotations.NotNull`
- `edu.umd.cs.findbugs.annotations.Nullable` / `edu.umd.cs.findbugs.annotations.NonNull`
- `org.checkerframework.checker.nullness.qual.Nullable` / `org.checkerframework.checker.nullness.qual.NonNull`

If you prefer not to add an external dependency, you can define your own minimal annotations:

```
import java.lang.annotation.*

@Target([ElementType.PARAMETER, ElementType.METHOD, ElementType.FIELD])
@Retention(RetentionPolicy.CLASS)
@interface Nullable {}

@Target([ElementType.PARAMETER, ElementType.METHOD, ElementType.FIELD])
```

```
@Retention(RetentionPolicy.CLASS)
@interface NonNull {}
```

The following class-level annotations are also recognized:

- `@NullMarked` (JSpecify) — when applied to a class, all parameters, return types, and fields are treated as `@NonNull` unless explicitly annotated `@Nullable`. Use `@NullUnmarked` on a nested class to opt out of this default.
- `@NonNullByDefault` (SpotBugs, Eclipse JDT) — same semantics as `@NullMarked`.
- `@ParametersAreNonnullByDefault` (JSR-305) — when applied to a class, all non-primitive parameters are treated as `@NonNull` unless explicitly annotated `@Nullable`.
- `@groovy.transform.NullCheck` — when applied to a method or class, all non-primitive parameters are treated as effectively `@NonNull` (unless explicitly annotated `@Nullable`). This provides compile-time null checking complementing the runtime checks that `@NullCheck` generates.

Nullable type arguments are also honoured wherever the static type checker's inference carries them through a class or method level type variable: calling `get(0)`, the idiomatic subscript `xs[0]`, or GDK methods like `head()` on a `List<@Nullable String>`, and `get(key)` or `m[key]` on a `Map<String, @Nullable Integer>`, all yield a nullable result which must be guarded or safely dereferenced. This works whether the annotated type appears in source or is read from a compiled dependency:

```
@TypeChecked(extensions='groovy.typecheckers.NullChecker')
class Catalog {
    static int firstTitleLength(List<@Nullable String> titles) {
        titles.get(0).length()              // get(0) may return null
    }
}
```

We would see the following error at compile-time:

```
[Static type checking] - Potential null dereference: 'get()' may return null
```

Wider generics support — nullable type parameters and bounds (`<T extends @Nullable Object>`), wildcards at use sites, `@Nullable` checking of arguments against instantiated parameter types (e.g. `add(null)` on a `List<@NonNull String>`), and array component positions — is not yet covered.

Null-safe access patterns

The checker recognizes several patterns for safely working with nullable values.

Safe navigation using the `?.` operator:

```
@TypeChecked(extensions='groovy.typecheckers.NullChecker')
String greet(@Nullable String name) {
    name?.toUpperCase()                         // ok: safe navigation
}
```

```
assert greet('world') == 'WORLD'
assert greet(null) == null
```

Null guards using `if (x != null)`:

```
@TypeChecked(extensions='groovy.typecheckers.NullChecker')
String process(@Nullable String input) {
    if (input != null) {
        return input.toUpperCase()              // ok: null guard
    }
    return 'default'
}
```

Early exit using `if (x == null) return` or `throw`:

```
@TypeChecked(extensions='groovy.typecheckers.NullChecker')
String process(@Nullable String input) {
    if (input == null) return 'default'         // early exit
    input.toUpperCase()                         // ok: input is non-null here
}
```

Broader guard patterns are also recognized: Groovy-truth guards (`if (x)`), `instanceof` checks, `Objects.nonNull(x)`/`Objects.isNull(x)`, assert statements (`assert x`, `assert x != null`), negated forms (`!x`, `x !instanceof Foo`), and boolean conjunctions and disjunctions with short-circuit semantics (`x != null && x.length() > 0`, `x == null || x.isEmpty()`). Each of these patterns also combines with the early-exit form, e.g. `if (x == null || x.isEmpty()) return`. Guard conditions are honored in `if` statements, `while` loops, and ternary expressions:

```
@TypeChecked(extensions='groovy.typecheckers.NullChecker')
int firstWordLength(@Nullable String phrase) {
    if (phrase == null || phrase.isEmpty()) {  // ok: short-circuit guard
        return 0
    }
    phrase.split(' ')[0].length()              // ok: phrase is non-null here
}

@TypeChecked(extensions='groovy.typecheckers.NullChecker')
String shout(@Nullable String word) {
    word ? word.toUpperCase() + '!' : 'nothing' // ok: Groovy-truth ternary guard
}
```

Validator and assertion methods which guarantee an argument is non-null when they complete normally also narrow that argument: `Objects.requireNonNull(x)` and Guava-style `checkNotNull(x)` throw for a null argument, `assertNotNull(x)` (JUnit 4/5, TestNG, or similar — message parameters are recognized in any position) fails for one, and a fluent `assertThat(x).isNotNull()` chain (AssertJ,

Truth, or similar) fails when `x` is null. Like annotations, these methods are matched by simple name, so any library following the common naming conventions is recognized:

```
@TypeChecked(extensions='groovy.typecheckers.NullChecker')
int labelWidth(@Nullable String label) {
    Objects.requireNonNull(label, 'label must be supplied')
    label.length()                              // ok: requireNonNull throws on null
}
```

Ternary and elvis expressions: the checker examines the branches of ternary (`condition ? a : b`) and elvis (`x ?: fallback`) expressions. A ternary is treated as nullable if either branch can be null, judging each branch under the condition's own guard facts, so `s != null ? s : 'default'` is not considered nullable. An elvis expression is nullable only if its fallback is, since a Groovy-truthy value is never null — making `x ?: 'default'` a recognized way to pass a `@Nullable` value where a non-null value is required:

```
@TypeChecked(extensions='groovy.typecheckers.NullChecker')
class Greeter {
    static String greet(@NonNull String name) { 'Hello ' + name }

    static String welcome(@Nullable String visitor) {
        // greet(visitor)                       // error: visitor may be null
        // greet(visitor?.trim())              // error: safe navigation may yield null
        greet(visitor ?: 'guest')               // ok: elvis fallback is non-null
    }
}
```

In flow-sensitive mode (`NullChecker(strict: true)`), nullable expression results (safe-navigation results, `@Nullable`-returning calls, nullable ternaries) are also tracked through assignments to unannotated variables. The elvis assignment `x ?= nonNullValue` clears any nullable state on `x`.

Errors detected

Here are some examples of the kinds of errors detected.

- Dereferencing a `@Nullable` variable without a null check:

  ```
  def greet = { @Nullable String name ->
      name.toUpperCase()                        // potential null dereference
  }
  ```

  We would see the following error at compile-time:

  ```
  [Static type checking] - Potential null dereference: 'name' is @Nullable
  ```

- Passing `null` to a `@NonNull` parameter:

```
@TypeChecked(extensions='groovy.typecheckers.NullChecker')
class Greeter {
    static String greet(@NonNull String name) {
        name.toUpperCase()
    }
    static void main(String[] args) {
        greet(null)                                 // cannot pass null
    }
}
```

We would see the following error at compile-time:

```
[Static type checking] - Cannot pass null to @NonNull parameter 'name' of 'greet'
```

- Returning `null` from a `@NonNull` method:

```
@TypeChecked(extensions='groovy.typecheckers.NullChecker')
class Greeter {
    @NonNull static String greet() {
        return null                                 // cannot return null
    }
}
```

We would see the following error at compile-time:

```
[Static type checking] - Cannot return null from @NonNull method 'greet'
```

- Assigning `null` to a `@NonNull` field:

```
@TypeChecked(extensions='groovy.typecheckers.NullChecker')
class Config {
    @NonNull String name
    void clear() {
        name = null                                 // cannot assign null
    }
}
```

We would see the following error at compile-time:

```
[Static type checking] - Cannot assign null to @NonNull variable 'name'
```

- Dereferencing the return value of a `@Nullable` method:

```
@TypeChecked(extensions='groovy.typecheckers.NullChecker')
class UserService {
    @Nullable static String findUser() { null }
    static void main(String[] args) {
        findUser().toUpperCase()                  // may return null
    }
}
```

We would see the following error at compile-time:

```
[Static type checking] - Potential null dereference: 'findUser()' may return null
```

The same applies when a `@Nullable` getter is accessed using property syntax, e.g. `user.name` for a `@Nullable getName()` method.

- Dereferencing a safe-navigation result without a further guard:

```
@TypeChecked(extensions='groovy.typecheckers.NullChecker')
String shorten(@Nullable String text) {
    text?.trim().substring(0, 1)              // text?.trim() may be null
}
```

We would see the following error at compile-time:

```
[Static type checking] - Potential null dereference: 'text?.trim()' may be null
```

Since the safe navigation propagates a possible null onwards, the subsequent unguarded dereference should use safe navigation too: `text?.trim()?.substring(0, 1)`.

@MonotonicNonNull and @Lazy

The `@MonotonicNonNull` annotation is useful for lazy initialization patterns. A field annotated with `@MonotonicNonNull` starts as `null` but once assigned a non-null value, it must remain non-null. The checker treats such fields as nullable (requiring null checks before use) but prevents re-assignment to `null` after initialization:

```
@TypeChecked(extensions='groovy.typecheckers.NullChecker')
class LazyService {
    @MonotonicNonNull String cachedValue

    String getValue() {
        if (cachedValue != null) {
            return cachedValue.toUpperCase()    // ok: null guard
        }
        cachedValue = 'computed'
```

```
        return cachedValue.toUpperCase()        // ok: just assigned non-null
    }
}
```

Re-assigning `null` to a `@MonotonicNonNull` field after it has been set is an error:

```
void reset() {
    value = "hello"
    value = null                                // cannot re-null
}
```

We would see the following error at compile-time:

```
[Static type checking] - Cannot assign null to @MonotonicNonNull variable 'value'
after non-null assignment
```

Groovy's `@Lazy` annotation is recognized as an implicit `@MonotonicNonNull`. A `@Lazy` field starts as `null` internally but once its generated getter initializes it, it stays non-null — exactly the monotonic non-null contract. Since `@Lazy` generates a getter that handles initialization automatically, property access to a `@Lazy` field is always safe and will not trigger null dereference warnings. The monotonic guarantee means the checker will also prevent re-assignment to `null` on the backing field after initialization.

Integration with @NullCheck

Groovy's `@NullCheck` annotation generates runtime null checks for method parameters. When the null checkers see `@NullCheck` on a method or class, they treat all non-primitive parameters as effectively `@NonNull`, catching violations at compile time before they would fail at runtime:

```
@NullCheck
@TypeChecked(extensions='groovy.typecheckers.NullChecker')
class Greeter {
    static String greet(String name) {
        "Hello, $name!"
    }
    static void main(String[] args) {
        greet(null)                             // caught at compile time
    }
}
```

We would see the following error at compile-time:

```
[Static type checking] - Cannot pass null to @NonNull parameter 'name' of 'greet'
```

Parameters explicitly annotated `@Nullable` override the `@NullCheck` behavior for the purpose of the

null checkers (though `@NullCheck` will still generate a runtime check for those parameters).

Integration with @NonNullByDefault

Class-level `@NonNullByDefault` (from SpotBugs, Eclipse JDT, or your own definition) makes all parameters, return types, and fields in a class effectively `@NonNull`. Members explicitly annotated `@Nullable` override this default. Similarly, `@ParametersAreNonnullByDefault` (JSR-305) applies only to parameters.

```
@NonNullByDefault
@TypeChecked(extensions='groovy.typecheckers.NullChecker')
class UserService {
    String name

    static String greet(String name) {
        "Hello, $name!"
    }

    static void main(String[] args) {
        greet(null)                             // compile error: params are @NonNull
    }
}
```

We would see the following error at compile-time:

```
[Static type checking] - Cannot pass null to @NonNull parameter 'name' of 'greet'
```

Integration with @Requires and @Ensures

When `groovy-contracts` is on the classpath, `NullChecker` recognises null-safety facts expressed through Design-by-Contract annotations. Top-level `x != null` conjuncts inside a `@Requires` closure mark the corresponding parameter as effectively `@NonNull`; a `result != null` conjunct inside an `@Ensures` closure marks the method return the same way. Both orderings (`x != null` and `null != x`) are recognised. Disjunctions (`||`) are deliberately not used for inference, since they do not establish non-nullness of any individual operand.

```
@TypeChecked(extensions='groovy.typecheckers.NullChecker')
class Greeter {
    @Requires({ name != null })
    static String greet(name) { "Hello, $name!" }

    static void main(String[] args) {
        greet(null)                             // caught at compile time
    }
}
```

We would see the following error at compile-time:

```
[Static type checking] - Cannot pass null to @NonNull parameter 'name' of 'greet'
```

The same applies on the return side with `@Ensures`:

```
@TypeChecked(extensions='groovy.typecheckers.NullChecker')
class Greeter {
    @Ensures({ result != null })
    String greet() {
        return null                              // caught at compile time
    }
}
```

We would see the following error at compile-time:

```
[Static type checking] - Cannot return null from @NonNull method 'greet'
```

This coexists with explicit `@NonNull` annotations: a method marked `@NonNull` that also carries any `@Ensures` postcondition, or that lives in a class with an `@Invariant` class invariant, still has its literal `return null` statements flagged. Without this coordination, the `groovy-contracts` AST transform would rewrite the return statement before the type-checking pass runs, hiding the literal from the checker. The runtime postcondition continues to guard non-literal cases.

Flow-sensitive mode with strict option

`NullChecker` only flags issues involving annotated code. `NullChecker(strict: true)` adds flow-sensitive tracking, detecting potential null dereferences in unannotated code by tracking nullability through variable assignments and control flow:

With `NullChecker(strict: true)`, the following code would be flagged:

```
def x = null
x.toString()                             // x may be null
```

We would see the following error at compile-time:

```
[Static type checking] - Potential null dereference: 'x' may be null
```

Reassigning a non-null value clears the nullable state:

```
@TypeChecked(extensions='groovy.typecheckers.NullChecker(strict: true)')
static main(args) {
    def x = null
    x = 'hello'
    assert x.toString() == 'hello'           // ok: reassigned non-null
```

```
}
```

Strict mode also verifies that each explicitly-annotated `@NonNull` instance field is definitely initialized: at its declaration, in an instance initializer block, or by every declared constructor (a constructor delegating via `this(…)` relies on its delegate). Fields that are merely non-null by default (e.g. under `@NullMarked` without an explicit annotation) are not checked, so idiomatic Groovy property classes constructed via named arguments stay noise-free:

```
@TypeChecked(extensions='groovy.typecheckers.NullChecker(strict: true)')
class Library {
    @NonNull String catalog                 // never assigned
    Library(String catalog) { }
}
```

We would see the following error at compile-time:

```
[Static type checking] - @NonNull field 'catalog' is not initialized by all
constructors
```

The complete list of errors detected include:

- assigning `null` to a `@NonNull` field or variable (including implicit `@NonNull` from `@NonNullByDefault`)
- passing `null` or a `@Nullable` value to a `@NonNull` parameter (including implicit `@NonNull` from `@NullCheck`, `@NonNullByDefault`, or `@ParametersAreNonnullByDefault`)
- returning `null` or a `@Nullable` value from a `@NonNull` method (including implicit `@NonNull` from `@NonNullByDefault`)
- dereferencing a `@Nullable` variable without a null check or safe navigation
- dereferencing the return value of a `@Nullable` method without a null check
- dereferencing a result whose inferred type carries a `@Nullable` type argument reaching it through a class or method level type variable (e.g. `get(0)`, `xs[0]`, or `head()` on a `List<@Nullable String>`)
- re-assigning `null` to a `@MonotonicNonNull` or `@Lazy` field after initialization
- dereferencing a variable known to be null through flow analysis (`strict` mode only)
- an explicitly-annotated `@NonNull` instance field not definitely initialized at its declaration, in an instance initializer block, or by every constructor (`strict` mode only)



For complementary null-related checks such as detecting broken null-check logic, unnecessary null guards before `instanceof`, or `Boolean` methods returning `null`, consider using CodeNarc's null-related rules alongside these type checkers.

Checking @Modifies Frame Conditions (Incubating)

The `ModifiesChecker` verifies that method bodies comply with their `@Modifies` frame condition declarations from the `groovy-contracts` module. It checks that methods only modify the fields and parameters they declare, helping both humans and AI reason about what a method changes — and crucially, what it does not.

Frame conditions are a well-established concept in formal verification. Dafny's `modifies` clause and JML's `assignable` clause serve the same purpose — declaring what a method may change so that everything else is known to be preserved. Groovy's `@Modifies` brings this idea to the JDK ecosystem in a pragmatic, opt-in form. For projects already using JetBrains annotations, the checker also recognises `@Contract(mutates = "…")` which expresses similar semantics.

This checker is opt-in and only activates for methods annotated with `@Modifies`:

```
@TypeChecked(extensions = 'groovy.typecheckers.ModifiesChecker')
class InventoryItem {
    int quantity = 0
    int reserved = 0

    @Modifies({ [this.quantity, this.reserved] })
    void restock(int amount) {
        quantity += amount       // OK: quantity is declared
        reserved = 0             // OK: reserved is declared
    }
}
```

What the checker verifies

Direct field assignments

Assignments to fields (via `this.field = …`, `field = …`, `field++`, `field += …`) must target fields listed in `@Modifies`. Local variable assignments are always allowed.

```
@Modifies({ this.count })
void increment() {
    count++            // OK: count is declared
    def temp = count   // OK: local variable
}

@Modifies({ this.count })
void broken() {
    other = 5          // ERROR: 'other' not declared in @Modifies
}
```

Method calls on `this`

When a method calls another method on `this`, the checker verifies compatibility:

- If the callee has `@Modifies`, its frame must be a subset of the caller's frame.
- If the callee is annotated `@Pure`, the call is always safe.
- If the callee's name is in a known-safe list (e.g., `toString`, `hashCode`), the call is allowed.
- Otherwise, a warning is produced.

```
@Modifies({ [this.items, this.count] })
void addItem(String item) {
    doAdd(item)                 // OK if doAdd has compatible @Modifies
    def n = currentCount()      // OK if currentCount() is @Pure
}
```

Method calls on parameters and variables

When a method calls a method on a variable or parameter:

- If the receiver is in the `@Modifies` set, any call is allowed (the contract declares the caller may modify it).
- If the receiver is **not** in the `@Modifies` set, the checker determines whether the call is non-mutating using a layered approach:
  1. **Immutable receiver type**: calls on `String`, `Integer`, `BigDecimal`, `LocalDate`, etc. are always safe.
  2. **`@Pure` method**: if the called method is annotated `@Pure`, it is safe.
  3. **Known-safe method name**: a curated whitelist of non-mutating method names including `size`, `get`, `contains`, `isEmpty`, `toString`, `iterator`, `stream`, `toList`, etc.
  4. **Unknown**: a warning is produced suggesting the method may modify the receiver.

```
@Modifies({ this.count })
void countItems(List items) {
    count = items.size()         // OK: size() is known-safe
    count = items.toString().length()  // OK: String is immutable
}

@Modifies({ list1 })
void process(List list1, List list2) {
    list1.add('x')              // OK: list1 is in @Modifies
    list2.size()                 // OK: size() is known-safe
    list2.clear()                // WARNING: clear() may modify list2
}
```

Supported annotations from other libraries

The `ModifiesChecker` recognises purity and frame condition annotations from multiple libraries by simple name:

Annotation	Library	Semantics
`@Pure`	Groovy, Checker Framework, Xtext	Implies `@Modifies({})` — no field mutations
`@SideEffectFree`	Checker Framework	Implies `@Modifies({})` — no field mutations
`@Contract(pure = true)`	JetBrains Annotations	Implies `@Modifies({})` — works with CLASS retention
`@Contract(mutates = "…")`	JetBrains Annotations	Parsed as a frame condition on callees (see below)

@Contract(mutates) support

When a callee has `@Contract(mutates = "…")`, the checker parses the mutates string to determine what the callee modifies:

- `mutates = ""` — callee mutates nothing (equivalent to `@Pure`)
- `mutates = "this"` — callee mutates its receiver
- `mutates = "param1"` — callee mutates its first parameter
- `mutates = "this,param1"` — callee mutates both

The checker maps `param1`, `param2`, etc. to the actual arguments at the call site and warns if a mutated argument is not in the caller's `@Modifies` set:

```
@Modifies({ this.items })
void process() {
    addToList(items, 'x')   // OK: items is in @Modifies
    addToList(other, 'x')   // WARNING: other is not in @Modifies
}

@Contract(mutates = "param1")
static void addToList(List list, String item) { list.add(item) }
```

Interaction with other contract annotations

The `ModifiesChecker` works alongside the other design-by-contract annotations to provide a complete specification for modular reasoning:

- `@Requires` declares what must be true before a call
- `@Ensures` declares what is guaranteed after a call
- `@Modifies` declares what may change (and implicitly, what does not)
- `@Pure` declares a method has no side effects

Together, these annotations allow analysis of method call sequences without reading method bodies — each method becomes a self-contained specification.

Checking @Pure Purity (Incubating)

The `PurityChecker` verifies that `@Pure` methods have no side effects. By default, strict purity is enforced. The `allows` option declares which effect categories are tolerated.

Languages like Haskell enforce purity through the type system — all side effects must be expressed via IO and State monads, making impure code structurally impossible. Frege (a Haskell dialect for the JVM) takes the same approach. This is powerful but requires incorporating these patterns into your application's type hierarchies, which can be impractical when integrating with JDK libraries that don't follow these conventions.

Groovy's `PurityChecker` provides an opt-in way to achieve similar levels of reasoning about method purity while staying pragmatic about the JDK ecosystem. The `allows` option acknowledges that real-world "pure" methods sometimes log or emit metrics — effects that don't affect program correctness but would violate strict Haskell-style purity.

Basic usage

```
// Strict: no side effects at all
@TypeChecked(extensions = 'groovy.typecheckers.PurityChecker')
class Calculator {
    @Pure
    int add(int a, int b) { a + b }              // OK: no effects

    @Pure
    int broken() { count++; return count }         // ERROR: field mutation
}
```

Effect categories

The `allows` option accepts a `|`-separated list of effect categories:

```
@TypeChecked(extensions = 'groovy.typecheckers.PurityChecker(allows:
"LOGGING|METRICS")')
```

Category	What it allows	Examples
`LOGGING`	Calls to logging frameworks and `println`/`print`	`log.debug(…)`, `println(…)`, `System.out.println(…)`
`METRICS`	Calls to metrics instruments	Micrometer counters/timers, OpenTelemetry metrics
`IO`	File, network, database, and console I/O	`new File(…)`, `Files.readAllLines(…)`, JDBC calls
`NONDETERMINISM`	Time-dependent, random, and environment-dependent calls	`System.nanoTime()`, `Math.random()`, `new Date()`, `UUID.randomUUID()`

Supported annotations

The checker recognises purity annotations from multiple libraries by simple name:

Annotation	Library	Semantics
`@Pure`	Groovy, Checker Framework, Xtext	Strict purity (side-effect-free + deterministic)
`@SideEffectFree`	Checker Framework	No mutations, but non-determinism is implicitly allowed
`@Contract(pure = true)`	JetBrains Annotations	Strict purity (works with CLASS retention)
`@Memoized`	Groovy	Effectively pure (checked for side effects)

Annotations are matched by simple name, so `@Pure` from any package is recognised. The `@Contract` annotation is checked for its `pure` attribute being `true`.

How calls are classified

The checker uses a layered approach to determine if a call is pure:

1. **Callee has `@Pure`, `@SideEffectFree`, or `@Contract(pure = true)`** — the call is pure
2. **Callee has `@Memoized`** — the call is effectively pure
3. **Receiver type is immutable** (String, Integer, BigDecimal, etc.) — all calls are pure
4. **Method is in the known-pure list** (`size`, `get`, `contains`, `isEmpty`, `toString`, etc.) — pure
5. **Call matches an effect category** — error unless that category is in `allows`
6. **None of the above** — warning: purity cannot be verified

Reasoning guarantees

The `allows` declaration is machine-readable, enabling both humans and AI to derive precise guarantees about `@Pure` methods:

Property	strict	+LOGGING	+METRICS	+IO	+NONDETERMINISM
No field mutations	Yes	Yes	Yes	Yes	Yes
Can carry forward state across call	Yes	Yes	Yes	Yes	Yes
Deterministic (same inputs → same output)	Yes	Yes	Yes	Yes	**No**
Can cache/memoize result	Yes	Yes	Yes	Yes	**No**
Can reorder call freely	Yes	Yes	Yes	**No**	Yes
Can eliminate dead call	Yes	**No**	**No**	**No**	Yes
No observable output	Yes	**No**	**No**	**No**	Yes

Each column represents a different `allows` configuration. For example, a `@Pure` method under `PurityChecker(allows: "LOGGING")` is guaranteed to have no field mutations, be deterministic, and be safely reorderable — but may produce log output, so dead call elimination would lose that output.

Examples

Strict purity (default):

```
@TypeChecked(extensions = 'groovy.typecheckers.PurityChecker')
class MathUtils {
    @Pure
    int square(int x) { x * x }                              // OK

    @Pure
    long timestamp() { System.nanoTime() }                    // ERROR: non-
deterministic

    @Pure
    int logged(int x) { println("x=$x"); x * 2 }             // ERROR: logging
}
```

Allowing logging:

```
@TypeChecked(extensions = 'groovy.typecheckers.PurityChecker(allows: "LOGGING")')
class Service {
    @Pure
    int compute(int x) {
        println("computing $x")          // OK: logging allowed
        return x * x
    }

    @Pure
    long getTime() { System.nanoTime() } // ERROR: non-deterministic (not allowed)
}
```

Allowing logging and non-determinism:

```
@TypeChecked(extensions = 'groovy.typecheckers.PurityChecker(allows:
"LOGGING|NONDETERMINISM")')
class Diagnostics {
    @Pure
    String snapshot(Map state) {
        println("snapshot at ${System.nanoTime()}")   // OK: both allowed
        return state.toString()
    }
}
```

What the checker detects

The checker flags:

- Direct field assignments on `this` (`this.x = ⋯`, `x++` where `x` is a field)
- Property assignments on parameters and local variables (`param.x = 1`)
- Calls to methods not known to be pure
- Calls matching effect categories (logging, metrics, I/O, non-determinism) unless allowed
- Construction of non-deterministic or I/O types

The checker does **not** currently detect:

- Mutation through aliases (`def ref = this.list; ref.clear()`)
- Deep heap mutation (`this.items.get(0).setName("x")`)
- Mutation via chained property access (`a.b.c = 1`)
- Side effects within closures passed to higher-order methods

These limitations are acceptable for an opt-in checker — the common cases are caught, and unverifiable calls produce warnings.

Relationship to @Modifies and ModifiesChecker

`@Pure` implies `@Modifies({})` — a method with no field mutations. The `ModifiesChecker` recognises `@Pure` and verifies the no-mutation guarantee. The `PurityChecker` goes further, verifying additional purity properties (no impure calls, no I/O, etc.).

Both checkers can be used independently or together:

```
// Just frame conditions
@TypeChecked(extensions = 'groovy.typecheckers.ModifiesChecker')

// Just purity
@TypeChecked(extensions = 'groovy.typecheckers.PurityChecker')

// Both -- ModifiesChecker handles @Modifies, PurityChecker handles @Pure
@TypeChecked(extensions = ['groovy.typecheckers.ModifiesChecker',
                           'groovy.typecheckers.PurityChecker'])
```

Checking @Associative/@Reducer Combiners (Incubating)

The parallel `Collection` reductions split, reorder, and recombine their input. That is only correct when the combining function is **associative** — `combine(a, combine(b, c))` must equal `combine(combine(a, b), c)` — and, for the seeded `injectParallel`, when the seed is an **identity** for the combiner. A non-associative combiner (subtraction, division, exponentiation) compiles and runs, but produces a different, non-deterministic result depending on how the work was partitioned. This is among the hardest bug classes to detect by testing alone. (Note: associativity is

distinct from commutativity — concatenation, for example, is associative but not commutative, and is a perfectly valid parallel combiner over an ordered source; this checker is about associativity and identity, not encounter order.)

`@Associative` and `@Reducer` (in `groovy.transform`) let a combiner **declare** the contract; `CombinerChecker` verifies, at compile time, that combiners reaching `sumParallel`/`injectParallel` carry it.

```
import groovy.transform.Associative
import groovy.transform.Reducer

class Maths {
    @Associative              static int add(int a, int b) { a + b }
    @Reducer(zero = '0')      static int plus(int a, int b) { a + b }
}
```

Modes

Associativity is undecidable in general, so the checker is deliberately conservative and false-positive averse (a type-checking extension has only an error channel).

`default` **(lenient)**

flags only high-confidence problems — a non-associative operator (`-`, `/`, `%`, `**`) applied directly to the two combiner parameters, and a `injectParallel` seed that contradicts a `@Reducer`'s declared `zero()`. An ordinary associative inline closure such as `{ a, b → a + b }` is never flagged.

`strict`

additionally requires the combiner to be a method reference to an `@Associative`/`@Reducer` method; inline closures and un-annotated references are rejected.

```
// lenient (default)
@TypeChecked(extensions = 'groovy.typecheckers.CombinerChecker')

// strict -- only declared, verifiable combiners allowed
@TypeChecked(extensions = "groovy.typecheckers.CombinerChecker(mode: 'strict')")
```

Examples

```
[1,2,3].injectParallel(0, Maths.&add)       // ✔ declared @Associative
[1,2,3].injectParallel(0) { a,b -> a + b }  // ✔ lenient: associative
[1,2,3].injectParallel(0) { a,b -> a - b }  // ✖ non-associative
[1,2,3].injectParallel(5, Maths.&plus)      // ✖ seed != @Reducer zero '0'
```

Recognising Monoid/Semigroup carriers

Groovy deliberately ships no `Monoid`/`Semigroup` of its own, but a combiner sourced from such a type

in an external library (Functional Java, Palatable Lambda, Purefun, ...) already **carries** the associativity contract — a `Semigroup` is an associative operation; a `Monoid` is that plus an identity. The checker therefore accepts a method-reference (or thin delegating closure) whose owner is such a carrier, with the same trust level as `@Associative` (an asserted contract, not a proof). Carriers are matched by simple type name (`Monoid`/`Semigroup` by default, extensible via the `monoids`/`semigroups` options) so there is no dependency on any library.

```
def m = new IntSum()   // implements Monoid
[1,2,3].injectParallel(m.zero(), m.&append)  // ✔ Monoid carries identity
[1,2,3].sumParallel(s.&append) // s: Semigroup // ✔ associative; no seed needed
[1,2,3].injectParallel(0, s.&append)         // ✖ Semigroup has no identity
```

This last case is a high-confidence positive the type distinction makes possible: a seeded `injectParallel` needs an identity, which a `Semigroup` categorically lacks — only a `Monoid` provides one.

JDK stream reductions

Beyond Groovy's parallel `Collection` methods, the checker also covers `java.util.stream.{Stream,IntStream,LongStream,DoubleStream}#reduce` (the same way `RegexChecker` covers `java.util.regex` types). All combiner shapes, the identity check, and Monoid/Semigroup carrier recognition apply unchanged; the three overloads map onto the existing model — `reduce(op)` is unseeded, `reduce(id, op)` and `reduce(id, acc, combiner)` are seeded (the last argument, i.e. the merge operator, is the associativity-critical one).

Because these resolve to a real interface method, recognition is gated on the receiver type (`java.util.stream.`) rather than name alone, keeping the generic name reduce from causing false positives. Note that Stream.reduce is contractually required to be associative *even for sequential streams*, so flagging is spec-correct independently of `.parallel()` — unlike Groovy's sequential `inject`, which is a deterministic left fold and is deliberately never flagged.

```
[1,2,3].stream().reduce(0) { a,b -> a + b }  // ✔ associative
[1,2,3].stream().reduce(0) { a,b -> a - b }  // ✖ non-associative
[1,2,3].stream().reduce(0, s.&append) // Semigroup // ✖ no identity
```

Limitations

This is a safety net, not a guarantee:

- It runs only under `@TypeChecked`/`@CompileStatic`; dynamic code is unchecked.
- Associativity cannot be proven, only asserted (via the annotations) or ruled out for obvious cases. The annotations are intended to be backed by tests (for example auto-derived property-based tests).
- An inline lambda has no declaration site for the annotation; the declarable contract is available only for named, referenced combiner methods.
- The inline-closure check is purely syntactic: it matches the operator token ({{-}}, {{/}}, {{%}},

{{*}}) and assumes operators carry their conventional meaning. Groovy lets operators be overloaded (and Groovy 6 adds {{=}} via {{plusEqual}}), which breaks that assumption both ways: a denylisted operator could be overloaded to something associative (a possible false positive), and -- the converse -- a normally-associative operator such as {{}} or {{}} could be overloaded to be non-associative. The latter is poor style and is an explicit **non-goal**: the checker does not attempt to detect it. The check deliberately stays minimal and errs toward silence; the declaration paths (`@Associative`/`@Reducer`, Monoid/Semigroup carriers) are the reliable, overloading-immune way to assert the contract.

Relationship to @Pure and @Memoized

A parallel combiner should normally also be side-effect-free; pairing this checker with `PurityChecker` (which already treats `@Memoized` methods as purity-obligated) gives both guarantees.

Checking the DO Macro (Incubating)

New in Groovy 6.0.0.

The `MonadicChecker` type-checking extension supports the `DO` macro for monadic comprehensions, teaching `@CompileStatic` / `@TypeChecked` how to look through the macro's desugared output (a runtime-dispatched `Comprehensions.bind` / `Comprehensions.map` chain). Without it, generator bound names erase to `Object` and the body fails to type-check.

```
@CompileStatic(extensions = 'groovy.typecheckers.MonadicChecker')
def addOne = DO { x <- Optional.of(1); yield x + 1 }
```

The checker also rejects a non-participating carrier with a compile error naming the missing shape (the carrier must be in the registry allow-list, expose structural `flatMap` / `map`, or be annotated with `@Monadic`).

The macro itself, its desugaring strategy, the participating-carrier rules, and the `@Monadic` annotation are documented in the Monadic Comprehensions chapter. This section covers only the type-checking surface.

Checking Hand-written Monadic Chains (Incubating)

New in Groovy 6.0.0.

The `MonadicShapeChecker` is a lint for monadic chains you write by hand (without the `DO` macro) over the standard carriers (`Optional`, `Stream`, `Awaitable`). It catches the two classic shape mistakes:

- `bind` (a.k.a. `flatMap`) closure returning a non-carrier value
- `map` closure returning the same carrier, where `flatMap` was meant

Activate via:

```
@TypeChecked(extensions = 'groovy.typecheckers.MonadicShapeChecker')
```

The checker has a `strict` option that tightens what it flags; in lenient (default) mode only high-confidence violations are reported. The sister `MonadicChecker` (above) covers the `DO`-macro path; this checker covers the hand-written path.

Groovy Contracts – design by contract support for Groovy

This module provides contract annotations that support the specification of class-invariants, pre- and post-conditions on Groovy classes and interfaces, loop invariants on `for`, `while`, and `do-while` loops, and frame conditions declaring which fields a method may modify. Special support is provided so that post-conditions may refer to the old value of variables or to the result value associated with calling a method.

Applying @Invariant, @Requires and @Ensures

With Groovy contracts in your classpath, contracts can be applied on a Groovy class or interface by using one of the annotations found in the `groovy.contracts` package.

```
package acme

import groovy.contracts.*

@Invariant({ speed() >= 0 })
class Rocket {
    int speed = 0
    boolean started = true

    @Requires({ isStarted() })
    @Ensures({ old.speed < speed })
    def accelerate(inc) { speed += inc }

    def speed() { speed }
}

def r = new Rocket()
r.accelerate(5)
```

Enabling contracts with @Contracted

Contract processing is opt-in. By default, classes that simply mention the contract annotations will have them processed; the `@Contracted` marker is used when you want to make the opt-in explicit at the package level, or when you want to apply contract processing to a type that doesn't itself carry any contract annotations (for example, an empty class whose contract comes entirely from inherited interfaces).

`@Contracted` may be placed on a package declaration (in a `package-info.groovy` / `package-info.java`) to enable contract processing across every type in that package:

```
@Contracted
package com.acme.banking

import groovy.contracts.*
```

or on an individual type:

```
@Contracted
class Account { /* ... */ }
```

Once enabled, runtime assertion checking still honours the JVM's `-ea` / `-da` flags; `@Contracted` controls whether the AST transforms generate assertion code at compile time, not whether the resulting assertions execute at run time.

More Features

Groovy contracts supports the following feature set:

- definition of class invariants, pre- and post-conditions via @Invariant, @Requires and @Ensures
- definition of loop invariants on `for`, `while`, and `do-while` loops via @Invariant
- declaration of frame conditions (which fields a method may modify) via @Modifies
- declaration of loop and recursion termination measures via @Decreases
- inheritance of class invariants, pre- and post-conditions of concrete predecessor classes
- inheritance of class invariants, pre- and post-conditions in implemented interfaces
- usage of old and result variable in post-condition assertions
- assertion injection in Plain Old Groovy Objects (POGOs)
- human-readable assertion messages, based on Groovy power asserts
- enabling contracts at package- or class-level with @Contracted
- enable or disable contract checking with Java's -ea and -da VM parameters
- annotation contracts: a way to reuse reappearing contract elements in a project domain model
- detection of circular assertion method calls

The Stack Example

Currently, Groovy contracts supports 3 annotations: @Invariant, @Requires and @Ensures – all of them work as annotations with closures, where closures allow you to specify arbitrary code pieces as annotation parameters:

```
import groovy.contracts.*

@Invariant({ elements != null })
class Stack<T> {
```

```
    List<T> elements

    @Ensures({ is_empty() })
    def Stack()  {
        elements = []
    }

    @Requires({ preElements?.size() > 0 })
    @Ensures({ !is_empty() })
    def Stack(List<T> preElements)  {
        elements = preElements
    }

    boolean is_empty()  {
        elements.isEmpty()
    }

    @Requires({ !is_empty() })
    T last_item()  {
        elements.get(count() - 1)
    }

    def count() {
        elements.size()
    }

    @Ensures({ result == true ? count() > 0 : count() >= 0  })
    boolean has(T item)  {
        elements.contains(item)
    }

    @Ensures({ last_item() == item })
    def push(T item)  {
       elements.add(item)
    }

    @Requires({ !is_empty() })
    @Ensures({ last_item() == item })
    def replace(T item)  {
        remove()
        elements.add(item)
    }

    @Requires({ !is_empty() })
    @Ensures({ result != null })
    T remove()  {
        elements.remove(count() - 1)
    }

    String toString() { elements.toString() }
```

```
}

def stack = new Stack<Integer>()
```

The example above specifies a class-invariant and methods with pre- and post-conditions. Note, that preconditions may reference method arguments and post-conditions have access to the method's result with the result variable and old instance variables values with old.

Indeed, Groovy AST transformations change these assertion annotations into Java assertion statements (can be turned on and off with a JVM param) and inject them at appropriate places, e.g. class-invariants are used to check an object's state before and after each method call.

Annotation closure rules

- Invariants should reference class fields only.
- Preconditions may reference class fields and arguments.
- Postconditions may reference class fields, arguments, the `result` and the old value of class fields using the syntax `old.fieldname`.

Old values of class fields will only be stored where Groovy contracts knows how to make a copy. Currently, this means fields with a `Cloneable` class type, primitives, the primitive wrapper types, `BigInteger`, `BigDecimal`, and G/Strings.

Use within scripts

Currently, Groovy contracts intentionally doesn't support scripts, but you can use them with JEP-445 compatible scripts from Groovy 5 as shown in this example:

```
import groovy.contracts.*
import org.apache.groovy.contracts.*
import static groovy.test.GroovyAssert.shouldFail

@Requires({ arg > 0 })
@Ensures({ result < arg })
def sqrt(arg) { Math.sqrt(arg) }

def main() {
    assert sqrt(4) == 2
    shouldFail(PreconditionViolation) { sqrt(-1) }
}
```

You could even place an `@Invariant` annotation on the `main` method and it will be moved to the generated script class.

Loop Invariants

In addition to class-level invariants, `@Invariant` can be placed directly on `for`, `while`, and `do-while`

loops. A loop invariant is a condition that must hold at the start of each iteration. If the condition is violated, a `LoopInvariantViolation` (a subclass of `AssertionError`) is thrown.

This is inspired by design-by-contract constructs found in languages like Dafny and Eiffel.

For loop

```
import groovy.contracts.Invariant

int sum = 0
@Invariant({ 0 <= i && i <= 4 })
for (int i in 0..4) {
    sum += i
}
assert sum == 10
```

While loop

```
import groovy.contracts.Invariant

int n = 10
@Invariant({ n >= 0 })
while (n > 0) {
    n--
}
assert n == 0
```

Multiple invariants

Multiple `@Invariant` annotations can be stacked on a single loop, and each condition is checked independently at the start of every iteration:

```
import groovy.contracts.Invariant

int sum = 0
@Invariant({ sum >= 0 })
@Invariant({ sum <= 100 })
for (int i in 1..5) {
    sum += i
}
assert sum == 15
```

Rules for loop invariant closures

- The closure should be a boolean expression (or multiple expressions).
- The closure may reference any variables visible in the enclosing scope, including the loop variable.

- Assignment operators and state-changing postfix/prefix operators are not supported inside the closure.

Termination with @Decreases

`@Decreases` declares a *termination measure* (a *variant*): a value that must strictly decrease through a well-founded sequence, guaranteeing progress towards termination. It applies in two places:

- on a **loop** (`for`/`while`/`do-while`) — the measure must decrease on every iteration; and
- on a **method** — the measure must decrease on every *recursive* re-entry.

In both cases the closure must yield a value whose successive values form a strictly decreasing, well-founded sequence. Two value shapes are supported:

- A single `Comparable` value — usually numeric — that strictly decreases on every iteration. If the value is also a `Number`, it must additionally remain non-negative.
- A `List` of `Comparable` elements, compared lexicographically: the first position at which consecutive values differ must show a strict decrease, and earlier positions must be equal. This is the natural way to express termination for nested loops (e.g. `[i, j]`).

If the variant fails to decrease (or a numeric scalar variant becomes negative), a `LoopVariantViolation` is thrown for a loop and a `RecursionVariantViolation` for a method (both subclasses of `AssertionError`).

This is inspired by similar constructs in design-by-contract languages such as Dafny and Eiffel.



`@Decreases` is incubating in Groovy 6.0.0. The contract — in particular the scalar non-negativity rule and the lexicographic shape for `List` variants — may be relaxed or extended in a future release based on usage feedback.

While loop

```
import groovy.contracts.Decreases

int n = 10
@Decreases({ n })
while (n > 0) {
    n--
}
assert n == 0
```

For loop

```
import groovy.contracts.Decreases

int remaining = 5
@Decreases({ remaining })
```

```
for (int i = 0; i < 5; i++) {
    remaining--
}
assert remaining == 0
```

Lexicographic (tuple) variants

For nested loops where no single counter strictly decreases on every iteration, return a `List` whose elements form a tuple compared lexicographically. In the example below the outer counter stays equal while the inner one decreases, then the outer one decreases and the inner one is allowed to reset:

```
import groovy.contracts.Decreases

int outer = 2, inner = 3
@Decreases({ [outer, inner] })
while (outer > 0) {
    if (inner > 0) {
        inner--
    } else {
        outer--
        inner = 3
    }
}
assert outer == 0
```

Recursive methods

On a method, the measure is a function of the method's **parameters** and must strictly decrease on every *recursive* re-entry — the recursion analogue of a loop variant. At runtime the value is captured on entry and compared against the nearest enclosing invocation of the same method (per thread); a non-decreasing or negative measure throws a `RecursionVariantViolation`, turning a non-terminating recursion into an immediate, localised error rather than a `StackOverflowError`:

```
import groovy.contracts.Decreases

@Decreases({ n })                       // strictly decreases on every recursive call
int factorial(int n) {
    n <= 1 ? 1 : n * factorial(n - 1)
}
assert factorial(5) == 120
```

The bookkeeping is per method and per thread, so it also handles *mutual* recursion (each participant must decrease its own measure on its own re-entry) and recursion through callbacks. Like `@Requires`/`@Ensures`, a method `@Decreases` is **inherited**: an override that does not redeclare it inherits its parent's measure (with the parent's parameter names re-mapped to the override's); an override may declare its own `@Decreases` to refine it.

Rules for @Decreases closures

- The closure must return a `Comparable` value or a `List` of `Comparable` elements.
- For a loop, the value is evaluated at the start and end of each iteration; for a method, on each entry. The new value must be strictly less than the previous one (lexicographically, for `List` variants).
- For numeric scalar variants, the value must also remain `>= 0`. This non-negativity rule does *not* apply to non-`Number` `Comparable` values (e.g. `String`) or to `List` variants.
- A loop measure may reference any variable visible in the enclosing scope; a method measure is over the method's parameters (it is evaluated on entry, so it cannot reference the method's `result`).
- Checks honour the JVM's `-ea`/`-da` assertion configuration for the enclosing class, as preconditions and postconditions do.
- Recursion measure inheritance currently requires the parent to be compiled in the same unit; an override of a method from a precompiled (binary) supertype is not yet instrumented.

Unwoven preconditions

By default `@Requires` *weaves* its assertion: a violating caller observes a `PreconditionViolation`. Real methods often already enforce their obligations — a hand-written guard, an `Objects.requireNonNull`, Guava's `Preconditions` — and adding a woven `@Requires` on top would double the check and change the exception a violating caller observes. Two members, shared with `@ThrowsIf`, cover this:

- `woven = false` — no assertion is generated. The annotation remains the caller's *documented obligation*, consumable by readers and tools (a verifier discharges it at call sites regardless of where enforcement lives); a violating caller observes whatever the existing enforcement does. An unwoven precondition never contributes to generated assertions, including the inherited precondition weaving of overriding methods.
- `direct` — pure information, no effect on bytecode, ignored (implicitly `true`) for woven preconditions: `true` (default) says a hand-written check enforces the obligation in this body; `false` says the enforcement lives in code the method *executes* (a validator call, possibly transitive), so there is no check to find in this body. Most users never set it.

```
import groovy.contracts.Requires
import static groovy.test.GroovyAssert.shouldFail

class Greeter {
    // the obligation is already enforced by requireNonNull: the annotation documents
it,
    // no assertion is generated, and a violating caller sees the library's NPE
    @Requires(value = { name != null }, woven = false, direct = false)
    static String greet(String name) {
        Objects.requireNonNull(name)
        "Hello, $name!"
    }
```

```
}

assert Greeter.greet('World') == 'Hello, World!'
shouldFail(NullPointerException) { Greeter.greet(null) }   // the enforcement,
untouched
```

The members are per-annotation, so repeated preconditions mix modes — woven arms are asserted, unwoven arms are left to their existing enforcement:

```
import groovy.contracts.Requires
import org.apache.groovy.contracts.PreconditionViolation
import static groovy.test.GroovyAssert.shouldFail

class Adjuster {
    @Requires({ amount <= 1000 })                              // woven: asserted
    @Requires(value = { amount >= 0 }, woven = false)          // enforced in the body
    static int adjust(int amount) {
        if (amount < 0) throw new IllegalArgumentException('negative')
        amount
    }
}

assert Adjuster.adjust(5) == 5
shouldFail(PreconditionViolation) { Adjuster.adjust(2000) }        // the woven arm
shouldFail(IllegalArgumentException) { Adjuster.adjust(-1) }       // the body's own
guard
```

Choosing between an unwoven `@Requires` and a `@ThrowsIf` over the same guard is a statement of intent: `@Requires` says a violating call is the *caller's bug* (callers may not rely on the violation behaviour); `@ThrowsIf` says the throw is *defined behaviour* callers may rely on and catch. Both are spellable over the identical `requireNonNull` body — they are different claims.



The `woven` and `direct` members of `@Requires` are incubating in Groovy 6.0.0 and may be refined in a future release based on usage feedback.

Exceptional contracts with @ThrowsIf

`@ThrowsIf` declares a method's *exceptional contract*: the method throws the given exception when the given condition holds on entry. It is the exceptional counterpart of `@Requires` — a precondition says *the caller must not do this* (violating it is the caller's bug), whereas a `@ThrowsIf` arm says *this input is handled, by throwing*: it is **defined behaviour**, part of the contract, that callers may rely on (and catch). Guard clauses of this shape are among the most common contract-shaped code written by hand (`if (b == 0) throw …`, `Objects.requireNonNull(y)`, Guava's `Preconditions`); `@ThrowsIf` makes that behaviour a first-class, machine-readable annotation. The condition is a closure over the method's parameters, using the same conventions as `@Requires`, and the annotation is repeatable — each *arm* pairs one condition with one exception type.

The contract is read as an *iff*: the method throws a matching exception *exactly when* some arm's

condition holds. Two attributes generate code; the others are runtime-retained metadata for readers and tools (documentation generators, static analysers, verifiers, AI coding agents):

Member	Effect on generated code
`woven`	`true` (default): the guard — `if (cond) throw new E(…)` — is inserted at method entry (the general form of the pattern `groovy.transform.NullCheck` provides for the null-check special case). `false`: nothing is generated — the throw already exists.
`checked`	`true`: a verification wrapper is generated (see below). `false` (default): nothing.
`exception`	The type the woven guard constructs (it should provide a `(String)` constructor, as the standard exceptions do), and the type the checked wrapper matches escaping throws against.
`exhaustive`	Nothing on its own. Under `checked = true` it gates the escaping-throw check: `false` says the method may throw the same exception for other, unlisted reasons, so escaping throws are never judged.
`direct`	Nothing, ever — pure information: `true` (default) says a hand-written `throw` statement lives in this body; `false` says the exception arises from code the method *executes* (a call, possibly transitive, or a runtime operation), so there is no throw statement in this body to find. Ignored (implicitly `true`) for woven arms. This is documentation for a human reader but exists primarily for tools - if you are not using a tool that needs it, you can safely ignore it.

Woven guards

With the defaults, the annotation *is* the guard:

```
import groovy.contracts.ThrowsIf
import static groovy.test.GroovyAssert.shouldFail

class Calculator {
    @ThrowsIf(value = { b == 0 }, exception = ArithmeticException)
    static int divide(int a, int b) { a.intdiv(b) }
}

assert Calculator.divide(4, 2) == 2
shouldFail(ArithmeticException) { Calculator.divide(1, 0) }   // the woven guard
```

On a constructor whose first statement is an explicit `super(…)` or `this(…)` call, the guard is inserted immediately *after* that call, which must stay first — a (current) language constraint.

Documenting an existing throw

With `woven = false` nothing is generated; the annotation is checkable documentation of a guard that is already there:

```
import groovy.contracts.ThrowsIf
```

```
import static groovy.test.GroovyAssert.shouldFail

class MathUtil {
    @ThrowsIf(value = { n < 0 }, exception = IllegalArgumentException, woven = false)
    static int fact(int n) {
        if (n < 0) throw new IllegalArgumentException('negative')   // the guard is
already here
        n <= 1 ? 1 : n * fact(n - 1)
    }
}

assert MathUtil.fact(4) == 24
shouldFail(IllegalArgumentException) { MathUtil.fact(-1) }
```

When the throw originates in code the method merely *calls* (a third-party library), add `direct = false` — there is no throw statement in this body to find, and weaving a wrong claim about invoked code would silently change behaviour, which is why such arms are never woven:

```
import groovy.contracts.ThrowsIf

class Wrapper {
    @ThrowsIf(value = { s == null }, exception = NullPointerException, woven = false,
direct = false)
    static Object describe(Object s) {
        Objects.requireNonNull(s)   // the library throws; the annotation records the
contract
        "value: $s"
    }
}

assert Wrapper.describe('x') == 'value: x'
```

Most users never set `direct`: the defaults are right for woven arms and hand-written guards alike, and the usual prompt to add `direct = false` is a verification or analysis tool reporting that it cannot find the promised throw in the body.

Runtime checking

Weaving *implements* the contract; `checked = true` additionally *verifies* it at runtime, in the same assertion style as `@Ensures`: on a normal return, no non-woven arm's condition may have held on entry (*must-throw*), and an escaping exception matching some arm's type must be justified by a matching arm's condition having held (*only-when* — checked only for `exhaustive` arm-sets). A broken implementation raises `ThrowsIfViolation` — never the declared exception, which is defined behaviour delivered at entry; a justified throw always propagates untouched:

```
import groovy.contracts.ThrowsIf
import org.apache.groovy.contracts.ThrowsIfViolation
import static groovy.test.GroovyAssert.shouldFail
```

```
class Broken {
    // the body SHOULD throw when n < 0 but does not — a broken implementation,
    // reported as a violation (never the declared exception)
    @ThrowsIf(value = { n < 0 }, exception = IllegalArgumentException, woven = false,
checked = true)
    static int identity(int n) { n }
}

assert Broken.identity(5) == 5
shouldFail(ThrowsIfViolation) { Broken.identity(-5) }
```

`checked` is *set-level*: if any arm is `checked`, the whole arm-set is checked — every non-woven arm is must-throw checked, and every arm serves as an only-when justifier.

Rules for @ThrowsIf closures

- The closure is a boolean expression over the method's parameters (the same conventions as `@Requires`; it cannot reference `result` or `old`).
- Multiple arms are independent must-throws; the only-when direction is a property of the whole arm-set. One `exhaustive = false` arm disclaims it for the set.
- Exhaustiveness is (at most) over the *conditions for the exception types mentioned*, never over exception *types*: declaring an `ArithmeticException` arm says nothing about whether the method can throw anything else.
- No claim, however `exhaustive`, reasons about VM resource conditions: an `OutOfMemoryError` or `StackOverflowError` is outside contract semantics, and the checked wrapper passes any `VirtualMachineError` through unjudged.
- When both apply on the same input, a `@Requires` violation is reported first (`PreconditionViolation`): the caller's obligation is judged before the method's defined behaviour.



`@ThrowsIf` is incubating in Groovy 6.0.0. The attribute set — in particular the `direct` metadata member and the set-level `checked` rule — may be refined in a future release based on usage feedback.

Frame Conditions with @Modifies

The `@Modifies` annotation declares which fields or parameters a method is allowed to modify. All other state is implicitly declared unchanged. This is known as a *frame condition* in design-by-contract terminology, inspired by similar constructs in Dafny and JML.

`@Modifies` does not generate runtime assertion code — it serves as a specification for humans and tools (including AI) to reason about what a method changes.

Basic usage

Use `this.fieldName` to declare that a field may be modified:

```
@Modifies({ this.items })
void addItem(String item) {
    items.add(item)
}
```

For multiple fields, use a list expression:

```
@Modifies({ [this.items, this.count] })
void addAndCount(String item) {
    items.add(item)
    count++
}
```

Or use multiple `@Modifies` annotations (via `@Repeatable`):

```
@Modifies({ this.items })
@Modifies({ this.count })
void addAndCount(String item) { ... }
```

Parameter references

Method parameters can also be declared as modifiable:

```
@Modifies({ arr })
void fillArray(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        arr[i] = i
    }
}
```

Interaction with @Ensures

When both `@Modifies` and `@Ensures` are present on a method, a compile-time check ensures that the `old` variable in the postcondition only references fields declared in `@Modifies`. Referencing a field via `old` that is not in the modification set produces a compile error:

```
@Modifies({ this.items })
@Ensures({ old -> old.items.size() < items.size() })  // OK: items is declared
void addItem(String item) { ... }

@Modifies({ this.items })
@Ensures({ old -> old.count == count })  // ERROR: count not declared in @Modifies
void addItem(String item) { ... }
```

Compile-time validation

The `@Modifies` annotation validates at compile time that:

- Field references (`this.fieldName`) refer to fields that exist on the class.
- Parameter references refer to actual parameters of the method.
- If `@Ensures` is also present, `old.fieldName` references only access fields declared in `@Modifies`.

Rules for @Modifies closures

- The closure must contain a single field reference (`this.field`), parameter reference (`param`), or a list of such references.
- Field references use the `this.fieldName` syntax.
- Parameter references use the bare parameter name.
- The closure is not executed at runtime — it is only analyzed at compile time.

Verifying method bodies with ModifiesChecker

While `@Modifies` alone is a compile-time-checked specification (validating field/parameter existence and `@Ensures` consistency), it does not verify the method body itself. For that, the `groovy-typecheckers` module provides the `ModifiesChecker` type checking extension:

```
@TypeChecked(extensions = 'groovy.typecheckers.ModifiesChecker')
```

When enabled, the checker verifies that method bodies only modify fields declared in `@Modifies`, and that method calls on non-modifiable receivers use only non-mutating operations. See the *Type Checkers* documentation for details.

HttpBuilder – lightweight HTTP client DSL (incubating)

The `groovy-http-builder` module provides a tiny declarative DSL over the JDK `java.net.http.HttpClient`. It is designed for scripting and simple automation tasks where a full-blown HTTP library would be overkill.

Goals

- Keep the implementation small and easy to maintain.
- Use only JDK HTTP client primitives (Jsoup is optionally supported for HTML parsing).
- Make common request setup declarative with Groovy closures.
- Handle only the simple cases that often pop up in scripting — not the full use cases that Apache Geb covers.
- Include JSON/XML/HTML response parsing hooks while intentionally keeping request hooks minimal.

Basic Usage

Create a client with `HttpBuilder.http`, configure shared settings in the closure, and issue requests:

```
import groovy.http.HttpBuilder

def http = HttpBuilder.http {
    baseUri '${rootUri}/'
    header 'User-Agent', 'my-app/1.0'
}

def res = http.get('/api/items') {
    query page: 1, size: 10
}

assert res.status == 200
```

`query(…)` encodes keys and values as URI query components using RFC 3986 style percent-encoding — for example, spaces become `%20`.

Non-DSL Equivalent (JDK HttpClient)

The snippet above is equivalent to the following plain JDK code:

```
import java.net.URI
import java.net.URLEncoder
import java.net.http.HttpClient
import java.net.http.HttpRequest
import java.net.http.HttpResponse
import java.nio.charset.StandardCharsets

def encodeQueryComponent = { Object value ->
    URLEncoder.encode(value.toString(), StandardCharsets.UTF_8)
            .replace('+', '%20')
            .replace('*', '%2A')
            .replace('%7E', '~')
}

def baseUri = 'https://example.com/'
def query = [page: 1, size: 10]
        .collect { k, v ->
            "${encodeQueryComponent(k)}=${encodeQueryComponent(v)}"
        }
        .join('&')

def target = URI.create(baseUri).resolve("/api/items?${query}")

def client = HttpClient.newHttpClient()
def request = HttpRequest.newBuilder(target)
```

```
        .header('User-Agent', 'my-app/1.0')
        .GET()
        .build()

def response = client.send(request, HttpResponse.BodyHandlers.ofString())

assert response.statusCode() == 200
println response.body()
```

JSON

GET

```
import static groovy.http.HttpBuilder.http

def client = http '${rootUri}'
def res = client.get('/api/items')

assert res.status == 200
assert res.json.items[0].name == 'book'
assert res.parsed.items[0].name == 'book' // auto-parsed from Content-Type
```

`res.json` lazily parses the response body as JSON. `res.parsed` auto-dispatches by the response `Content-Type` header, so for `application/json` it behaves identically to `res.json`.

POST

```
def result = http.post('/api/items') {
    json([name: 'book', qty: 2])
}

assert result.status == 200
assert result.json.ok
```

The `json(…)` helper serializes the supplied object as JSON and sets `Content-Type: application/json` automatically.

XML

```
def result = http.get('/api/repo.xml')

assert result.status == 200
assert result.xml.license.text() == 'Apache License 2.0'
assert result.parsed.license.text() == 'Apache License 2.0' // auto-parsed from
Content-Type
```

`result.xml` parses the response body with `XmlSlurper`. `result.parsed` dispatches to `xml` for XML content types.

HTML (jsoup)

If jsoup is on the classpath, `result.html` returns a jsoup `Document`:

```
// @Grab('org.jsoup:jsoup:1.23.2') // needed if running as standalone script
def client = http('${mvnrepositoryUri}')
def res = client.get('/artifact/org.apache.groovy/groovy-all') {
    header 'User-Agent', 'Mozilla/5.0 (Macintosh)'
}

assert res.status == 200

def license = res.parsed.select('span.badge.badge-license')*.text().join(', ')
assert license == 'Apache 2.0'
```

`result.parsed` dispatches to jsoup for `text/html` content types when jsoup is available, otherwise it falls back to the raw string body.

Form URL-Encoding

The `form(…)` helper sends `application/x-www-form-urlencoded` POST bodies:

```
def result = http.post('/login') {
    form(username: 'admin', password: 'p@ssw0rd')
}

assert result.status == 200
```

`form(…)` encodes values as `application/x-www-form-urlencoded` and sets `Content-Type` automatically (unless you override it with `header`).

Unlike `query(…)`, `form(…)` uses *form semantics*, so spaces become `+`.

HTML Login Example

Combining `form(…)` with HTML parsing enables simple login flows:

```
def app = http {
    baseUri '${rootUri}'
    followRedirects true
    header 'User-Agent', 'Mozilla/5.0 (Macintosh)'
}

def loginPage = app.get('/login')
```

```
assert loginPage.status == 200
assert loginPage.html.select('h1').text() == 'Please Login'

def afterLogin = app.post('/login') {
    form(username: 'admin', password: 'p@ssw0rd')
}

assert afterLogin.status == 200
assert afterLogin.html.select('h1').text() == 'Admin Section'
```

Content-Type Auto-Parsing

`result.parsed` dispatches by the response `Content-Type`:

Content-Type	Parsed as
`application/json`, `application/\*+json`	JSON object (`JsonSlurper`)
`application/xml`, `text/xml`, `application/*+xml`	XML object (`XmlSlurper`)
`text/html`	jsoup `Document` (if jsoup is on the classpath, otherwise raw string)
anything else	raw string body

Advanced Client Configuration

The `clientConfig` hook gives direct access to the JDK `HttpClient.Builder` for advanced configuration — authenticator, SSL context, proxy, cookie handler:

```
def http = HttpBuilder.http {
    baseUri 'https://api.example.com'
    header 'Authorization', "Bearer ${token}"
    clientConfig { builder ->
        builder.authenticator(myAuthenticator)
               .sslContext(mySSLContext)
               .proxy(ProxySelector.of(new InetSocketAddress('proxy.corp', 8080)))
    }
}
```

The `clientConfig` closure receives the `HttpClient.Builder` before `build()` is called, so any JDK-supported configuration is available — with one exception: the redirect policy is owned by `followRedirects` (see Redirects), so a policy set on the raw builder is overridden.

Redirects

Redirects are not followed by default: a 3xx response is returned to the caller, which can inspect its `Location` header and decide what to do. Opt in with `followRedirects true`:

```
def http = HttpBuilder.http {
```

```
    baseUri 'https://api.example.com'
    followRedirects true
    header 'Authorization', "Bearer ${token}"
}
```

At most 5 hops are followed. Methods are rewritten the same way as the JDK's `HttpClient.Redirect.NORMAL` policy: `301` and `302` downgrade `POST` to `GET` (dropping the body), `303` downgrades everything except `HEAD` to `GET`, and `307`/`308` preserve the method and body.

Headers do not leave their origin

Headers configured on the builder or on an individual request are treated as intended for one origin only: the scheme, host, and port of the first request. Which of them hold credentials is not something `HttpBuilder` can know — a token may sit in `Authorization`, `Cookie`, `X-Api-Key`, or any custom header — so the policy does not name header names:

- A hop that stays on the origin keeps every header.
- A hop that leaves the origin sheds *all* caller-supplied headers, builder defaults and per-request headers alike, for the remainder of the chain. Returning to the original origin later does not restore them: once another server directs the chain, no hop gets the credentials back.
- The one exception is `Content-Type` on a `307`/`308` hop, which forwards the request body: the header describing that body travels with it, as in browsers.
- A hop from an `https` URL to a plain `http` URL is never followed (matching `HttpClient.Redirect.NORMAL`); the 3xx is returned to the caller.

Note that the scheme is part of the origin, so a redirect from `http://api.example.com` to `https://api.example.com` also sheds headers.

Streaming requests (`streamAsync` and friends) never follow redirects; a 3xx is always returned as-is.

Which settings for which scenario

Scenario	What to configure
Same-origin API that occasionally redirects (`/old` → `/new`)	`followRedirects true` — headers survive every same-origin hop
API that redirects downloads to a CDN or pre-signed URL (e.g. an API host redirecting to cloud storage)	`followRedirects true` — credentials are shed on the cross-origin hop, which such targets expect (pre-signed URLs typically reject requests that also carry an `Authorization` header)
Service that redirects to a sibling host which must receive your credentials (e.g. a regional endpoint on another hostname)	Send the request to the final host directly; or leave `followRedirects` off, read `Location` from the 3xx, and re-issue the request yourself — an explicit decision to send credentials to the second origin
Server upgrades `http` to `https` with a redirect	Configure the `https` URL as the `baseUri` so no upgrade hop is needed

Scenario	What to configure
Requests (and redirect hops) must never leave the API at all	`confineToBaseUri true` — see below
Streaming a response served behind a redirect	Resolve the `Location` of the 3xx yourself and stream from the target

Confining requests to the base URI

When request URIs are built from untrusted input, `confineToBaseUri true` rejects any request — and any redirect hop — that resolves outside the `baseUri` namespace, whether to another origin or above the base path (e.g. via `..` segments or an absolute path). Violations throw `SecurityException` instead of executing. It requires `baseUri` to be set:

```
def http = HttpBuilder.http {
    baseUri 'https://api.example.com/v2/'
    confineToBaseUri true
    followRedirects true
}

http.get('items/42')              // ok: https://api.example.com/v2/items/42
http.get('../admin')              // SecurityException: escapes /v2/
http.get('https://evil.example')  // SecurityException: different origin
```

Async Requests

Every HTTP method has an async variant that returns a `CompletableFuture<HttpResult>` using the JDK `HttpClient.sendAsync()` natively (no extra threads):

```
def http = HttpBuilder.http('https://api.example.com')

def future = http.getAsync('/users/alice')
// ... do other work while the request is in flight ...
def result = future.get()
assert result.json.name == 'alice'
```

Available methods: `getAsync`, `postAsync`, `putAsync`, `deleteAsync`, `patchAsync`, and the generic `requestAsync(method, uri, spec)`.

These compose naturally with `CompletableFuture` methods:

```
http.getAsync('/data')
    .thenApply { it.json }
    .thenAccept { data -> println "Got: $data" }
```

If the Groovy async module is on the classpath, these futures are automatically `await`-able:

```
def result = await http.getAsync('/data')
```

Declarative HTTP Clients

For APIs with multiple endpoints, you can define a typed interface and let Groovy generate the implementation at compile time. Annotate an interface with `@HttpBuilderClient` and each method with an HTTP method annotation:

```
@HttpBuilderClient('https://api.example.com')
@Header(name = 'Accept', value = 'application/json')
interface BookApi {
    @Get('/books/{id}')
    Map getBook(String id)

    @Get('/books')
    List searchBooks(@Query('q') String query)

    @Post('/books')
    Map createBook(@Body Map book)

    @Delete('/books/{id}')
    void deleteBook(String id)
}

def api = BookApi.create()
def book = api.getBook('978-0-321-12521-7')
```

The AST transform generates an implementation class that uses `HttpBuilder` under the hood. Three `create()` factory methods are added to the interface:

- `create()` — uses the annotation URL and default settings
- `create(String baseUrl)` — overrides the base URL at runtime
- `create(Closure config)` — full control over the underlying `HttpBuilder` for runtime values (e.g. auth tokens) or anything not covered by the annotations, including `clientConfig` for JDK-level settings (authenticator, SSL, proxy)

```
// Runtime auth token
def api = MyApi.create {
    baseUri 'https://api.example.com'
    header 'Authorization', "Bearer ${token}"
}

// Full JDK client customization
def api = MyApi.create {
    baseUri 'https://internal.corp'
    clientConfig { builder ->
```

```
        builder.sslContext(mySSLContext)
    }
}
```

Parameter Mapping

Method parameters are mapped automatically by convention — no annotation is needed for the common case:

Condition	Mapping
Name matches `{placeholder}` in URL	Path variable — substituted into the URL (URL-encoded)
Annotated with `@Body`	Request body — serialized as JSON
Everything else	Query parameter — the parameter name is used as the query key

The `@Query` annotation is only needed when the query parameter name differs from the method parameter name:

```
@Get('/search')
List search(String q)                    // ?q=...  (implied)

@Get('/search')
List search(@Query('q') String query)    // ?q=...  (explicit, different name)
```

HTTP Methods

All standard HTTP methods are supported via annotations: `@Get`, `@Post`, `@Put`, `@Delete`, `@Patch`.

```
@Patch('/items/{id}')
Map patchItem(String id, @Body Map updates)
```

Headers

`@Header` annotations can be placed on the interface (applies to all methods) or on individual methods. Method-level headers are merged with interface-level headers.

Timeouts and Redirects

Connection timeout, default request timeout, and redirect following can be configured on the `@HttpBuilderClient` annotation:

```
@HttpBuilderClient(value = 'https://api.example.com',
                   connectTimeout = 5,
                   requestTimeout = 10,
                   followRedirects = true)
```

```
interface MyApi {
    @Get('/users/{id}')
    Map getUser(String id)              // uses default 10s timeout

    @Get('/reports/generate')
    @Timeout(60)
    Map generateReport()                // overrides to 60s for this method
}
```

- `connectTimeout` — how long to wait for the TCP connection (client-level, in seconds)
- `requestTimeout` — default request timeout applied to all methods (in seconds)
- `@Timeout(value)` — per-method override of the request timeout (in seconds)
- `followRedirects` — whether to follow HTTP redirects, with the origin-confined header semantics described in Redirects

Default `0` means no timeout.

Return Types

The return type of each method determines how the response is processed:

Return type	Behaviour
`Map` or `List`	Response parsed as JSON
Typed class (e.g. `User`)	Response parsed as JSON, then coerced to the target type
`GPathResult`	Response parsed as XML (via `XmlSlurper`)
jsoup `Document`	Response parsed as HTML (requires jsoup on classpath)
`String`	Raw response body
`HttpResult`	Full response (status, headers, body)
`void`	Response discarded
`CompletableFuture<T>`	Asynchronous execution; inner type determines parsing

For typed responses, the JSON is parsed and coerced to the target class using Groovy's `as` coercion:

```
class User {
    String name
    String bio
}

@HttpBuilderClient('https://api.example.com')
interface UserApi {
    @Get('/users/{id}')
    User getUser(String id)
}
```

Request Bodies

By default, `@Body` parameters are serialized as JSON. Additional body modes are available:

Annotation	Behaviour
`@Body`	JSON body (default)
`@BodyText`	Plain text body (sent as-is)
`@Form` (on method)	All non-path parameters become form-encoded fields (`application/x-www-form-urlencoded`)

```
@Post('/login')
@Form
Map login(String username, String password)    // form-encoded POST

@Post('/notes')
Map createNote(@BodyText String content)       // plain text body
```

Error Handling

By default, HTTP 4xx/5xx responses throw a `RuntimeException`. You can declare a specific exception type in the method's `throws` clause, and the generated client will throw that type instead:

```
@Get('/users/{id}')
Map getUser(String id) throws NotFoundException
```

The exception class is instantiated by trying constructors in order: `(int status, String body)`, then `(String message)`, then no-arg.

Async

Methods returning `CompletableFuture` execute asynchronously:

```
@HttpBuilderClient('https://api.example.com')
interface AsyncApi {
    @Get('/data/{id}')
    CompletableFuture<Map> getData(String id)
}

def api = AsyncApi.create()
def future = api.getData('42')
// ... do other work ...
def data = future.get()
```

Scripting Ant tasks

Groovy integrates very well with Apache Ant thanks to AntBuilder.

The <groovy> Ant Task

`<groovy>`



Here we describe an Ant task for using Groovy from within an Ant build file. You may also be interested in Ant's built-in `script` task which supports Groovy and other languages, or `AntBuilder` which lets you write Ant build scripts in Groovy rather than XML.

Executes a series of Groovy statements from Apache Ant. Statements can either be read in from a resource or as direct text between the enclosing Groovy tags.

Required taskdef

Assuming all the groovy jars you need are in *my.classpath* (this will be `groovy-VERSION.jar`, `groovy-ant-VERSION.jar` plus any modules and transitive dependencies you might be using) you will need to declare this task at some point in the `build.xml` prior to the `groovy` task being invoked.

```
<taskdef name="groovy"
         classname="org.codehaus.groovy.ant.Groovy"
         classpathref="my.classpath"/>
```

You can simply place statements between the `groovy` tags like this:

```
<groovy>
...
</groovy>
```

Or you can supply the Groovy source script as a resource. You can specify the pathname using the `src` attribute like this:

```
<groovy src="/some/path/MyGroovyScript.groovy" otherAttributes="...">
```

Or as a nested `fileset` like this (though the fileset definition is expected to select just one file):

```
<groovy>
    <fileset file="MyGroovyScript.groovy"/>
</groovy>
```

Or as a nested single element resource collection which could look like any of these:

```
<groovy>
    <file file="MyGroovyScript.groovy"/>
</groovy>

<groovy>
    <url url="https://some.domain/some/path/to/MyGroovyScript.groovy"/>
</groovy>

<groovy>
    <javaconstant name="some.packagename.SomeClass.MY_CODE_FRAGMENT"/>
</groovy>
```

You may also supply a filter chain like this:

```
<groovy>
    <fileset file="MyGroovyScript.groovy"/>
    <!-- take 5 lines after skipping 18 lines, just as an example -->
    <filterchain>
        <headfilter lines="5" skip="18"/>
    </filterchain>
</groovy>
```

You might need to use the *contextClassLoader* attribute (see below) if any of your modules load services via the classpath, e.g. `groovy-json`.

<groovy> attributes

Attribute	Description	Required
src	File containing Groovy statements. The directory containing the file is added to the classpath.	Yes, unless statements enclosed within tags
classpath	The classpath to use.	No
classpathref	The classpath to use, given as reference to a PATH defined elsewhere.	No
output	Set the output file; defaults to the Ant log.	No
append	If enabled and output is to a file, append to existing file rather than overwrite. Defaults to false.	No
fork	If enabled the script will be executed in a forked JVM process (disabled by default).	No
inheritAll	If enabled and `fork` is true, all properties from the parent Ant project are passed to the forked JVM as system properties. For selective passing, use a nested `<syspropertyset>` instead. Defaults to false.	No

Attribute	Description	Required
scriptBaseClass	The name of the base class for scripts.	No
parameters	Generates metadata for reflection on method parameter names on JDK 8 and above. Defaults to false.	No
useGroovyShell	If enabled a new GroovyShell is used to run the script. Special variables won't be available but you don't need Ant in the classpath. Defaults to false.	No
includeAntRuntime	If enabled the system classpath will be included on the classpath when forking. Defaults to true.	No
stacktrace	If enabled a stacktrace will be reported if an error occurs during compilation. Defaults to false.	No
configScript	Sets the configuration script for the groovy compiler configuration.	No
contextClassLoader	If enabled, the contextClassLoader to be set with the classLoader of the shell used to run the script. Not used if fork is true.	No

Parameters specified as nested elements

<classpath>

Groovy's classpath attribute is a PATH like structure and can also be set via a nested classpath element.

<arg>

Arguments can be set via one or more nested <arg> elements using the standard Ant command line conventions.

<sysproperty> and <syspropertyset>

Because `<groovy>` extends Ant's `<java>` task, you can use nested `<sysproperty>` and `<syspropertyset>` elements to pass specific properties into the forked JVM when `fork="true"`. The forked script reads them via `System.getProperty("name")`.

If you want to pass *all* properties from the parent Ant project, set `inheritAll="true"` instead. Explicit `<sysproperty>` and `<syspropertyset>` entries take precedence on name collision.



The `properties` binding (see below) reflects only the child JVM's Ant project when `fork="true"`, not the parent project's properties. Use system properties (via `<sysproperty>`, `<syspropertyset>`, or `inheritAll`) to communicate values to a forked script.

Available bindings

A number of bindings are in scope for use within your Groovy statements.

Name	Description
ant	an instance of `AntBuilder` that knows about the current ant project
project	the current ant project
properties	a `Map` of ant properties
target	the owning target that invoked this groovy script
task	the wrapping task, can access anything needed in `org.apache.tools.ant.Task`
args	command line arguments, if any

Examples

Hello world, version 1:

```
<groovy>
println "Hello World"
</groovy>
```

Hello world, version 2:

```
<groovy>
ant.echo "Hello World"
</groovy>
```

List all xml files in the current directory:

```
<groovy>
xmlfiles = new File(".").listFiles().findAll{ it =~ "\.xml$" }
xmlfiles.sort().each { println it.toString() }
</groovy>
```

List all xml files within a jar:

```
<zipfileset id="found" src="foobar.jar"
            includes="**/*.xml"/>
<groovy>
    project.references.found.each {
        println it.name
    }
</groovy>
```

Run a script:

```
<groovy src="/some/directory/some/file.groovy">
  <classpath>
    <pathelement location="/my/groovy/classes/directory"/>
  </classpath>
</groovy>
```

Find all `Builder` classes having an `org.*` package within a directory of jars:

```
<property name="local.target" value="C:/Projects/GroovyExamples"/>
<groovy>
import java.util.jar.JarFile
def classes = []
def resourceNamePattern = /org\/.*\/.*Builder.class/
def jarNamePattern = /.*(beta|commons).*jar$/

def libdir = new File("${properties['local.target']}/lib")
libdir.listFiles().grep(~jarNamePattern).each { candidate ->
    new JarFile(candidate).entries().each { entry ->
        if (entry.name ==~ resourceNamePattern) classes += entry.name
    }
}
properties["builder-classes"] = classes.join(' ')
</groovy>
<echo message='${builder-classes}'/>
```

Which might result in something like:

```
org/apache/commons/cli/PatternOptionBuilder.class
org/apache/commons/cli/OptionBuilder.class
org/codehaus/groovy/tools/groovydoc/GroovyRootDocBuilder.class
org/custommonkey/xmlunit/HTMLDocumentBuilder.class
org/custommonkey/xmlunit/TolerantSaxDocumentBuilder.class
```

FileScanner version of above (with a slight variation on collecting the names):

```
<groovy>
import java.util.jar.JarFile
def resourceNamePattern = /org\/.*\/.*Builder.class/
def candidates = ant.fileScanner {
    fileset(dir: '${local.target}/lib') {
        include(name: '*beta*.jar')
        include(name: '*commons*.jar')
    }
}
def classes = candidates.collect {
```

```
    new JarFile(it).entries().collect { it.name }.findAll {
        it ==~ resourceNamePattern
    }
}.flatten()
properties["builder-classes"] = classes.join(' ')
</groovy>
```

Calling out to a web service from your Ant script:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="SOAP example" default="main" basedir=".">
    <property environment="env"/>
    <property name="celsius" value="0"/>
    <target name="main">
        <taskdef name="groovy" classname="org.codehaus.groovy.ant.Groovy">
            <classpath>
                <fileset dir="${env.GROOVY_HOME}" includes="lib/groovy-
*.jar,lib/ivy*.jar"/>
            </classpath>
        </taskdef>
        <groovy>
            @Grab('org.codehaus.groovy.modules:groovyws:0.5.1')
            import groovyx.net.ws.WSClient
            def url = 'http://www.w3schools.com/webservices/tempconvert.asmx?WSDL'
            def proxy = new WSClient(url, this.class.classLoader)
            proxy.initialize()
            ant.echo "I'm freezing at ${properties.celsius} degrees Celsius"
            properties.result = proxy.CelsiusToFahrenheit(properties.celsius)
        </groovy>
        <antcall target="results"/>
    </target>
    <target name="results">
        <echo message="I'm freezing at ${result} degrees Fahrenheit"/>
    </target>
</project>
```

Which will output the following (along with some informational messages):

```
main:
     ...
     [echo] I'm freezing at 0 degrees Celsius
results:
     [echo] I'm freezing at 32 degrees Fahrenheit

BUILD SUCCESSFUL
```

Setting arguments:

```
<target name="run">
    <groovy>
        <arg line="1 2 3"/>
        <arg value="4 5"/>
        println args.size()
        println args[2]
        args.each{ ant.echo(message:it) }
    </groovy>
</target>
```

Passing Ant properties into a forked JVM selectively via `<syspropertyset>`:

```
<property name="myapp.url" value="https://example.com"/>
<property name="myapp.timeout" value="30"/>
<groovy fork="true">
    <syspropertyset>
        <propertyref prefix="myapp."/>
    </syspropertyset>
    println "Calling ${System.getProperty('myapp.url')} with timeout
${System.getProperty('myapp.timeout')}"
</groovy>
```

Or pass every parent property at once with `inheritAll`:

```
<property name="target.env" value="staging"/>
<groovy fork="true" inheritAll="true">
    println "Deploying to ${System.getProperty('target.env')}"
</groovy>
```

Output:

```
Buildfile: build.xml

run:
   [groovy] 4
   [groovy] 3
     [echo] 1
     [echo] 2
     [echo] 3
     [echo] 4 5

BUILD SUCCESSFUL
```

The <groovyc> Ant Task

`<groovyc>`

Description

Compiles Groovy source files and, if joint compilation option is used, Java source files from Apache Ant.

Required taskdef

Assuming the groovy jars are in *groovy.libs*, you will need to declare this task at some point in the `build.xml` prior to the `groovyc` task being invoked. Consider also adding any additional Groovy module jars, libraries and potentially transitive dependencies you might be using.

```
<taskdef name="groovyc" classname="org.codehaus.groovy.ant.Groovyc">
  <classpath>
    <fileset file="${groovy.libs}/groovy-ant-VERSION.jar"/>
    <fileset file="${groovy.libs}/groovy-VERSION.jar"/>
  </classpath>
</taskdef>
```

<groovyc> Attributes

Attribute	Description	Required
srcdir	Location of the Groovy (and possibly Java) source files.	Yes
destdir	Location to store the class files.	Yes
classpath	The classpath to use.	No
classpathref	The classpath to use given as a path references.	No
sourcepath	The sourcepath to use.	No
sourcepathref	The sourcepath to use given as a path reference.	No
encoding	Encoding of source files.	No
verbose	Asks the compiler for verbose output; defaults to no.	No
includeAntRuntime	Whether to include the Ant run-time libraries in the classpath; defaults to yes.	No

Attribute	Description	Required
includeJavaRuntime	Whether to include the default run-time libraries from the executing VM in the classpath; defaults to no.	No
includeDestClasses	This property controls whether to include the destination classes directory in the classpath given to the compiler. The default value is "true".	No
fork	Whether to execute groovyc using a spawned instance of the JVM; defaults to no.	No
memoryInitialSize	The initial size of the memory for the underlying VM, if using fork mode; ignored otherwise. Defaults to the standard VM memory setting. (Examples: 83886080, 81920k, or 80m)	No
memoryMaximumSize	The maximum size of the memory for the underlying VM, if using fork mode; ignored otherwise. Defaults to the standard VM memory setting. (Examples: 83886080, 81920k, or 80m)	No
inheritAll	If enabled and `fork` is true, all properties from the parent Ant project are passed to the forked compiler JVM as system properties. For selective passing, use a nested `<sysproperty>` or `<syspropertyset>` instead. Defaults to false.	No
failonerror	Indicates whether compilation errors will fail the build; defaults to true.	No
proceed	Inverse alias for *failonerror*.	No
listfiles	Indicates whether the source files to be compiled will be listed; defaults to no.	No

Attribute	Description	Required
stacktrace	if true each compile error message will contain a stacktrace	No
indy	Enable compilation with the ``invoke dynamic'' support when using Groovy 2.0 and beyond and running on JDK 7	No
scriptBaseClass	Sets the base class for Groovy scripts	No
stubdir	Set the stub directory into which the Java source stub files should be generated. The directory need not exist and will not be deleted automatically - though its contents will be cleared unless 'keepStubs' is true. Ignored when forked.	No
keepStubs	Set the keepStubs flag. Defaults to false. Set to true for debugging. Ignored when forked.	No
forceLookupUnnamedFiles	The Groovyc Ant task is frequently used in the context of a build system that knows the complete list of source files to be compiled. In such a context, it is wasteful for the Groovy compiler to go searching the classpath when looking for source files and hence by default the Groovyc Ant task calls the compiler in a special mode with such searching turned off. If you wish the compiler to search for source files then you need to set this flag to true. Defaults to false.	No
configscript	Set the configuration file used to customize the compilation configuration.	No

Attribute	Description	Required
errorFormat	How errors and warnings are rendered: `full` (the default, with the source line and a caret) or `short` (one `file:line:column: severity: message` line per diagnostic).	No
tolerance	The number of non-fatal errors to collect before compilation is abandoned. Zero means unlimited. Defaults to 10.	No
parameters	Generates metadata for reflection on method parameter names on JDK 8 and above. Defaults to false.	No
previewFeatures	Enables the JEP preview features on JDK 12 and above. Defaults to false.	No
targetBytecode	Sets the bytecode compatibility level.	No
javahome	Sets the `java.home` value to use, default is the current JDK's home.	No
executable	Sets the name of the java executable to use when invoking the compiler in forked mode, ignored otherwise.	No
scriptExtension	Set the extension to use when searching for Groovy source files. Accepts extensions in the form *.groovy, .groovy or groovy.	No
updatedProperty	The property to set on compilation success. This property will not be set if the compilation fails, or if there are no files to compile.	No
errorProperty	The property to set on compilation failure. This property will be set if the compilation fails.	No

Example:

```
<path id="classpath.main">
  <fileset dir="${groovy.libs}" includes="*.jar" excludes="groovy-ant-*.jar"/>
  ...
</path>
<groovyc srcdir="${dir.sources}" destdir="${dir.classes}"
classpathref="classpath.main"
         fork="true" includeantruntime="false" configscript="config.groovy"
targetBytecode="1.8"/>
```

<groovyc> Nested Elements

element	kind	Required	Replaces Attribute
src	a path structure	Yes (unless srcdir is used)	srcdir
classpath	a path structure	No	classpath or classpathref
javac	javac task	No	N/A
jvmarg	command line argument	No	N/A
sysproperty	an Ant `<sysproperty>`	No	N/A
syspropertyset	an Ant `<propertyset>`	No	N/A

Notes:

- For path structures see for example https://ant.apache.org/manual/using.html#path
- For usages of the `javac` task see https://ant.apache.org/manual/Tasks/javac.html
- The nested `javac` task behaves more or less as documented for the top-level `javac` task. `srcdir`, `destdir`, `fork`, `memoryInitialSize`, and `memoryMaximumSize` for the nested `javac` task are taken from the enclosing `groovyc` task. If these attributes or any else that are not explicitly supported are specified then a warning is logged, and they are ignored completely. `classpath` and `classpathref` specified on the nested `javac` task is merged with the values taken from the enclosing `groovyc` task and also used for the Groovy compilation. Nested inside the nested `javac` task the only element supported is `compilerarg`, and this only with the `value` attribute, which is treated like the `line` attribute of the top-level `javac` task, i.e. it is split by spaces into separate arguments. Only arguments starting with `-W`, `-X`, or `-proc:` are properly translated as needed. Anything else is supplied as-is to groovyc and has to be manually prefixed with `-F` or `-J`.

Joint Compilation

Joint compilation is enabled by using an embedded `javac` element, as shown in the following example:

```
<groovyc srcdir="${testSourceDirectory}" destdir="${testClassesDirectory}"
targetBytecode="1.8">
```

```
  <classpath>
    <pathelement path="${mainClassesDirectory}"/>
    <pathelement path="${testClassesDirectory}"/>
    <path refid="testPath"/>
  </classpath>
  <javac debug="true" source="1.8" target="1.8" />
</groovyc>
```

More details about joint compilation can be found in the joint compilation section.

Passing JVM arguments and system properties to a forked compiler

When `fork="true"`, you can attach arbitrary JVM arguments and system properties to the spawned compiler JVM via nested `<jvmarg>`, `<sysproperty>`, and `<syspropertyset>` elements (mirroring Ant's `<java>` task). This is useful for compile-time configuration consumed by AST transforms, the parser/lexer, or annotation processors.

```
<groovyc srcdir="src" destdir="build/classes" fork="true">
  <jvmarg value="--add-opens=java.base/java.lang=ALL-UNNAMED"/>
  <sysproperty key="my.compile.flag" value="true"/>
  <syspropertyset>
    <propertyref prefix="myapp."/>
  </syspropertyset>
</groovyc>
```

To pass *every* parent project property at once, set `inheritAll="true"`. Explicit `<sysproperty>` and `<syspropertyset>` entries take precedence on name collision.

Template engines

Introduction

Groovy supports multiple ways to generate text dynamically including `GStrings`, `printf` and MarkupBuilder just to name a few. In addition to these, there is a dedicated template framework which is well-suited to applications where the text to be generated follows the form of a static template.

Template framework

The template framework in Groovy consists of a `TemplateEngine` abstract base class that engines must implement and a `Template` interface that the resulting templates they generate must implement.

Included with Groovy are several template engines:

- `SimpleTemplateEngine` - for basic templates
- `StreamingTemplateEngine` - functionally equivalent to `SimpleTemplateEngine`, but can handle

strings larger than 64k

- `GStringTemplateEngine` - stores the template as writeable closures (useful for streaming scenarios)
- `XmlTemplateEngine` - works well when the template and output are valid XML
- `MarkupTemplateEngine` - a very complete, optimized, template engine

SimpleTemplateEngine

Shown here is the `SimpleTemplateEngine` that allows you to use JSP-like scriptlets (see example below), script, and EL expressions in your template in order to generate parametrized text. Here is an example of using the system:

```
def text = 'Dear "$firstname $lastname",\nSo nice to meet you in <% print city
%>.\nSee you in ${month},\n${signed}'

def binding = ["firstname":"Sam", "lastname":"Pullara", "city":"San Francisco",
"month":"December", "signed":"Groovy-Dev"]

def engine = new groovy.text.SimpleTemplateEngine()
def template = engine.createTemplate(text).make(binding)

def result = 'Dear "Sam Pullara",\nSo nice to meet you in San Francisco.\nSee you in
December,\nGroovy-Dev'

assert result == template.toString()
```

While it is generally not deemed good practice to mix processing logic in your template (or view), sometimes very simple logic can be useful. E.g. in the example above, we could change this:

```
$firstname
```

to this (assuming we have set up a static import for capitalize **inside** the template):

```
${firstname.capitalize()}
```

or this:

```
<% print city %>
```

to this:

```
<% print city == "New York" ? "The Big Apple" : city %>
```

Advanced Usage Note

If you happen to be embedding your template directly in your script (as we did above) you have to be careful about backslash escaping. Because the template string itself will be parsed by Groovy before it is passed to the templating framework, you have to escape any backslashes inside GString expressions or scriptlet 'code' that are entered as part of a Groovy program. E.g. if we wanted quotes around *The Big Apple* above, we would use:

```
<% print city == "New York" ? "\\"The Big Apple\\"" : city %>
```

Similarly, if we wanted a newline, we would use:

```
\\n
```

in any GString expression or scriptlet 'code' that appears inside a Groovy script. A normal “\n” is fine within the static template text itself or if the entire template itself is in an external template file. Similarly, to represent an actual backslash in your text you would need

```
\\\\
```

in an external file or

```
\\\\
```

in any GString expression or scriptlet 'code'. (Note: the necessity to have this extra slash may go away in a future version of Groovy if we can find an easy way to support such a change.)

StreamingTemplateEngine

The `StreamingTemplateEngine` engine is functionally equivalent to the `SimpleTemplateEngine`, but creates the template using writable closures making it more scalable for large templates. Specifically this template engine can handle strings larger than 64k.

It uses JSP style <% %> script and <%= %> expression syntax or GString style expressions. The variable 'out' is bound to the writer that the template is being written to.

Frequently, the template source will be a file but here we show a simple example providing the template as a string:

```
def text = '''\
Dear <% out.print firstname %> ${lastname},

We <% if (accepted) out.print 'are pleased' else out.print 'regret' %> \
to inform you that your paper entitled
'$title' was ${ accepted ? 'accepted' : 'rejected' }.
```

```
The conference committee.'''

def template = new groovy.text.StreamingTemplateEngine().createTemplate(text)

def binding = [
    firstname : "Grace",
    lastname  : "Hopper",
    accepted  : true,
    title     : 'Groovy for COBOL programmers'
]

String response = template.make(binding)

assert response == '''Dear Grace Hopper,

We are pleased to inform you that your paper entitled
'Groovy for COBOL programmers' was accepted.

The conference committee.'''
```

GStringTemplateEngine

As an example of using the `GStringTemplateEngine`, here is the example above done again (with a few changes to show some other options). First we will store the template in a file this time:

test.template

```
Dear "$firstname $lastname",
So nice to meet you in <% out << (city == "New York" ? "\\"The Big Apple\\"" : city)
%>.
See you in ${month},
${signed}
```

Note that we used `out` instead of `print` to support the streaming nature of `GStringTemplateEngine`. Because we have the template in a separate file, there is no need to escape the backslashes. Here is how we call it:

```
def f = new File('test.template')
def engine = new groovy.text.GStringTemplateEngine()
def template = engine.createTemplate(f).make(binding)
println template.toString()
```

and here is the output:

```
Dear "Sam Pullara",
So nice to meet you in "The Big Apple".
See you in December,
```

```
Groovy-Dev
```

XmlTemplateEngine

`XmlTemplateEngine` for use in templating scenarios where both the template source and the expected output are intended to be XML. Templates may use the normal `${expression}` and `$variable` notations to insert an arbitrary expression into the template. In addition, support is also provided for special tags: `<gsp:scriptlet>` (for inserting code fragments) and `<gsp:expression>` (for code fragments which produce output).

Comments and processing instructions will be removed as part of processing and special XML characters such as `<`, `>`, `"` and `'` will be escaped using the respective XML notation. The output will also be indented using standard XML pretty printing.

The xmlns namespace definition for gsp: tags will be removed but other namespace definitions will be preserved (but may change to an equivalent position within the XML tree).

Normally, the template source will be in a file but here is a simple example providing the XML template as a string:

```
def binding = [firstname: 'Jochen', lastname: 'Theodorou', nickname: 'blackdrag',
salutation: 'Dear']
def engine = new groovy.text.XmlTemplateEngine()
def text = '''\
    <document xmlns:gsp='http://groovy.codehaus.org/2005/gsp' xmlns:foo='baz'
type='letter'>
        <gsp:scriptlet>def greeting = "${salutation}est"</gsp:scriptlet>
        <gsp:expression>greeting</gsp:expression>
        <foo:to>$firstname "$nickname" $lastname</foo:to>
        How are you today?
    </document>
'''
def template = engine.createTemplate(text).make(binding)
println template.toString()
```

This example will produce this output:

```
<document type='letter'>
  Dearest
  <foo:to xmlns:foo='baz'>
    Jochen &amp;quot;blackdrag&amp;quot; Theodorou
  </foo:to>
  How are you today?
</document>
```

The MarkupTemplateEngine

This template engine is a template engine primarily aimed at generating XML-like markup (XML, XHTML, HTML5, ...), but that can be used to generate any text based content. Unlike traditional template engines, this one relies on a DSL that uses the builder syntax. Here is a sample template:

```
xmlDeclaration()
cars {
   cars.each {
       car(make: it.make, model: it.model)
   }
}
```

If you feed it with the following model:

```
model = [cars: [new Car(make: 'Peugeot', model: '508'), new Car(make: 'Toyota', model:
'Prius')]]
```

It would be rendered as:

```
<?xml version='1.0'?>
<cars><car make='Peugeot' model='508'/><car make='Toyota' model='Prius'/></cars>
```

The key features of this template engine are:

- a *markup builder like* syntax
- templates are compiled into bytecode
- fast rendering
- optional type checking of the model
- includes
- internationalization support
- fragments/layouts

The template format

Basics

Templates consist of Groovy code. Let's explore the first example more thoroughly:

```
xmlDeclaration()                                ①
cars {                                          ②
   cars.each {                                  ③
       car(make: it.make, model: it.model)      ④
   }                                            ⑤
```

```
}
```

① renders the XML declaration string.

② opens a `cars` tag

③ `cars` is a variable found in the *template model*, which is a list of `Car` instances

④ for each item, we create a `car` tag with the attributes from the `Car` instance

⑤ closes the `cars` tag

As you can see, regular Groovy code can be used in the template. Here, we are calling `each` on a list (retrieved from the model), allowing us to render one `car` tag per entry.

In a similar fashion, rendering HTML code is as simple as this:

```
yieldUnescaped '<!DOCTYPE html>'                                          ①
html(lang:'en') {                                                         ②
    head {                                                                ③
        meta('http-equiv':'"Content-Type" content="text/html; charset=utf-8"')  ④
        title('My page')                                                  ⑤
    }                                                                     ⑥
    body {                                                                ⑦
        p('This is an example of HTML contents')                          ⑧
    }                                                                     ⑨
}                                                                         ⑩
```

① renders the HTML doctype special tag

② opens the `html` tag with an attribute

③ opens the `head` tag

④ renders a `meta` tag with one `http-equiv` attribute

⑤ renders the `title` tag

⑥ closes the `head` tag

⑦ opens the `body` tag

⑧ renders a `p` tag

⑨ closes the `body` tag

⑩ closes the `html` tag

The output is straightforward:

```
<!DOCTYPE html><html lang='en'><head><meta http-equiv='"Content-Type"
content="text/html; charset=utf-8"'/><title>My page</title></head><body><p>This is an
example of HTML contents</p></body></html>
```



With some configuration, you can have the output pretty printed, with newlines

> and indent automatically added.

Support methods

In the previous example, the doctype declaration was rendered using the `yieldUnescaped` method. We have also seen the `xmlDeclaration` method. The template engine provides several support methods that will help you render contents appropriately:

Method	Description	Example

yield	Renders contents, but escapes it before rendering	**Template**:
		`yield 'Some text with <angle brackets>'`
		Output:
		`Some text with <angle brackets>`

yieldUnescaped	Renders raw contents. The argument is rendered as is, without escaping.	**Template**:
		`yieldUnescaped 'Some text with <angle brackets>'`
		Output:
		`Some text with <angle brackets>`

xmlDeclaration	Renders an XML declaration String. If the encoding is specified in the configuration, it is written in the declaration.	**Template**:
		`xmlDeclaration()`
		Output:
		`<?xml version='1.0'?>`
		If `TemplateConfiguration#getDeclarationEncoding` is not null:
		Output:
		`<?xml version='1.0' encoding='UTF-8'?>`

Method	Description	Example

comment	Renders raw contents inside an XML comment	**Template**:
		`comment 'This is <a href="foo.html">commented out</a>'`
		Output:
		`<!--This is <a href="foo.html">commented out</a>-->`

newLine	Renders a new line. See also `TemplateConfiguration#setAutoNewLine` and `TemplateConfiguration#setNewLineString`.	**Template**:
		`p('text')`
		`newLine()`
		`p('text on new line')`
		Output:
		`<p>text</p>`
		`<p>text on new line</p>`

pi	Renders an XML processing instruction.	**Template**:
		`pi("xml-stylesheet":[href:"mystyle.css", type:"text/css"])`
		Output:
		`<?xml-stylesheet href='mystyle.css' type='text/css'?>`

tryEscape	Returns an escaped string for an object, if it is a `String` (or any type derived from `CharSequence`). Otherwise returns the object itself.	**Template**:
		`yieldUnescaped tryEscape('Some text with <angle brackets>')`
		Output:
		`Some text with <angle brackets>`

Includes

The `MarkupTemplateEngine` supports inclusion of contents from another file. Included contents may be:

- another template
- raw contents
- contents to be escaped

Including another template can be done using:

```
include template: 'other_template.tpl'
```

Including a file as raw contents, without escaping it, can be done like this:

```
include unescaped: 'raw.txt'
```

Eventually, inclusion of text that should be escaped before rendering can be done using:

```
include escaped: 'to_be_escaped.txt'
```

Alternatively, you can use the following helper methods instead:

- `includeGroovy(<name>)` to include another template
- `includeEscaped(<name>)` to include another file with escaping
- `includeUnescaped(<name>)` to include another file without escaping

Calling those methods instead of the `include xxx:` syntax can be useful if the name of the file to be included is dynamic (stored in a variable for example). Files to be included (independently of their type, template or text) are found on **classpath**. This is one of the reasons why the `MarkupTemplateEngine` takes an optional `ClassLoader` as constructor argument (the other reason being that you can include code referencing other classes in a template).

If you don't want your templates to be on classpath, the `MarkupTemplateEngine` accepts a convenient constructor that lets you define the directory where templates are to be found.

Fragments

Fragments are nested templates. They can be used to provide improved composition in a single template. A fragment consists of a string, the inner template, and a model, used to render this template. Consider the following template:

```
ul {
    pages.each {
        fragment "li(line)", line:it
    }
```

```
}
```

The `fragment` element creates a nested template, and renders it with a model which is specific to this template. Here, we have the `li(line)` fragment, where `line` is bound to `it`. Since `it` corresponds to the iteration of `pages`, we will generate a single `li` element for each page in our model:

```
<ul><li>Page 1</li><li>Page 2</li></ul>
```

Fragments are interesting to factorize template elements. They come at the price of the compilation of a fragment per template, and they cannot be externalized.

Layouts

Layouts, unlike fragments, refer to other templates. They can be used to compose templates and share common structures. This is often interesting if you have, for example, a common HTML page setup, and that you only want to replace the body. This can be done easily with a *layout*. First of all, you need to create a layout template:

layout-main.tpl

```
html {
    head {
        title(title)                ①
    }
    body {
        bodyContents()              ②
    }
}
```

① the `title` variable (inside the title tag) is a layout variable

② the `bodyContents` call will render the body

Then what you need is a template that includes the layout:

```
layout 'layout-main.tpl',                                   ①
    title: 'Layout example',                                ②
    bodyContents: contents { p('This is the body') }        ③
```

① use the `main-layout.tpl` layout file

② set the `title` variable

③ set the `bodyContents`

As you can see, `bodyContents` will be rendered inside the layout, thanks to the `bodyContents()` call in the layout file. As a result, the template will be rendered as this:

```
<html><head><title>Layout example</title></head><body><p>This is the
```

```
body</p></body></html>
```

The call to the `contents` method is used to tell the template engine that the block of code is in fact a specification of a template, instead of a helper function to be rendered directly. If you don't add `contents` before your specification, then the contents would be rendered, but you would also see a random string generated, corresponding to the result value of the block.

Layouts are a powerful way to share common elements across multiple templates, without having to rewrite everything or use includes.

Layouts use, by default, a model which is independent from the model of the page where they are used. It is however possible to make them inherit from the parent model. Imagine that the model is defined like this:

```
model = new HashMap<String,Object>();
model.put('title','Title from main model');
```

and the following template:

```
layout 'layout-main.tpl', true,                                 ①
    bodyContents: contents { p('This is the body') }
```

① note the use of `true` to enable model inheritance

then it is not necessary to pass the `title` value to the layout as in the previous example. The result will be:

```
<html><head><title>Title from main model</title></head><body><p>This is the
body</p></body></html>
```

But it is also possible to override a value from the parent model:

```
layout 'layout-main.tpl', true,                                 ①
    title: 'overridden title',                                   ②
    bodyContents: contents { p('This is the body') }
```

① `true` means inherit from the parent model

② but `title` is overridden

then the output will be:

```
<html><head><title>overridden title</title></head><body><p>This is the
body</p></body></html>
```

Rendering contents

Creation of a template engine

On the server side, rendering templates require an instance of `groovy.text.markup.MarkupTemplateEngine` and a `groovy.text.markup.TemplateConfiguration`:

```
TemplateConfiguration config = new TemplateConfiguration();          ①
MarkupTemplateEngine engine = new MarkupTemplateEngine(config);      ②
Template template = engine.createTemplate("p('test template')");     ③
Map<String, Object> model = new HashMap<>();                         ④
Writable output = template.make(model);                              ⑤
output.writeTo(writer);                                              ⑥
```

① creates a template configuration

② creates a template engine with this configuration

③ creates a template instance from a `String`

④ creates a model to be used in the template

⑤ bind the model to the template instance

⑥ render output

There are several possible options to parse templates:

- from a `String`, using `createTemplate(String)`
- from a `Reader`, using `createTemplate(Reader)`
- from a `URL`, using `createTemplate(URL)`
- given a template name, using `createTemplateByPath(String)`

The last version should in general be preferred:

```
Template template = engine.createTemplateByPath("main.tpl");
Writable output = template.make(model);
output.writeTo(writer);
```

Configuration options

The behavior of the engine can be tweaked with several configuration options accessible through the `TemplateConfiguration` class:

Option	Default value	Description	Example

declarationEncoding	null	Determines the value of the encoding to be written when `xmlDeclaration` is called. It does **not** influence the writer you are using as output.	**Template**:
			`xmlDeclaration()`
			Output:
			`<?xml version='1.0'?>`
			If `TemplateConfiguration#getDeclarationEncoding` is not null:
			Output:
			`<?xml version='1.0' encoding='UTF-8'?>`

expandEmptyElements	false	If true, empty tags are rendered in their expanded form.	**Template**:
			`p()`
			Output:
			`<p/>`
			If `expandEmptyElements` is true:
			Output:
			`<p></p>`

Option	Default value	Description	Example

useDoubleQuotes	false	If true, use double quotes for attributes instead of simple quotes	**Template**:
			`tag(attr:'value')`
			Output:
			`<tag attr='value'/>`
			If `useDoubleQuotes` is true:
			Output:
			`<tag attr="value"/>`

newLineString	System default (system property `line.separator`)	Allows to choose what string is used when a new line is rendered	**Template**:
			`p('foo')`
			`newLine()`
			`p('baz')`
			If `newLineString='BAR'`:
			Output:
			`<p>foo</p>BAR<p>baz</p>`

autoEscape	false	If true, variables from models are automatically escaped before rendering.	See the auto escape section
autoIndent	false	If true, performs automatic indentation after new lines	See the auto formatting section
autoIndentString	four (4) spaces	The string to be used as indent.	See the auto formatting section
autoNewLine	false	If true, performs automatic insertion of new lines	See the auto formatting section
baseTemplateClass	`groovy.text.markup.BaseTemplate`	Sets the super class of compiled templates. This can be used to provide application specific templates.	See the custom templates section

Option	Default value	Description	Example
locale	Default locale	Sets the default locale for templates.	See the internationalization section



Once the template engine has been created, it is **unsafe** to change the configuration.

Automatic formatting

By default, the template engine will render output without any specific formatting. Some configuration options can improve the situation:

- `autoIndent` is responsible for auto-indenting after a new line is inserted
- `autoNewLine` is responsible for automatically inserting new lines based on the original formatting of the template source

In general, it is recommended to set both `autoIndent` and `autoNewLine` to true if you want human-readable, pretty printed, output:

```
config.setAutoNewLine(true);
config.setAutoIndent(true);
```

Using the following template:

```
html {
    head {
        title('Title')
    }
}
```

The output will now be:

```
<html>
    <head>
        <title>Title</title>
    </head>
</html>
```

We can slightly change the template so that the `title` instruction is found on the same line as the `head` one:

```
html {
    head { title('Title')
    }
```

```
}
```

And the output will reflect that:

```
<html>
    <head><title>Title</title>
    </head>
</html>
```

New lines are **only** inserted where curly braces for tags are found, and the insertion corresponds to where the nested content is found. This means that tags in the body of another tag will **not** trigger new lines unless they use curly braces themselves:

```
html {
    head {
        meta(attr:'value')          ①
        title('Title')              ②
        newLine()                   ③
        meta(attr:'value2')         ④
    }
}
```

① a new line is inserted because `meta` is not on the same line as `head`

② no new line is inserted, because we're on the same depth as the previous tag

③ we can force rendering of a new line by explicitly calling `newLine`

④ and this tag will be rendered on a separate line

This time, the output will be:

```
<html>
    <head>
        <meta attr='value'/><title>Title</title>
        <meta attr='value2'/>
    </head>
</html>
```

By default, the renderer uses four(4) spaces as indent, but you can change it by setting the `TemplateConfiguration#autoIndentString` property.

Automatic escaping

By default, contents which is read from the model is rendered **as is**. If this contents comes from user input, it can be sensible, and you might want to escape it by default, for example to avoid XSS injection. For that, the template configuration provides an option which will automatically escape objects from the model, as long as they inherit from `CharSequence` (typically, `String`s).

Let's imagine the following setup:

```
config.setAutoEscape(false);
model = new HashMap<String,Object>();
model.put("unsafeContents", "I am an <html> hacker.");
```

and the following template:

```
html {
    body {
        div(unsafeContents)
    }
}
```

Then you wouldn't want the HTML from `unsafeContents` to be rendered as is, because of potential security issues:

```
<html><body><div>I am an <html> hacker.</div></body></html>
```

Automatic escaping will fix this:

```
config.setAutoEscape(true);
```

And now the output is properly escaped:

```
<html><body><div>I am an &lt;html&gt; hacker.</div></body></html>
```

Note that using automatic escaping doesn't prevent you from including unescaped contents from the model. To do this, your template should then explicitly mention that a model variable should not be escaped by prefixing it with `unescaped.`, like in this example:

Explicit bypass of automatic escaping

```
html {
    body {
        div(unescaped.unsafeContents)
    }
}
```

Common gotchas

Strings containing markup

Say that you want to generate a `<p>` tag which contains a string containing markup:

```
p {
    yield "This is a "
    a(href:'target.html', "link")
    yield " to another page"
}
```

and generates:

```
<p>This is a <a href='target.html'>link</a> to another page</p>
```

Can't this be written shorter? A naive alternative would be:

```
p {
    yield "This is a ${a(href:'target.html', "link")} to another page"
}
```

but the result will not look as expected:

```
<p><a href='target.html'>link</a>This is a  to another page</p>
```

The reason is that the markup template engine is a *streaming* engine. In the original version, the first `yield` call generates a string which is streamed to the output, then the `a` link is generated and streamed, and then the last `yield` call is streamed, leading in an execution **in order**. But with the string version above, the order of execution is different:

- the `yield` call requires an argument, a *string*
- that arguments need to be evaluated *before* the *yield* call is generated

so evaluating the string leads to an execution of the `a(href:⋯)` call **before** `yield` is itself called. This is not what you want to do. Instead, you want to generate a *string* which contains markup, which is then passed to the `yield` call. This can be done this way:

```
p("This is a ${stringOf {a(href:'target.html', "link")}} to another page")
```

Note the `stringOf` call, which basically tells the markup template engine that the underlying markup needs to be rendered separately and exported as a string. Note that for simple expressions, `stringOf` can be replaced by an alternate tag notation that starts with a *dollar* sign:

```
p("This is a ${$a(href:'target.html', "link")} to another page")
```



It is worth noting that using `stringOf` or the special `$tag` notation triggers the creation of a distinct string writer which is then used to render the markup. It is

> slower than using the version with calls to `yield` which perform direct streaming of the markup instead.

Internationalization

The template engine has native support for internationalization. For that, when you create the `TemplateConfiguration`, you can provide a `Locale` which is the default locale to be used for templates. Each template may have different versions, one for each locale. The name of the template makes the difference:

- `file.tpl`: default template file
- `file_fr_FR.tpl`: french version of the template
- `file_en_US.tpl`: american english version of the template
- ...

When a template is rendered or included, then:

- if the template name or include name **explicitly** sets a locale, the **specific** version is included, or the default version if not found
- if the template name doesn't include a locale, the version for the `TemplateConfiguration` locale is used, or the default version if not found

For example, imagine the default locale is set to `Locale.ENGLISH` and that the main template includes:

Use an explicit locale in include

```
include template: 'locale_include_fr_FR.tpl'
```

then the template is rendered using the specific template:

Bypass the template configuration

```
Texte en français
```

Using an include without specifying a locale will make the template engine look for a template with the configured locale, and if not, fallback to the default, like here:

Don't use a locale in include

```
include template: 'locale_include.tpl'
```

Fallback to the default template

```
Default text
```

However, changing the default locale of the template engine to `Locale.FRANCE` will change the

output, because the template engine will now look for a file with the `fr_FR` locale:

Don't fall back to the default template because a locale specific template was found

```
Texte en français
```

This strategy lets you translate your templates one by one, by relying on default templates, for which no locale is set in the file name.

Custom template classes

By default, templates created inherit the `groovy.text.markup.BaseTemplate` class. It may be interesting for an application to provide a different template class, for example to provide additional helper methods which are aware of the application, or customized rendering primitives (for HTML, for example).

The template engine provides this ability by setting an alternative `baseTemplateClass` in the `TemplateConfiguration`:

```
config.setBaseTemplateClass(MyTemplate.class);
```

The custom base class has to extend `BaseClass` like in this example:

```
public abstract class MyTemplate extends BaseTemplate {
    private List<Module> modules
    public MyTemplate(
            final MarkupTemplateEngine templateEngine,
            final Map model,
            final Map<String, String> modelTypes,
            final TemplateConfiguration configuration) {
        super(templateEngine, model, modelTypes, configuration)
    }

    List<Module> getModules() {
        return modules
    }

    void setModules(final List<Module> modules) {
        this.modules = modules
    }

    boolean hasModule(String name) {
        modules?.any { it.name == name }
    }
}
```

This example shows a class which provides an additional method named `hasModule`, which can then be used directly in the template:

```
if (hasModule('foo')) {
    p 'Found module [foo]'
} else {
    p 'Module [foo] not found'
}
```

Type checked templates

Optional type checking

Even if templates are not type checked, they are statically compiled. This means that once the templates are compiled, performance should be very good. For some applications, it might be good to make sure that templates are valid before they are actually rendered. This means failing template compilation, for example, if a method on a model variable doesn't exist.

The `MarkupTemplateEngine` provides such a facility. Templates can be optionally type checked. For that, the developer must provide additional information at template creation time, which is the types of the variables found in the model. Imagine a model exposing a list of pages, where a page is defined as:

Page.groovy

```
public class Page {

    Long id
    String title
    String body
}
```

Then a list of pages can be exposed in the model, like this:

```
Page p = new Page();
p.setTitle("Sample page");
p.setBody("Page body");
List<Page> pages = new LinkedList<>();
pages.add(p);
model = new HashMap<String,Object>();
model.put("pages", pages);
```

A template can use it easily:

```
pages.each { page ->                    ①
    p("Page title: $page.title")        ②
    p(page.text)                        ③
}
```

① iterate on pages from the model

② `page.title` is valid

③ `page.text` is **not** (should be `page.body`)

Without type checking, the compilation of the template succeeds, because the template engine doesn't know about the model until a page is actually rendered. This means that the problem would only surface at runtime, once the page is rendered:

Runtime error

```
No such property: text
```

In some situations, this can be complicated to sort out or even notice. By declaring the type of the `pages` to the template engine, we're now capable of failing at compile time:

```
modelTypes = new HashMap<String,String>();                                  ①
modelTypes.put("pages", "List<Page>");                                      ②
Template template = engine.createTypeCheckedModelTemplate("main.tpl", modelTypes)   ③
```

① create a map which will hold the model types

② declare the type of the `pages` variables (note the use of a string for the type)

③ use `createTypeCheckedModelTemplate` instead of `createTemplate`

This time, when the template is compiled at the last line, an error occurs:

Template compilation time error

```
[Static type checking] - No such property: text for class: Page
```

This means that you don't need to wait for the page to be rendered to see an error. The use of `createTypeCheckedModelTemplate` is mandatory.

Alternative declaration of types

Alternatively, if the developer is also the one who writes the templates, it is possible to declare the types of the expected variables directly in the template. In this case, even if you call `createTemplate`, it will be type checked:

Inline declaration of types

```
modelTypes = {                          ①
    List<Page> pages                    ②
}

pages.each { page ->
    p("Page title: $page.title")
    p(page.text)
}
```

① types need to be declared in the `modelTypes` header

② declare one variable per object in the model

Performance of type checked templates

An additional interest of using type checked models is that performance should improve. By telling the type checker what are the expected types, you also let the compiler generate optimized code for that, so if you are looking for the best performance, consider using type checked templates.

Other solutions

Also, there are other templating solutions that can be used along with Groovy, such as FreeMarker, Velocity, StringTemplate and others.

Servlet support

The `groovy-servlet` module supports the following features:

- A `GroovyServlet` class let you use Groovy scripts as Servlets (called *Groovylets*).
- A `TemplateServlet` class which supports running Groovy's template engines to generate servlet content. A typical use case is to write Groovy Server Pages (GSPs) which are similar to Java Server Pages (JSPs) but embed Groovy code rather than Java code.
- An `AbstractHttpServlet` class for defining traditional Servlets.
- Helper classes like `ServletBinding` and `ServletCategory` which let you access some of the features offered by the `groovy-servlet` module in additional contexts.

From Groovy 6, `groovy-servlet` targets Jakarta EE 11 (Servlet 6.1). It is compatible with Jakarta EE 10 (Servlet 6.0) containers as it does not use Servlet 6.1-exclusive APIs. The `javax` classifier is available for pre-Jakarta containers (e.g. Tomcat 9, Jetty 9).

Groovlets

You can write (Java) Servlets in Groovy (called Groovlets).

This is supported by the `GroovyServlet` class. It will automatically compile your `.groovy` source files, turn them into bytecode, load the Class and cache it until you change the source file.

Here's a simple example to show you the kind of things you can do from a Groovlet.

Notice the use of implicit variables to access the `session`, `output` and `request`. Also notice that this is more like a script as the code isn't wrapped with a class definition.

```
if (!session) {
  session = request.getSession(true)
}

if (!session.counter) {
  session.counter = 1
```

```
}

println """
<html>
    <head>
        <title>Groovy Servlet</title>
    </head>
    <body>
        <p>
Hello, ${request.remoteHost}: ${session.counter}! ${new Date()}
        </p>
    </body>
</html>
"""
session.counter = session.counter + 1
```

Or, do the same thing using MarkupBuilder:

```
if (!session) {
    session = request.getSession(true)
}

if (!session.counter) {
    session.counter = 1
}

html.html { // html is implicitly bound to new MarkupBuilder(out)
  head {
      title('Groovy Servlet')
  }
  body {
    p("Hello, ${request.remoteHost}: ${session.counter}! ${new Date()}")
  }
}
session.counter = session.counter + 1
```

Implicit variables

The following variables are ready for use in Groovlets:

variable name	bound to	note
request	ServletRequest	-
response	ServletResponse	-
context	ServletContext	-
application	ServletContext	-
session	getSession(false)	can be null! see <1>

variable name	bound to	note
params		a Map object
headers		a Map object
out	response.getWriter()	see <2>
sout	response.getOutputStream()	see <2>
html	new MarkupBuilder(out)	see <2>
json	new StreamingJsonBuilder(out)	see <2>

1. The session variable is only set, if there was already a session object. See the `if (session == null)` checks in the examples above.
2. These variables cannot be re-assigned inside a `Groovlet`. They are bound on first access, allowing to e.g. calling methods on the `response` object before using `out`.

Setting up Groovlets

The traditional configuration approach for declaring new servlets is to add them to your Servlet engine configuration, e.g. you might add the following to your `web.xml`:

```
<servlet>
    <servlet-name>Groovy</servlet-name>
    <servlet-class>groovy.servlet.GroovyServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>Groovy</servlet-name>
    <url-pattern>*.groovy</url-pattern>
</servlet-mapping>
```

Then put the required groovy jar files into `WEB-INF/lib`.

Now put the .groovy files in, say, the root directory (i.e. where you would put your html files). The `GroovyServlet` takes care of compiling the .groovy files.

So for example using Apache Tomcat you could edit `tomcat/conf/server.xml` like this:

```
<Context path="/groovy" docBase="/path_to_servlet_base"/>
```

Then access it with http://localhost:8080/groovy/hello.groovy

Some Servlet engines let you define Servlets programmatically, e.g. here is how you might use Jetty with Groovlets enabled:

```
import groovy.servlet.GroovyServlet
import org.eclipse.jetty.server.Server
```

```
import org.eclipse.jetty.ee9.servlet.ServletContextHandler

var server = new Server(8080)
var handler = new ServletContextHandler(server, '/', ServletContextHandler.SESSIONS)
handler.baseResourceAsString = 'path_to_servlet_base'
handler.addServlet(GroovyServlet, '*.groovy')
server.start()
server.join()
```

Error responses

When a Groovlet fails, `GroovyServlet` returns a generic `500` response to the client by default and writes the full detail — the script path, the exception message and a stack frame — to the servlet and application logs. This keeps internal detail out of responses sent to callers.

While developing, you can restore the detailed client responses by setting the `groovy.servlet.verbose.errors` system property to `true` (for example, `-Dgroovy.servlet.verbose.errors=true`), or by overriding the `isVerboseErrors()` method in a `GroovyServlet` subclass.

Integrating Groovy in a Java application

Groovy integration mechanisms

The Groovy language proposes several ways to integrate itself into applications (Java or even Groovy) at runtime, from the most basic, simple code execution to the most complete, integrating caching and compiler customization.



All the examples written in this section are using Groovy, but the same integration mechanisms can be used from Java.

Eval

The `groovy.util.Eval` class is the simplest way to execute Groovy dynamically at runtime. This can be done by calling the `me` method:

```
import groovy.util.Eval

assert Eval.me('33*3') == 99
assert Eval.me('"foo".toUpperCase()') == 'FOO'
```

`Eval` supports multiple variants that accept parameters for simple evaluation:

```
assert Eval.x(4, '2*x') == 8                ①
assert Eval.me('k', 4, '2*k') == 8          ②
assert Eval.xy(4, 5, 'x*y') == 20           ③
```

```
assert Eval.xyz(4, 5, 6, 'x*y+z') == 26     ④
```

① Simple evaluation with one bound parameter named `x`

② Same evaluation, with a custom bound parameter named `k`

③ Simple evaluation with two bound parameters named `x` and `y`

④ Simple evaluation with three bound parameters named `x`, `y` and `z`

The `Eval` class makes it very easy to evaluate simple scripts, but doesn't scale: there is no caching of the script, and it isn't meant to evaluate more than one-liners.

GroovyShell

Multiple sources

The `groovy.lang.GroovyShell` class is the preferred way to evaluate scripts with the ability to cache the resulting script instance. Although the `Eval` class returns the result of the execution of the compiled script, the `GroovyShell` class offers more options.

```
def shell = new GroovyShell()                          ①
def result = shell.evaluate '3*5'                      ②
def result2 = shell.evaluate(new StringReader('3*5'))  ③
assert result == result2
def script = shell.parse '3*5'                         ④
assert script instanceof groovy.lang.Script
assert script.run() == 15                              ⑤
```

① create a new `GroovyShell` instance

② can be used as `Eval` with direct execution of the code

③ can read from multiple sources (`String`, `Reader`, `File`, `InputStream`)

④ can defer execution of the script. `parse` returns a `Script` instance

⑤ `Script` defines a `run` method

Sharing data between a script and the application

It is possible to share data between the application and the script using a `groovy.lang.Binding`:

```
def sharedData = new Binding()                         ①
def shell = new GroovyShell(sharedData)                ②
def now = new Date()
sharedData.setProperty('text', 'I am shared data!')    ③
sharedData.setProperty('date', now)                    ④

String result = shell.evaluate('"At $date, $text"')    ⑤

assert result == "At $now, I am shared data!"
```

① create a new `Binding` that will contain shared data

② create a `GroovyShell` using this shared data

③ add a string to the binding

④ add a date to the binding (you are not limited to simple types)

⑤ evaluate the script

Note that it is also possible to write from the script into the binding:

```
def sharedData = new Binding()                  ①
def shell = new GroovyShell(sharedData)         ②

shell.evaluate('foo=123')                       ③

assert sharedData.getProperty('foo') == 123     ④
```

① create a new `Binding` instance

② create a new `GroovyShell` using that shared data

③ use an **undeclared** variable to store the result into the binding

④ read the result from the caller

It is important to understand that you need to use an undeclared variable if you want to write into the binding. Using `def` or an `explicit` type like in the example below would fail because you would then create a *local variable*:

```
def sharedData = new Binding()
def shell = new GroovyShell(sharedData)

shell.evaluate('int foo=123')

try {
    assert sharedData.getProperty('foo')
} catch (MissingPropertyException e) {
    println "foo is defined as a local variable"
}
```



You must be very careful when using shared data in a multithreaded environment. The `Binding` instance that you pass to `GroovyShell` is **not** thread safe, and shared by all scripts.

It is possible to work around the shared instance of `Binding` by leveraging the `Script` instance which is returned by `parse`:

```
def shell = new GroovyShell()
```

```
def b1 = new Binding(x:3)                       ①
def b2 = new Binding(x:4)                       ②
def script = shell.parse('x = 2*x')
script.binding = b1
script.run()
script.binding = b2
script.run()
assert b1.getProperty('x') == 6
assert b2.getProperty('x') == 8
assert b1 != b2
```

① will store the x variable inside b1

② will store the x variable inside b2

However, you must be aware that you are still sharing the **same instance** of a script. So this technique cannot be used if you have two threads working on the same script. In that case, you must make sure of creating two distinct script instances:

```
def shell = new GroovyShell()

def b1 = new Binding(x:3)
def b2 = new Binding(x:4)
def script1 = shell.parse('x = 2*x')            ①
def script2 = shell.parse('x = 2*x')            ②
assert script1 != script2
script1.binding = b1                            ③
script2.binding = b2                            ④
def t1 = Thread.start { script1.run() }         ⑤
def t2 = Thread.start { script2.run() }         ⑥
[t1,t2]*.join()                                 ⑦
assert b1.getProperty('x') == 6
assert b2.getProperty('x') == 8
assert b1 != b2
```

① create an instance of script for thread 1

② create an instance of script for thread 2

③ assign first binding to script 1

④ assign second binding to script 2

⑤ start first script in a separate thread

⑥ start second script in a separate thread

⑦ wait for completion

In case you need thread safety like here, it is more advisable to use the GroovyClassLoader directly instead.

Custom script class

We have seen that the `parse` method returns an instance of `groovy.lang.Script`, but it is possible to use a custom class, given that it extends `Script` itself. It can be used to provide additional behavior to the script like in the example below:

```
abstract class MyScript extends Script {
    String name

    String greet() {
        "Hello, $name!"
    }
}
```

The custom class defines a property called `name` and a new method called `greet`. This class can be used as the script base class by using a custom configuration:

```
import org.codehaus.groovy.control.CompilerConfiguration

def config = new CompilerConfiguration()                                    ①
config.scriptBaseClass = 'MyScript'                                         ②

def shell = new GroovyShell(this.class.classLoader, new Binding(), config)  ③
def script = shell.parse('greet()')                                         ④
assert script instanceof MyScript
script.setName('Michel')
assert script.run() == 'Hello, Michel!'
```

① create a `CompilerConfiguration` instance

② instruct it to use `MyScript` as the base class for scripts

③ then use the compiler configuration when you create the shell

④ the script now has access to the new method `greet`



You are not limited to the sole *scriptBaseClass* configuration. You can use any of the compiler configuration tweaks, including the compilation customizers.

GroovyClassLoader

In the previous section, we have shown that `GroovyShell` was an easy tool to execute scripts, but it makes it complicated to compile anything but scripts. Internally, it makes use of the `groovy.lang.GroovyClassLoader`, which is at the heart of the compilation and loading of classes at runtime.

By leveraging the `GroovyClassLoader` instead of `GroovyShell`, you will be able to load classes, instead of instances of scripts:

```
import groovy.lang.GroovyClassLoader
```

```
def gcl = new GroovyClassLoader()                                          ①
def clazz = gcl.parseClass('class Foo { void doIt() { println "ok" } }')   ②
assert clazz.name == 'Foo'                                                 ③
def o = clazz.newInstance()                                                ④
o.doIt()                                                                   ⑤
```

① create a new `GroovyClassLoader`

② `parseClass` will return an instance of `Class`

③ you can check that the class which is returns is really the one defined in the script

④ and you can create a new instance of the class, which is not a script

⑤ then call any method on it



A GroovyClassLoader keeps a reference of all the classes it created, so it is easy to create a memory leak. In particular, if you execute the same script twice, if it is a String, then you obtain two distinct classes!

```
import groovy.lang.GroovyClassLoader
import org.junit.jupiter.api.Test
import static groovy.test.GroovyAssert.*

def gcl = new GroovyClassLoader()
def clazz1 = gcl.parseClass('class Foo { }')                               ①
def clazz2 = gcl.parseClass('class Foo { }')                               ②
assert clazz1.name == 'Foo'                                                ③
assert clazz2.name == 'Foo'
assert clazz1 != clazz2                                                    ④
```

① dynamically create a class named "Foo"

② create an identical looking class, using a separate `parseClass` call

③ make sure both classes have the same name

④ but they are actually different!

The reason is that a `GroovyClassLoader` doesn't keep track of the source text. If you want to have the same instance, then the source **must** be a file, like in this example:

```
def gcl = new GroovyClassLoader()
def clazz1 = gcl.parseClass(file)                                          ①
def clazz2 = gcl.parseClass(new File(file.absolutePath))                   ②
assert clazz1.name == 'Foo'                                                ③
assert clazz2.name == 'Foo'
assert clazz1 == clazz2                                                    ④
```

① parse a class from a `File`

② parse a class from a distinct file instance, but pointing to the same physical file

③ make sure our classes have the same name

④ but now, they are the same instance

Using a `File` as input, the `GroovyClassLoader` is capable of **caching** the generated class file, which avoids creating multiple classes at runtime for the same source.

GroovyScriptEngine

The `groovy.util.GroovyScriptEngine` class provides a flexible foundation for applications which rely on script reloading and script dependencies. While `GroovyShell` focuses on standalone `Script`s and `GroovyClassLoader` handles dynamic compilation and loading of any Groovy class, the `GroovyScriptEngine` will add a layer on top of `GroovyClassLoader` to handle both script dependencies and reloading.

To illustrate this, we will create a script engine and execute code in an infinite loop. First of all, you need to create a directory with the following script inside:

ReloadingTest.groovy

```
class Greeter {
    String sayHello() {
        def greet = "Hello, world!"
        greet
    }
}

new Greeter()
```

then you can execute this code using a `GroovyScriptEngine`:

```
def binding = new Binding()
def engine = new GroovyScriptEngine([tmpDir.toURI().toURL()] as URL[])          ①

while (true) {
    def greeter = engine.run('ReloadingTest.groovy', binding)                   ②
    println greeter.sayHello()                                                  ③
    Thread.sleep(1000)
}
```

① create a script engine which will look for sources into our source directory

② execute the script, which will return an instance of `Greeter`

③ print the greeting message

At this point, you should see a message printed every second:

```
Hello, world!
```

```
Hello, world!
...
```

Without interrupting the script execution, now replace the contents of the `ReloadingTest` file with:

ReloadingTest.groovy

```
class Greeter {
    String sayHello() {
        def greet = "Hello, Groovy!"
        greet
    }
}

new Greeter()
```

And the message should change to:

```
Hello, world!
...
Hello, Groovy!
Hello, Groovy!
...
```

But it is also possible to have a dependency on another script. To illustrate this, create the following file into the same directory, without interrupting the executing script:

Dependency.groovy

```
class Dependency {
    String message = 'Hello, dependency 1'
}
```

and update the `ReloadingTest` script like this:

ReloadingTest.groovy

```
import Dependency

class Greeter {
    String sayHello() {
        def greet = new Dependency().message
        greet
    }
}

new Greeter()
```

And this time, the message should change to:

```
Hello, Groovy!
...
Hello, dependency 1!
Hello, dependency 1!
...
```

And as a last test, you can update the `Dependency.groovy` file without touching the `ReloadingTest` file:

Dependency.groovy

```
class Dependency {
    String message = 'Hello, dependency 2'
}
```

And you should observe that the dependent file was reloaded:

```
Hello, dependency 1!
...
Hello, dependency 2!
Hello, dependency 2!
```

JavaShell

While `GroovyShell` and `GroovyClassLoader` compile Groovy source, `org.apache.groovy.util.JavaShell` compiles *Java* source in-memory using the platform `javax.tools.JavaCompiler`. It is useful when a Groovy or polyglot application needs to compile and load Java classes at runtime: code generation, plug-in loaders, scripting-style Java endpoints, or mixed-language testing setups.



`JavaShell` requires a JDK at runtime, not just a JRE — `ToolProvider.getSystemJavaCompiler()` returns `null` on a JRE.

The simplest usage compiles a single source string and returns the loaded `Class`:

```
import org.apache.groovy.util.JavaShell

def js = new JavaShell()                                                    ①
def src = """
    package demo;
    public class Foo {
        public static String greet(String who) { return "hello, " + who; }
    }
"""
Class<?> foo = js.compile('demo.Foo', src)                                  ②
assert foo.getDeclaredMethod('greet', String).invoke(null, 'world') == 'hello, world'
```

```
③
```

① create a new `JavaShell` (an optional `ClassLoader` argument sets the parent loader)

② compile the source; `className` is the fully-qualified binary name and must match the `package` declaration — see the `JavaShell` javadoc for the full naming rules

③ invoke the compiled method reflectively

`compileAll` returns *every* class produced from the source unit — useful when the source declares auxiliary or nested classes:

```
import org.apache.groovy.util.JavaShell

def js = new JavaShell()
def src = """
    package demo;
    public class Box { public static int hidden() { return Helper.value(); } }
    class Helper { static int value() { return 42; } }
"""
Map<String, Class<?>> classes = js.compileAll('demo.Box', src)              ①
assert classes.keySet() == ['demo.Box', 'demo.Helper'] as Set               ②
assert classes['demo.Box'].getDeclaredMethod('hidden').invoke(null) == 42
```

① compile a primary public class together with its package-private helper

② the returned map is keyed by binary class name

To additionally persist the generated class files to disk (for tools that consume `.class` files or to seed a classpath outside the JVM), use `compileAllTo`. It accepts a `Path` output directory and writes each class to its conventional package subdirectory:

```
import org.apache.groovy.util.JavaShell
import java.nio.file.Files

def out = Files.createTempDirectory('javashell-')
try {
    def js = new JavaShell()
    def src = """
        package demo;
        public class Deep { public static String tag(String label) { return label; } }
    """
    def written = js.compileAllTo('demo.Deep', ['-parameters'], src, out)   ①
    def classFile = out.resolve('demo/Deep.class')                          ②
    assert written['demo.Deep'] == classFile && Files.exists(classFile)
    // classes also remain available through js.classLoader, same as compileAll
    assert js.classLoader.loadClass('demo.Deep').getDeclaredMethod('tag', String
).invoke(null, 'x') == 'x'
} finally {
    out.toFile().deleteDir()
```

```
}
```

① compile and write to disk in one call; the second argument passes `javac` options (here `-parameters` to retain formal parameter names)

② packaged classes land under `<outputDir>/<package as path>/<ClassName>.class`; default-package classes land directly under `outputDir`

The compiled classes remain available through `js.getClassLoader()` after `compileAllTo` returns, so the same instance covers both "compile to memory" and "compile to memory and disk".

When the source fails to compile, `JavaShell` throws `JavaShellCompilationException` carrying the `javac` diagnostics. See the `JavaShell` javadoc for the full set of overloads (with/without an `Iterable<String>` options argument) and the expected `className`/`options` formats.



`compileAllTo` and its no-options overload are marked `@Incubating` — their signatures may evolve in a subsequent 6.x release based on usage feedback.

CompilationUnit

Ultimately, it is possible to perform more operations during compilation by relying directly on the `org.codehaus.groovy.control.CompilationUnit` class. This class is responsible for determining the various steps of compilation and would let you introduce new steps or even stop compilation at various phases. This is for example how stub generation is done, for the joint compiler.

However, overriding `CompilationUnit` is not recommended and should only be done if no other standard solution works.

Class loader release in managed environments

Applications that load and discard Groovy repeatedly — a web container performing (parallel) redeployments with Groovy inside each web application, plugin systems, or any host that expects class loaders to be garbage collected — need one extra consideration.

Groovy associates runtime metadata (`ClassInfo`) with every class it dispatches on, including JDK classes such as `String`. By default those associations use `java.lang.ClassValue`, which gives the fastest possible lookup. A `ClassValue` association lives exactly as long as the key class does — by design (see JDK-8136353, resolved as working as intended) — so an association on a long-lived class such as `String` never releases its value, and with it holds the Groovy class loader the value belongs to. In a container this shows up as metaspace growth on every redeployment, even after the old application is undeployed.

Three supported measures release the class loader; any one suffices:

- Start the JVM with `-Dgroovy.use.classvalue=soft`. Groovy keeps the `ClassValue` per-class fast path but holds each association through a `SoftReference`, so immortal JDK classes no longer hold a strong chain to the Groovy class loader. Metadata carrying state that cannot be rebuilt — an installed or per-instance metaclass, registry-written extension methods — is strong-rooted while in use, and associations cleared under memory pressure are rebuilt on demand, with call-site guards remaining sound for `invokedynamic` and classic call sites alike. This preserves default-

mode dispatch performance while letting the collector reclaim the loader.

- Start the JVM with `-Dgroovy.use.classvalue=false`. Groovy then stores the associations in a weak-key map instead of `ClassValue`. The trade-off is a hash lookup where `ClassValue` has a per-class fast path. Compilation speed is unaffected (within 1% in our measurements), and for applications that are not dispatch-heavy the difference is unlikely to be noticed; heavily dynamic code, however, can see a few percent overall, with hot dynamic idioms on JDK-class receivers (string coercions, ranges, safe navigation, map property access) up to roughly 1.4 times slower. Measure with your own workload if dispatch performance matters.
- Clean up explicitly when the application is discarded (for example from `ServletContextListener#contextDestroyed`):

```
import org.codehaus.groovy.reflection.ClassInfo

for (ClassInfo ci : ClassInfo.getAllClassInfo()) {
    Class<?> c = ci.theClass
    if (c != null) ClassInfo.remove(c)
}
```



Groovy itself starts no non-terminating background threads, so no thread of Groovy's will pin the class loader. Threads started by user code (including via `Thread.start` from scripts, or executors created by scripts) capture the creating context and must be shut down by the application as usual.

JSR 223 javax.script API



JSR-223 is a standard API for calling scripting frameworks in Java. It is available since Java 6 and aims at providing a common framework for calling multiple languages from Java. Groovy provides its own richer integration mechanisms, and if you don't plan to use multiple languages in the same application, it is recommended that you use the Groovy integration mechanisms instead of the limited JSR-223 API.

Here is how you need to initialize the JSR-223 engine to talk to Groovy from Java:

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;
...
ScriptEngineManager factory = new ScriptEngineManager();
ScriptEngine engine = factory.getEngineByName("groovy");
```

Then you can execute Groovy scripts easily:

```
Integer sum = (Integer) engine.eval("(1..10).sum()");
```

```
assertEquals(Integer.valueOf(55), sum);
```

It is also possible to share variables:

```
engine.put("first", "HELLO");
engine.put("second", "world");
String result = (String) engine.eval("first.toLowerCase() + ' ' +
second.toUpperCase()");
assertEquals("hello WORLD", result);
```

This next example illustrates calling an invokable function:

```
import javax.script.Invocable;
...
ScriptEngineManager factory = new ScriptEngineManager();
ScriptEngine engine = factory.getEngineByName("groovy");
String fact = "def factorial(n) { n == 1 ? 1 : n * factorial(n - 1) }";
engine.eval(fact);
Invocable inv = (Invocable) engine;
Object[] params = {5};
Object result = inv.invokeFunction("factorial", params);
assertEquals(Integer.valueOf(120), result);
```

The engine keeps per default hard references to the script functions. To change this you should set an engine level scoped attribute to the script context of the name `#jsr223.groovy.engine.keep.globals` with a String being `phantom` to use phantom references, `weak` to use weak references or `soft` to use soft references - casing is ignored. Any other string will cause the use of hard references.

Domain-Specific Languages

Command chains

Groovy lets you omit parentheses around the arguments of a method call for top-level statements. "command chain" feature extends this by allowing us to chain such parentheses-free method calls, requiring neither parentheses around arguments, nor dots between the chained calls. The general idea is that a call like `a b c d` will actually be equivalent to `a(b).c(d)`. This also works with multiple arguments, closure arguments, and even named arguments. Furthermore, such command chains can also appear on the right-hand side of assignments. Let's have a look at some examples supported by this new syntax:

```
// equivalent to: turn(left).then(right)
turn left then right

// equivalent to: take(2.pills).of(chloroquinine).after(6.hours)
take 2.pills of chloroquinine after 6.hours
```

```
// equivalent to: paint(wall).with(red, green).and(yellow)
paint wall with red, green and yellow

// with named parameters too
// equivalent to: check(that: margarita).tastes(good)
check that: margarita tastes good

// with closures as parameters
// equivalent to: given({}).when({}).then({})
given { } when { } then { }
```

It is also possible to use methods in the chain which take no arguments, but in that case, the parentheses are needed:

```
// equivalent to: select(all).unique().from(names)
select all unique() from names
```

If your command chain contains an odd number of elements, the chain will be composed of method / arguments, and will finish by a final property access:

```
// equivalent to: take(3).cookies
// and also this: take(3).getCookies()
take 3 cookies
```

This command chain approach opens up interesting possibilities in terms of the much wider range of DSLs which can now be written in Groovy.

The above examples illustrate using a command chain based DSL but not how to create one. There are various strategies that you can use, but to illustrate creating such a DSL, we will show a couple of examples - first using maps and Closures:

```
show = { println it }
square_root = { Math.sqrt(it) }

def please(action) {
  [the: { what ->
    [of: { n -> action(what(n)) }]
  }]
}

// equivalent to: please(show).the(square_root).of(100)
please show the square_root of 100
// ==> 10.0
```

As a second example, consider how you might write a DSL for simplifying one of your existing APIs.

Maybe you need to put this code in front of customers, business analysts or testers who might be not hard-core Java developers. We'll use the `Splitter` from the Google Guava libraries project as it already has a nice Fluent API. Here is how we might use it out of the box:

```
@Grab('com.google.guava:guava:r09')
import com.google.common.base.*
def result = Splitter.on(',').trimResults(CharMatcher.is('_' as char)).split("_a ,_b_
,c__").iterator().toList()
```

It reads fairly well for a Java developer but if that is not your target audience or you have many such statements to write, it could be considered a little verbose. Again, there are many options for writing a DSL. We'll keep it simple with Maps and Closures. We'll first write a helper method:

```
@Grab('com.google.guava:guava:r09')
import com.google.common.base.*
def split(string) {
  [on: { sep ->
    [trimming: { trimChar ->
      Splitter.on(sep).trimResults(CharMatcher.is(trimChar as char)).split(string
).iterator().toList()
    }]
  }]
}
```

now instead of this line from our original example:

```
def result = Splitter.on(',').trimResults(CharMatcher.is('_' as char)).split("_a ,_b_
,c__").iterator().toList()
```

we can write this:

```
def result = split "_a ,_b_ ,c__" on ',' trimming '_\'
```

Operator overloading

Various operators in Groovy are mapped onto regular method calls on objects.

This allows you to provide your own Java or Groovy objects which can take advantage of operator overloading. The following table describes the operators supported in Groovy and the methods they map to.

Operator	Method
`a + b`	a.plus(b)
`a - b`	a.minus(b)

Operator	Method
`a * b`	a.multiply(b)
`a ** b`	a.power(b)
`a / b`	a.div(b)
`a % b`	a.mod(b)
`a | b`	a.or(b)
`a & b`	a.and(b)
`a ^ b`	a.xor(b)
`a++` or `++a`	a.next()
`a--` or `--a`	a.previous()
`a[b]`	a.getAt(b)
`a[b] = c`	a.putAt(b, c)
`a << b`	a.leftShift(b)
`a >> b`	a.rightShift(b)
`a >>> b`	a.rightShiftUnsigned(b)
`switch(a) { case(b) : }`	b.isCase(a)
`if(a)`	a.asBoolean()
`~a`	a.bitwiseNegate()
`-a`	a.negative()
`+a`	a.positive()
`a as b`	a.asType(b)
`a == b`	a.equals(b)
`a != b`	! a.equals(b)
`a <=> b`	a.compareTo(b)
`a > b`	a.compareTo(b) > 0
`a >= b`	a.compareTo(b) >= 0
`a \< b`	a.compareTo(b) < 0
`a <= b`	a.compareTo(b) <= 0

Script base classes

The Script class

Groovy scripts are always compiled to classes. For example, a script as simple as:

```
println 'Hello from Groovy'
```

is compiled to a class extending the abstract groovy.lang.Script class. This class contains a single abstract method called *run*. When a script is compiled, then its body will become the *run* method, while the other methods found in the script are found in the implementing class. The `Script` class provides base support for integration with your application through the `Binding` object, as illustrated in this example:

```
def binding = new Binding()             ①
def shell = new GroovyShell(binding)    ②
binding.setVariable('x',1)              ③
binding.setVariable('y',3)
shell.evaluate 'z=2*x+y'                ④
assert binding.getVariable('z') == 5    ⑤
```

① a binding is used to share data between the script and the calling class

② a `GroovyShell` can be used with this binding

③ input variables are set from the calling class inside the binding

④ then the script is evaluated

⑤ and the `z` variable has been "exported" into the binding

This is a very practical way to share data between the caller and the script, however it may be insufficient or not practical in some cases. For that purpose, Groovy allows you to set your own base script class. A base script class has to extend groovy.lang.Script and be a single abstract method type:

```
abstract class MyBaseClass extends Script {
    String name
    public void greet() { println "Hello, $name!" }
}
```

Then the custom script base class can be declared in the compiler configuration, for example:

```
def config = new CompilerConfiguration()                                ①
config.scriptBaseClass = 'MyBaseClass'                                  ②
def shell = new GroovyShell(this.class.classLoader, config)             ③
shell.evaluate """
    setName 'Judith'                                                    ④
    greet()
"""
```

① create a custom compiler configuration

② set the base script class to our custom base script class

③ then create a `GroovyShell` using that configuration

④ the script will then extend the base script class, giving direct access to the `name` property and `greet` method

The @BaseScript annotation

As an alternative, it is also possible to use the `@BaseScript` annotation directly into a script:

```
import groovy.transform.BaseScript

@BaseScript MyBaseClass baseScript
setName 'Judith'
greet()
```

where `@BaseScript` should annotate a variable which type is the class of the base script. Alternatively, you can set the base script class as a member of the `@BaseScript` annotation itself:

```
@BaseScript(MyBaseClass)
import groovy.transform.BaseScript

setName 'Judith'
greet()
```

When using the special no-args `run` method, you can even annotate that method as shown here:

```
abstract class CustomScript extends Script {
    int getTheMeaningOfLife() { 42 }
}

@BaseScript(CustomScript)
def run() {
    assert theMeaningOfLife == 42
}
```

Alternate abstract method

We have seen that the base script class is a single abstract method type that needs to implement the `run` method. The `run` method is executed by the script engine automatically. In some circumstances it may be interesting to have a base class which implements the `run` method, but provides an alternative abstract method to be used for the script body. For example, the base script `run` method might perform some initialization before the `run` method is executed. This is possible by doing this:

```
abstract class MyBaseClass extends Script {
    int count
    abstract void scriptBody()                          ①
    def run() {
        count++                                         ②
        scriptBody()                                    ③
        count                                           ④
    }
```

```
}
```

① the base script class should define one (and only one) abstract method

② the `run` method can be overridden and perform a task before executing the script body

③ `run` calls the abstract `scriptBody` method which will delegate to the user script

④ then it can return something else than the value from the script

If you execute this code:

```
def result = shell.evaluate """
    println 'Ok'
"""
assert result == 1
```

Then you will see that the script is executed, but the result of the evaluation is `1` as returned by the `run` method of the base class. It is even clearer if you use `parse` instead of `evaluate`, because it would allow you to execute the `run` method several times on the same script instance:

```
def script = shell.parse("println 'Ok'")
assert script.run() == 1
assert script.run() == 2
```

Adding properties to numbers

In Groovy number types are considered equal to any other types. As such, it is possible to enhance numbers by adding properties or methods to them. This can be very handy when dealing with measurable quantities for example. Details about how existing classes can be enhanced in Groovy are found in the extension modules section or the categories section.

An illustration of this can be found in Groovy using the `TimeCategory`:

```
use(TimeCategory) {
    println 1.minute.from.now       ①
    println 10.hours.ago

    def someDate = new Date()       ②
    println someDate - 3.months
}
```

① using the `TimeCategory`, a property `minute` is added to the `Integer` class

② similarly, the `months` method returns a `groovy.time.DatumDependentDuration` which can be used in calculus

Categories are lexically bound, making them a great fit for internal DSLs.

@DelegatesTo

Explaining delegation strategy at compile time

`@groovy.lang.DelegatesTo` is a documentation and compile-time annotation aimed at:

- documenting APIs that use closures as arguments
- providing type information for the static type checker and compiler

The Groovy language is a platform of choice for building DSLs. Using closures, it's quite easy to create custom control structures, as well as it is simple to create builders. Imagine that you have the following code:

```
email {
    from 'dsl-guru@mycompany.com'
    to 'john.doe@waitaminute.com'
    subject 'The pope has resigned!'
    body {
        p 'Really, the pope has resigned!'
    }
}
```

One way of implementing this is using the builder strategy, which implies a method, named `email` which accepts a closure as an argument. The method may delegate subsequent calls to an object that implements the `from`, `to`, `subject` and `body` methods. Again, `body` is a method which accepts a closure as an argument and that uses the builder strategy.

Implementing such a builder is usually done the following way:

```
def email(Closure cl) {
    def email = new EmailSpec()
    def code = cl.rehydrate(email, this, this)
    code.resolveStrategy = Closure.DELEGATE_ONLY
    code()
}
```

the `EmailSpec` class implements the `from`, `to`, ... methods. By calling `rehydrate`, we're creating a copy of the closure for which we set the `delegate`, `owner` and `thisObject` values. Setting the owner and the `this` object is not very important here since we will use the `DELEGATE_ONLY` strategy which says that the method calls will be resolved only against the delegate of the closure.

```
class EmailSpec {
    void from(String from) { println "From: $from"}
    void to(String... to) { println "To: $to"}
    void subject(String subject) { println "Subject: $subject"}
    void body(Closure body) {
        def bodySpec = new BodySpec()
```

```
        def code = body.rehydrate(bodySpec, this, this)
        code.resolveStrategy = Closure.DELEGATE_ONLY
        code()
    }
}
```

The `EmailSpec` class has itself a `body` method accepting a closure that is cloned and executed. This is what we call the builder pattern in Groovy.

One of the problems with the code that we've shown is that the user of the `email` method doesn't have any information about the methods that he's allowed to call inside the closure. The only possible information is from the method documentation. There are two issues with this: first of all, documentation is not always written, and if it is, it's not always available (javadoc not downloaded, for example). Second, it doesn't help IDEs. What would be really interesting, here, is for IDEs to help the developer by suggesting, once they are in the closure body, methods that exist on the `email` class.

Moreover, if the user calls a method in the closure which is not defined by the `EmailSpec` class, the IDE should at least issue a warning (because it's very likely that it will break at runtime).

One more problem with the code above is that it is not compatible with static type checking. Type checking would let the user know if a method call is authorized at compile time instead of runtime, but if you try to perform type checking on this code:

```
email {
    from 'dsl-guru@mycompany.com'
    to 'john.doe@waitaminute.com'
    subject 'The pope has resigned!'
    body {
        p 'Really, the pope has resigned!'
    }
}
```

Then the type checker will know that there's an `email` method accepting a `Closure`, but it will complain about every method call **inside** the closure, because `from`, for example, is not a method which is defined in the class. Indeed, it's defined in the `EmailSpec` class and it has absolutely no hint to help it knowing that the closure delegate will, at runtime, be of type `EmailSpec`:

```
@groovy.transform.TypeChecked
void sendEmail() {
    email {
        from 'dsl-guru@mycompany.com'
        to 'john.doe@waitaminute.com'
        subject 'The pope has resigned!'
        body {
            p 'Really, the pope has resigned!'
        }
    }
```

```
}
```

will fail compilation with errors like this one:

```
[Static type checking] - Cannot find matching method MyScript#from(java.lang.String).
Please check if the declared type is correct and if the method exists.
 @ line 31, column 21.
                       from 'dsl-guru@mycompany.com'
```

@DelegatesTo

For those reasons, Groovy 2.1 introduced a new annotation named `@DelegatesTo`. The goal of this annotation is to solve both the documentation issue, that will let your IDE know about the expected methods in the closure body, and it will also solve the type checking issue, by giving hints to the compiler about what are the potential receivers of method calls in the closure body.

The idea is to annotate the `Closure` parameter of the `email` method:

```
def email(@DelegatesTo(EmailSpec) Closure cl) {
    def email = new EmailSpec()
    def code = cl.rehydrate(email, this, this)
    code.resolveStrategy = Closure.DELEGATE_ONLY
    code()
}
```

What we've done here is telling the compiler (or the IDE) that when the method will be called with a closure, the delegate of this closure will be set to an object of type `email`. But there is still a problem: the default delegation strategy is not the one which is used in our method. So we will give more information and tell the compiler (or the IDE) that the delegation strategy is also changed:

```
def email(@DelegatesTo(strategy=Closure.DELEGATE_ONLY, value=EmailSpec) Closure cl) {
    def email = new EmailSpec()
    def code = cl.rehydrate(email, this, this)
    code.resolveStrategy = Closure.DELEGATE_ONLY
    code()
}
```

Now, both the IDE and the type checker (if you are using `@TypeChecked`) will be aware of the delegate and the delegation strategy. This is very nice because it will both allow the IDE to provide smart completion, but it will also remove errors at compile time that exist only because the behaviour of the program is normally only known at runtime!

The following code will now pass compilation:

```
@TypeChecked
void doEmail() {
```

```
    email {
        from 'dsl-guru@mycompany.com'
        to 'john.doe@waitaminute.com'
        subject 'The pope has resigned!'
        body {
            p 'Really, the pope has resigned!'
        }
    }
}
```

DelegatesTo modes

`@DelegatesTo` supports multiple modes that we will describe with examples in this section.

Simple delegation

In this mode, the only mandatory parameter is the *value* which says to which class we delegate calls. Nothing more. We're telling the compiler that the type of the delegate will **always** be of the type documented by `@DelegatesTo` (note that it can be a subclass, but if it is, the methods defined by the subclass will not be visible to the type checker).

```
void body(@DelegatesTo(BodySpec) Closure cl) {
    // ...
}
```

Delegation strategy

In this mode, you must specify both the delegate class **and** a delegation strategy. This must be used if the closure will not be called with the default delegation strategy, which is `Closure.OWNER_FIRST`.

```
void body(@DelegatesTo(strategy=Closure.DELEGATE_ONLY, value=BodySpec) Closure cl) {
    // ...
}
```

Delegate to parameter

In this variant, we will tell the compiler that we are delegating to another parameter of the method. Take the following code:

```
def exec(Object target, Closure code) {
   def clone = code.rehydrate(target, this, this)
   clone()
}
```

Here, the delegate which will be used is **not** created inside the `exec` method. In fact, we take an argument of the method and delegate to it. Usage may look like this:

```
def email = new Email()
exec(email) {
   from '...'
   to '...'
   send()
}
```

Each of the method calls are delegated to the `email` parameter. This is a widely used pattern which is also supported by `@DelegatesTo` using a companion annotation:

```
def exec(@DelegatesTo.Target Object target, @DelegatesTo Closure code) {
   def clone = code.rehydrate(target, this, this)
   clone()
}
```

A closure is annotated with `@DelegatesTo`, but this time, without specifying any class. Instead, we're annotating another parameter with `@DelegatesTo.Target`. The type of the delegate is then determined at compile time. One could think that we are using the parameter type, which in this case is `Object` but this is not true. Take this code:

```
class Greeter {
   void sayHello() { println 'Hello' }
}
def greeter = new Greeter()
exec(greeter) {
   sayHello()
}
```

Remember that this works out of the box **without** having to annotate with `@DelegatesTo`. However, to make the IDE aware of the delegate type, or the **type checker** aware of it, we need to add `@DelegatesTo`. And in this case, it will know that the `Greeter` variable is of type `Greeter`, so it will not report errors on the *sayHello* method **even if the exec method doesn't explicitly define the target as of type Greeter**. This is a very powerful feature, because it prevents you from writing multiple versions of the same `exec` method for different receiver types!

In this mode, the `@DelegatesTo` annotation also supports the `strategy` parameter that we've described upper.

Multiple closures

In the previous example, the `exec` method accepted only one closure, but you may have methods that take multiple closures:

```
void fooBarBaz(Closure foo, Closure bar, Closure baz) {
    ...
```

```
}
```

Then nothing prevents you from annotating each closure with `@DelegatesTo`:

```
class Foo { void foo(String msg) { println "Foo ${msg}!" } }
class Bar { void bar(int x) { println "Bar ${x}!" } }
class Baz { void baz(Date d) { println "Baz ${d}!" } }

void fooBarBaz(@DelegatesTo(Foo) Closure foo, @DelegatesTo(Bar) Closure bar,
@DelegatesTo(Baz) Closure baz) {
   ...
}
```

But more importantly, if you have multiple closures **and** multiple arguments, you can use several targets:

```
void fooBarBaz(
    @DelegatesTo.Target('foo') foo,
    @DelegatesTo.Target('bar') bar,
    @DelegatesTo.Target('baz') baz,

    @DelegatesTo(target='foo') Closure cl1,
    @DelegatesTo(target='bar') Closure cl2,
    @DelegatesTo(target='baz') Closure cl3) {
    cl1.rehydrate(foo, this, this).call()
    cl2.rehydrate(bar, this, this).call()
    cl3.rehydrate(baz, this, this).call()
}

def a = new Foo()
def b = new Bar()
def c = new Baz()
fooBarBaz(
    a, b, c,
    { foo('Hello') },
    { bar(123) },
    { baz(new Date()) }
)
```



At this point, you may wonder why we don't use the parameter names as references. The reason is that the information (the parameter name) is not always available (it's a debug-only information), so it's a limitation of the JVM.

Delegating to a generic type

In some situations, it is interesting to instruct the IDE or the compiler that the delegate type will not be a parameter but a generic type. Imagine a configurator that runs on a list of elements:

```
public <T> void configure(List<T> elements, Closure configuration) {
   elements.each { e->
      def clone = configuration.rehydrate(e, this, this)
      clone.resolveStrategy = Closure.DELEGATE_FIRST
      clone.call()
   }
}
```

Then this method can be called with any list like this:

```
@groovy.transform.ToString
class Realm {
   String name
}
List<Realm> list = []
3.times { list << new Realm() }
configure(list) {
   name = 'My Realm'
}
assert list.every { it.name == 'My Realm' }
```

To let the type checker and the IDE know that the `configure` method calls the closure on each element of the list, you need to use `@DelegatesTo` differently:

```
public <T> void configure(
    @DelegatesTo.Target List<T> elements,
    @DelegatesTo(strategy=Closure.DELEGATE_FIRST, genericTypeIndex=0) Closure
configuration) {
   def clone = configuration.rehydrate(e, this, this)
   clone.resolveStrategy = Closure.DELEGATE_FIRST
   clone.call()
}
```

`@DelegatesTo` takes an optional `genericTypeIndex` argument that tells what is the index of the generic type that will be used as the delegate type. This **must** be used in conjunction with `@DelegatesTo.Target` and the index starts at 0. In the example above, that means that the delegate type is resolved against `List<T>`, and since the generic type at index 0 is `T` and inferred as a `Realm`, the type checker infers that the delegate type will be of type `Realm`.



We're using a `genericTypeIndex` instead of a placeholder (`T`) because of JVM limitations.

Delegating to an arbitrary type

It is possible that none of the options above can represent the type you want to delegate to. For example, let's define a mapper class which is parametrized with an object and defines a map

method which returns an object of another type:

```
class Mapper<T,U> {                                ①
    final T value                                  ②
    Mapper(T value) { this.value = value }
    U map(Closure<U> producer) {                   ③
        producer.delegate = value
        producer()
    }
}
```

① The mapper class takes two generic type arguments: the source type and the target type

② The source object is stored in a final field

③ The `map` method asks to convert the source object to a target object

As you can see, the method signature from `map` does not give any information about what object will be manipulated by the closure. Reading the method body, we know that it will be the `value` which is of type `T`, but `T` is not found in the method signature, so we are facing a case where none of the available options for `@DelegatesTo` is suitable. For example, if we try to statically compile this code:

```
def mapper = new Mapper<String,Integer>('Hello')
assert mapper.map { length() } == 5
```

Then the compiler will fail with:

```
Static type checking] - Cannot find matching method TestScript0#length()
```

In that case, you can use the `type` member of the `@DelegatesTo` annotation to reference `T` as a type token:

```
class Mapper<T,U> {
    final T value
    Mapper(T value) { this.value = value }
    U map(@DelegatesTo(type="T") Closure<U> producer) {  ①
        producer.delegate = value
        producer()
    }
}
```

① The `@DelegatesTo` annotation references a generic type which is not found in the method signature

Note that you are not limited to generic type tokens. The `type` member can be used to represent complex types, such as `List<T>` or `Map<T,List<U>>`. The reason why you should use that in last resort is that the type is only checked when the type checker finds usage of `@DelegatesTo`, not when the annotated method itself is compiled. This means that type safety is only ensured at the call site.

Additionally, compilation will be slower (though probably unnoticeable for most cases).

Compilation customizers

Introduction

Whether you are using `groovyc` to compile classes or a `GroovyShell`, for example, to execute scripts, under the hood, a *compiler configuration* is used. This configuration holds information like the source encoding or the classpath but it can also be used to perform more operations like adding imports by default, applying AST transformations transparently or disabling global AST transformations.

The goal of compilation customizers is to make those common tasks easy to implement. For that, the `CompilerConfiguration` class is the entry point. The general schema will always be based on the following code:

```
import org.codehaus.groovy.control.CompilerConfiguration
// create a configuration
def config = new CompilerConfiguration()
// tweak the configuration
config.addCompilationCustomizers(...)
// run your script
def shell = new GroovyShell(config)
shell.evaluate(script)
```

Compilation customizers must extend the *org.codehaus.groovy.control.customizers.CompilationCustomizer* class. A customizer works:

- on a specific compilation phase
- on *every* class node being compiled

You can implement your own compilation customizer but Groovy includes some of the most common operations.

Import customizer

Using this compilation customizer, your code will have imports added transparently. This is in particular useful for scripts implementing a DSL where you want to avoid users from having to write imports. The import customizer will let you add all the variants of imports the Groovy language allows, that is:

- class imports, optionally aliased
- star imports
- static imports, optionally aliased
- static star imports

```
import org.codehaus.groovy.control.customizers.ImportCustomizer
```

```
def icz = new ImportCustomizer()
// "normal" import
icz.addImports('java.util.concurrent.atomic.AtomicInteger',
'java.util.concurrent.ConcurrentHashMap')
// "aliases" import
icz.addImport('CHM', 'java.util.concurrent.ConcurrentHashMap')
// "static" import
icz.addStaticImport('java.lang.Math', 'PI') // import static java.lang.Math.PI
// "aliased static" import
icz.addStaticImport('pi', 'java.lang.Math', 'PI') // import static java.lang.Math.PI
as pi
// "star" import
icz.addStarImports 'java.util.concurrent' // import java.util.concurrent.*
// "static star" import
icz.addStaticStars 'java.lang.Math' // import static java.lang.Math.*
```

A detailed description of all shortcuts can be found in org.codehaus.groovy.control.customizers.ImportCustomizer

AST transformation customizer

The AST transformation customizer is meant to apply AST transformations transparently. Unlike global AST transformations that apply on every class being compiled as long as the transform is found on classpath (which has drawbacks like increasing the compilation time or side effects due to transformations applied where they should not), the customizer will allow you to selectively apply a transform only for specific scripts or classes.

As an example, let's say you want to be able to use `@Log` in a script. The problem is that `@Log` is normally applied on a class node and a script, by definition, doesn't require one. But implementation wise, scripts are classes, it's just that you cannot annotate this implicit class node with `@Log`. Using the AST customizer, you have a workaround to do it:

```
import org.codehaus.groovy.control.customizers.ASTTransformationCustomizer
import groovy.util.logging.Log

def acz = new ASTTransformationCustomizer(Log)
config.addCompilationCustomizers(acz)
```

That's all! Internally, the `@Log` AST transformation is applied to every class node in the compilation unit. This means that it will be applied to the script, but also to classes defined within the script.

If the AST transformation that you are using accepts parameters, you can use parameters in the constructor too:

```
def acz = new ASTTransformationCustomizer(Log, value: 'LOGGER')
// use name 'LOGGER' instead of the default 'log'
```

```
config.addCompilationCustomizers(acz)
```

As the AST transformation customizers works with objects instead of AST nodes, not all values can be converted to AST transformation parameters. For example, primitive types are converted to `ConstantExpression` (that is `LOGGER` is converted to `new ConstantExpression('LOGGER')`, but if your AST transformation takes a closure as an argument, then you have to give it a `ClosureExpression`, like in the following example:

```
def configuration = new CompilerConfiguration()
def expression = new AstBuilder().buildFromCode(CompilePhase.CONVERSION) { -> true
}.expression[0]
def customizer = new ASTTransformationCustomizer(ConditionalInterrupt, value:
expression, thrown: SecurityException)
configuration.addCompilationCustomizers(customizer)
def shell = new GroovyShell(configuration)
shouldFail(SecurityException) {
    shell.evaluate("""
        // equivalent to adding @ConditionalInterrupt(value={true}, thrown:
SecurityException)
        class MyClass {
            void doIt() { }
        }
        new MyClass().doIt()
    """)
}
```

For a complete list of options, please refer to org.codehaus.groovy.control.customizers.ASTTransformationCustomizer

Secure AST customizer



Despite its name, this customizer is **not** a sandbox and **not** a security boundary. It secures the **grammar** not the **execution** - it is a best-effort restriction of the grammar available to a DSL, and nothing more. It cannot constrain what otherwise-permitted code does at runtime, its coverage of the source is deliberately partial (see Limitations below), and an author of the source being compiled can work around it. The Apache Groovy threat model treats it as a hardening aid rather than a security boundary: a demonstrated bypass is expected behaviour, not a vulnerability. If you must run partially-trusted scripts, layer real isolation **outside** the language — a separate process or JVM, an OS or container sandbox, least-privilege credentials, and network egress controls — and treat this customizer as one layer within that, never the perimeter.

This customizer will allow the developer of a DSL to restrict the **grammar** of the language, for example, to prevent users from using particular constructs. It is only ``secure'' in that one aspect, i.e. limiting the allowable constructs within a DSL. It does **not** replace other aspects of security which might additionally be needed. The only reason for it to exist is to limit the expressiveness of

the language. This customizer only works at the AST (abstract syntax tree) level, not at runtime! It can be strange at first glance, but it makes much more sense if you think of Groovy as a platform to build DSLs. You may not want a user to have a complete language at hand. In the example below, we will demonstrate it using an example of language that only allows arithmetic operations, but this customizer allows you to:

- allow/disallow creation of closures
- allow/disallow imports
- allow/disallow package definition
- allow/disallow definition of methods
- restrict the receivers of method calls
- restrict the kind of AST expressions a user can use
- restrict the tokens (grammar-wise) a user can use
- restrict the types of the constants that can be used in code

For all those features, the secure AST customizer works using either an allowed list (list of elements that are permitted) **or** a disallowed list (list of elements that are not permitted). For each type of feature (imports, tokens, ...) you have the choice to use either an allowed or disallowed list, but you can mix dis/allowed lists for distinct features. Typically, you will choose allowed lists (which permits only the constructs listed and disallows all others).

```
import org.codehaus.groovy.control.customizers.SecureASTCustomizer
import static org.codehaus.groovy.syntax.Types.* ①

def scz = new SecureASTCustomizer()
scz.with {
    closuresAllowed = false // user will not be able to write closures
    methodDefinitionAllowed = false // user will not be able to define methods
    allowedImports = [] // empty allowed list means imports are disallowed
    allowedStaticImports = [] // same for static imports
    allowedStaticStarImports = ['java.lang.Math'] // only java.lang.Math is allowed
    // the list of tokens the user can find
    // constants are defined in org.codehaus.groovy.syntax.Types
    allowedTokens = [ ①
            PLUS,
            MINUS,
            MULTIPLY,
            DIVIDE,
            REMAINDER,
            POWER,
            PLUS_PLUS,
            MINUS_MINUS,
            COMPARE_EQUAL,
            COMPARE_NOT_EQUAL,
            COMPARE_LESS_THAN,
            COMPARE_LESS_THAN_EQUAL,
            COMPARE_GREATER_THAN,
```

```
            COMPARE_GREATER_THAN_EQUAL,
    ].asImmutable()
    // limit the types of constants that a user can define to number types only
    allowedConstantTypesClasses = [ ②
            Integer,
            Float,
            Long,
            Double,
            BigDecimal,
            Integer.TYPE,
            Long.TYPE,
            Float.TYPE,
            Double.TYPE
    ].asImmutable()
    // method calls are only allowed if the receiver is of one of those types
    // be careful, it's not a runtime type!
    allowedReceiversClasses = [ ②
            Math,
            Integer,
            Float,
            Double,
            Long,
            BigDecimal
    ].asImmutable()
}
```

① use for token types from org.codehaus.groovy.syntax.Types

② you can use class literals here

If what the secure AST customizer provides out of the box isn't enough for your needs, before creating your own compilation customizer, you might be interested in the expression and statement checkers that the AST customizer supports. Basically, it allows you to add custom checks on the AST tree, on expressions (expression checkers) or statements (statement checkers). For this, you must implement `org.codehaus.groovy.control.customizers.SecureASTCustomizer.StatementChecker` or `org.codehaus.groovy.control.customizers.SecureASTCustomizer.ExpressionChecker`.

Those interfaces define a single method called `isAuthorized`, returning a boolean, and taking a `Statement` (or `Expression`) as a parameter. It allows you to perform complex logic over expressions or statements to tell if a user is allowed to do it or not.

For example, there's no predefined configuration flag in the customizer which will let you prevent people from using an attribute expression. Using a custom checker, it is trivial:

```
def scz = new SecureASTCustomizer()
def checker = { expr ->
    !(expr instanceof AttributeExpression)
} as SecureASTCustomizer.ExpressionChecker
scz.addExpressionCheckers(checker)
```

Then we can make sure that this works by evaluating a simple script:

```
new GroovyShell(config).evaluate '''
    class A {
        int val
    }

    def a = new A(val: 123)
    a.@val ①
'''
```

① will fail compilation

Statements can be checked using org.codehaus.groovy.control.customizers.SecureASTCustomizer.StatementChecker Expressions can be checked using org.codehaus.groovy.control.customizers.SecureASTCustomizer.ExpressionChecker

Limitations of the secure AST customizer

Coverage is partial by design, so it is worth knowing where the checks do and do not reach. The customizer visits the script statement block, method and constructor bodies, static and instance initializer blocks, and field initializer expressions. It does **not** visit the following, so restrictions such as `disallowedReceivers`, the statement and expression allow/disallow lists, and your own custom checkers do not apply to code appearing there:

- annotation members, including closure arguments to annotations
- code carrying no source position, which is how constructors, initializers and fields added by the compiler or by an AST transformation are told apart from those written by the author of the source being secured — a script class always has generated constructors, for example, and checking those would reject valid programs

A generated member may still **contain** code you wrote, because a transformation can move it there: `@TupleConstructor(pre=⋯)` and `@MapConstructor(pre=⋯)` relocate the supplied closure body into the constructor they generate, and `@ConditionalInterrupt` relocates its condition into a synthetic method. Such code keeps its original source position and is checked wherever it was moved to.

Note that a constructor is not a `` `method definition'' `` as far as `methodDefinitionAllowed` is concerned: its body is checked, but declaring one remains permitted.

Import restrictions apply to actual `import` statements, so they have no effect on a fully-qualified reference such as `new java.lang.ProcessBuilder(⋯)`. The `indirectImportCheckEnabled` flag exists to catch some of those, but only within the code the customizer visits.

More fundamentally, **compiling** Groovy source is itself code execution, whether or not you subsequently run the result: global AST transforms found on the compile classpath, `@groovy.transform.ASTTest`, `@Grab` dependency resolution and static initializers all run during compilation, before and independently of any customizer you configure. Restricting the grammar

of source that you nevertheless compile does not change this. When compiling source you do not fully trust, you should also set the `groovy.grape.enable` system property to `false`, keep the compile classpath under your control, and apply the out-of-language isolation described above.

Source aware customizer

This customizer may be used as a filter on other customizers. The filter, in that case, is the `org.codehaus.groovy.control.SourceUnit`. For this, the source aware customizer takes another customizer as a delegate, and it will apply customization of that delegate only and only if predicates on the source unit match.

`SourceUnit` gives you access to multiple things but in particular the file being compiled (if compiling from a file, of course). It gives you the potential to perform operation based on the file name, for example. Here is how you would create a source aware customizer:

```
import org.codehaus.groovy.control.customizers.SourceAwareCustomizer
import org.codehaus.groovy.control.customizers.ImportCustomizer

def delegate = new ImportCustomizer()
def sac = new SourceAwareCustomizer(delegate)
```

Then you can use predicates on the source aware customizer:

```
// the customizer will only be applied to classes contained in a file name ending with
'Bean'
sac.baseNameValidator = { baseName ->
    baseName.endsWith 'Bean'
}

// the customizer will only be applied to files which extension is '.spec'
sac.extensionValidator = { ext -> ext == 'spec' }

// source unit validation
// allow compilation only if the file contains at most 1 class
sac.sourceUnitValidator = { SourceUnit sourceUnit -> sourceUnit.AST.classes.size() ==
1 }

// class validation
// the customizer will only be applied to classes ending with 'Bean'
sac.classValidator = { ClassNode cn -> cn.endsWith('Bean') }
```

Customizer builder

If you are using compilation customizers in Groovy code (like the examples above) then you can use an alternative syntax to customize compilation. A builder (`org.codehaus.groovy.control.customizers.builder.CompilerCustomizationBuilder`) simplifies the creation of customizers using a hierarchical DSL.

```
import org.codehaus.groovy.control.CompilerConfiguration
import static org.codehaus.groovy.control.customizers.builder
.CompilerCustomizationBuilder.withConfig ①

def conf = new CompilerConfiguration()
withConfig(conf) {
    // ... ②
}
```

① static import of the builder method

② configuration goes here

The code sample above shows how to use the builder. A static method, *withConfig*, takes a closure corresponding to the builder code, and automatically registers compilation customizers to the configuration. Every compilation customizer available in the distribution can be configured this way:

Import customizer

```
withConfig(configuration) {
   imports { // imports customizer
      normal 'my.package.MyClass' // a normal import
      alias 'AI', 'java.util.concurrent.atomic.AtomicInteger' // an aliased import
      star 'java.util.concurrent' // star imports
      staticMember 'java.lang.Math', 'PI' // static import
      staticMember 'pi', 'java.lang.Math', 'PI' // aliased static import
   }
}
```

AST transformation customizer

```
withConfig(conf) {
   ast(Log) ①
}

withConfig(conf) {
   ast(Log, value: 'LOGGER') ②
}
```

① apply @Log transparently

② apply @Log with a different name for the logger

Secure AST customizer

```
withConfig(conf) {
   secureAst {
       closuresAllowed = false
```

```
        methodDefinitionAllowed = false
    }
}
```

Source aware customizer

```
withConfig(configuration){
    source(extension: 'sgroovy') {
        ast(CompileStatic) ①
    }
}

withConfig(configuration){
    source(extensions: ['sgroovy','sg']) {
        ast(CompileStatic) ②
    }
}

withConfig(configuration) {
    source(extensionValidator: { it.name in ['sgroovy','sg']}) {
        ast(CompileStatic) ②
    }
}

withConfig(configuration) {
    source(basename: 'foo') {
        ast(CompileStatic) ③
    }
}

withConfig(configuration) {
    source(basenames: ['foo', 'bar']) {
        ast(CompileStatic) ④
    }
}

withConfig(configuration) {
    source(basenameValidator: { it in ['foo', 'bar'] }) {
        ast(CompileStatic) ④
    }
}

withConfig(configuration) {
    source(unitValidator: { unit -> !unit.AST.classes.any { it.name == 'Baz' } }) {
        ast(CompileStatic) ⑤
    }
}
```

① apply CompileStatic AST annotation on .sgroovy files

② apply CompileStatic AST annotation on .sgroovy or .sg files

③ apply CompileStatic AST annotation on files whose name is 'foo'

④ apply CompileStatic AST annotation on files whose name is 'foo' or 'bar'

⑤ apply CompileStatic AST annotation on files that do not contain a class named 'Baz'

Inlining a customizer

Inlined customizer allows you to write a compilation customizer directly, without having to create a class for it.

```
withConfig(configuration) {
    inline(phase:'CONVERSION') { source, context, classNode ->  ①
        println "visiting $classNode"                           ②
    }
}
```

① define an inlined customizer which will execute at the CONVERSION phase

② prints the name of the class node being compiled

Multiple customizers

Of course, the builder allows you to define multiple customizers at once:

```
withConfig(configuration) {
   ast(ToString)
   ast(EqualsAndHashCode)
}
```

The `configscript` commandline parameter

So far, we have described how you can customize compilation using a `CompilationConfiguration` class, but this is only possible if you embed Groovy and that you create your own instances of `CompilerConfiguration` (then use it to create a `GroovyShell`, `GroovyScriptEngine`, ...).

If you want it to be applied on the classes you compile with the normal Groovy compiler (that is to say with `groovyc`, `ant` or `gradle`, for example), it is possible to use a commandline parameter named `configscript` that takes a Groovy configuration script as argument.

This script gives you access to the `CompilerConfiguration` instance **before** the files are compiled (exposed into the configuration script as a variable named `configuration`), so that you can tweak it.

It also transparently integrates the compiler configuration builder above. As an example, let's see how you would activate static compilation by default on all classes.

Configscript example: Static compilation by default

Normally, classes in Groovy are compiled with a dynamic runtime. You can activate static

compilation by placing an annotation named `@CompileStatic` on any class. Some people would like to have this mode activated by default, that is to say not having to annotate (potentially many) classes. Using `configscript`, makes this possible. First of all, you need to create a file named `config.groovy` into say `src/conf` with the following contents:

```
withConfig(configuration) { ①
   ast(groovy.transform.CompileStatic)
}
```

① *configuration* references a `CompilerConfiguration` instance

That is actually all you need. You don't have to import the builder, it's automatically exposed in the script. Then, compile your files using the following command line:

```
groovyc -configscript src/conf/config.groovy src/main/groovy/MyClass.groovy
```

We strongly recommend you to separate configuration files from classes, hence why we suggest using the `src/main` and `src/conf` directories above.

Configscript example: Setting system properties

In a configuration script you can also set system properties, e.g.:

```
System.setProperty('spock.iKnowWhatImDoing.disableGroovyVersionCheck', 'true')
```

If you have numerous system properties to set, then using a configuration file will reduce the need to set a bunch of system properties with a long command line or appropriately defined environment variable. You can also share all the settings by simply sharing the config file.

AST transformations

If:

- runtime metaprogramming doesn't allow you to do what you want
- you need to improve the performance of the execution of your DSLs
- you want to leverage the same syntax as Groovy but with different semantics
- you want to improve support for type checking in your DSLs

Then AST transformations are the way to go. Unlike the techniques used so far, AST transformations are meant to change or generate code before it is compiled to bytecode. AST transformations are capable of adding new methods at compile time for example, or totally changing the body of a method based on your needs. They are a very powerful tool but also come at the price of not being easy to write. For more information about AST transformations, please take a look at the compile-time metaprogramming section of this manual.

Custom type checking extensions

It may be interesting, in some circumstances, to provide feedback about wrong code to the user as soon as possible, that is to say when the DSL script is compiled, rather than having to wait for the execution of the script. However, this is not often possible with dynamic code. Groovy actually provides a practical answer to this known as type checking extensions.

Builders

Many tasks require building things and the builder pattern is one technique used by developers to make building things easier, especially building of structures which are hierarchical in nature. This pattern is so ubiquitous that Groovy has special built-in support. Firstly, there are many built-in builders. Secondly, there are classes which make it easier to write your own builders.

Existing builders

Groovy comes with many built-in builders. Let's look at some of them.

MarkupBuilder

See Creating Xml - MarkupBuilder.

StreamingMarkupBuilder

See Creating Xml - StreamingMarkupBuilder.

SaxBuilder

A builder for generating Simple API for XML (SAX) events.

If you have the following SAX handler:

```
class LogHandler extends org.xml.sax.helpers.DefaultHandler {

    String log = ''

    void startElement(String uri, String localName, String qName, org.xml.sax
.Attributes attributes) {
        log += "Start Element: $localName, "
    }

    void endElement(String uri, String localName, String qName) {
        log += "End Element: $localName, "
    }
}
```

You can use `SaxBuilder` to generate SAX events for the handler like this:

```
def handler = new LogHandler()
def builder = new groovy.xml.SAXBuilder(handler)
```

```
builder.root() {
    helloWorld()
}
```

And then check that everything worked as expected:

```
assert handler.log == 'Start Element: root, Start Element: helloWorld, End Element:
helloWorld, End Element: root, '
```

StaxBuilder

A Groovy builder that works with Streaming API for XML (StAX) processors.

Here is a simple example using the StAX implementation of Java to generate XML:

```
def factory = javax.xml.stream.XMLOutputFactory.newInstance()
def writer = new StringWriter()
def builder = new groovy.xml.StaxBuilder(factory.createXMLStreamWriter(writer))

builder.root(attribute:1) {
    elem1('hello')
    elem2('world')
}

def pretty= writer.toString()
    .replaceAll(/<\?xml[^>]*>/, '') // remove XML declaration
assert pretty == '<root attribute="1"><elem1>hello</elem1><elem2>world</elem2></root>'
```

An external library such as Jettison can be used as follows:

```
@Grab('org.codehaus.jettison:jettison:1.3.3')
@GrabExclude('stax:stax-api') // part of Java 6 and later
import org.codehaus.jettison.mapped.*

def writer = new StringWriter()
def mappedWriter = new MappedXMLStreamWriter(new MappedNamespaceConvention(), writer)
def builder = new groovy.xml.StaxBuilder(mappedWriter)

builder.root(attribute:1) {
     elem1('hello')
     elem2('world')
}

assert writer.toString() ==
'{"root":{"@attribute":"1","elem1":"hello","elem2":"world"}}'
```

Reading XML with StAX

`StaxBuilder` writes XML through a user-supplied `XMLStreamWriter`. For **reading** XML with StAX, Groovy provides hardened entry points:

- `groovy.xml.FactorySupport.createXMLInputFactory()` — returns a hardened `XMLInputFactory` (DTD support disabled, external entity resolution disabled). Pass `true` to allow DOCTYPE declarations.
- `groovy.xml.XmlUtil.events(reader)` — returns a `Stream<XMLEvent>` over the supplied reader. Closes the underlying reader when the stream is closed.
- `groovy.xml.XmlUtil.streamElements(reader, localName)` — pulls each matching subtree as a small DOM `Node`, suitable for streaming multi-gigabyte documents without loading them into memory.

See XML security for a worked example including parallel per-record processing using virtual threads.

DOMBuilder

A builder for parsing HTML, XHTML and XML into a W3C DOM tree.

For example this XML `String`:

```
String recordsXML = '''
    <records>
      <car name='HSV Maloo' make='Holden' year='2006'>
        <country>Australia</country>
        <record type='speed'>Production Pickup Truck with speed of 271kph</record>
      </car>
      <car name='P50' make='Peel' year='1962'>
        <country>Isle of Man</country>
        <record type='size'>Smallest Street-Legal Car at 99cm wide and 59 kg in
weight</record>
      </car>
      <car name='Royale' make='Bugatti' year='1931'>
        <country>France</country>
        <record type='price'>Most Valuable Car at $15 million</record>
      </car>
    </records>'''
```

Can be parsed into a DOM tree with a `DOMBuilder` like this:

```
def reader = new StringReader(recordsXML)
def doc = groovy.xml.DOMBuilder.parse(reader)
```

And then processed further e.g. by using DOMCategory:

```
def records = doc.documentElement
```

```
use(groovy.xml.dom.DOMCategory) {
    assert records.car.size() == 3
}
```



`DOMBuilder.parse(…)` and `DOMBuilder.newInstance()` reject DOCTYPE declarations by default. The `newInstance(validating, namespaceAware, allowDocTypeDeclaration)` overload exposes a relax knob when DTD support is required. See XML security for details.

NodeBuilder

`NodeBuilder` is used for creating nested trees of groovy.util.Node objects for handling arbitrary data. To create a simple user list you use a `NodeBuilder` like this:

```
def nodeBuilder = new NodeBuilder()
def userlist = nodeBuilder.userlist {
    user(id: '1', firstname: 'John', lastname: 'Smith') {
        address(type: 'home', street: '1 Main St.', city: 'Springfield', state: 'MA',
zip: '12345')
        address(type: 'work', street: '2 South St.', city: 'Boston', state: 'MA', zip:
'98765')
    }
    user(id: '2', firstname: 'Alice', lastname: 'Doe')
}
```

Now you can process the data further, e.g. by using GPath expressions:

```
assert userlist.user.@firstname.join(', ') == 'John, Alice'
assert userlist.user.find { it.@lastname == 'Smith' }.address.size() == 2
```

JsonBuilder

Groovy's `JsonBuilder` makes it easy to create Json. For example to create this Json string:

```
String carRecords = '''
    {
        "records": {
        "car": {
            "name": "HSV Maloo",
            "make": "Holden",
            "year": 2006,
            "country": "Australia",
            "record": {
              "type": "speed",
              "description": "production pickup truck with speed of 271kph"
            }
          }
```

```
      }
    }
'''
```

you can use a `JsonBuilder` like this:

```
JsonBuilder builder = new JsonBuilder()
builder.records {
  car {
        name 'HSV Maloo'
        make 'Holden'
        year 2006
        country 'Australia'
        record {
            type 'speed'
            description 'production pickup truck with speed of 271kph'
        }
  }
}
String json = JsonOutput.prettyPrint(builder.toString())
```

We use JsonUnit to check that the builder produced the expected result:

```
JsonAssert.assertJsonEquals(json, carRecords)
```

If you need to customize the generated output you can pass a `JsonGenerator` instance when creating a `JsonBuilder`:

```
import groovy.json.*

def generator = new JsonGenerator.Options()
        .excludeNulls()
        .excludeFieldsByName('make', 'country', 'record')
        .excludeFieldsByType(Number)
        .addConverter(URL) { url -> "http://groovy-lang.org" }
        .build()

JsonBuilder builder = new JsonBuilder(generator)
builder.records {
  car {
        name 'HSV Maloo'
        make 'Holden'
        year 2006
        country 'Australia'
        homepage new URL('http://example.org')
        record {
            type 'speed'
```

```
            description 'production pickup truck with speed of 271kph'
        }
    }
}

assert builder.toString() == '{"records":{"car":{"name":"HSV
Maloo","homepage":"http://groovy-lang.org"}}}'
```

StreamingJsonBuilder

Unlike `JsonBuilder` which creates a data structure in memory, which is handy in those situations where you want to alter the structure programmatically before output, `StreamingJsonBuilder` directly streams to a writer without any intermediate memory data structure. If you do not need to modify the structure and want a more memory-efficient approach, use `StreamingJsonBuilder`.

The usage of `StreamingJsonBuilder` is similar to `JsonBuilder`. In order to create this Json string:

```
String carRecords = """
    {
      "records": {
        "car": {
          "name": "HSV Maloo",
          "make": "Holden",
          "year": 2006,
          "country": "Australia",
          "record": {
            "type": "speed",
            "description": "production pickup truck with speed of 271kph"
          }
        }
      }
    }
"""
```

you use a `StreamingJsonBuilder` like this:

```
StringWriter writer = new StringWriter()
StreamingJsonBuilder builder = new StreamingJsonBuilder(writer)
builder.records {
    car {
        name 'HSV Maloo'
        make 'Holden'
        year 2006
        country 'Australia'
        record {
            type 'speed'
            description 'production pickup truck with speed of 271kph'
        }
    }
```

```
}
String json = JsonOutput.prettyPrint(writer.toString())
```

We use JsonUnit to check the expected result:

```
JsonAssert.assertJsonEquals(json, carRecords)
```

If you need to customize the generated output you can pass a `JsonGenerator` instance when creating a `StreamingJsonBuilder`:

```
def generator = new JsonGenerator.Options()
        .excludeNulls()
        .excludeFieldsByName('make', 'country', 'record')
        .excludeFieldsByType(Number)
        .addConverter(URL) { url -> "http://groovy-lang.org" }
        .build()

StringWriter writer = new StringWriter()
StreamingJsonBuilder builder = new StreamingJsonBuilder(writer, generator)

builder.records {
    car {
        name 'HSV Maloo'
        make 'Holden'
        year 2006
        country 'Australia'
        homepage new URL('http://example.org')
        record {
            type 'speed'
            description 'production pickup truck with speed of 271kph'
        }
    }
}

assert writer.toString() == '{"records":{"car":{"name":"HSV
Maloo","homepage":"http://groovy-lang.org"}}}'
```

SwingBuilder

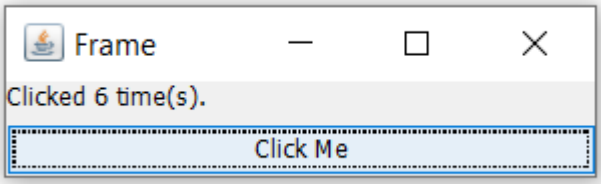
`SwingBuilder` allows you to create full-fledged Swing GUIs in a declarative and concise fashion. It accomplishes this by employing a common idiom in Groovy, builders. Builders handle the busywork of creating complex objects for you, such as instantiating children, calling Swing methods, and attaching these children to their parents. As a consequence, your code is much more readable and maintainable, while still allowing you to access to the full range of Swing components.

Here's a simple example of using `SwingBuilder`:

```
import groovy.swing.SwingBuilder
import java.awt.BorderLayout as BL

count = 0
new SwingBuilder().edt {
  frame(title: 'Frame', size: [250, 75], show: true) {
    borderLayout()
    textlabel = label(text: 'Click the button!', constraints: BL.NORTH)
    button(text:'Click Me',
         actionPerformed: {count++; textlabel.text = "Clicked ${count} time(s).";
println "clicked"}, constraints:BL.SOUTH)
  }
}
```

Here is what it will look like:



This hierarchy of components would normally be created through a series of repetitive instantiations, setters, and finally attaching this child to its respective parent. Using `SwingBuilder`, however, allows you to define this hierarchy in its native form, which makes the interface design understandable simply by reading the code.

The flexibility shown here is made possible by leveraging the many programming features built-in to Groovy, such as closures, implicit constructor calling, import aliasing, and string interpolation. Of course, these do not have to be fully understood in order to use `SwingBuilder`; as you can see from the code above, their uses are intuitive.

Here is a slightly more involved example, with an example of `SwingBuilder` code re-use via a closure.

```
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

def swing = new SwingBuilder()

def sharedPanel = {
     swing.panel() {
        label("Shared Panel")
    }
}
```

```
count = 0
swing.edt {
    frame(title: 'Frame', defaultCloseOperation: JFrame.EXIT_ON_CLOSE, pack: true,
show: true) {
        vbox {
            textlabel = label('Click the button!')
            button(
                text: 'Click Me',
                actionPerformed: {
                    count++
                    textlabel.text = "Clicked ${count} time(s)."
                    println "Clicked!"
                }
            )
            widget(sharedPanel())
            widget(sharedPanel())
        }
    }
}
```

Here's another variation that relies on observable beans and binding:

```
import groovy.swing.SwingBuilder
import groovy.beans.Bindable

class MyModel {
   @Bindable int count = 0
}

def model = new MyModel()
new SwingBuilder().edt {
  frame(title: 'Java Frame', size: [100, 100], locationRelativeTo: null, show: true) {
    gridLayout(cols: 1, rows: 2)
    label(text: bind(source: model, sourceProperty: 'count', converter: { v ->  v?
"Clicked $v times": ''}))
    button('Click me!', actionPerformed: { model.count++ })
  }
}
```

`@Bindable` is one of the core AST Transformations. It generates all the required boilerplate code to turn a simple bean into an observable one. The `bind()` node creates appropriate `PropertyChangeListeners` that will update the interested parties whenever a `PropertyChangeEvent` is fired.

SwingBuilder Widgets Reference

The following tables list all nodes available in `SwingBuilder`, grouped by category. Each node name links to its detail section showing properties.

Support Nodes

Node Name	Swing Class	Leaf
action	`Action`	false
actions	-	false
buttonGroup	`ButtonGroup`	true
imageIcon	`ImageIcon`	false
keyStrokeAction	-	false
map	-	false
noparent	-	false
shortcut	-	false

Binding

Node Name	Swing Class	Leaf
bind	-	false
bindGroup	-	false
bindProxy	-	true

Pass Thru Nodes

Node Name	Swing Class	Leaf
bean	`Object`	true
container	`Component`	false
widget	`Component`	true

Windows

Node Name	Swing Class	Leaf
dialog	`JDialog`	false
fileChooser	`JFileChooser`	false
frame	`JFrame`	false
optionPane	`JOptionPane`	false
window	`JWindow`	false

Action Button Widgets

Node Name	Swing Class	Leaf
button	`JButton`	false

Node Name	Swing Class	Leaf
checkBox	JCheckBox	false
checkBoxMenuItem	JCheckBoxMenuItem	false
menuItem	JMenuItem	false
radioButton	JRadioButton	false
radioButtonMenuItem	JRadioButtonMenuItem	false
toggleButton	JToggleButton	false

Text Widgets

Node Name	Swing Class	Leaf
editorPane	JEditorPane	false
formattedTextField	JFormattedTextField	false
label	JLabel	false
passwordField	JPasswordField	false
textArea	JTextArea	false
textField	JTextField	false
textPane	JTextPane	false

Basic Widgets

Node Name	Swing Class	Leaf
colorChooser	JColorChooser	false
comboBox	JComboBox	false
list	JList	false
progressBar	JProgressBar	false
scrollBar	JScrollBar	false
separator	JSeparator	false
slider	JSlider	false
spinner	JSpinner	false
tree	JTree	false

MDI Widgets

Node Name	Swing Class	Leaf
desktopPane	JDesktopPane	false
internalFrame	JInternalFrame	false

Menu Widgets

Node Name	Swing Class	Leaf
menu	`JMenu`	false
menuBar	`JMenuBar`	false
popupMenu	`JPopupMenu`	false

Containers

Node Name	Swing Class	Leaf
layeredPane	`JLayeredPane`	false
panel	`JPanel`	false
scrollPane	`JScrollPane`	false
splitPane	`JSplitPane`	false
tabbedPane	`JTabbedPane`	false
toolBar	`JToolBar`	false
viewport	`JViewport`	false

Data Models

Node Name	Swing Class	Leaf
boundedRangeModel	`DefaultBoundedRangeModel`	false
spinnerDateModel	`SpinnerDateModel`	false
spinnerListModel	`SpinnerListModel`	false
spinnerNumberModel	`SpinnerNumberModel`	false

Table Components

Node Name	Swing Class	Leaf
closureColumn	-	false
column	-	false
columnModel	-	false
propertyColumn	-	false
table	`JTable`	false
tableColumn	`TableColumn`	false
tableModel	-	false

Renderers

Node Name	Swing Class	Leaf
cellRenderer	-	false
headerRenderer	-	false
listCellRenderer	-	false
onRender	-	false
tableCellRenderer	-	false

Editors

Node Name	Swing Class	Leaf
cellEditor	-	false
editorValue	-	false
prepareEditor	-	false

Basic Layouts

Node Name	Swing Class	Leaf
borderLayout	`BorderLayout`	false
cardLayout	`CardLayout`	false
flowLayout	`FlowLayout`	false
gbc	`GridBagConstraints`	false
gridBagConstraints	`GridBagConstraints`	false
gridBagLayout	`GridBagLayout`	true
gridLayout	`GridLayout`	false
overlayLayout	`OverlayLayout`	false
springLayout	`SpringLayout`	false

Box Layout

Node Name	Swing Class	Leaf
box	`Box`	false
boxLayout	-	false
glue	-	false
hbox	`Box`	false
hglue	-	false
hstrut	-	false
rigidArea	-	false
vbox	`Box`	false

Node Name	Swing Class	Leaf
vglue	-	false
vstrut	-	false

Table Layout

Node Name	Swing Class	Leaf
tableLayout	-	false
td	-	false
tr	-	false

Borders

Node Name	Swing Class	Leaf
compoundBorder	-	true
emptyBorder	-	true
etchedBorder	-	true
lineBorder	-	true
loweredBevelBorder	-	true
loweredEtchedBorder	-	true
matteBorder	-	true
raisedBevelBorder	-	true
raisedEtchedBorder	-	true
titledBorder	-	true

Threading

Node Name	Swing Class	Leaf
doLater	-	false
doOutside	-	false
edt	-	false

Utility Methods

These are explicit methods (not node factories) available on `SwingBuilder`:

Method	Description
`doLater()`	Run a closure on the EDT asynchronously (non-blocking)
`doOutside()`	Run a closure outside the EDT in a new thread

Method	Description
`edt()`	Run a closure on the Event Dispatch Thread (blocking)
`keyStrokeAction()`	Create keyboard shortcut bindings on a component
`shortcut()`	Create a platform-appropriate KeyStroke (uses Cmd on Mac, Ctrl elsewhere)

Widget Details

---

action

Swing Class `javax.swing.Action`

Leaf false

API Docs javax.swing.Action

Basic usage

```
swing.action(name: 'MyAction', closure: { println 'action' })
```

---

actions

Basic usage

```
swing.actions()
```

---

bean

Swing Class `java.lang.Object`

Leaf true

Basic usage

```
swing.bean()
```

---

bind

Basic usage

```
swing.bind(source: model, sourceProperty: 'name')
```

bindGroup

Basic usage

```
swing.bindGroup()
```

bindProxy

Basic usage

```
swing.bindProxy()
```

borderLayout

Swing Class	`java.awt.BorderLayout`
Leaf	false
API Docs	java.awt.BorderLayout

Basic usage

```
swing.borderLayout()
```

Properties

Property	Type	Readable	Writable
`hgap`	`int`	✔	✔
`vgap`	`int`	✔	✔

boundedRangeModel

Swing Class	`javax.swing.DefaultBoundedRangeModel`
Leaf	false
API Docs	javax.swing.DefaultBoundedRangeModel

Basic usage

```
swing.boundedRangeModel()
```

Properties

Property	Type	Readable	Writable
changeListeners	event.ChangeListener[]	✔	
extent	int	✔	✔
maximum	int	✔	✔
minimum	int	✔	✔
value	int	✔	✔
valueIsAdjusting	boolean	✔	✔

---

box

Swing Class javax.swing.Box

Leaf false

API Docs javax.swing.Box

Basic usage

```
swing.box()
```

Properties

Property	Type	Readable	Writable
UI	plaf.ComponentUI	✔	
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	

Property	Type	Readable	Writable
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	

Property	Type	Readable	Writable
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

boxLayout

Basic usage

```
swing.boxLayout()
```

button

Swing Class javax.swing.JButton

Leaf false

API Docs javax.swing.JButton

Basic usage

```
swing.button('Click Me')
```

Properties

Property	Type	Readable	Writable
UI	plaf.ButtonUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
action	Action	✔	✔
actionCommand	String	✔	✔
actionListeners	awt.event.ActionListener[]	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔

Property	Type	Readable	Writable
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
borderPainted	boolean	✔	✔
changeListeners	event.ChangeListener[]	✔	
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
contentAreaFilled	boolean	✔	✔
debugGraphicsOptions	int	✔	✔
defaultButton	boolean	✔	
defaultCapable	boolean	✔	✔
disabledIcon	Icon	✔	✔
disabledSelectedIcon	Icon	✔	✔
displayedMnemonicIndex	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusPainted	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
hideActionText	boolean	✔	✔
horizontalAlignment	int	✔	✔
horizontalTextPosition	int	✔	✔
icon	Icon	✔	✔
iconTextGap	int	✔	✔
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	

Property	Type	Readable	Writable
itemListeners	awt.event.ItemListener[]	✔	
label	String	✔	✔
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
margin	awt.Insets	✔	✔
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
mnemonic	int	✔	✔
model	ButtonModel	✔	✔
multiClickThreshhold	long	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
pressedIcon	Icon	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rolloverEnabled	boolean	✔	✔
rolloverIcon	Icon	✔	✔
rolloverSelectedIcon	Icon	✔	✔
rootPane	JRootPane	✔	
selected	boolean	✔	✔
selectedIcon	Icon	✔	✔
selectedObjects	Object[]	✔	
text	String	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
verticalAlignment	int	✔	✔
verticalTextPosition	int	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	

Property	Type	Readable	Writable
width	int	✔	
x	int	✔	
y	int	✔	

buttonGroup

Swing Class javax.swing.ButtonGroup

Leaf true

API Docs javax.swing.ButtonGroup

Basic usage

```
swing.buttonGroup()
```

Properties

Property	Type	Readable	Writable
buttonCount	int	✔	
elements	Enumeration	✔	
selection	ButtonModel	✔	

cardLayout

Swing Class java.awt.CardLayout

Leaf false

API Docs java.awt.CardLayout

Basic usage

```
swing.cardLayout()
```

Properties

Property	Type	Readable	Writable
hgap	int	✔	✔
vgap	int	✔	✔

cellEditor

Basic usage

```
swing.cellEditor()
```

---

cellRenderer

Basic usage

```
swing.cellRenderer()
```

---

checkBox

Swing Class	javax.swing.JCheckBox
Leaf	false
API Docs	javax.swing.JCheckBox

Basic usage

```
swing.checkBox('Click Me')
```

Properties

Property	Type	Readable	Writable
UI	plaf.ButtonUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
action	Action	✔	✔
actionCommand	String	✔	✔
actionListeners	awt.event.ActionListener[]	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	

Property	Type	Readable	Writable
border	border.Border	✔	✔
borderPainted	boolean	✔	✔
borderPaintedFlat	boolean	✔	✔
changeListeners	event.ChangeListener[]	✔	
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
contentAreaFilled	boolean	✔	✔
debugGraphicsOptions	int	✔	✔
disabledIcon	Icon	✔	✔
disabledSelectedIcon	Icon	✔	✔
displayedMnemonicIndex	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusPainted	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
hideActionText	boolean	✔	✔
horizontalAlignment	int	✔	✔
horizontalTextPosition	int	✔	✔
icon	Icon	✔	✔
iconTextGap	int	✔	✔
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
itemListeners	awt.event.ItemListener[]	✔	
label	String	✔	✔

Property	Type	Readable	Writable
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
margin	awt.Insets	✔	✔
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
mnemonic	int	✔	✔
model	ButtonModel	✔	✔
multiClickThreshhold	long	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
pressedIcon	Icon	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rolloverEnabled	boolean	✔	✔
rolloverIcon	Icon	✔	✔
rolloverSelectedIcon	Icon	✔	✔
rootPane	JRootPane	✔	
selected	boolean	✔	✔
selectedIcon	Icon	✔	✔
selectedObjects	Object[]	✔	
text	String	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
verticalAlignment	int	✔	✔
verticalTextPosition	int	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	

Property	Type	Readable	Writable
y	int	✔	

checkBoxMenuItem

Swing Class	javax.swing.JCheckBoxMenuItem
Leaf	false
API Docs	javax.swing.JCheckBoxMenuItem

Basic usage

```
swing.checkBoxMenuItem('Click Me')
```

Properties

Property	Type	Readable	Writable
UI	plaf.ButtonUI	✔	✔
UIClassID	String	✔	
accelerator	KeyStroke	✔	✔
accessibleContext	javax.accessibility.AccessibleContext	✔	
action	Action	✔	✔
actionCommand	String	✔	✔
actionListeners	awt.event.ActionListener[]	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
armed	boolean	✔	✔
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
borderPainted	boolean	✔	✔
changeListeners	event.ChangeListener[]	✔	
component	awt.Component	✔	
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	

Property	Type	Readable	Writable
containerListeners	awt.event.ContainerListener[]	✔	
contentAreaFilled	boolean	✔	✔
debugGraphicsOptions	int	✔	✔
disabledIcon	Icon	✔	✔
disabledSelectedIcon	Icon	✔	✔
displayedMnemonicIndex	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusPainted	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
hideActionText	boolean	✔	✔
horizontalAlignment	int	✔	✔
horizontalTextPosition	int	✔	✔
icon	Icon	✔	✔
iconTextGap	int	✔	✔
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
itemListeners	awt.event.ItemListener[]	✔	
label	String	✔	✔
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
margin	awt.Insets	✔	✔
maximumSize	awt.Dimension	✔	✔
menuDragMouseListeners	event.MenuDragMouseListener[]	✔	
menuKeyListeners	event.MenuKeyListener[]	✔	
minimumSize	awt.Dimension	✔	✔
mnemonic	int	✔	✔

Property	Type	Readable	Writable
model	ButtonModel	✔	✔
multiClickThreshhold	long	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
pressedIcon	Icon	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rolloverEnabled	boolean	✔	✔
rolloverIcon	Icon	✔	✔
rolloverSelectedIcon	Icon	✔	✔
rootPane	JRootPane	✔	
selected	boolean	✔	✔
selectedIcon	Icon	✔	✔
selectedObjects	Object[]	✔	
state	boolean	✔	✔
subElements	MenuElement[]	✔	
text	String	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
verticalAlignment	int	✔	✔
verticalTextPosition	int	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

closureColumn

Basic usage

```
swing.closureColumn()
```

---

colorChooser

Swing Class	javax.swing.JColorChooser
Leaf	false
API Docs	javax.swing.JColorChooser

Basic usage

```
swing.colorChooser()
```

Properties

Property	Type	Readable	Writable
UI	plaf.ColorChooserUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
chooserPanels	colorchooser.AbstractColorChooserPanel[]	✔	✔
color	awt.Color	✔	✔
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔

Property	Type	Readable	Writable
doubleBuffered	boolean	✔	✔
dragEnabled	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
previewPanel	JComponent	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
selectionModel	colorchooser.ColorSelectionModel	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔

Property	Type	Readable	Writable
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

column

Basic usage

```
swing.column()
```

columnModel

Basic usage

```
swing.columnModel()
```

comboBox

Swing Class	javax.swing.JComboBox
Leaf	false
API Docs	javax.swing.JComboBox

Basic usage

```
swing.comboBox()
```

Properties

Property	Type	Readable	Writable
UI	plaf.ComboBoxUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
action	Action	✔	✔

Property	Type	Readable	Writable
actionCommand	String	✔	✔
actionListeners	awt.event.ActionListener[]	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
editable	boolean	✔	✔
editor	ComboBoxEditor	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
itemAt	Object		
itemCount	int	✔	
itemListeners	awt.event.ItemListener[]	✔	
keySelectionManager	JComboBox$KeySelectionManager	✔	✔

Property	Type	Readable	Writable
layout	awt.LayoutManager	✔	✔
lightWeightPopupEnabled	boolean	✔	✔
managingFocus	boolean	✔	
maximumRowCount	int	✔	✔
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
model	ComboBoxModel	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
popupMenuListeners	event.PopupMenuListener[]	✔	
popupVisible	boolean	✔	✔
preferredSize	awt.Dimension	✔	✔
prototypeDisplayValue	Object	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
renderer	ListCellRenderer	✔	✔
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
selectedIndex	int	✔	✔
selectedItem	Object	✔	✔
selectedObjects	Object[]	✔	
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

compoundBorder

Basic usage

```
swing.compoundBorder()
```

---

container

Swing Class	`java.awt.Component`
Leaf	false
API Docs	java.awt.Component

Basic usage

```
swing.container()
```

Properties

Property	Type	Readable	Writable
`background`	`awt.Color`	✔	✔
`enabled`	`boolean`	✔	✔
`focusable`	`boolean`	✔	✔
`font`	`awt.Font`	✔	✔
`foreground`	`awt.Color`	✔	✔
`name`	`String`	✔	✔
`visible`	`boolean`	✔	✔

---

desktopPane

Swing Class	`javax.swing.JDesktopPane`
Leaf	false
API Docs	javax.swing.JDesktopPane

Basic usage

```
swing.desktopPane()
```

Properties

Property	Type	Readable	Writable
UI	plaf.DesktopPaneUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
allFrames	JInternalFrame[]	✔	
allFramesInLayer	Object		
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
component	Object		
componentCount	int	✔	
componentCountInLayer	Object		
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
componentsInLayer	Object		
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
desktopManager	DesktopManager	✔	✔
doubleBuffered	boolean	✔	✔
dragMode	int	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	

Property	Type	Readable	Writable
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
selectedFrame	JInternalFrame	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

dialog

Swing Class	javax.swing.JDialog
Leaf	false
API Docs	javax.swing.JDialog

Basic usage

```
swing.dialog(title: 'My Window')
```

Properties

Property	Type	Readable	Writable
JMenuBar	JMenuBar	✔	✔
accessibleContext	javax.accessibility.AccessibleContext	✔	
active	boolean	✔	
alignmentX	float	✔	
alignmentY	float	✔	
alwaysOnTop	boolean	✔	✔
alwaysOnTopSupported	boolean	✔	
autoRequestFocus	boolean	✔	✔
background	awt.Color	✔	✔
bounds	awt.Rectangle		✔
bufferStrategy	awt.image.BufferStrategy	✔	
component	Object		
componentCount	int	✔	
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
contentPane	awt.Container	✔	✔
cursor	awt.Cursor		✔
defaultCloseOperation	int	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusCycleRootAncestor	awt.Container	✔	
focusOwner	awt.Component	✔	
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
focusableWindow	boolean	✔	
focusableWindowState	boolean	✔	✔
focused	boolean	✔	
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
glassPane	awt.Component	✔	✔

Property	Type	Readable	Writable
graphics	awt.Graphics	✔	
iconImage	awt.Image		✔
iconImages	List	✔	✔
inputContext	awt.im.InputContext	✔	
insets	awt.Insets	✔	
layeredPane	JLayeredPane	✔	✔
layout	awt.LayoutManager	✔	✔
locale	Locale	✔	
location	awt.Point		✔
locationByPlatform	boolean	✔	✔
locationRelativeTo	awt.Component		✔
maximumSize	awt.Dimension	✔	
minimumSize	awt.Dimension	✔	✔
modal	boolean	✔	✔
modalExclusionType	awt.Dialog$ModalExclusionType	✔	✔
modalityType	awt.Dialog$ModalityType	✔	✔
mostRecentFocusOwner	awt.Component	✔	
name	String	✔	✔
opacity	float	✔	✔
opaque	boolean	✔	
ownedWindows	awt.Window[]	✔	
owner	awt.Window	✔	
preferredSize	awt.Dimension	✔	
resizable	boolean	✔	✔
rootPane	JRootPane	✔	
shape	awt.Shape	✔	✔
showing	boolean	✔	
size	awt.Dimension		✔
title	String	✔	✔
toolkit	awt.Toolkit	✔	
transferHandler	TransferHandler	✔	✔
type	awt.Window$Type	✔	✔
undecorated	boolean	✔	✔
validateRoot	boolean	✔	
visible	boolean	✔	✔
warningString	String	✔	
windowFocusListeners	awt.event.WindowFocusListener[]	✔	
windowListeners	awt.event.WindowListener[]	✔	

Property	Type	Readable	Writable
windowStateListeners	awt.event.WindowStateListener[]	✔	

editorPane

Swing Class javax.swing.JEditorPane

Leaf false

API Docs javax.swing.JEditorPane

Basic usage

```
swing.editorPane(text: 'initial text', columns: 20)
```

Properties

Property	Type	Readable	Writable
UI	plaf.TextUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
actions	Action[]	✔	
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
caret	text.Caret	✔	✔
caretColor	awt.Color	✔	✔
caretListeners	event.CaretListener[]	✔	
caretPosition	int	✔	✔
component	Object		
componentCount	int	✔	
componentOrientation	awt.ComponentOrientation		✔
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	

Property	Type	Readable	Writable
containerListeners	awt.event.ContainerListener[]	✔	
contentType	String	✔	✔
debugGraphicsOptions	int	✔	✔
disabledTextColor	awt.Color	✔	✔
document	text.Document	✔	✔
doubleBuffered	boolean	✔	✔
dragEnabled	boolean	✔	✔
dropLocation	text.JTextComponent$DropLocation	✔	
dropMode	DropMode	✔	✔
editable	boolean	✔	✔
editorKit	text.EditorKit	✔	✔
enabled	boolean	✔	✔
focusAccelerator	char	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
highlighter	text.Highlighter	✔	✔
hyperlinkListeners	event.HyperlinkListener[]	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputMethodRequests	awt.im.InputMethodRequests	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
keymap	text.Keymap	✔	✔
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
margin	awt.Insets	✔	✔
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
navigationFilter	text.NavigationFilter	✔	✔

Property	Type	Readable	Writable
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
page	java.net.URL	✔	✔
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredScrollableViewportSize	awt.Dimension	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
scrollableTracksViewportHeight	boolean	✔	
scrollableTracksViewportWidth	boolean	✔	
selectedText	String	✔	
selectedTextColor	awt.Color	✔	✔
selectionColor	awt.Color	✔	✔
selectionEnd	int	✔	✔
selectionStart	int	✔	✔
text	String	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

editorValue

Basic usage

```
swing.editorValue()
```

emptyBorder

Basic usage

```
swing.emptyBorder()
```

---

etchedBorder

Basic usage

```
swing.etchedBorder()
```

---

fileChooser

Swing Class	`javax.swing.JFileChooser`
Leaf	false
API Docs	javax.swing.JFileChooser

Basic usage

```
swing.fileChooser()
```

Properties

Property	Type	Readable	Writable
`UI`	`plaf.FileChooserUI`	✔	
`UIClassID`	`String`	✔	
`acceptAllFileFilter`	`filechooser.FileFilter`	✔	
`acceptAllFileFilterUsed`	`boolean`	✔	✔
`accessibleContext`	`javax.accessibility.AccessibleContext`	✔	
`accessory`	`JComponent`	✔	✔
`actionListeners`	`awt.event.ActionListener[]`	✔	
`actionMap`	`ActionMap`	✔	✔
`alignmentX`	`float`	✔	✔
`alignmentY`	`float`	✔	✔
`ancestorListeners`	`event.AncestorListener[]`	✔	
`approveButtonMnemonic`	`int`	✔	✔
`approveButtonText`	`String`	✔	✔

Property	Type	Readable	Writable
approveButtonToolTipText	String	✔	✔
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
choosableFileFilters	filechooser.FileFilter[]	✔	
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
controlButtonsAreShown	boolean	✔	✔
currentDirectory	java.io.File	✔	✔
debugGraphicsOptions	int	✔	✔
dialogTitle	String	✔	✔
dialogType	int	✔	✔
directorySelectionEnabled	boolean	✔	
doubleBuffered	boolean	✔	✔
dragEnabled	boolean	✔	✔
enabled	boolean	✔	✔
fileFilter	filechooser.FileFilter	✔	✔
fileHidingEnabled	boolean	✔	✔
fileSelectionEnabled	boolean	✔	
fileSelectionMode	int	✔	✔
fileSystemView	filechooser.FileSystemView	✔	✔
fileView	filechooser.FileView	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	

Property	Type	Readable	Writable
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
multiSelectionEnabled	boolean	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
selectedFile	java.io.File	✔	✔
selectedFiles	java.io.File[]	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

flowLayout

Swing Class java.awt.FlowLayout

Leaf false

API Docs java.awt.FlowLayout

Basic usage

```
swing.flowLayout()
```

Properties

Property	Type	Readable	Writable
alignOnBaseline	boolean	✔	✔
alignment	int	✔	✔
hgap	int	✔	✔
vgap	int	✔	✔

---

formattedTextField

Swing Class javax.swing.JFormattedTextField

Leaf false

API Docs javax.swing.JFormattedTextField

Basic usage

```
swing.formattedTextField()
```

Properties

Property	Type	Readable	Writable
UI	plaf.TextUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
action	Action	✔	✔
actionCommand	String		✔
actionListeners	awt.event.ActionListener[]	✔	
actionMap	ActionMap	✔	✔
actions	Action[]	✔	
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	

Property	Type	Readable	Writable
border	border.Border	✔	✔
caret	text.Caret	✔	✔
caretColor	awt.Color	✔	✔
caretListeners	event.CaretListener[]	✔	
caretPosition	int	✔	✔
columns	int	✔	✔
component	Object		
componentCount	int	✔	
componentOrientation	awt.ComponentOrientation		✔
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
disabledTextColor	awt.Color	✔	✔
document	text.Document	✔	✔
doubleBuffered	boolean	✔	✔
dragEnabled	boolean	✔	✔
dropLocation	text.JTextComponent$DropLocation	✔	
dropMode	DropMode	✔	✔
editValid	boolean	✔	
editable	boolean	✔	✔
enabled	boolean	✔	✔
focusAccelerator	char	✔	✔
focusCycleRoot	boolean	✔	✔
focusLostBehavior	int	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
formatter	JFormattedTextField$AbstractFormatter	✔	
formatterFactory	JFormattedTextField$AbstractFormatterFactory	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	

Property	Type	Readable	Writable
highlighter	text.Highlighter	✔	✔
horizontalAlignment	int	✔	✔
horizontalVisibility	BoundedRangeModel	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputMethodRequests	awt.im.InputMethodRequests	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
keymap	text.Keymap	✔	✔
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
margin	awt.Insets	✔	✔
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
navigationFilter	text.NavigationFilter	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredScrollableViewportSize	awt.Dimension	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
scrollOffset	int	✔	✔
scrollableTracksViewportHeight	boolean	✔	
scrollableTracksViewportWidth	boolean	✔	
selectedText	String	✔	
selectedTextColor	awt.Color	✔	✔
selectionColor	awt.Color	✔	✔
selectionEnd	int	✔	✔
selectionStart	int	✔	✔
text	String	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔

Property	Type	Readable	Writable
validateRoot	boolean	✔	
value	Object	✔	✔
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

frame

Swing Class	javax.swing.JFrame
Leaf	false
API Docs	javax.swing.JFrame

Basic usage

```
swing.frame(title: 'My Window')
```

Properties

Property	Type	Readable	Writable
JMenuBar	JMenuBar	✔	✔
accessibleContext	javax.accessibility.AccessibleContext	✔	
active	boolean	✔	
alignmentX	float	✔	
alignmentY	float	✔	
alwaysOnTop	boolean	✔	✔
alwaysOnTopSupported	boolean	✔	
autoRequestFocus	boolean	✔	✔
background	awt.Color	✔	✔
bounds	awt.Rectangle		✔
bufferStrategy	awt.image.BufferStrategy	✔	
component	Object		
componentCount	int	✔	
components	awt.Component[]	✔	

Property	Type	Readable	Writable
containerListeners	awt.event.ContainerListener[]	✔	
contentPane	awt.Container	✔	✔
cursor	int		✔
cursorType	int	✔	
defaultCloseOperation	int	✔	✔
enabled	boolean	✔	✔
extendedState	int	✔	✔
focusCycleRoot	boolean	✔	✔
focusCycleRootAncestor	awt.Container	✔	
focusOwner	awt.Component	✔	
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
focusableWindow	boolean	✔	
focusableWindowState	boolean	✔	✔
focused	boolean	✔	
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
glassPane	awt.Component	✔	✔
graphics	awt.Graphics	✔	
iconImage	awt.Image	✔	✔
iconImages	List	✔	✔
inputContext	awt.im.InputContext	✔	
insets	awt.Insets	✔	
layeredPane	JLayeredPane	✔	✔
layout	awt.LayoutManager	✔	✔
locale	Locale	✔	
location	awt.Point		✔
locationByPlatform	boolean	✔	✔
locationRelativeTo	awt.Component		✔
maximizedBounds	awt.Rectangle	✔	✔
maximumSize	awt.Dimension	✔	
menuBar	awt.MenuBar	✔	✔
minimumSize	awt.Dimension	✔	✔
modalExclusionType	awt.Dialog$ModalExclusionType	✔	✔
mostRecentFocusOwner	awt.Component	✔	

Property	Type	Readable	Writable
name	String	✔	✔
opacity	float	✔	✔
opaque	boolean	✔	
ownedWindows	awt.Window[]	✔	
owner	awt.Window	✔	
preferredSize	awt.Dimension	✔	
resizable	boolean	✔	✔
rootPane	JRootPane	✔	
shape	awt.Shape	✔	✔
showing	boolean	✔	
size	awt.Dimension		✔
state	int	✔	✔
title	String	✔	✔
toolkit	awt.Toolkit	✔	
transferHandler	TransferHandler	✔	✔
type	awt.Window$Type	✔	✔
undecorated	boolean	✔	✔
validateRoot	boolean	✔	
visible	boolean	✔	✔
warningString	String	✔	
windowFocusListeners	awt.event.WindowFocusListener[]	✔	
windowListeners	awt.event.WindowListener[]	✔	
windowStateListeners	awt.event.WindowStateListener[]	✔	

gbc

Swing Class java.awt.GridBagConstraints

Leaf false

API Docs java.awt.GridBagConstraints

Basic usage

```
swing.gbc()
```

glue

Basic usage

```
swing.glue()
```

---

gridBagConstraints

Swing Class	`java.awt.GridBagConstraints`
Leaf	false
API Docs	java.awt.GridBagConstraints

Basic usage

```
swing.gridBagConstraints()
```

---

gridBagLayout

Swing Class	`java.awt.GridBagLayout`
Leaf	true
API Docs	java.awt.GridBagLayout

Basic usage

```
swing.gridBagLayout()
```

Properties

Property	Type	Readable	Writable
`layoutDimensions`	`int[][]`	✔	
`layoutOrigin`	`awt.Point`	✔	
`layoutWeights`	`double[][]`	✔	

---

gridLayout

Swing Class	`java.awt.GridLayout`
Leaf	false
API Docs	java.awt.GridLayout

Basic usage

```
swing.gridLayout()
```

Properties

Property	Type	Readable	Writable
columns	int	✔	✔
hgap	int	✔	✔
rows	int	✔	✔
vgap	int	✔	✔

---

hbox

Swing Class javax.swing.Box

Leaf false

API Docs javax.swing.Box

Basic usage

```
swing.hbox()
```

Properties

Property	Type	Readable	Writable
UI	plaf.ComponentUI	✔	
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔

Property	Type	Readable	Writable
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔

Property	Type	Readable	Writable
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

headerRenderer

Basic usage

```
swing.headerRenderer()
```

hglue

Basic usage

```
swing.hglue()
```

hstrut

Basic usage

```
swing.hstrut()
```

imageIcon

Swing Class	javax.swing.ImageIcon
Leaf	false
API Docs	javax.swing.ImageIcon

Basic usage

```
swing.imageIcon(url: getClass().getResource('/icon.png'))
```

Properties

Property	Type	Readable	Writable
accessibleContext	javax.accessibility.AccessibleContext	✔	
description	String	✔	✔
iconHeight	int	✔	
iconWidth	int	✔	
image	awt.Image	✔	✔
imageLoadStatus	int	✔	
imageObserver	awt.image.ImageObserver	✔	✔

internalFrame

Swing Class javax.swing.JInternalFrame

Leaf false

API Docs javax.swing.JInternalFrame

Basic usage

```
swing.internalFrame()
```

Properties

Property	Type	Readable	Writable
JMenuBar	JMenuBar	✔	✔
UI	plaf.InternalFrameUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
closable	boolean	✔	✔
closed	boolean	✔	✔

Property	Type	Readable	Writable
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
contentPane	awt.Container	✔	✔
cursor	awt.Cursor		✔
debugGraphicsOptions	int	✔	✔
defaultCloseOperation	int	✔	✔
desktopIcon	JInternalFrame$JDesktopIcon	✔	✔
desktopPane	JDesktopPane	✔	
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusCycleRootAncestor	awt.Container	✔	
focusOwner	awt.Component	✔	
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
frameIcon	Icon	✔	✔
glassPane	awt.Component	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
icon	boolean	✔	✔
iconifiable	boolean	✔	✔
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
internalFrameListeners	event.InternalFrameListener[]	✔	
lastCursor	awt.Cursor	✔	
layer	int	✔	✔
layeredPane	JLayeredPane	✔	✔
layout	awt.LayoutManager	✔	✔

Property	Type	Readable	Writable
managingFocus	boolean	✔	
maximizable	boolean	✔	✔
maximum	boolean	✔	✔
maximumSize	awt.Dimension	✔	✔
menuBar	JMenuBar	✔	✔
minimumSize	awt.Dimension	✔	✔
mostRecentFocusOwner	awt.Component	✔	
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
normalBounds	awt.Rectangle	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
resizable	boolean	✔	✔
rootPane	JRootPane	✔	
selected	boolean	✔	✔
title	String	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
warningString	String	✔	
width	int	✔	
x	int	✔	
y	int	✔	

label

Swing Class javax.swing.JLabel

Leaf	false
API Docs	javax.swing.JLabel

Basic usage

```
swing.label('Hello')
```

Properties

Property	Type	Readable	Writable
UI	plaf.LabelUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
disabledIcon	Icon	✔	✔
displayedMnemonic	int	✔	✔
displayedMnemonicIndex	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔

Property	Type	Readable	Writable
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
horizontalAlignment	int	✔	✔
horizontalTextPosition	int	✔	✔
icon	Icon	✔	✔
iconTextGap	int	✔	✔
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
labelFor	awt.Component	✔	✔
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
text	String	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
verticalAlignment	int	✔	✔
verticalTextPosition	int	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	

Property	Type	Readable	Writable
x	int	✔	
y	int	✔	

layeredPane

Swing Class javax.swing.JLayeredPane

Leaf false

API Docs javax.swing.JLayeredPane

Basic usage

```
swing.layeredPane()
```

Properties

Property	Type	Readable	Writable
UI	plaf.ComponentUI	✔	
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
component	Object		
componentCount	int	✔	
componentCountInLayer	Object		
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
componentsInLayer	Object		
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔

Property	Type	Readable	Writable
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	

Property	Type	Readable	Writable
y	int	✔	

lineBorder

Basic usage

```
swing.lineBorder()
```

list

Swing Class	javax.swing.JList
Leaf	false
API Docs	javax.swing.JList

Basic usage

```
swing.list()
```

Properties

Property	Type	Readable	Writable
UI	plaf.ListUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
anchorSelectionIndex	int	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
cellRenderer	ListCellRenderer	✔	✔
component	Object		
componentCount	int	✔	

Property	Type	Readable	Writable
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
dragEnabled	boolean	✔	✔
dropLocation	JList$DropLocation	✔	
dropMode	DropMode	✔	✔
enabled	boolean	✔	✔
firstVisibleIndex	int	✔	
fixedCellHeight	int	✔	✔
fixedCellWidth	int	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
lastVisibleIndex	int	✔	
layout	awt.LayoutManager	✔	✔
layoutOrientation	int	✔	✔
leadSelectionIndex	int	✔	
listData	Object[]		✔
listSelectionListeners	event.ListSelectionListener[]	✔	
managingFocus	boolean	✔	
maxSelectionIndex	int	✔	
maximumSize	awt.Dimension	✔	✔
minSelectionIndex	int	✔	
minimumSize	awt.Dimension	✔	✔
model	ListModel	✔	✔

Property	Type	Readable	Writable
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredScrollableViewportSize	awt.Dimension	✔	
preferredSize	awt.Dimension	✔	✔
prototypeCellValue	Object	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
scrollableTracksViewportHeight	boolean	✔	
scrollableTracksViewportWidth	boolean	✔	
selectedIndex	int	✔	✔
selectedIndices	int[]	✔	✔
selectedValue	Object	✔	
selectedValues	Object[]	✔	
selectedValuesList	List	✔	
selectionBackground	awt.Color	✔	✔
selectionEmpty	boolean	✔	
selectionForeground	awt.Color	✔	✔
selectionInterval	Object		
selectionMode	int	✔	✔
selectionModel	ListSelectionModel	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
valueIsAdjusting	boolean	✔	✔
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
visibleRowCount	int	✔	✔
width	int	✔	
x	int	✔	

Property	Type	Readable	Writable
y	int	✔	

listCellRenderer

Basic usage

```
swing.listCellRenderer()
```

loweredBevelBorder

Basic usage

```
swing.loweredBevelBorder()
```

loweredEtchedBorder

Basic usage

```
swing.loweredEtchedBorder()
```

map

Basic usage

```
swing.map()
```

matteBorder

Basic usage

```
swing.matteBorder()
```

menu

Swing Class `javax.swing.JMenu`

Leaf	false
API Docs	javax.swing.JMenu

Basic usage

```
swing.menu()
```

Properties

Property	Type	Readable	Writable
UI	plaf.ButtonUI	✔	✔
UIClassID	String	✔	
accelerator	KeyStroke	✔	✔
accessibleContext	javax.accessibility.AccessibleContext	✔	
action	Action	✔	✔
actionCommand	String	✔	✔
actionListeners	awt.event.ActionListener[]	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
armed	boolean	✔	✔
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
borderPainted	boolean	✔	✔
changeListeners	event.ChangeListener[]	✔	
component	awt.Component	✔	
componentCount	int	✔	
componentOrientation	awt.ComponentOrientation		✔
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
contentAreaFilled	boolean	✔	✔
debugGraphicsOptions	int	✔	✔
delay	int	✔	✔
disabledIcon	Icon	✔	✔
disabledSelectedIcon	Icon	✔	✔

Property	Type	Readable	Writable
displayedMnemonicIndex	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusPainted	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
hideActionText	boolean	✔	✔
horizontalAlignment	int	✔	✔
horizontalTextPosition	int	✔	✔
icon	Icon	✔	✔
iconTextGap	int	✔	✔
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
item	Object		
itemCount	int	✔	
itemListeners	awt.event.ItemListener[]	✔	
label	String	✔	✔
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
margin	awt.Insets	✔	✔
maximumSize	awt.Dimension	✔	✔
menuComponent	Object		
menuComponentCount	int	✔	
menuComponents	awt.Component[]	✔	
menuDragMouseListeners	event.MenuDragMouseListener[]	✔	
menuKeyListeners	event.MenuKeyListener[]	✔	
menuListeners	event.MenuListener[]	✔	
menuLocation	Object		

Property	Type	Readable	Writable
minimumSize	awt.Dimension	✔	✔
mnemonic	int	✔	✔
model	ButtonModel	✔	✔
multiClickThreshhold	long	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
popupMenu	JPopupMenu	✔	
popupMenuVisible	boolean	✔	✔
preferredSize	awt.Dimension	✔	✔
pressedIcon	Icon	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rolloverEnabled	boolean	✔	✔
rolloverIcon	Icon	✔	✔
rolloverSelectedIcon	Icon	✔	✔
rootPane	JRootPane	✔	
selected	boolean	✔	✔
selectedIcon	Icon	✔	✔
selectedObjects	Object[]	✔	
subElements	MenuElement[]	✔	
tearOff	boolean	✔	
text	String	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
topLevelMenu	boolean	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
verticalAlignment	int	✔	✔
verticalTextPosition	int	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	

Property	Type	Readable	Writable
x	int	✔	
y	int	✔	

menuBar

Swing Class javax.swing.JMenuBar

Leaf false

API Docs javax.swing.JMenuBar

Basic usage

```
swing.menuBar()
```

Properties

Property	Type	Readable	Writable
UI	plaf.MenuBarUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
borderPainted	boolean	✔	✔
component	awt.Component	✔	
componentAtIndex	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔

Property	Type	Readable	Writable
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
helpMenu	JMenu	✔	✔
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
margin	awt.Insets	✔	✔
maximumSize	awt.Dimension	✔	✔
menu	Object		
menuCount	int	✔	
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
selected	boolean	✔	
selectionModel	SingleSelectionModel	✔	✔
subElements	MenuElement[]	✔	
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔

Property	Type	Readable	Writable
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

menuItem

Swing Class javax.swing.JMenuItem

Leaf false

API Docs javax.swing.JMenuItem

Basic usage

```
swing.menuItem('Click Me')
```

Properties

Property	Type	Readable	Writable
UI	plaf.ButtonUI	✔	✔
UIClassID	String	✔	
accelerator	KeyStroke	✔	✔
accessibleContext	javax.accessibility.AccessibleContext	✔	
action	Action	✔	✔
actionCommand	String	✔	✔
actionListeners	awt.event.ActionListener[]	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
armed	boolean	✔	✔
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔

Property	Type	Readable	Writable
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
borderPainted	boolean	✔	✔
changeListeners	event.ChangeListener[]	✔	
component	awt.Component	✔	
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
contentAreaFilled	boolean	✔	✔
debugGraphicsOptions	int	✔	✔
disabledIcon	Icon	✔	✔
disabledSelectedIcon	Icon	✔	✔
displayedMnemonicIndex	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusPainted	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
hideActionText	boolean	✔	✔
horizontalAlignment	int	✔	✔
horizontalTextPosition	int	✔	✔
icon	Icon	✔	✔
iconTextGap	int	✔	✔
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
itemListeners	awt.event.ItemListener[]	✔	
label	String	✔	✔

Property	Type	Readable	Writable
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
margin	awt.Insets	✔	✔
maximumSize	awt.Dimension	✔	✔
menuDragMouseListeners	event.MenuDragMouseListener[]	✔	
menuKeyListeners	event.MenuKeyListener[]	✔	
minimumSize	awt.Dimension	✔	✔
mnemonic	int	✔	✔
model	ButtonModel	✔	✔
multiClickThreshhold	long	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
pressedIcon	Icon	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rolloverEnabled	boolean	✔	✔
rolloverIcon	Icon	✔	✔
rolloverSelectedIcon	Icon	✔	✔
rootPane	JRootPane	✔	
selected	boolean	✔	✔
selectedIcon	Icon	✔	✔
selectedObjects	Object[]	✔	
subElements	MenuElement[]	✔	
text	String	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
verticalAlignment	int	✔	✔
verticalTextPosition	int	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔

Property	Type	Readable	Writable
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

noparent

Basic usage

```
swing.noparent()
```

onRender

Basic usage

```
swing.onRender()
```

optionPane

Swing Class	javax.swing.JOptionPane
Leaf	false
API Docs	javax.swing.JOptionPane

Basic usage

```
swing.optionPane()
```

Properties

Property	Type	Readable	Writable
UI	plaf.OptionPaneUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	

Property	Type	Readable	Writable
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
icon	Icon	✔	✔
inheritsPopupMenu	boolean	✔	✔
initialSelectionValue	Object	✔	✔
initialValue	Object	✔	✔
inputMap	InputMap	✔	
inputValue	Object	✔	✔
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
maxCharactersPerLineCount	int	✔	
maximumSize	awt.Dimension	✔	✔
message	Object	✔	✔
messageType	int	✔	✔
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔

Property	Type	Readable	Writable
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
optionType	int	✔	✔
options	Object[]	✔	✔
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
selectionValues	Object[]	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
value	Object	✔	✔
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
wantsInput	boolean	✔	✔
width	int	✔	
x	int	✔	
y	int	✔	

overlayLayout

Swing Class javax.swing.OverlayLayout

Leaf false

API Docs javax.swing.OverlayLayout

Basic usage

```
swing.overlayLayout()
```

Properties

Property	Type	Readable	Writable
target	awt.Container	✔	

---

panel

Swing Class javax.swing.JPanel

Leaf false

API Docs javax.swing.JPanel

Basic usage

```
swing.panel()
```

Properties

Property	Type	Readable	Writable
UI	plaf.PanelUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔

Property	Type	Readable	Writable
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

passwordField

Swing Class	javax.swing.JPasswordField
Leaf	false
API Docs	javax.swing.JPasswordField

Basic usage

```
swing.passwordField(text: 'initial text', columns: 20)
```

Properties

Property	Type	Readable	Writable
UI	plaf.TextUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
action	Action	✔	✔
actionCommand	String		✔
actionListeners	awt.event.ActionListener[]	✔	
actionMap	ActionMap	✔	✔
actions	Action[]	✔	
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
caret	text.Caret	✔	✔
caretColor	awt.Color	✔	✔
caretListeners	event.CaretListener[]	✔	
caretPosition	int	✔	✔
columns	int	✔	✔
component	Object		
componentCount	int	✔	
componentOrientation	awt.ComponentOrientation		✔
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	

Property	Type	Readable	Writable
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
disabledTextColor	awt.Color	✔	✔
document	text.Document	✔	✔
doubleBuffered	boolean	✔	✔
dragEnabled	boolean	✔	✔
dropLocation	text.JTextComponent$DropLocation	✔	
dropMode	DropMode	✔	✔
echoChar	char	✔	✔
editable	boolean	✔	✔
enabled	boolean	✔	✔
focusAccelerator	char	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
highlighter	text.Highlighter	✔	✔
horizontalAlignment	int	✔	✔
horizontalVisibility	BoundedRangeModel	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputMethodRequests	awt.im.InputMethodRequests	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
keymap	text.Keymap	✔	✔
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
margin	awt.Insets	✔	✔
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
navigationFilter	text.NavigationFilter	✔	✔

Property	Type	Readable	Writable
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
password	char[]	✔	
preferredScrollableViewportSize	awt.Dimension	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
scrollOffset	int	✔	✔
scrollableTracksViewportHeight	boolean	✔	
scrollableTracksViewportWidth	boolean	✔	
selectedText	String	✔	
selectedTextColor	awt.Color	✔	✔
selectionColor	awt.Color	✔	✔
selectionEnd	int	✔	✔
selectionStart	int	✔	✔
text	String	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

popupMenu

Swing Class javax.swing.JPopupMenu

Leaf false

API Docs javax.swing.JPopupMenu

Basic usage

```
swing.popupMenu()
```

Properties

Property	Type	Readable	Writable
UI	plaf.PopupMenuUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
borderPainted	boolean	✔	✔
component	awt.Component	✔	
componentAtIndex	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	

Property	Type	Readable	Writable
height	int	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
invoker	awt.Component	✔	✔
label	String	✔	✔
layout	awt.LayoutManager	✔	✔
lightWeightPopupEnabled	boolean	✔	✔
location	Object		
managingFocus	boolean	✔	
margin	awt.Insets	✔	
maximumSize	awt.Dimension	✔	✔
menuKeyListeners	event.MenuKeyListener[]	✔	
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
popupMenuListeners	event.PopupMenuListener[]	✔	
popupSize	awt.Dimension		✔
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
selected	awt.Component		✔
selectionModel	SingleSelectionModel	✔	✔
subElements	MenuElement[]	✔	
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	

Property	Type	Readable	Writable
width	int	✔	
x	int	✔	
y	int	✔	

prepareEditor

Basic usage

```
swing.prepareEditor()
```

progressBar

Swing Class	javax.swing.JProgressBar
Leaf	false
API Docs	javax.swing.JProgressBar

Basic usage

```
swing.progressBar()
```

Properties

Property	Type	Readable	Writable
UI	plaf.ProgressBarUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
borderPainted	boolean	✔	✔
changeListeners	event.ChangeListener[]	✔	

Property	Type	Readable	Writable
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
indeterminate	boolean	✔	✔
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
maximum	int	✔	✔
maximumSize	awt.Dimension	✔	✔
minimum	int	✔	✔
minimumSize	awt.Dimension	✔	✔
model	BoundedRangeModel	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
orientation	int	✔	✔
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
percentComplete	double	✔	

Property	Type	Readable	Writable
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
string	String	✔	✔
stringPainted	boolean	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
value	int	✔	✔
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

propertyColumn

Basic usage

```
swing.propertyColumn()
```

radioButton

Swing Class	javax.swing.JRadioButton
Leaf	false
API Docs	javax.swing.JRadioButton

Basic usage

```
swing.radioButton('Click Me')
```

Properties

Property	Type	Readable	Writable
UI	plaf.ButtonUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
action	Action	✔	✔
actionCommand	String	✔	✔
actionListeners	awt.event.ActionListener[]	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
borderPainted	boolean	✔	✔
changeListeners	event.ChangeListener[]	✔	
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
contentAreaFilled	boolean	✔	✔
debugGraphicsOptions	int	✔	✔
disabledIcon	Icon	✔	✔
disabledSelectedIcon	Icon	✔	✔
displayedMnemonicIndex	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusPainted	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔

Property	Type	Readable	Writable
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
hideActionText	boolean	✔	✔
horizontalAlignment	int	✔	✔
horizontalTextPosition	int	✔	✔
icon	Icon	✔	✔
iconTextGap	int	✔	✔
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
itemListeners	awt.event.ItemListener[]	✔	
label	String	✔	✔
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
margin	awt.Insets	✔	✔
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
mnemonic	int	✔	✔
model	ButtonModel	✔	✔
multiClickThreshhold	long	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
pressedIcon	Icon	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rolloverEnabled	boolean	✔	✔
rolloverIcon	Icon	✔	✔
rolloverSelectedIcon	Icon	✔	✔
rootPane	JRootPane	✔	
selected	boolean	✔	✔
selectedIcon	Icon	✔	✔

Property	Type	Readable	Writable
selectedObjects	Object[]	✔	
text	String	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
verticalAlignment	int	✔	✔
verticalTextPosition	int	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

radioButtonMenuItem

Swing Class javax.swing.JRadioButtonMenuItem

Leaf false

API Docs javax.swing.JRadioButtonMenuItem

Basic usage

```
swing.radioButtonMenuItem('Click Me')
```

Properties

Property	Type	Readable	Writable
UI	plaf.ButtonUI	✔	✔
UIClassID	String	✔	
accelerator	KeyStroke	✔	✔
accessibleContext	javax.accessibility.AccessibleContext	✔	
action	Action	✔	✔
actionCommand	String	✔	✔
actionListeners	awt.event.ActionListener[]	✔	
actionMap	ActionMap	✔	✔

Property	Type	Readable	Writable
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
armed	boolean	✔	✔
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
borderPainted	boolean	✔	✔
changeListeners	event.ChangeListener[]	✔	
component	awt.Component	✔	
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
contentAreaFilled	boolean	✔	✔
debugGraphicsOptions	int	✔	✔
disabledIcon	Icon	✔	✔
disabledSelectedIcon	Icon	✔	✔
displayedMnemonicIndex	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusPainted	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
hideActionText	boolean	✔	✔
horizontalAlignment	int	✔	✔
horizontalTextPosition	int	✔	✔
icon	Icon	✔	✔
iconTextGap	int	✔	✔

Property	Type	Readable	Writable
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
itemListeners	awt.event.ItemListener[]	✔	
label	String	✔	✔
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
margin	awt.Insets	✔	✔
maximumSize	awt.Dimension	✔	✔
menuDragMouseListeners	event.MenuDragMouseListener[]	✔	
menuKeyListeners	event.MenuKeyListener[]	✔	
minimumSize	awt.Dimension	✔	✔
mnemonic	int	✔	✔
model	ButtonModel	✔	✔
multiClickThreshhold	long	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
pressedIcon	Icon	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rolloverEnabled	boolean	✔	✔
rolloverIcon	Icon	✔	✔
rolloverSelectedIcon	Icon	✔	✔
rootPane	JRootPane	✔	
selected	boolean	✔	✔
selectedIcon	Icon	✔	✔
selectedObjects	Object[]	✔	
subElements	MenuElement[]	✔	
text	String	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔

Property	Type	Readable	Writable
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
verticalAlignment	int	✔	✔
verticalTextPosition	int	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

raisedBevelBorder

Basic usage

```
swing.raisedBevelBorder()
```

raisedEtchedBorder

Basic usage

```
swing.raisedEtchedBorder()
```

rigidArea

Basic usage

```
swing.rigidArea()
```

scrollBar

Swing Class javax.swing.JScrollBar

Leaf false

API Docs javax.swing.JScrollBar

Basic usage

```
swing.scrollBar()
```

Properties

Property	Type	Readable	Writable
UI	plaf.ScrollBarUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
adjustmentListeners	awt.event.AdjustmentListener[]	✔	
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
blockIncrement	int	✔	✔
border	border.Border	✔	✔
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
inheritsPopupMenu	boolean	✔	✔

Property	Type	Readable	Writable
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
maximum	int	✔	✔
maximumSize	awt.Dimension	✔	✔
minimum	int	✔	✔
minimumSize	awt.Dimension	✔	✔
model	BoundedRangeModel	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
orientation	int	✔	✔
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
unitIncrement	int	✔	✔
validateRoot	boolean	✔	
value	int	✔	✔
valueIsAdjusting	boolean	✔	✔
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleAmount	int	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

scrollPane

Swing Class	`javax.swing.JScrollPane`
Leaf	false
API Docs	javax.swing.JScrollPane

Basic usage

```
swing.scrollPane()
```

Properties

Property	Type	Readable	Writable
UI	plaf.ScrollPaneUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
columnHeader	JViewport	✔	✔
columnHeaderView	awt.Component		✔
component	Object		
componentCount	int	✔	
componentOrientation	awt.ComponentOrientation		✔
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔

Property	Type	Readable	Writable
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
horizontalScrollBar	JScrollBar	✔	✔
horizontalScrollBarPolicy	int	✔	✔
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
rowHeader	JViewport	✔	✔
rowHeaderView	awt.Component		✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
verticalScrollBar	JScrollBar	✔	✔
verticalScrollBarPolicy	int	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
viewport	JViewport	✔	✔
viewportBorder	border.Border	✔	✔

Property	Type	Readable	Writable
viewportBorderBounds	awt.Rectangle	✔	
viewportView	awt.Component		✔
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
wheelScrollingEnabled	boolean	✔	✔
width	int	✔	
x	int	✔	
y	int	✔	

separator

Swing Class javax.swing.JSeparator

Leaf false

API Docs javax.swing.JSeparator

Basic usage

```
swing.separator()
```

Properties

Property	Type	Readable	Writable
UI	plaf.SeparatorUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	

Property	Type	Readable	Writable
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
orientation	int	✔	✔
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔

Property	Type	Readable	Writable
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

slider

Swing Class	javax.swing.JSlider
Leaf	false
API Docs	javax.swing.JSlider

Basic usage

```
swing.slider()
```

Properties

Property	Type	Readable	Writable
UI	plaf.SliderUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
changeListeners	event.ChangeListener[]	✔	
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	

Property	Type	Readable	Writable
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
extent	int	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
inverted	boolean	✔	✔
labelTable	Dictionary	✔	✔
layout	awt.LayoutManager	✔	✔
majorTickSpacing	int	✔	✔
managingFocus	boolean	✔	
maximum	int	✔	✔
maximumSize	awt.Dimension	✔	✔
minimum	int	✔	✔
minimumSize	awt.Dimension	✔	✔
minorTickSpacing	int	✔	✔
model	BoundedRangeModel	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
orientation	int	✔	✔
paintLabels	boolean	✔	✔
paintTicks	boolean	✔	✔
paintTrack	boolean	✔	✔

Property	Type	Readable	Writable
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
snapToTicks	boolean	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
value	int	✔	✔
valueIsAdjusting	boolean	✔	✔
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

spinner

Swing Class javax.swing.JSpinner

Leaf false

API Docs javax.swing.JSpinner

Basic usage

```
swing.spinner()
```

Properties

Property	Type	Readable	Writable
UI	plaf.SpinnerUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	

Property	Type	Readable	Writable
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
changeListeners	event.ChangeListener[]	✔	
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
editor	JComponent	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
model	SpinnerModel	✔	✔
name	String	✔	✔

Property	Type	Readable	Writable
nextFocusableComponent	awt.Component	✔	✔
nextValue	Object	✔	
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
previousValue	Object	✔	
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
value	Object	✔	✔
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

spinnerDateModel

Swing Class javax.swing.SpinnerDateModel

Leaf false

API Docs javax.swing.SpinnerDateModel

Basic usage

```
swing.spinnerDateModel()
```

Properties

Property	Type	Readable	Writable
calendarField	int	✔	✔
changeListeners	event.ChangeListener[]	✔	
date	Date	✔	
end	Comparable	✔	✔
nextValue	Object	✔	
previousValue	Object	✔	
start	Comparable	✔	✔
value	Object	✔	✔

spinnerListModel

Swing Class javax.swing.SpinnerListModel

Leaf false

API Docs javax.swing.SpinnerListModel

Basic usage

```
swing.spinnerListModel()
```

Properties

Property	Type	Readable	Writable
changeListeners	event.ChangeListener[]	✔	
list	List	✔	✔
nextValue	Object	✔	
previousValue	Object	✔	
value	Object	✔	✔

spinnerNumberModel

Swing Class javax.swing.SpinnerNumberModel

Leaf false

API Docs javax.swing.SpinnerNumberModel

Basic usage

```
swing.spinnerNumberModel()
```

Properties

Property	Type	Readable	Writable
changeListeners	event.ChangeListener[]	✔	
maximum	Comparable	✔	✔
minimum	Comparable	✔	✔
nextValue	Object	✔	
number	Number	✔	
previousValue	Object	✔	
stepSize	Number	✔	✔
value	Object	✔	✔

splitPane

Swing Class javax.swing.JSplitPane

Leaf false

API Docs javax.swing.JSplitPane

Basic usage

```
swing.splitPane()
```

Properties

Property	Type	Readable	Writable
UI	plaf.SplitPaneUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
bottomComponent	awt.Component	✔	✔
component	Object		

Property	Type	Readable	Writable
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
continuousLayout	boolean	✔	✔
debugGraphicsOptions	int	✔	✔
dividerLocation	int	✔	✔
dividerSize	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
lastDividerLocation	int	✔	✔
layout	awt.LayoutManager	✔	✔
leftComponent	awt.Component	✔	✔
managingFocus	boolean	✔	
maximumDividerLocation	int	✔	
maximumSize	awt.Dimension	✔	✔
minimumDividerLocation	int	✔	
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
oneTouchExpandable	boolean	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
orientation	int	✔	✔

Property	Type	Readable	Writable
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
resizeWeight	double	✔	✔
rightComponent	awt.Component	✔	✔
rootPane	JRootPane	✔	
toolTipText	String	✔	✔
topComponent	awt.Component	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

springLayout

Swing Class javax.swing.SpringLayout

Leaf false

API Docs javax.swing.SpringLayout

Basic usage

```
swing.springLayout()
```

tabbedPane

Swing Class javax.swing.JTabbedPane

Leaf false

API Docs javax.swing.JTabbedPane

Basic usage

```
swing.tabbedPane()
```

Properties

Property	Type	Readable	Writable
UI	plaf.TabbedPaneUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
backgroundAt	Object		
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
boundsAt	Object		
changeListeners	event.ChangeListener[]	✔	
component	Object		
componentAt	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
disabledIconAt	Object		
displayedMnemonicIndexAt	Object		
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
enabledAt	Object		
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔

Property	Type	Readable	Writable
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
foregroundAt	Object		
graphics	awt.Graphics	✔	
height	int	✔	
iconAt	Object		
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
mnemonicAt	Object		
model	SingleSelectionModel	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
selectedComponent	awt.Component	✔	✔
selectedIndex	int	✔	✔
tabComponentAt	Object		
tabCount	int	✔	
tabLayoutPolicy	int	✔	✔
tabPlacement	int	✔	✔
tabRunCount	int	✔	
titleAt	Object		
toolTipText	String	✔	✔
toolTipTextAt	Object		
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔

Property	Type	Readable	Writable
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

table

Swing Class javax.swing.JTable

Leaf false

API Docs javax.swing.JTable

Basic usage

```
swing.table()
```

Properties

Property	Type	Readable	Writable
UI	plaf.TableUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoCreateColumnsFromModel	boolean	✔	✔
autoCreateRowSorter	boolean	✔	✔
autoResizeMode	int	✔	✔
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔

Property	Type	Readable	Writable
cellEditor	table.TableCellEditor	✔	✔
cellSelectionEnabled	boolean	✔	✔
columnClass	Object		
columnCount	int	✔	
columnModel	table.TableColumnModel	✔	✔
columnName	Object		
columnSelectionAllowed	boolean	✔	✔
columnSelectionInterval	Object		
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
dragEnabled	boolean	✔	✔
dropLocation	JTable$DropLocation	✔	
dropMode	DropMode	✔	✔
editing	boolean	✔	
editingColumn	int	✔	✔
editingRow	int	✔	✔
editorComponent	awt.Component	✔	
enabled	boolean	✔	✔
fillsViewportHeight	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
gridColor	awt.Color	✔	✔
height	int	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔

Property	Type	Readable	Writable
insets	awt.Insets	✔	
intercellSpacing	awt.Dimension	✔	✔
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
model	table.TableModel	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredScrollableViewportSize	awt.Dimension	✔	✔
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
rowCount	int	✔	
rowHeight	int	✔	✔
rowMargin	int	✔	✔
rowSelectionAllowed	boolean	✔	✔
rowSelectionInterval	Object		
rowSorter	RowSorter	✔	✔
scrollableTracksViewportHeight	boolean	✔	
scrollableTracksViewportWidth	boolean	✔	
selectedColumn	int	✔	
selectedColumnCount	int	✔	
selectedColumns	int[]	✔	
selectedRow	int	✔	
selectedRowCount	int	✔	
selectedRows	int[]	✔	
selectionBackground	awt.Color	✔	✔
selectionForeground	awt.Color	✔	✔
selectionMode	int		✔
selectionModel	ListSelectionModel	✔	✔
showGrid	boolean		✔
showHorizontalLines	boolean	✔	✔

Property	Type	Readable	Writable
showVerticalLines	boolean	✔	✔
surrendersFocusOnKeystroke	boolean	✔	✔
tableHeader	table.JTableHeader	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
updateSelectionOnSort	boolean	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

tableCellRenderer

Basic usage

```
swing.tableCellRenderer()
```

tableColumn

Swing Class javax.swing.table.TableColumn

Leaf false

API Docs javax.swing.table.TableColumn

Basic usage

```
swing.tableColumn()
```

Properties

Property	Type	Readable	Writable
cellEditor	table.TableCellEditor	✔	✔
cellRenderer	table.TableCellRenderer	✔	✔

Property	Type	Readable	Writable
headerRenderer	table.TableCellRenderer	✔	✔
headerValue	Object	✔	✔
identifier	Object	✔	✔
maxWidth	int	✔	✔
minWidth	int	✔	✔
modelIndex	int	✔	✔
preferredWidth	int	✔	✔
propertyChangeListeners	java.beans.PropertyChangeListener[]	✔	
resizable	boolean	✔	✔
width	int	✔	✔

tableLayout

Basic usage

```
swing.tableLayout()
```

tableModel

Basic usage

```
swing.tableModel()
```

td

Basic usage

```
swing.td()
```

textArea

Swing Class	javax.swing.JTextArea
Leaf	false
API Docs	javax.swing.JTextArea

Basic usage

```
swing.textArea(text: 'initial text', columns: 20)
```

Properties

Property	Type	Readable	Writable
UI	plaf.TextUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
actions	Action[]	✔	
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
caret	text.Caret	✔	✔
caretColor	awt.Color	✔	✔
caretListeners	event.CaretListener[]	✔	
caretPosition	int	✔	✔
columns	int	✔	✔
component	Object		
componentCount	int	✔	
componentOrientation	awt.ComponentOrientation		✔
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
disabledTextColor	awt.Color	✔	✔
document	text.Document	✔	✔
doubleBuffered	boolean	✔	✔
dragEnabled	boolean	✔	✔
dropLocation	text.JTextComponent$DropLocation	✔	
dropMode	DropMode	✔	✔
editable	boolean	✔	✔

Property	Type	Readable	Writable
enabled	boolean	✔	✔
focusAccelerator	char	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
highlighter	text.Highlighter	✔	✔
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputMethodRequests	awt.im.InputMethodRequests	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
keymap	text.Keymap	✔	✔
layout	awt.LayoutManager	✔	✔
lineCount	int	✔	
lineEndOffset	Object		
lineOfOffset	Object		
lineStartOffset	Object		
lineWrap	boolean	✔	✔
managingFocus	boolean	✔	
margin	awt.Insets	✔	✔
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
navigationFilter	text.NavigationFilter	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredScrollableViewportSize	awt.Dimension	✔	
preferredSize	awt.Dimension	✔	✔

Property	Type	Readable	Writable
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
rows	int	✔	✔
scrollableTracksViewportHeight	boolean	✔	
scrollableTracksViewportWidth	boolean	✔	
selectedText	String	✔	
selectedTextColor	awt.Color	✔	✔
selectionColor	awt.Color	✔	✔
selectionEnd	int	✔	✔
selectionStart	int	✔	✔
tabSize	int	✔	✔
text	String	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
wrapStyleWord	boolean	✔	✔
x	int	✔	
y	int	✔	

textField

Swing Class javax.swing.JTextField

Leaf false

API Docs javax.swing.JTextField

Basic usage

```
swing.textField(text: 'initial text', columns: 20)
```

Properties

Property	Type	Readable	Writable
UI	plaf.TextUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
action	Action	✔	✔
actionCommand	String		✔
actionListeners	awt.event.ActionListener[]	✔	
actionMap	ActionMap	✔	✔
actions	Action[]	✔	
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
caret	text.Caret	✔	✔
caretColor	awt.Color	✔	✔
caretListeners	event.CaretListener[]	✔	
caretPosition	int	✔	✔
columns	int	✔	✔
component	Object		
componentCount	int	✔	
componentOrientation	awt.ComponentOrientation		✔
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
disabledTextColor	awt.Color	✔	✔
document	text.Document	✔	✔
doubleBuffered	boolean	✔	✔
dragEnabled	boolean	✔	✔
dropLocation	text.JTextComponent$DropLocation	✔	
dropMode	DropMode	✔	✔
editable	boolean	✔	✔
enabled	boolean	✔	✔

Property	Type	Readable	Writable
focusAccelerator	char	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
highlighter	text.Highlighter	✔	✔
horizontalAlignment	int	✔	✔
horizontalVisibility	BoundedRangeModel	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputMethodRequests	awt.im.InputMethodRequests	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
keymap	text.Keymap	✔	✔
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
margin	awt.Insets	✔	✔
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
navigationFilter	text.NavigationFilter	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredScrollableViewportSize	awt.Dimension	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
scrollOffset	int	✔	✔

Property	Type	Readable	Writable
scrollableTracksViewportHeight	boolean	✔	
scrollableTracksViewportWidth	boolean	✔	
selectedText	String	✔	
selectedTextColor	awt.Color	✔	✔
selectionColor	awt.Color	✔	✔
selectionEnd	int	✔	✔
selectionStart	int	✔	✔
text	String	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

textPane

Swing Class javax.swing.JTextPane

Leaf false

API Docs javax.swing.JTextPane

Basic usage

```
swing.textPane(text: 'initial text', columns: 20)
```

Properties

Property	Type	Readable	Writable
UI	plaf.TextUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔

Property	Type	Readable	Writable
actions	Action[]	✔	
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
caret	text.Caret	✔	✔
caretColor	awt.Color	✔	✔
caretListeners	event.CaretListener[]	✔	
caretPosition	int	✔	✔
characterAttributes	text.AttributeSet	✔	
component	Object		
componentCount	int	✔	
componentOrientation	awt.ComponentOrientation		✔
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
contentType	String	✔	✔
debugGraphicsOptions	int	✔	✔
disabledTextColor	awt.Color	✔	✔
document	text.Document	✔	✔
doubleBuffered	boolean	✔	✔
dragEnabled	boolean	✔	✔
dropLocation	text.JTextComponent$DropLocation	✔	
dropMode	DropMode	✔	✔
editable	boolean	✔	✔
editorKit	text.EditorKit	✔	✔
enabled	boolean	✔	✔
focusAccelerator	char	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔

Property	Type	Readable	Writable
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
highlighter	text.Highlighter	✔	✔
hyperlinkListeners	event.HyperlinkListener[]	✔	
inheritsPopupMenu	boolean	✔	✔
inputAttributes	text.MutableAttributeSet	✔	
inputMap	InputMap	✔	
inputMethodRequests	awt.im.InputMethodRequests	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
keymap	text.Keymap	✔	✔
layout	awt.LayoutManager	✔	✔
logicalStyle	text.Style	✔	✔
managingFocus	boolean	✔	
margin	awt.Insets	✔	✔
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
navigationFilter	text.NavigationFilter	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
page	java.net.URL	✔	✔
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
paragraphAttributes	text.AttributeSet	✔	
preferredScrollableViewportSize	awt.Dimension	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
scrollableTracksViewportHeight	boolean	✔	
scrollableTracksViewportWidth	boolean	✔	
selectedText	String	✔	
selectedTextColor	awt.Color	✔	✔
selectionColor	awt.Color	✔	✔

Property	Type	Readable	Writable
selectionEnd	int	✔	✔
selectionStart	int	✔	✔
styledDocument	text.StyledDocument	✔	✔
text	String	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

titledBorder

Basic usage

```
swing.titledBorder()
```

toggleButton

Swing Class	javax.swing.JToggleButton
Leaf	false
API Docs	javax.swing.JToggleButton

Basic usage

```
swing.toggleButton('Click Me')
```

Properties

Property	Type	Readable	Writable
UI	plaf.ButtonUI	✔	✔
UIClassID	String	✔	

Property	Type	Readable	Writable
accessibleContext	javax.accessibility.AccessibleContext	✔	
action	Action	✔	✔
actionCommand	String	✔	✔
actionListeners	awt.event.ActionListener[]	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
borderPainted	boolean	✔	✔
changeListeners	event.ChangeListener[]	✔	
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
contentAreaFilled	boolean	✔	✔
debugGraphicsOptions	int	✔	✔
disabledIcon	Icon	✔	✔
disabledSelectedIcon	Icon	✔	✔
displayedMnemonicIndex	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusPainted	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	

Property	Type	Readable	Writable
hideActionText	boolean	✔	✔
horizontalAlignment	int	✔	✔
horizontalTextPosition	int	✔	✔
icon	Icon	✔	✔
iconTextGap	int	✔	✔
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
itemListeners	awt.event.ItemListener[]	✔	
label	String	✔	✔
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
margin	awt.Insets	✔	✔
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
mnemonic	int	✔	✔
model	ButtonModel	✔	✔
multiClickThreshhold	long	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
pressedIcon	Icon	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rolloverEnabled	boolean	✔	✔
rolloverIcon	Icon	✔	✔
rolloverSelectedIcon	Icon	✔	✔
rootPane	JRootPane	✔	
selected	boolean	✔	✔
selectedIcon	Icon	✔	✔
selectedObjects	Object[]	✔	
text	String	✔	✔
toolTipText	String	✔	✔

Property	Type	Readable	Writable
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
verticalAlignment	int	✔	✔
verticalTextPosition	int	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

toolBar

Swing Class javax.swing.JToolBar

Leaf false

API Docs javax.swing.JToolBar

Basic usage

```
swing.toolBar()
```

Properties

Property	Type	Readable	Writable
UI	plaf.ToolBarUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	

Property	Type	Readable	Writable
border	border.Border	✔	✔
borderPainted	boolean	✔	✔
component	Object		
componentAtIndex	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
floatable	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
margin	awt.Insets	✔	✔
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
orientation	int	✔	✔
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	

Property	Type	Readable	Writable
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rollover	boolean	✔	✔
rootPane	JRootPane	✔	
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

tr

Basic usage

```
swing.tr()
```

tree

Swing Class javax.swing.JTree

Leaf false

API Docs javax.swing.JTree

Basic usage

```
swing.tree()
```

Properties

Property	Type	Readable	Writable
UI	plaf.TreeUI	✔	✔

Property	Type	Readable	Writable
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
anchorSelectionPath	tree.TreePath	✔	✔
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
cellEditor	tree.TreeCellEditor	✔	✔
cellRenderer	tree.TreeCellRenderer	✔	✔
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
dragEnabled	boolean	✔	✔
dropLocation	JTree$DropLocation	✔	
dropMode	DropMode	✔	✔
editable	boolean	✔	✔
editing	boolean	✔	
editingPath	tree.TreePath	✔	
enabled	boolean	✔	✔
expandsSelectedPaths	boolean	✔	✔
fixedRowHeight	boolean	✔	
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔

Property	Type	Readable	Writable
graphics	awt.Graphics	✔	
height	int	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
invokesStopCellEditing	boolean	✔	✔
largeModel	boolean	✔	✔
lastSelectedPathComponent	Object	✔	
layout	awt.LayoutManager	✔	✔
leadSelectionPath	tree.TreePath	✔	✔
leadSelectionRow	int	✔	
managingFocus	boolean	✔	
maxSelectionRow	int	✔	
maximumSize	awt.Dimension	✔	✔
minSelectionRow	int	✔	
minimumSize	awt.Dimension	✔	✔
model	tree.TreeModel	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
pathForRow	Object		
preferredScrollableViewportSize	awt.Dimension	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
rootVisible	boolean	✔	✔
rowBounds	Object		
rowCount	int	✔	
rowHeight	int	✔	✔
scrollableTracksViewportHeight	boolean	✔	
scrollableTracksViewportWidth	boolean	✔	
scrollsOnExpand	boolean	✔	✔
selectionCount	int	✔	

Property	Type	Readable	Writable
selectionEmpty	boolean	✔	
selectionInterval	Object		
selectionModel	tree.TreeSelectionModel	✔	✔
selectionPath	tree.TreePath	✔	✔
selectionPaths	tree.TreePath[]	✔	✔
selectionRow	int		✔
selectionRows	int[]	✔	✔
showsRootHandles	boolean	✔	✔
toggleClickCount	int	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
treeExpansionListeners	event.TreeExpansionListener[]	✔	
treeSelectionListeners	event.TreeSelectionListener[]	✔	
treeWillExpandListeners	event.TreeWillExpandListener[]	✔	
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
visibleRowCount	int	✔	✔
width	int	✔	
x	int	✔	
y	int	✔	

vbox

Swing Class javax.swing.Box

Leaf false

API Docs javax.swing.Box

Basic usage

```
swing.vbox()
```

Properties

Property	Type	Readable	Writable
UI	plaf.ComponentUI	✔	
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔

Property	Type	Readable	Writable
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

vglue

Basic usage

```
swing.vglue()
```

viewport

Swing Class	javax.swing.JViewport
Leaf	false
API Docs	javax.swing.JViewport

Basic usage

```
swing.viewport()
```

Properties

Property	Type	Readable	Writable
UI	plaf.ViewportUI	✔	✔
UIClassID	String	✔	
accessibleContext	javax.accessibility.AccessibleContext	✔	
actionMap	ActionMap	✔	✔
alignmentX	float	✔	✔
alignmentY	float	✔	✔
ancestorListeners	event.AncestorListener[]	✔	
autoscrolls	boolean	✔	✔
background	awt.Color	✔	✔
backingStoreEnabled	boolean	✔	✔
baselineResizeBehavior	awt.Component$BaselineResizeBehavior	✔	
border	border.Border	✔	✔
changeListeners	event.ChangeListener[]	✔	
component	Object		
componentCount	int	✔	
componentPopupMenu	JPopupMenu	✔	✔
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
debugGraphicsOptions	int	✔	✔
doubleBuffered	boolean	✔	✔
enabled	boolean	✔	✔
extentSize	awt.Dimension	✔	✔
focusCycleRoot	boolean	✔	✔
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
graphics	awt.Graphics	✔	
height	int	✔	

Property	Type	Readable	Writable
inheritsPopupMenu	boolean	✔	✔
inputMap	InputMap	✔	
inputVerifier	InputVerifier	✔	✔
insets	awt.Insets	✔	
layout	awt.LayoutManager	✔	✔
managingFocus	boolean	✔	
maximumSize	awt.Dimension	✔	✔
minimumSize	awt.Dimension	✔	✔
name	String	✔	✔
nextFocusableComponent	awt.Component	✔	✔
opaque	boolean	✔	✔
optimizedDrawingEnabled	boolean	✔	
paintingForPrint	boolean	✔	
paintingTile	boolean	✔	
preferredSize	awt.Dimension	✔	✔
registeredKeyStrokes	KeyStroke[]	✔	
requestFocusEnabled	boolean	✔	✔
rootPane	JRootPane	✔	
scrollMode	int	✔	✔
toolTipText	String	✔	✔
topLevelAncestor	awt.Container	✔	
transferHandler	TransferHandler	✔	✔
validateRoot	boolean	✔	
verifyInputWhenFocusTarget	boolean	✔	✔
vetoableChangeListeners	java.beans.VetoableChangeListener[]	✔	
view	awt.Component	✔	✔
viewPosition	awt.Point	✔	✔
viewRect	awt.Rectangle	✔	
viewSize	awt.Dimension	✔	✔
visible	boolean	✔	✔
visibleRect	awt.Rectangle	✔	
width	int	✔	
x	int	✔	
y	int	✔	

vstrut

Basic usage

```
swing.vstrut()
```

---

widget

Swing Class	`java.awt.Component`
Leaf	true
API Docs	java.awt.Component

Basic usage

```
swing.widget()
```

Properties

Property	Type	Readable	Writable
`background`	`awt.Color`	✔	✔
`enabled`	`boolean`	✔	✔
`focusable`	`boolean`	✔	✔
`font`	`awt.Font`	✔	✔
`foreground`	`awt.Color`	✔	✔
`name`	`String`	✔	✔
`visible`	`boolean`	✔	✔

---

window

Swing Class	`javax.swing.JWindow`
Leaf	false
API Docs	javax.swing.JWindow

Basic usage

```
swing.window(title: 'My Window')
```

Properties

Property	Type	Readable	Writable
accessibleContext	javax.accessibility.AccessibleContext	✔	
active	boolean	✔	
alignmentX	float	✔	
alignmentY	float	✔	
alwaysOnTop	boolean	✔	✔
alwaysOnTopSupported	boolean	✔	
autoRequestFocus	boolean	✔	✔
background	awt.Color	✔	✔
bounds	awt.Rectangle		✔
bufferStrategy	awt.image.BufferStrategy	✔	
component	Object		
componentCount	int	✔	
components	awt.Component[]	✔	
containerListeners	awt.event.ContainerListener[]	✔	
contentPane	awt.Container	✔	✔
cursor	awt.Cursor		✔
enabled	boolean	✔	✔
focusCycleRoot	boolean	✔	✔
focusCycleRootAncestor	awt.Container	✔	
focusOwner	awt.Component	✔	
focusTraversalKeys	Object		
focusTraversalPolicy	awt.FocusTraversalPolicy	✔	✔
focusTraversalPolicyProvider	boolean	✔	✔
focusTraversalPolicySet	boolean	✔	
focusable	boolean	✔	✔
focusableWindow	boolean	✔	
focusableWindowState	boolean	✔	✔
focused	boolean	✔	
font	awt.Font	✔	✔
foreground	awt.Color	✔	✔
glassPane	awt.Component	✔	✔
graphics	awt.Graphics	✔	
iconImage	awt.Image		✔
iconImages	List	✔	✔
inputContext	awt.im.InputContext	✔	
insets	awt.Insets	✔	
layeredPane	JLayeredPane	✔	✔
layout	awt.LayoutManager	✔	✔

Property	Type	Readable	Writable
locale	Locale	✔	
location	awt.Point		✔
locationByPlatform	boolean	✔	✔
locationRelativeTo	awt.Component		✔
maximumSize	awt.Dimension	✔	
minimumSize	awt.Dimension	✔	✔
modalExclusionType	awt.Dialog$ModalExclusionType	✔	✔
mostRecentFocusOwner	awt.Component	✔	
name	String	✔	✔
opacity	float	✔	✔
opaque	boolean	✔	
ownedWindows	awt.Window[]	✔	
owner	awt.Window	✔	
preferredSize	awt.Dimension	✔	
rootPane	JRootPane	✔	
shape	awt.Shape	✔	✔
showing	boolean	✔	
size	awt.Dimension		✔
toolkit	awt.Toolkit	✔	
transferHandler	TransferHandler	✔	✔
type	awt.Window$Type	✔	✔
validateRoot	boolean	✔	
visible	boolean	✔	✔
warningString	String	✔	
windowFocusListeners	awt.event.WindowFocusListener[]	✔	
windowListeners	awt.event.WindowListener[]	✔	
windowStateListeners	awt.event.WindowStateListener[]	✔	

AntBuilder



Here we describe `AntBuilder` which lets you write Ant build scripts in Groovy rather than XML. You may also be interested in using Groovy from Ant using the Groovy Ant task.

Despite being primarily a build tool, Apache Ant is a very practical tool for manipulating files including zip files, copy, resource processing, and more. But if ever you've been working with a `build.xml` file or some *Jelly script* and found yourself a little restricted by all those pointy brackets, or found it a bit weird using XML as a scripting language and wanted something a little cleaner and more straight forward, then maybe Ant scripting with Groovy might be what you're after.

Groovy has a helper class called `AntBuilder` which makes the scripting of Ant tasks really easy; allowing a real scripting language to be used for programming constructs (variables, methods, loops, logical branching, classes etc). It still looks like a neat concise version of Ant's XML without all those pointy brackets; though you can mix and match this markup inside your script. Ant itself is a collection of jar files. By adding them to your classpath, you can easily use them within Groovy as is. We believe using `AntBuilder` leads to more concise and readily understood syntax.

`AntBuilder` exposes Ant tasks directly using the convenient builder notation that we are used to in Groovy. Here is the most basic example, which is printing a message on the standard output:

```
def ant = new groovy.ant.AntBuilder()          ①
ant.echo('hello from Ant!')         ②
```

① creates an instance of `AntBuilder`

② executes the `echo` task with the message in parameter

Imagine that you need to create a ZIP file. It can be as simple as:

```
def ant = new AntBuilder()
ant.zip(destfile: 'sources.zip', basedir: 'src')
```

In the next example, we demonstrate the use of `AntBuilder` to copy a list of files using a classical Ant pattern directly in Groovy:

```
// let's just call one task
ant.echo("hello")

// here is an example of a block of Ant inside GroovyMarkup
ant.sequential {
    echo("inside sequential")
    def myDir = "build/AntTest/"
    mkdir(dir: myDir)
    copy(todir: myDir) {
        fileset(dir: "src/test") {
            include(name: "**/*.groovy")
        }
    }
    echo("done")
}

// now let's do some normal Groovy again
def file = new File(ant.project.baseDir,"build/AntTest/some/pkg/MyTest.groovy")
assert file.exists()
```

Another example would be iterating over a list of files matching a specific pattern:

```
// let's create a scanner of filesets
def scanner = ant.fileScanner {
    fileset(dir:"src/test") {
        include(name:"**/My*.groovy")
    }
}

// now let's iterate over
def found = false
for (f in scanner) {
    println("Found file $f")
    found = true
    assert f instanceof File
    assert f.name.endsWith(".groovy")
}
assert found
```

Or executing a JUnit test:

```
ant.junit {
    classpath { pathelement(path: '.') }
    test(name:'some.pkg.MyTest')
}
```

We can even go further by compiling and executing a Java file directly from Groovy:

```
ant.echo(file:'Temp.java', '''
    class Temp {
        public static void main(String[] args) {
            System.out.println("Hello");
        }
    }
''')
ant.javac(srcdir:'.', includes:'Temp.java', fork:'true')
ant.java(classpath:'.', classname:'Temp', fork:'true')
ant.echo('Done')
```

It is worth mentioning that `AntBuilder` is included in Gradle, so you can use it in Gradle just like you would in Groovy. Additional documentation can be found in the Gradle manual.

CliBuilder

`CliBuilder` provides a compact way to specify the available options for a commandline application and then automatically parse the application's commandline parameters according to that specification. By convention, a distinction is made between *option* commandline parameters and any remaining parameters which are passed to an application as its arguments. Typically, several types of options might be supported such as `-V` or `--tabsize=4`. `CliBuilder` removes the burden of

developing lots of code for commandline processing. Instead, it supports a somewhat declarative approach to declaring your options and then provides a single call to parse the commandline parameters with a simple mechanism to interrogate the options (you can think of this as a simple model for your options).

Even though the details of each commandline you create could be quite different, the same main steps are followed each time. First, a `CliBuilder` instance is created. Then, allowed commandline options are defined. This can be done using a *dynamic api* style or an *annotation* style. The commandline parameters are then parsed according to the options specification resulting in a collection of options which are then interrogated.

Here is a simple example `Greeter.groovy` script illustrating usage:

```
// import of CliBuilder not shown                              ①
// specify parameters
def cli = new CliBuilder(usage: 'groovy Greeter [option]') ②
cli.a(longOpt: 'audience', args: 1, 'greeting audience')   ③
cli.h(longOpt: 'help', 'display usage')                    ④

// parse and process parameters
def options = cli.parse(args)                              ⑤
if (options.h) cli.usage()                                 ⑥
else println "Hello ${options.a ? options.a : 'World'}"    ⑦
```

① Earlier versions of Groovy had a CliBuilder in the `groovy.util` package and no import was necessary. In Groovy 2.5, this approach became deprecated: applications should instead choose the `groovy.cli.picocli` or `groovy.cli.commons` version. The groovy.util version in Groovy 2.5 points to the commons-cli version for backwards compatibility but has been removed in Groovy 3.0.

② define a new `CliBuilder` instance specifying an optional usage string

③ specify a `-a` option taking a single argument with an optional long variant `--audience`

④ specify a `-h` option taking no arguments with an optional long variant `--help`

⑤ parse the commandline parameters supplied to the script

⑥ if the `h` option is found display a usage message

⑦ display a standard greeting or, if the `a` option is found, a customized greeting

Running this script with no commandline parameters, i.e.:

```
> groovy Greeter
```

results in the following output:

```
Hello World
```

Running this script with `-h` as the single commandline parameter, i.e.:

```
> groovy Greeter -h
```

results in the following output:

```
usage: groovy Greeter [option]
 -a,--audience <arg>   greeting audience
 -h,--help             display usage
```

Running this script with `--audience Groovologist` as the commandline parameters, i.e.:

```
> groovy Greeter --audience Groovologist
```

results in the following output:

```
Hello Groovologist
```

When creating the `CliBuilder` instance in the above example, we set the optional `usage` property within the constructor call. This follows Groovy's normal ability to set additional properties of the instance during construction. There are numerous other properties which can be set such as `header` and `footer`. For the complete set of available properties, see the available properties for the groovy.util.CliBuilder class.

When defining an allowed commandline option, both a short name (e.g. "h" for the `help` option shown previously) and a short description (e.g. "display usage" for the `help` option) must be supplied. In our example above, we also set some additional properties such as `longOpt` and `args`. The following additional properties are supported when specifying an allowed commandline option:

Name	Description	Type
argName	the name of the argument for this option used in output	`String`
longOpt	the long representation or long name of the option	`String`
args	the number of argument values	`int` or `String` (1)
optionalArg	whether the argument value is optional	`boolean`
required	whether the option is mandatory	`boolean`
type	the type of this option	`Class`

Name	Description	Type
valueSeparator	the character that is the value separator	`char` (2)
defaultValue	a default value	`String`
convert	converts the incoming String to the required type	`Closure` (1)

(1) More details later
(2) Single character Strings are coerced to chars in special cases in Groovy

If you have an option with only a `longOpt` variant, you can use the special shortname of '_' to specify the option, e.g. : `cli._(longOpt: 'verbose', 'enable verbose logging')`. Some of the remaining named parameters should be fairly self-explanatory while others deserve a bit more explanation. But before further explanations, let's look at ways of using `CliBuilder` with annotations.

Using Annotations and an interface

Rather than making a series of method calls (albeit in a very declarative mini-DSL form) to specify the allowable options, you can provide an interface specification of the allowable options where annotations are used to indicate and provide details for those options and for how unprocessed parameters are handled. Two annotations are used: groovy.cli.Option and groovy.cli.Unparsed.

Here is how such a specification can be defined:

```
interface GreeterI {
    @Option(shortName='h', description='display usage') Boolean help()        ①
    @Option(shortName='a', description='greeting audience') String audience() ②
    @Unparsed(description = "positional parameters") List remaining()         ③
}
```

① Specify a Boolean option set using `-h` or `--help`

② Specify a String option set using `-a` or `--audience`

③ Specify where any remaining parameters will be stored

Note how the long name is automatically determined from the interface method name. You can use the `longName` annotation attribute to override that behavior and specify a custom long name if you wish or use a longName of '_' to indicate that no long name is to be provided. You will need to specify a shortName in such a case.

Here is how you could use the interface specification:

```
// import CliBuilder not shown
def cli = new CliBuilder(usage: 'groovy Greeter')  ①
def argz = '--audience Groovologist'.split()
def options = cli.parseFromSpec(GreeterI, argz)            ②
assert options.audience() == 'Groovologist'                ③
```

```
argz = '-h Some Other Args'.split()
options = cli.parseFromSpec(GreeterI, argz)                ④
assert options.help()
assert options.remaining() == ['Some', 'Other', 'Args']    ⑤
```

① Create a `CliBuilder` instance as before with optional properties

② Parse parameters using the interface specification

③ Interrogate options using the methods from the interface

④ Parse a different set of parameters

⑤ Interrogate the remaining parameters

When `parseFromSpec` is called, `CliBuilder` automatically creates an instance implementing the interface and populates it. You simply call the interface methods to interrogate the option values.

Using Annotations and an instance

Alternatively, perhaps you already have a domain class containing the option information. You can simply annotate properties or setters from that class to enable `CliBuilder` to appropriately populate your domain object. Each annotation both describes that option's properties through the annotation attributes and indicates the setter the `CliBuilder` will use to populate that option in your domain object.

Here is how such a specification can be defined:

```
class GreeterC {
    @Option(shortName='h', description='display usage')
    Boolean help                          ①

    private String audience
    @Option(shortName='a', description='greeting audience')
    void setAudience(String audience) { ②
        this.audience = audience
    }
    String getAudience() { audience }

    @Unparsed(description = "positional parameters")
    List remaining                        ③
}
```

① Indicate that a Boolean property is an option

② Indicate that a String property (with explicit setter) is an option

③ Specify where any remaining args will be stored

And here is how you could use the specification:

```
// import CliBuilder not shown
def cli = new CliBuilder(usage: 'groovy Greeter [option]') ①
```

```
def options = new GreeterC()                                    ②
def argz = '--audience Groovologist foo'.split()
cli.parseFromInstance(options, argz)                            ③
assert options.audience == 'Groovologist'                       ④
assert options.remaining == ['foo']                             ⑤
```

① Create a `CliBuilder` instance as before with optional parameters

② Create an instance for `CliBuilder` to populate

③ Parse arguments populating the supplied instance

④ Interrogate the String option property

⑤ Interrogate the remaining arguments property

When `parseFromInstance` is called, `CliBuilder` automatically populates your instance. You simply interrogate the instance properties (or whatever accessor methods you have provided in your domain object) to access the option values.

Using Annotations and a script

Finally, there are two additional convenience annotation aliases specifically for scripts. They simply combine the previously mentioned annotations and groovy.transform.Field. The groovydoc for those annotations reveals the details: groovy.cli.OptionField and groovy.cli.UnparsedField.

Here is an example using those annotations in a self-contained script that would be called with the same arguments as shown for the instance example earlier:

```
// import CliBuilder not shown
import groovy.cli.OptionField
import groovy.cli.UnparsedField

@OptionField String audience
@OptionField Boolean help
@UnparsedField List remaining
new CliBuilder().parseFromInstance(this, args)
assert audience == 'Groovologist'
assert remaining == ['foo']
```

Options with arguments

We saw in our initial example that some options act like flags, e.g. `Greeter -h` but others take an argument, e.g. `Greeter --audience Groovologist`. The simplest cases involve options which act like flags or have a single (potentially optional) argument. Here is an example involving those cases:

```
// import CliBuilder not shown
def cli = new CliBuilder()
cli.a(args: 0, 'a arg') ①
cli.b(args: 1, 'b arg') ②
cli.c(args: 1, optionalArg: true, 'c arg') ③
```

```
def options = cli.parse('-a -b foo -c bar baz'.split()) ④

assert options.a == true
assert options.b == 'foo'
assert options.c == 'bar'
assert options.arguments() == ['baz']

options = cli.parse('-a -c -b foo bar baz'.split()) ⑤

assert options.a == true
assert options.c == true
assert options.b == 'foo'
assert options.arguments() == ['bar', 'baz']
```

① An option that is simply a flag - the default; setting args to 0 is allowed but not needed.

② An option with exactly one argument

③ An option with an optional argument; it acts like a flag if the option is left out

④ An example using this spec where an argument is supplied to the 'c' option

⑤ An example using this spec where no argument is supplied to the 'c' option; it's just a flag

Note: when an option with an optional argument is encountered, it will (somewhat) greedily consume the next parameter from the supplied commandline parameters. If however, the next parameter matches a known long or short option (with leading single or double hyphens), that will take precedence, e.g. `-b` in the above example.

Option arguments may also be specified using the annotation style. Here is an interface option specification illustrating such a definition:

```
interface WithArgsI {
    @Option boolean a()
    @Option String b()
    @Option(optionalArg=true) String[] c()
    @Unparsed List remaining()
}
```

And here is how it is used:

```
def cli = new CliBuilder()
def options = cli.parseFromSpec(WithArgsI, '-a -b foo -c bar baz'.split())
assert options.a()
assert options.b() == 'foo'
assert options.c() == ['bar']
assert options.remaining() == ['baz']

options = cli.parseFromSpec(WithArgsI, '-a -c -b foo bar baz'.split())
assert options.a()
assert options.c() == []
```

```
assert options.b() == 'foo'
assert options.remaining() == ['bar', 'baz']
```

This example makes use of an array-typed option specification. We cover this in more detail shortly when we discuss multiple arguments.

Specifying a type

Arguments on the commandline are by nature Strings (or arguably can be considered Booleans for flags) but can be converted to richer types automatically by supplying additional typing information. For the annotation-based argument definition style, these types are supplied using the field types for annotation properties or return types of annotated methods (or the setter argument type for setter methods). For the dynamic method style of argument definition a special 'type' property is supported which allows you to specify a Class name.

When an explicit type is defined, the `args` named-parameter is assumed to be 1 (except for Boolean-typed options where it is 0 by default). An explicit `args` parameter can still be provided if needed. Here is an example using types with the dynamic api argument definition style:

```
def argz = '''-a John -b -d 21 -e 1980 -f 3.5 -g 3.14159
    -h cv.txt -i DOWN and some more'''.split()
def cli = new CliBuilder()
cli.a(type: String, 'a-arg')
cli.b(type: boolean, 'b-arg')
cli.c(type: Boolean, 'c-arg')
cli.d(type: int, 'd-arg')
cli.e(type: Long, 'e-arg')
cli.f(type: Float, 'f-arg')
cli.g(type: BigDecimal, 'g-arg')
cli.h(type: File, 'h-arg')
cli.i(type: RoundingMode, 'i-arg')
def options = cli.parse(argz)
assert options.a == 'John'
assert options.b
assert !options.c
assert options.d == 21
assert options.e == 1980L
assert options.f == 3.5f
assert options.g == 3.14159
assert options.h == new File('cv.txt')
assert options.i == RoundingMode.DOWN
assert options.arguments() == ['and', 'some', 'more']
```

Primitives, numeric types, files, enums and arrays thereof, are supported (they are converted using org.codehaus.groovy.runtime.StringGroovyMethods#asType).

Custom parsing of the argument String

If the supported types aren't sufficient, you can supply a closure to handle the String to rich type

conversion for you. Here is a sample using the dynamic api style:

```
def argz = '''-a John -b Mary -d 2016-01-01 and some more'''.split()
def cli = new CliBuilder()
def lower = { it.toLowerCase() }
cli.a(convert: lower, 'a-arg')
cli.b(convert: { it.toUpperCase() }, 'b-arg')
cli.d(convert: { Date.parse('yyyy-MM-dd', it) }, 'd-arg')
def options = cli.parse(argz)
assert options.a == 'john'
assert options.b == 'MARY'
assert options.d.format('dd-MM-yyyy') == '01-01-2016'
assert options.arguments() == ['and', 'some', 'more']
```

Alternatively, you can use the annotation style by supplying the conversion closure as an annotation parameter. Here is an example specification:

```
interface WithConvertI {
    @Option(convert={ it.toLowerCase() }) String a()
    @Option(convert={ it.toUpperCase() }) String b()
    @Option(convert={ Date.parse("yyyy-MM-dd", it) }) Date d()
    @Unparsed List remaining()
}
```

And an example using that specification:

```
Date newYears = Date.parse("yyyy-MM-dd", "2016-01-01")
def argz = '''-a John -b Mary -d 2016-01-01 and some more'''.split()
def cli = new CliBuilder()
def options = cli.parseFromSpec(WithConvertI, argz)
assert options.a() == 'john'
assert options.b() == 'MARY'
assert options.d() == newYears
assert options.remaining() == ['and', 'some', 'more']
```

Options with multiple arguments

Multiple arguments are also supported using an `args` value greater than 1. There is a special named parameter, `valueSeparator`, which can also be optionally used when processing multiple arguments. It allows some additional flexibility in the syntax supported when supplying such argument lists on the commandline. For example, supplying a value separator of ',' allows a comma-delimited list of values to be passed on the commandline.

The `args` value is normally an integer. It can be optionally supplied as a String. There are two special String symbols: `` ` `` and `` `\*` ``. The `*` value means 0 or more. The `+` value means 1 or more. The * value is the same as using + and also setting the `optionalArg` value to true.

Accessing the multiple arguments follows a special convention. Simply add an 's' to the normal property you would use to access the argument option and you will retrieve all the supplied arguments as a list. So, for a short option named 'a', you access the first 'a' argument using `options.a` and the list of all arguments using `options.as`. It's fine to have a shortname or longname ending in 's' so long as you don't also have the singular variant without the 's'. So, if `name` is one of your options with multiple arguments and `guess` is another with a single argument, there will be no confusion using `options.names` and `options.guess`.

Here is an excerpt highlighting the use of multiple arguments:

```
// import CliBuilder not shown
def cli = new CliBuilder()
cli.a(args: 2, 'a-arg')
cli.b(args: '2', valueSeparator: ',', 'b-arg') ①
cli.c(args: '+', valueSeparator: ',', 'c-arg') ②

def options = cli.parse('-a 1 2 3 4'.split()) ③
assert options.a == '1' ④
assert options.as == ['1', '2'] ⑤
assert options.arguments() == ['3', '4']

options = cli.parse('-a1 -a2 3'.split()) ⑥
assert options.as == ['1', '2']
assert options.arguments() == ['3']

options = cli.parse(['-b1,2']) ⑦
assert options.bs == ['1', '2']

options = cli.parse(['-c', '1'])
assert options.cs == ['1']

options = cli.parse(['-c1'])
assert options.cs == ['1']

options = cli.parse(['-c1,2,3'])
assert options.cs == ['1', '2', '3']
```

① Args value supplied as a String and comma value separator specified

② One or more arguments are allowed

③ Two commandline parameters will be supplied as the 'b' option's list of arguments

④ Access the 'a' option's first argument

⑤ Access the 'a' option's list of arguments

⑥ An alternative syntax for specifying two arguments for the 'a' option

⑦ The arguments to the 'b' option supplied as a comma-separated value

As an alternative to accessing multiple arguments using the *plural name* approach, you can use an array-based type for the option. In this case, all options will always be returned via the array which

is accessed via the normal singular name. We'll see an example of this next when discussing types.

Multiple arguments are also supported using the annotation style of option definition by using an array type for the annotated class member (method or property) as this example shows:

```
interface ValSepI {
    @Option(numberOfArguments=2) String[] a()
    @Option(numberOfArgumentsString='2', valueSeparator=',') String[] b()
    @Option(numberOfArgumentsString='+', valueSeparator=',') String[] c()
    @Unparsed remaining()
}
```

And used as follows:

```
def cli = new CliBuilder()

def options = cli.parseFromSpec(ValSepI, '-a 1 2 3 4'.split())
assert options.a() == ['1', '2']
assert options.remaining() == ['3', '4']

options = cli.parseFromSpec(ValSepI, '-a1 -a2 3'.split())
assert options.a() == ['1', '2']
assert options.remaining() == ['3']

options = cli.parseFromSpec(ValSepI, ['-b1,2'] as String[])
assert options.b() == ['1', '2']

options = cli.parseFromSpec(ValSepI, ['-c', '1'] as String[])
assert options.c() == ['1']

options = cli.parseFromSpec(ValSepI, ['-c1'] as String[])
assert options.c() == ['1']

options = cli.parseFromSpec(ValSepI, ['-c1,2,3'] as String[])
assert options.c() == ['1', '2', '3']
```

Types and multiple arguments

Here is an example using types and multiple arguments with the dynamic api argument definition style:

```
def argz = '''-j 3 4 5 -k1.5,2.5,3.5 and some more'''.split()
def cli = new CliBuilder()
cli.j(args: 3, type: int[], 'j-arg')
cli.k(args: '+', valueSeparator: ',', type: BigDecimal[], 'k-arg')
def options = cli.parse(argz)
assert options.js == [3, 4, 5] ①
assert options.j == [3, 4, 5]  ①
```

```
assert options.k == [1.5, 2.5, 3.5]
assert options.arguments() == ['and', 'some', 'more']
```

① For an array type, the trailing 's' can be used but isn't needed

Setting a default value

Groovy makes it easy using the Elvis operator to provide a default value at the point of usage of some variable, e.g. `String x = someVariable ?: 'some default'`. But sometimes you wish to make such a default part of the options specification to minimise the interrogators work in later stages. `CliBuilder` supports the `defaultValue` property to cater for this scenario.

Here is how you could use it using the dynamic api style:

```
def cli = new CliBuilder()
cli.f longOpt: 'from', type: String, args: 1, defaultValue: 'one', 'f option'
cli.t longOpt: 'to', type: int, defaultValue: '35', 't option'

def options = cli.parse('-f two'.split())
assert options.hasOption('f')
assert options.f == 'two'
assert !options.hasOption('t')
assert options.t == 35

options = cli.parse('-t 45'.split())
assert !options.hasOption('from')
assert options.from == 'one'
assert options.hasOption('to')
assert options.to == 45
```

Similarly, you might want such a specification using the annotation style. Here is an example using an interface specification:

```
interface WithDefaultValueI {
    @Option(shortName='f', defaultValue='one') String from()
    @Option(shortName='t', defaultValue='35') int to()
}
```

Which would be used like this:

```
def cli = new CliBuilder()

def options = cli.parseFromSpec(WithDefaultValueI, '-f two'.split())
assert options.from() == 'two'
assert options.to() == 35

options = cli.parseFromSpec(WithDefaultValueI, '-t 45'.split())
assert options.from() == 'one'
```

```
assert options.to() == 45
```

You can also use the `defaultValue` annotation attribute when using annotations with an instance, though it's probably just as easy to provide an initial value for the property (or backing field).

Use with `TypeChecked`

The dynamic api style of using `CliBuilder` is inherently dynamic but you have a few options should you want to make use of Groovy's static type checking capabilities. Firstly, consider using the annotation style, for example, here is an interface option specification:

```
interface TypeCheckedI{
    @Option String name()
    @Option int age()
    @Unparsed List remaining()
}
```

And it can be used in combination with `@TypeChecked` as shown here:

```
@TypeChecked
void testTypeCheckedInterface() {
    def argz = "--name John --age 21 and some more".split()
    def cli = new CliBuilder()
    def options = cli.parseFromSpec(TypeCheckedI, argz)
    String n = options.name()
    int a = options.age()
    assert n == 'John' && a == 21
    assert options.remaining() == ['and', 'some', 'more']
}
```

Secondly, there is a feature of the dynamic api style which offers some support. The definition statements are inherently dynamic but actually return a value which we have ignored in earlier examples. The returned value is in fact a `TypedOption<Type>` and special `getAt` support allows the options to be interrogated using the typed option, e.g. `options[savedTypeOption]`. So, if you have statements similar to these in a non type checked part of your code:

```
def cli = new CliBuilder()
TypedOption<Integer> age = cli.a(longOpt: 'age', type: Integer, 'some age option')
```

Then, the following statements can be in a separate part of your code which is type checked:

```
def args = '--age 21'.split()
def options = cli.parse(args)
int a = options[age]
assert a == 21
```

Finally, there is one additional convenience method offered by `CliBuilder` to even allow the definition part to be type checked. It is a slightly more verbose method call. Instead of using the short name (the *opt* name) in the method call, you use a fixed name of `option` and supply the `opt` value as a property. You must also specify the type directly as shown in the following example:

```
import groovy.cli.TypedOption
import groovy.transform.TypeChecked

@TypeChecked
void testTypeChecked() {
    def cli = new CliBuilder()
    TypedOption<String> name = cli.option(String, opt: 'n', longOpt: 'name', 'name
option')
    TypedOption<Integer> age = cli.option(Integer, longOpt: 'age', 'age option')
    def argz = "--name John --age 21 and some more".split()
    def options = cli.parse(argz)
    String n = options[name]
    int a = options[age]
    assert n == 'John' && a == 21
    assert options.arguments() == ['and', 'some', 'more']
}
```

Advanced CLI Usage



NOTE Advanced CLI features

`CliBuilder` can be thought of as a Groovy friendly wrapper on top of either picocli or Apache Commons CLI. If there is a feature not provided by `CliBuilder` that you know is supported in the underlying library, the current `CliBuilder` implementation (and various Groovy language features) make it easy for you to call the underlying library methods directly. Doing so is a pragmatic way to leverage the Groovy-friendly syntax offered by `CliBuilder` and yet still access some of the underlying library's advanced features. A word of caution however; future versions of `CliBuilder` could potentially use another underlying library and in that event, some porting work may be required for your Groovy classes and/or scripts.

Apache Commons CLI

As an example, here is some code for making use of Apache Commons CLI's grouping mechanism:

```
import org.apache.commons.cli.*

def cli = new CliBuilder()
cli.f longOpt: 'from', 'f option'
cli.u longOpt: 'until', 'u option'
def optionGroup = new OptionGroup()
optionGroup.with {
  addOption cli.option('o', [longOpt: 'output'], 'o option')
```

```
  addOption cli.option('d', [longOpt: 'directory'], 'd option')
}
cli.options.addOptionGroup optionGroup
assert !cli.parse('-d -o'.split()) ①
```

① The parse will fail since only one option from a group can be used at a time.

Picocli

Below are some features available in the picocli version of `CliBuilder`.

New property: errorWriter

When users of your application give invalid command line arguments, CliBuilder writes an error message and the usage help message to the `stderr` output stream. It doesn't use the `stdout` stream to prevent the error message from being parsed when your program's output is used as input for another process. You can customize the destination by setting the `errorWriter` to a different value.

On the other hand, `CliBuilder.usage()` prints the usage help message to the `stdout` stream. This way, when users request help (e.g. with a `--help` parameter), they can pipe the output to a utility like `less` or `grep`.

You can specify different writers for testing. *Be aware that for backwards compatibility, setting the `writer` property to a different value will set **both** the `writer` and the `errorWriter` to the specified writer.*

ANSI colors

The picocli version of CliBuilder renders the usage help message in ANSI colors on supported platforms automatically. If desired you can customize this. (An example follows below.)

New property: name

As before, you can set the synopsis of the usage help message with the `usage` property. You may be interested in a small improvement: if you only set the command `name`, a synopsis will be generated automatically, with repeating elements followed by `...` and optional elements surrounded with `[` and `]`. (An example follows below.)

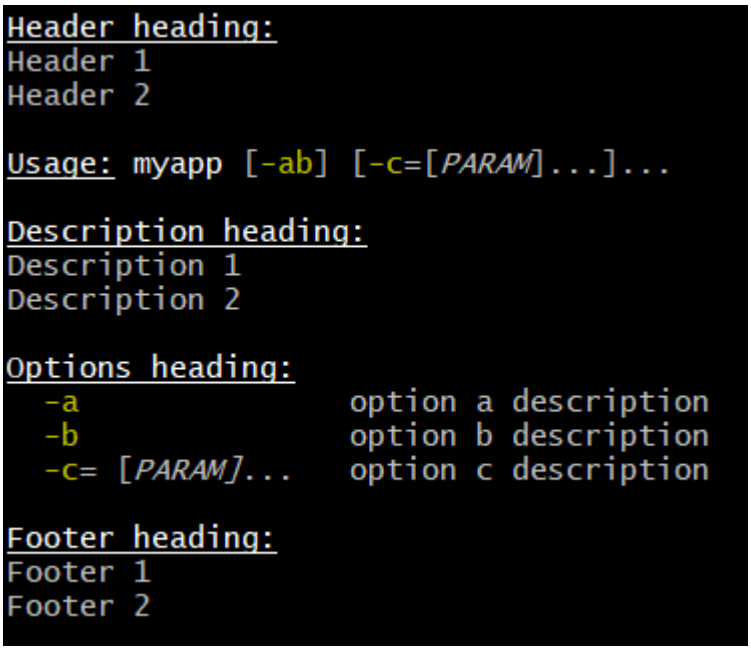
New property: usageMessage

This property exposes a `UsageMessageSpec` object from the underlying picocli library, which gives fine-grained control over various sections of the usage help message. For example:

```
def cli = new CliBuilder()
cli.name = "myapp"
cli.usageMessage.with {
    headerHeading("@|bold,underline Header heading:|@%n")
    header("Header 1", "Header 2")                    // before the synopsis
    synopsisHeading("%n@|bold,underline Usage:|@ ")
    descriptionHeading("%n@|bold,underline Description heading:|@%n")
    description("Description 1", "Description 2")     // after the synopsis
```

```
    optionListHeading("%n@|bold,underline Options heading:|@%n")
    footerHeading("%n@|bold,underline Footer heading:|@%n")
    footer("Footer 1", "Footer 2")
}
cli.a('option a description')
cli.b('option b description')
cli.c(args: '*', 'option c description')
cli.usage()
```

Gives this output:



Property: parser

The `parser` property gives access to the picocli `ParserSpec` object that can be used to customize the parser behavior.

This can be useful when the `CliBuilder` options to control the parser are not fine-grained enough. For example, for backward compatibility with the Commons CLI implementation of `CliBuilder`, by default `CliBuilder` stops looking for options when an unknown option is encountered, and subsequent command line arguments are treated as positional parameters. `CliBuilder` provides a `stopAtNonOption` property, and by setting this to `false` you can make the parser more strict, so an unknown option results in `error: Unknown option: '-x'`.

But what if you want to treat unknown options as positional parameters, and still process subsequent command line arguments as options?

This can be accomplished with the `parser` property. For example:

```
def cli = new CliBuilder()
cli.parser.stopAtPositional(false)
cli.parser.unmatchedOptionsArePositionalParams(true)
// ...
def opts = cli.parse(args)
```

```
// ...
```

See the documentation for details.

Map options

Finally, if your application has options that are key-value pairs, you may be interested in picocli's support for maps. For example:

```
import java.util.concurrent.TimeUnit

import static java.util.concurrent.TimeUnit.DAYS
import static java.util.concurrent.TimeUnit.HOURS


def cli = new CliBuilder()
cli.D(args: 2,   valueSeparator: '=', 'the old way')                          ①
cli.X(type: Map, 'the new way')                                               ②
cli.Z(type: Map, auxiliaryTypes: [TimeUnit, Integer].toArray(), 'typed map')  ③

def options = cli.parse('-Da=b -Dc=d -Xx=y -Xi=j -ZDAYS=2 -ZHOURS=23'.split())④
assert options.Ds == ['a', 'b', 'c', 'd']                                     ⑤
assert options.Xs == [ 'x':'y', 'i':'j' ]                                     ⑥
assert options.Zs == [ (DAYS as TimeUnit):2, (HOURS as TimeUnit):23 ]         ⑦
```

① Previously, `key=value` pairs were split up into parts and added to a list

② Picocli map support: simply specify `Map` as the type of the option

③ You can even specify the type of the map elements

④ To compare, let's specify two key-value pairs for each option

⑤ Previously, all key-value pairs end up in a list and it is up to the application to work with this list

⑥ Picocli returns the key-value pairs as a `Map`

⑦ Both keys and values of the map can be strongly typed

Controlling the Picocli version

To use a specific version of picocli, add a dependency to that version in your build configuration. If running scripts using a pre-installed version of Groovy, use the `@Grab` annotation to control the version of picocli to use in `CliBuilder`.

```
@GrabConfig(systemClassLoader=true)
@Grab('info.picocli:picocli:4.2.0')
import groovy.cli.picocli.CliBuilder

def cli = new CliBuilder()
```

ObjectGraphBuilder

`ObjectGraphBuilder` is a builder for an arbitrary graph of beans that follow the JavaBean convention. It is in particular useful for creating test data.

Let's start with a list of classes that belong to your domain:

```
package com.acme

class Company {
    String name
    Address address
    List employees = []
}

class Address {
    String line1
    String line2
    int zip
    String state
}

class Employee {
    String name
    int employeeId
    Address address
    Company company
}
```

Then using `ObjectGraphBuilder` building a `Company` with three employees is as easy as:

```
def builder = new ObjectGraphBuilder()                          ①
builder.classLoader = this.class.classLoader                    ②
builder.classNameResolver = "com.acme"                          ③

def acme = builder.company(name: 'ACME') {                      ④
    3.times {
        employee(id: it.toString(), name: "Drone $it") {        ⑤
            address(line1:"Post street")                        ⑥
        }
    }
}

assert acme != null
assert acme instanceof Company
assert acme.name == 'ACME'
assert acme.employees.size() == 3
def employee = acme.employees[0]
assert employee instanceof Employee
assert employee.name == 'Drone 0'
```

```
assert employee.address instanceof Address
```

① creates a new object graph builder

② sets the classloader where the classes will be resolved

③ sets the base package name for classes to be resolved

④ creates a `Company` instance

⑤ with 3 `Employee` instances

⑥ each of them having a distinct `Address`

Behind the scenes, the object graph builder:

- will try to match a node name into a `Class`, using a default `ClassNameResolver` strategy that requires a package name
- then will create an instance of the appropriate class using a default `NewInstanceResolver` strategy that calls a no-arg constructor
- resolves the parent/child relationship for nested nodes, involving two other strategies:
  - `RelationNameResolver` will yield the name of the child property in the parent, and the name of the parent property in the child (if any, in this case, `Employee` has a parent property aptly named `company`)
  - `ChildPropertySetter` will insert the child into the parent taking into account if the child belongs to a `Collection` or not (in this case `employees` should be a list of `Employee` instances in `Company`).

All 4 strategies have a default implementation that work as expected if the code follows the usual conventions for writing JavaBeans. In case any of your beans or objects do not follow the convention you may plug your own implementation of each strategy. For example imagine that you need to build a class which is immutable:

```
@Immutable
class Person {
    String name
    int age
}
```

Then if you try to create a `Person` with the builder:

```
def person = builder.person(name:'Jon', age:17)
```

It will fail at runtime with:

```
Cannot set read-only property: name for class: com.acme.Person
```

Fixing this can be done by changing the new instance strategy:

```
builder.newInstanceResolver = { Class klazz, Map attributes ->
    if (klazz.getConstructor(Map)) {
        def o = klazz.newInstance(attributes)
        attributes.clear()
        return o
    }
    klazz.newInstance()
}
```

`ObjectGraphBuilder` supports ids per node, meaning that you can store a reference to a node in the builder. This is useful when multiple objects reference the same instance. Because a property named `id` may be of business meaning in some domain models `ObjectGraphBuilder` has a strategy named `IdentifierResolver` that you may configure to change the default name value. The same may happen with the property used for referencing a previously saved instance, a strategy named `ReferenceResolver` will yield the appropriate value (default is `refId'):

```
def company = builder.company(name: 'ACME') {
    address(id: 'a1', line1: '123 Groovy Rd', zip: 12345, state: 'JV')          ①
    employee(name: 'Duke', employeeId: 1, address: a1)                          ②
    employee(name: 'John', employeeId: 2 ){
      address( refId: 'a1' )                                                    ③
    }
}
```

① an address can be created with an `id`

② an employee can reference the address directly with its id

③ or use the `refId` attribute corresponding to the `id` of the corresponding address

Its worth mentioning that you cannot modify the properties of a referenced bean.

JmxBuilder

See Working with JMX - JmxBuilder for details.

FileTreeBuilder

groovy.util.FileTreeBuilder is a builder for generating a file directory structure from a specification. For example, to create the following tree:

```
src/
 |--- main
 |      |--- groovy
 |             |--- Foo.groovy
 |--- test
        |--- groovy
               |--- FooTest.groovy
```

You can use a `FileTreeBuilder` like this:

```
tmpDir = File.createTempDir()
def fileTreeBuilder = new FileTreeBuilder(tmpDir)
fileTreeBuilder.dir('src') {
    dir('main') {
       dir('groovy') {
          file('Foo.groovy', 'println "Hello"')
       }
    }
    dir('test') {
       dir('groovy') {
          file('FooTest.groovy', 'class FooTest extends groovy.test.GroovyTestCase
{}')
       }
    }
 }
```

To check that everything worked as expected we use the following `assert`s:

```
assert new File(tmpDir, '/src/main/groovy/Foo.groovy').text == 'println "Hello"'
assert new File(tmpDir, '/src/test/groovy/FooTest.groovy').text == 'class FooTest
extends groovy.test.GroovyTestCase {}'
```

`FileTreeBuilder` also supports a shorthand syntax:

```
tmpDir = File.createTempDir()
def fileTreeBuilder = new FileTreeBuilder(tmpDir)
fileTreeBuilder.src {
    main {
       groovy {
          'Foo.groovy'('println "Hello"')
       }
    }
    test {
       groovy {
          'FooTest.groovy'('class FooTest extends groovy.test.GroovyTestCase {}')
       }
    }
 }
```

This produces the same directory structure as above, as shown by these `assert`s:

```
assert new File(tmpDir, '/src/main/groovy/Foo.groovy').text == 'println "Hello"'
assert new File(tmpDir, '/src/test/groovy/FooTest.groovy').text == 'class FooTest
extends groovy.test.GroovyTestCase {}'
```

Creating a builder

While Groovy has many built-in builders, the builder pattern is so common, you will no doubt eventually come across a building requirement that hasn't been catered for by those built-in builders. The good news is that you can build your own. You can do everything from scratch by relying on Groovy's metaprogramming capabilities. Alternatively, the `BuilderSupport` and `FactoryBuilderSupport` classes make designing your own builders much easier.

BuilderSupport

One approach to building a builder is to subclass `BuilderSupport`. With this approach, the general idea is to override one or more of a number of *lifecycle* methods including `setParent`, `nodeCompleted` and some or all of the `createNode` methods from the `BuilderSupport` abstract class.

As an example, suppose we want to create a builder of tracking athletic training programs. Each program is made up of a number of sets and each set has its own steps. A step might itself be a set of smaller steps. For each `set` or `step`, we might wish to record the `distance` required (or `time`), whether to `repeat` the steps a certain number of times, whether to take a `break` between each step and so forth.

For the simplicity of this example, we'll capture the training programming using maps and lists. A set has a list of steps. Information like `repeat` count or `distance` will be tracked in a map of attributes for each step and set.

The builder implementation is as follows:

- Override a couple of the `createNode` methods. We'll create a map capturing the set name, an empty list of steps, and potentially some attributes.
- Whenever we complete a node we'll add the node to the list of steps for the parent (if any).

The code looks like this:

```
class TrainingBuilder1 extends BuilderSupport {
    protected createNode(name) {
        [name: name, steps: []]
    }

    protected createNode(name, Map attributes) {
        createNode(name) + attributes
    }

    void nodeCompleted(maybeParent, node) {
        if (maybeParent) maybeParent.steps << node
    }

    // unused lifecycle methods
    protected void setParent(parent, child) { }
    protected createNode(name, Map attributes, value) { }
    protected createNode(name, value) { }
```

```
}
```

Next, we'll write a little helper method which recursively adds up the distances of all substeps, accounting for repeated steps as needed.

```
def total(map) {
    if (map.distance) return map.distance
    def repeat = map.repeat ?: 1
    repeat * map.steps.sum{ total(it) }
}
```

Finally, we can now use our builder and helper method to create a swimming training program and check its total distance:

```
def training = new TrainingBuilder1()

def monday = training.swimming {
    warmup(repeat: 3) {
        freestyle(distance: 50)
        breaststroke(distance: 50)
    }
    endurance(repeat: 20) {
        freestyle(distance: 50, break: 15)
    }
    warmdown {
        kick(distance: 100)
        choice(distance: 100)
    }
}

assert 1500 == total(monday)
```

FactoryBuilderSupport

A second approach to building a builder is to subclass `FactoryBuilderSupport`. This builder has similar goals to `BuilderSupport` but with extra features to simplify domain class construction.

With this approach, the general idea is to override one or more of a number of *lifecycle* methods including `resolveFactory`, `nodeCompleted` and `postInstantiate` methods from the `FactoryBuilderSupport` abstract class.

We'll use the same example as for the previous `BuilderSupport` example; a builder of tracking athletic training programs.

For this example, rather than capturing the training programming using maps and lists, we'll use some simple domain classes.

The builder implementation is as follows:

- Override the `resolveFactory` method to return a special factory which returns classes by capitalizing the names used in our mini DSL.
- Whenever we complete a node we'll add the node to the list of steps for the parent (if any).

The code, including the code for the special factory class, looks like this:

```
import static org.apache.groovy.util.BeanUtils.capitalize

class TrainingBuilder2 extends FactoryBuilderSupport {
    def factory = new TrainingFactory(loader: getClass().classLoader)

    protected Factory resolveFactory(name, Map attrs, value) {
        factory
    }

    void nodeCompleted(maybeParent, node) {
        if (maybeParent) maybeParent.steps << node
    }
}

class TrainingFactory extends AbstractFactory {
    ClassLoader loader
    def newInstance(FactoryBuilderSupport fbs, name, value, Map attrs) {
        def clazz = loader.loadClass(capitalize(name))
        value ? clazz.newInstance(value: value) : clazz.newInstance()
    }
}
```

Rather than using lists and maps, we'll have some simple domain classes and related traits:

```
trait HasDistance {
    int distance
}

trait Container extends HasDistance {
    List steps = []
    int repeat
}

class Cycling implements Container { }

class Interval implements Container { }

class Sprint implements HasDistance {}

class Tempo implements HasDistance {}
```

Just like for the `BuilderSupport` example, it is useful to have a helper method to calculate the total

distance covered during the training session. The implementation is very similar to our earlier example, but is adjusted to work well with our newly defined traits.

```
def total(HasDistance c) {
    c.distance
}

def total(Container c) {
    if (c.distance) return c.distance
    def repeat = c.repeat ?: 1
    repeat * c.steps.sum{ total(it) }
}
```

Finally, we can now use our new builder and helper methods to create a cycling training program and check its total distance:

```
def training = new TrainingBuilder2()

def tuesday = training.cycling {
    interval(repeat: 5) {
        sprint(distance: 400)
        tempo(distance: 3600)
    }
}

assert 20000 == total(tuesday)
```

Working with JMX

Introduction

The Java Management Extensions (JMX) technology provides a standard way of managing resources such as applications, devices, and services on the JDK. Each resource to be managed is represented by a *Managed Bean* (or *MBean*). Given that Groovy sits directly on top of Java, Groovy can leverage the tremendous amount of work already done for JMX with Java. In addition, Groovy provides a `GroovyMBean` class, in the `groovy-jmx` module, which makes an MBean look like a normal Groovy object and simplifies Groovy code for interacting with MBeans. For example, the following code:

```
println server.getAttribute(beanName, 'Age')
server.setAttribute(beanName, new Attribute('Name', 'New name'))
Object[] params = [5, 20]
String[] signature = [Integer.TYPE, Integer.TYPE]
println server.invoke(beanName, 'add', params, signature)
```

can be simplified to:

```
def mbean = new GroovyMBean(server, beanName)
println mbean.Age
mbean.Name = 'New name'
println mbean.add(5, 20)
```

The remainder of this page shows you how to:

- Monitor the JVM using MXBeans
- Monitor Apache Tomcat and display statistics
- Monitor WebLogic and display information
- Leverage Spring's MBean annotation support to export your Groovy beans as MBeans
- Use JmxBuilder to export and manage MBeans with a Groovy DSL



Since JDK 9, JMX classes are split across the `java.management` module (core API) and the `java.management.rmi` module (RMI connector). If your application runs on the module path, you may need to add `requires java.management.rmi` for remote JMX connectivity. On JDK 17+, remote JMX connections may require configuring deserialization filters via `jmx.remote.rmi.server.serial.filter.pattern` due to stricter default security.

Monitoring the JVM

MBeans are not accessed directly by an application but are managed by a repository called an *MBean server*. Java includes a special MBean server called the *platform MBean server*, which is built into the JVM. Platform MBeans are registered in this server using unique names.

You can monitor the JVM through its platform MBeans with the following code:

```
import java.lang.management.*

def os = ManagementFactory.operatingSystemMXBean
println """OPERATING SYSTEM:
\tarchitecture = $os.arch
\tname = $os.name
\tversion = $os.version
\tprocessors = $os.availableProcessors
"""

def rt = ManagementFactory.runtimeMXBean
println """RUNTIME:
\tname = $rt.name
\tspec name = $rt.specName
\tvendor = $rt.specVendor
\tspec version = $rt.specVersion
\tmanagement spec version = $rt.managementSpecVersion
"""
```

```
def cl = ManagementFactory.classLoadingMXBean
println """CLASS LOADING SYSTEM:
\tisVerbose = ${cl.isVerbose()}
\tloadedClassCount = $cl.loadedClassCount
\ttotalLoadedClassCount = $cl.totalLoadedClassCount
\tunloadedClassCount = $cl.unloadedClassCount
"""

def comp = ManagementFactory.compilationMXBean
println """COMPILATION:
\ttotalCompilationTime = $comp.totalCompilationTime
"""

def mem = ManagementFactory.memoryMXBean
def heapUsage = mem.heapMemoryUsage
def nonHeapUsage = mem.nonHeapMemoryUsage
println """MEMORY:
HEAP STORAGE:
\tcommitted = $heapUsage.committed
\tinit = $heapUsage.init
\tmax = $heapUsage.max
\tused = $heapUsage.used
NON-HEAP STORAGE:
\tcommitted = $nonHeapUsage.committed
\tinit = $nonHeapUsage.init
\tmax = $nonHeapUsage.max
\tused = $nonHeapUsage.used
"""

ManagementFactory.memoryPoolMXBeans.each { mp ->
    println "\tname: " + mp.name
    String[] mmnames = mp.memoryManagerNames
    mmnames.each{ mmname ->
        println "\t\tManager Name: $mmname"
    }
    println "\t\tmtype = $mp.type"
    println "\t\tUsage threshold supported = " + mp.isUsageThresholdSupported()
}
println()

def td = ManagementFactory.threadMXBean
println "THREADS:"
td.allThreadIds.each { tid ->
    println "\tThread name = ${td.getThreadInfo(tid).threadName}"
}
println()

println "GARBAGE COLLECTION:"
ManagementFactory.garbageCollectorMXBeans.each { gc ->
    println "\tname = $gc.name"
```

```
    println "\t\tcollection count = $gc.collectionCount"
    println "\t\tcollection time = $gc.collectionTime"
    String[] mpoolNames = gc.memoryPoolNames
    mpoolNames.each { mpoolName ->
        println "\t\tmpool name = $mpoolName"
    }
}
```

When run, you will see something like this:

```
OPERATING SYSTEM:
                architecture = amd64
                name = Windows 10
                version = 10.0
                processors = 12

RUNTIME:
                name = 724176@QUOKKA
                spec name = Java Virtual Machine Specification
                vendor = Oracle Corporation
                spec version = 11
                management spec version = 2.0

CLASS LOADING SYSTEM:
                isVerbose = false
                loadedClassCount = 6962
                totalLoadedClassCount = 6969
                unloadedClassCount = 0

COMPILATION:
                totalCompilationTime = 7548

MEMORY:
            HEAP STORAGE:
                committed = 645922816
                init = 536870912
                max = 8560574464
                used = 47808352
            NON-HEAP STORAGE:
                committed = 73859072
                init = 7667712
                max = -1
                used = 70599520

    name: CodeHeap 'non-nmethods'
        Manager Name: CodeCacheManager
        mtype = Non-heap memory
        Usage threshold supported = true
    name: Metaspace
        Manager Name: Metaspace Manager
```

```
        mtype = Non-heap memory
        Usage threshold supported = true
    name: CodeHeap 'profiled nmethods'
        Manager Name: CodeCacheManager
        mtype = Non-heap memory
        Usage threshold supported = true
    name: Compressed Class Space
        Manager Name: Metaspace Manager
        mtype = Non-heap memory
        Usage threshold supported = true
    name: G1 Eden Space
        Manager Name: G1 Old Generation
        Manager Name: G1 Young Generation
        mtype = Heap memory
        Usage threshold supported = false
    name: G1 Old Gen
        Manager Name: G1 Old Generation
        Manager Name: G1 Young Generation
        mtype = Heap memory
        Usage threshold supported = true
    name: G1 Survivor Space
        Manager Name: G1 Old Generation
        Manager Name: G1 Young Generation
        mtype = Heap memory
        Usage threshold supported = false
    name: CodeHeap 'non-profiled nmethods'
        Manager Name: CodeCacheManager
        mtype = Non-heap memory
        Usage threshold supported = true

THREADS:
    Thread name = Reference Handler
    Thread name = Finalizer
    Thread name = Signal Dispatcher
    Thread name = Attach Listener
    Thread name = Common-Cleaner
    Thread name = Java2D Disposer
    Thread name = AWT-Shutdown
    Thread name = AWT-Windows
    Thread name = Image Fetcher 0
    Thread name = AWT-EventQueue-0
    Thread name = D3D Screen Updater
    Thread name = DestroyJavaVM
    Thread name = TimerQueue
    Thread name = Thread-0

GARBAGE COLLECTION:
    name = G1 Young Generation
        collection count = 6
        collection time = 69
        mpool name = G1 Eden Space
```

```
        mpool name = G1 Survivor Space
        mpool name = G1 Old Gen
    name = G1 Old Generation
        collection count = 0
        collection time = 0
        mpool name = G1 Eden Space
        mpool name = G1 Survivor Space
        mpool name = G1 Old Gen
```

Monitoring Tomcat

First start up Tomcat with JMX monitoring enabled by setting the following:

```
set JAVA_OPTS=-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=9004\
 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl
=false
```

You can do this in your startup script and may choose any available port, we used 9004.

The following code uses JMX to discover the available MBeans in the running Tomcat, determine which are web modules, extract the processing time for each web module and displays the result in a graph using JFreeChart:

```
import groovy.swing.SwingBuilder
import groovy.jmx.GroovyMBean

import javax.management.ObjectName
import javax.management.remote.JMXConnectorFactory as JmxFactory
import javax.management.remote.JMXServiceURL as JmxUrl
import javax.swing.WindowConstants as WC

import org.jfree.chart.ChartFactory
import org.jfree.data.category.DefaultCategoryDataset as Dataset
import org.jfree.chart.plot.PlotOrientation as Orientation

def serverUrl = 'service:jmx:rmi:///jndi/rmi://localhost:9004/jmxrmi'
def server = JmxFactory.connect(new JmxUrl(serverUrl)).MBeanServerConnection
def serverInfo = new GroovyMBean(server, 'Catalina:type=Server').serverInfo
println "Connected to: $serverInfo"

def query = new ObjectName('Catalina:*')
String[] allNames = server.queryNames(query, null)
def modules = allNames.findAll { name ->
    name.contains('j2eeType=WebModule')
}.collect{ new GroovyMBean(server, it) }

println "Found ${modules.size()} web modules. Processing ..."
def dataset = new Dataset()
```

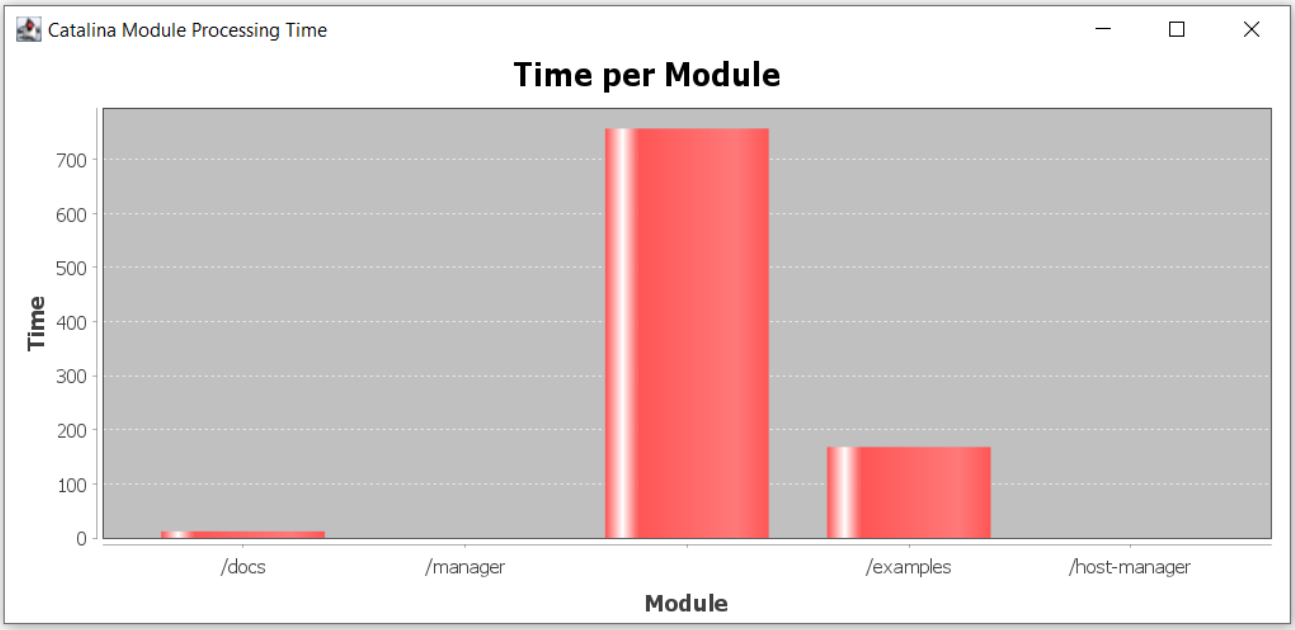
```
modules.each { m ->
    println m.name()
    dataset.addValue m.processingTime, 0, m.path
}

def labels = ['Time per Module', 'Module', 'Time']
def options = [false, true, true]
def chart = ChartFactory.createBarChart(*labels, dataset,
                Orientation.VERTICAL, *options)
def swing = new SwingBuilder()
def frame = swing.frame(title:'Catalina Module Processing Time',
defaultCloseOperation:WC.DISPOSE_ON_CLOSE) {
    panel(id:'canvas') { rigidArea(width:800, height:350) }
}
frame.pack()
frame.show()
chart.draw(swing.canvas.graphics, swing.canvas.bounds)
```

When run, we will see a trace of progress being made:

```
Connected to: Apache Tomcat/9.0.37
Found 5 web modules. Processing ...
Catalina:j2eeType=WebModule,name=//localhost/docs,J2EEApplication=none,J2EEServer=none
Catalina:j2eeType=WebModule,name=//localhost/manager,J2EEApplication=none,J2EEServer=n
one
Catalina:j2eeType=WebModule,name=//localhost/,J2EEApplication=none,J2EEServer=none
Catalina:j2eeType=WebModule,name=//localhost/examples,J2EEApplication=none,J2EEServer=
none
Catalina:j2eeType=WebModule,name=//localhost/host-
manager,J2EEApplication=none,J2EEServer=none
```

The output will look like this:



Note: if you get errors running this script, see the **Troubleshooting** section below.

WebLogic Example

This script prints out information about the server followed by information about JMS Destinations (as an example). Many other MBeans are available (consult the WebLogic MBean Reference for your version).

```
import javax.management.remote.*
import javax.management.*
import javax.naming.Context
import groovy.jmx.GroovyMBean

def urlRuntime = '/jndi/weblogic.management.mbeanservers.runtime'
def urlBase = 'service:jmx:t3://localhost:7001'

def serviceURL = new JMXServiceURL(urlBase + urlRuntime)
def h = new Hashtable()
h.put(Context.SECURITY_PRINCIPAL, 'weblogic')
h.put(Context.SECURITY_CREDENTIALS, 'weblogic')
h.put(JMXConnectorFactory.PROTOCOL_PROVIDER_PACKAGES, 'weblogic.management.remote')
def server = JMXConnectorFactory.connect(serviceURL, h).MBeanServerConnection
def domainName = new ObjectName
('com.bea:Name=RuntimeService,Type=weblogic.management.mbeanservers.runtime.RuntimeSer
viceMBean')
def rtName = server.getAttribute(domainName, 'ServerRuntime')
def rt = new GroovyMBean(server, rtName)
println "Server: name=$rt.Name, state=$rt.State, version=$rt.WeblogicVersion"
def destFilter = Query.match(Query.attr('Type'), Query.value('JMSDestinationRuntime'))
server.queryNames(new ObjectName('com.bea:*'), destFilter).each { name ->
    def jms = new GroovyMBean(server, name)
    println "JMS Destination: name=$jms.Name, type=$jms.DestinationType,
```

```
messages=$jms.MessagesReceivedCount"
}
```

Here is the output:

```
Server: name=examplesServer, state=RUNNING, version=WebLogic Server 10.0  Wed May 9
18:10:27 EDT 2007 933139
JMS Destination: name=examples-jms!exampleTopic, type=Topic, messages=0
JMS Destination: name=examples-jms!exampleQueue, type=Queue, messages=0
JMS Destination: name=examples-jms!jms/MULTIDATASOURCE_MDB_QUEUE, type=Queue,
messages=0
JMS Destination: name=examplesJMSServer!examplesJMSServer.TemporaryQueue0, type=Queue,
messages=68
JMS Destination: name=examples-jms!quotes, type=Topic, messages=0
JMS Destination: name=examples-jms!weblogic.wsee.wseeExamplesDestinationQueue,
type=Queue, messages=0
JMS Destination: name=examples-jms!weblogic.examples.ejb30.ExampleQueue, type=Queue,
messages=0
```

Spring Example

You can also use Spring to automatically register beans as JMX aware.

Here is an example class (Calculator.groovy):

```
import org.springframework.jmx.export.annotation.*

@ManagedResource(objectName="bean:name=calcMBean", description="Calculator MBean")
public class Calculator {

    private int invocations

    @ManagedAttribute(description="The Invocation Attribute")
    public int getInvocations() {
        return invocations
    }

    private int base = 10

    @ManagedAttribute(description="The Base to use when adding strings")
    public int getBase() {
        return base
    }

    @ManagedAttribute(description="The Base to use when adding strings")
    public void setBase(int base) {
        this.base = base
    }
```

```
    @ManagedOperation(description="Add two numbers")
    @ManagedOperationParameters([
        @ManagedOperationParameter(name="x", description="The first number"),
        @ManagedOperationParameter(name="y", description="The second number")])
    public int add(int x, int y) {
        invocations++
        return x + y
    }

    @ManagedOperation(description="Add two strings representing numbers of a
particular base")
    @ManagedOperationParameters([
        @ManagedOperationParameter(name="x", description="The first number"),
        @ManagedOperationParameter(name="y", description="The second number")])
    public String addStrings(String x, String y) {
        invocations++
        def result = Integer.valueOf(x, base) + Integer.valueOf(y, base)
        return Integer.toString(result, base)
    }
}
```

Here is the Spring configuration file (beans.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="mbeanServer"
          class="org.springframework.jmx.support.MBeanServerFactoryBean">
        <property name="locateExistingServerIfPossible" value="true"/>
    </bean>

    <bean id="exporter"
          class="org.springframework.jmx.export.MBeanExporter">
        <property name="assembler" ref="assembler"/>
        <property name="namingStrategy" ref="namingStrategy"/>
        <property name="beans">
            <map>
                <entry key="bean:name=defaultCalcName" value-ref="calcBean"/>
            </map>
        </property>
        <property name="server" ref="mbeanServer"/>
        <property name="autodetect" value="true"/>
    </bean>

    <bean id="jmxAttributeSource"
```

```
class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>

    <!-- will create management interface using annotation metadata -->
    <bean id="assembler"
          class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
        <property name="attributeSource" ref="jmxAttributeSource"/>
    </bean>

    <!-- will pick up the ObjectName from the annotation -->
    <bean id="namingStrategy"
          class="org.springframework.jmx.export.naming.MetadataNamingStrategy">
        <property name="attributeSource" ref="jmxAttributeSource"/>
    </bean>

    <bean id="calcBean"
          class="Calculator">
        <property name="base" value="10"/>
    </bean>
</beans>
```

Here is a script which uses this bean and configuration:

```
import org.springframework.context.support.ClassPathXmlApplicationContext
import java.lang.management.ManagementFactory
import javax.management.ObjectName
import javax.management.Attribute
import groovy.jmx.GroovyMBean

// get normal bean
def ctx = new ClassPathXmlApplicationContext("beans.xml")
def calc = ctx.getBean("calcBean")

Thread.start {
    // access bean via JMX, use a separate thread just to
    // show that we could access remotely if we wanted
    def server = ManagementFactory.platformMBeanServer
    def mbean = new GroovyMBean(server, 'bean:name=calcMBean')
    sleep 1000
    assert 8 == mbean.add(7, 1)
    mbean.Base = 8
    assert '10' == mbean.addStrings('7', '1')
    mbean.Base = 16
    sleep 2000
    println "Number of invocations: $mbean.Invocations"
    println mbean
}

assert 15 == calc.add(9, 6)
assert '11' == calc.addStrings('10', '1')
sleep 2000
```

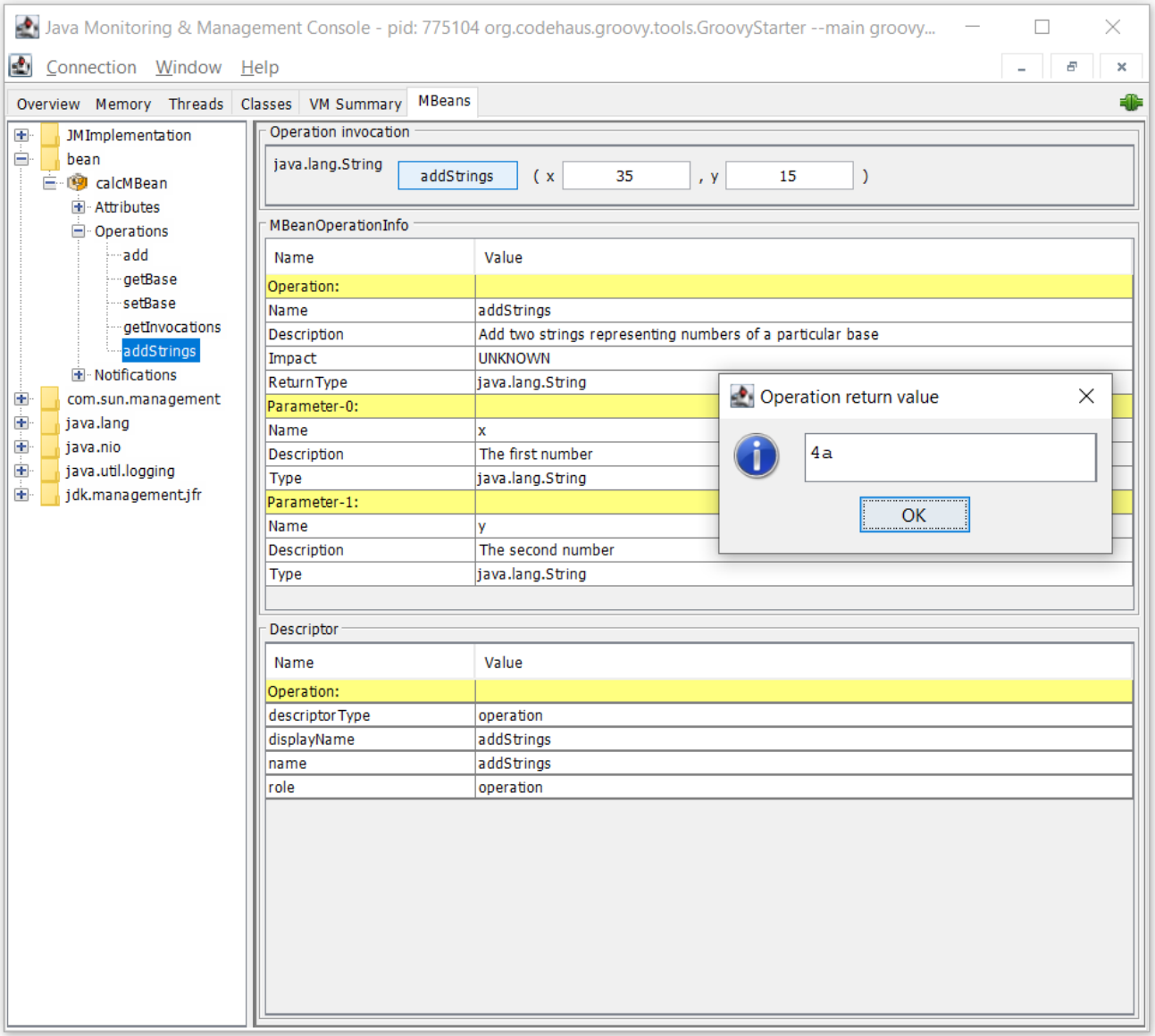
```
assert '20' == calc.addStrings('1f', '1')
```

And here is the resulting output:

```
Number of invocations: 5
MBean Name:
  bean:name=calcMBean

Attributes:
  (rw) int Base
  (r) int Invocations
Operations:
  int add(int x, int y)
  java.lang.String addStrings(java.lang.String x, java.lang.String y)
  int getInvocations()
  int getBase()
  void setBase(int p1)
```

You can even attach to the process while it is running with jconsole. It will look something like:



We started the Groovy application with the `-Dcom.sun.management.jmxremote` JVM argument.

See also:

- Dynamic language beans in Spring
- Spring JMX Documentation

Troubleshooting

java.lang.SecurityException

If you get the following error, your container's JMX access is password protected:

```
java.lang.SecurityException: Authentication failed! Credentials required
```

To fix that, add an environment with the credentials when connecting, like this (password has to be set before that):

```
def jmxEnv = null
if (password != null) {
    jmxEnv = [(JMXConnector.CREDENTIALS): (String[])["monitor", password]]
}
def connector = JMXConnectorFactory.connect(new JMXServiceURL(serverUrl), jmxEnv)
```

Details for the software you are trying to monitor/manage may differ slightly. Check out the other examples using credentials above if appropriate (e.g. WebLogic). If you still have troubles, you will have to consult the documentation for the software you are trying to monitor/manage for details on how to provide credentials.

JmxBuilder

JmxBuilder is a Groovy-based domain specific language for the Java Management Extension (JMX) API. It uses the builder pattern (FactoryBuilder) to create an internal DSL that facilitates the exposure of POJO's and Groovy beans as management components via the MBean server. JmxBuilder hides the complexity of creating and exporting management beans via the JMX API and provides a set of natural Groovy constructs to interact with the JMX infrastructure.

Instantiating JmxBuilder

To start using JmxBuilder, simply make sure the jar file is on your class path. Then you can do the following in your code:

```
def jmx = new JmxBuilder()
```

That's it! You are now ready to use the JmxBuilder.

NOTE

- You can pass in an instance of **your own MBeanServer** to the builder (**JmxBuilder(MBeanServer)**)
- If no MBeanServer is specified, the builder instance will default to the underlying platform MBeanServer.

Once you have an instance of JmxBuilder, you are now ready to invoke any of its builder nodes.

JMX Connectors

Remote connectivity is a crucial part of the JMX architecture. JmxBuilder facilitates the creation of connector servers and connector clients with a minimal amount of coding.

Connector Server

JmxBuilder.connectorServer() supports the full Connector api syntax and will let you specify properties, override the URL, specify your own host, etc.

Syntax

```
jmx.connectorServer(
    protocol:"rmi",
    host:"...",
    port:1099,
    url:"...",
    properties:[
        "authenticate":true|false,
        "passwordFile":"...",
        "accessFile":"...",
        "sslEnabled" : true | false
        // any valid connector property
    ]
)
```

Note that the serverConnector node will accept four ServerConnector property aliases (authenticate, passwordFile,accessFile, and sslEnabled). You can use these aliases or provided any of the RMI-supported properties.

Example - Connector Server (see correction below)

```
jmx.connectorServer(port: 9000).start()
```

The snippet above returns an RMI connector that will start listening on port 9000. By default, the builder will internally generate URL **"service:jmx:rmi:///jndi/rmi://localhost:9000/jmxrmi"**.

The snippet on its own fails, because it does not create the RMI registry the connector binds its stub into:

```
Caught: java.io.IOException: Cannot bind to URL [rmi://localhost:9000/jmxrmi]:
javax.naming.ServiceUnavailableException [Root exception is java.rmi.ConnectException:
Connection refused to host: localhost; ...]
```



The corrected example, which creates the registry first, is shown below.

The connector listens on the host it was given, which is `localhost` unless you say otherwise, so by default only clients on the same machine can reach it. Pass a `host` to listen more widely — `0.0.0.0` for every interface, or a single address to pick one.



A connector exposes the MBean server, and the default MBean server is the platform one, whose beans can dump the heap and run diagnostic commands. It authenticates nobody unless you configure it to, so a connector reachable from other hosts hands that to any peer that can reach the port. `JmxBuilder` is a development and experimentation aid; treat a connector left open the way you would treat an open shell on the host.

Connector Example (Corrected) - Connector Server

The example above does not create the RMI registry. So, in order to export, you have to first export the RMI object registry (make sure to import `java.rmi.registry.LocateRegistry`).

```
import java.rmi.registry.LocateRegistry
//...

LocateRegistry.createRegistry(9000)
jmx.connectorServer(port: 9000).start()
```

Connector Client

JmxBuilder.connectorClient() node lets you create JMX connector client object to connect to a JMX MBean Server.

Syntax

```
jmx.connectorClient (
    protocol:"rmi",
    host:"...",
    port:1099,
    url:"...",
)
```

Example - Client Connector

Creating a connector client can be done just as easily. With one line of code, you can create an instance of a JMX Connector Client as shown below.

```
def client = jmx.connectorClient(port: 9000)
client.connect()
```

You can then access the MBeanServerConnection associated with the connector using:

```
client.getMBeanServerConnection()
```

JmxBuilder MBean Export

You can **export a Java object or a Groovy object** with minimal coding. JmxBuilder will even find and **export dynamic Groovy methods** injected at runtime.

Implicit vs Explicit Descriptors

When using the builder, you can **let JmxBuilder implicitly generate** all of your MBean descriptor info. This is useful when you want to write minimal code to quickly export your beans. You can also explicitly declare all descriptor info for the bean. This gives you total control on how you want to describe every piece of information that you want to export for the underlying bean.

The JmxBuilder.export() Node

The **JmxBuilder.export() node provides a container** where all management entities to be exported to the MBeanServer are placed. You can place one or more bean() or timer() nodes as children of the export() node. JmxBuilder will **automatically batch export the entities described** by the nodes to the MBean server for management (see example below).

```
def beans = jmx.export {
    bean(new Foo())
    bean(new Bar())
    bean(new SomeBar())
}
```

In the code snippet above, **JmxBuilder.export() will export three management beans** to the MBean server.

JmxBuilder.export() Syntax

JmxBuilder.export() node supports the **registrationPolicy** parameter to specify how JmxBuilder will behave to resolve bean name collision during MBean registration:

```
jmx.export(policy:"replace|ignore|error")
or
jmx.export(regPolicy:"replace|ignore|error")
```

- **replace** - JmxBuilder.export() will replace any bean already registered with the MBean during export.
- **ignore** - The bean being exported will be ignored if the same bean is already registered.
- **error** - JmxBuilder.export() throws an error upon bean name collision during registration.

Integration with GroovyMBean Class

When you export an MBean to the MBeanServer, **JmxBuilder will return an instance of GroovyMBean** representing the management bean that have been exported by the builder. Nodes such as **bean()** and **timer()** will return an instances of GroovyMBean when they are invoked. The **export()** node returns an **array of all of GroovyMBean[]** representing all managed objects exported to the MBean server.

MBean Registration with JmxBuilder.bean()

This portion of this reference uses class **RequestController** to illustrate how to use JmxBuilder to export runtime management beans. The class is for illustration purpose and can be a POJO or a Groovy bean.

RequestController

```
class RequestController {
```

```
    // constructors
    RequestController() { super() }
    RequestController(Map resource) { }

    // attributes
    boolean isStarted() { true }
    int getRequestCount() { 0 }
    int getResourceCount() { 0 }
    void setRequestLimit(int limit) { }
    int getRequestLimit() { 0 }

    // operations
    void start() { }
    void stop() { }
    void putResource(String name, Object resource) { }
    void makeRequest(String res) { }
    void makeRequest() { }
}
```

Implicit Export

As mentioned earlier, you can use JmxBuilder's flexible syntax to export any POJO/POGO with no descriptor. The builder can automatically describe all aspects of the management beans using implicit defaults. These default values can easily be overridden as we'll see in this in the next section.

The simplest way to export a POJO or POGO is listed below.

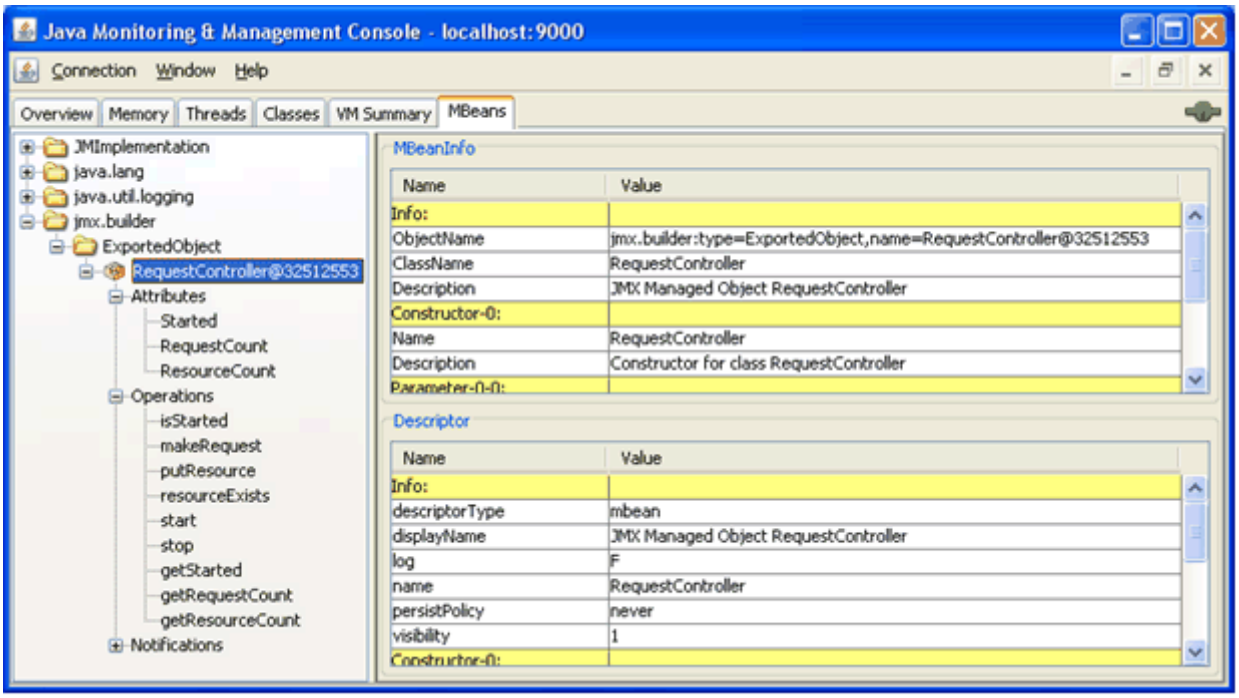
```
jmx.export {
    bean(new RequestController(resource: "Hello World"))
}
```

What this does:

- First, the **JmxBuilder.export() node will export** an MBean to the MBeanServer representing **the declared POJO** instance.
- The builder will **generate a default ObjectName** for the MBean and all other MBean descriptor information.
- **JmxBuilder will automatically export** all declared **attributes** (MBean getter/setters), **constructors**, and **operations** on the instance.
- The exported **attributes** will have **read-only** visibility.

Remember, **JmxBuilder.export() returns an array of GroovyMBean[] objects** for all exported instances. So, once you call JmxBuilder.export(), **you have immediate access to the underlying MBean proxy** (via GroovyMBean).

JConsole view of Exported Bean



JmxBuilder.bean() Syntax

The JmxBuilder.bean() node supports an extensive set of descriptors to describe your bean for management. The JMX MBeanServer uses these descriptors to expose metadata about the bean exposed for management.

```
jmx.export {
    bean(
        target:bean instance,
        name:ObjectName,
        desc:"...",
        attributes:"*",
        attributes:[]
        attributes:[ "AttrubuteName1","AttributeName2",...,"AttributeName_n" ]
        attributes:[
            "AttributeName":"*",
            "AttributeName":[
                desc:"...",
                defaultValue:value,
                writable:true|false,
                editable:true|false,
                onChange:{event-> // event handler}
            ]
        ],

        constructors:"*",
        constructors:[
            "Constructor Name":[],
            "Constructor Name":[ "ParamType1","ParamType2,...,ParamType_n" ],
            "Constructor Name":[
```

```
                desc:"...",
                params:[
                    "ParamType1":"*",
                    "ParamType2":[desc:"...", name:"..."],...,
                    "ParamType_n":[desc:"...", name:"..."]
                ]
            ]
        ],

        operations:"*",
        operations:[ "OperationName1", "OperationName2",...,"OperationNameN" ],
        operations:[
            "OperationName1":"*",
            "OperationName2":[ "type1","type2,"type3" ]
            "OperationName3":[
                desc:"...",
                params:[
                    "ParamType1":"*"
                    "ParamType2":[desc:"...", name:"..."],...,
                    "ParamType_n":[desc:"...", name:"..."]
                ],
                onInvoked:{event-> JmxBuilder.send(event:"", to:"")}
            ]
        ],

        listeners:[
            "ListenerName1":[event: "...", from:ObjectName, call:{event->}],
            "ListenerName2":[event: "...", from:ObjectName, call:&methodPointer]
        ]

    )
}
```

Instead of describing the entire node, the following section explore each attribute separately.

Bean() Node - Specifying MBean ObjectName

Using the bean() node descriptors, you can specify your own MBean ObjectName.

```
def ctrl = new RequestController(resource:"Hello World")
def beans = jmx.export {
    bean(target: ctrl, name: "jmx.tutorial:type=Object")
}
```

The ObjectName can be specified as a String or an instance of the ObjectName.

Bean() Node - Attribute Export

JMX attributes are the setters and getters on the underlying bean. The JmxBuilder.bean() node

provides several ways to flexibly describe and export MBean attributes. You can combine them however you want to achieve any level of attribute visibility. Let's take a look.

Export All Attributes with Wildcard "*"

The following code snippet **will describe and export all attributes** on the bean as read-only. **JmxBuilder will use default values** to describe the attributes that exported for management.

```
def objName = new ObjectName("jmx.tutorial:type=Object")
def beans = jmx.export {
    bean(target: new RequestController(),
    name: objName,
    attributes: "*")
}
```

Export Attribute List

JmxBuilder will let you specify a list of attributes to export.

```
def objName = new ObjectName("jmx.tutorial:type=Object")
def beans = jmx.export {
    bean(
        target: new RequestController(),
        name: objName,
        attributes: ["Resource", "RequestCount"]
    )
}
```

In the snippet above, **only the "Resource" and "RequestCount" attributes will be exported**. Again, since no descriptors are provided, **JmxBuilder will use sensible defaults** to describe the exported attributes.

Export Attribute with Explicit Descriptors

One of the strengths of JmxBuilder is its flexibility in describing MBean. With the builder you can describe all aspects of the MBeans attribute that you want to export to the MBeanServer (see syntax above).

```
def objName = new ObjectName("jmx.tutorial:type=Object")
def beans = jmx.export {
    bean(
        target: new RequestController(),
        name: objName,
        attributes: [
            "Resource": [desc: "The resource to request.", readable: true, writable:
true, defaultValue: "Hello"],
            "RequestCount": "*"
        ]
```

```
    )
}
```

In the snippet above, attribute **"Resource" is fully-described** using all supported descriptors (i.e. desc, readable, writable, defaultValue) for a JMX attribute. However, we use the wildcard to describe attribute **RequestCount** and it will be exported and described using defaults.

Bean() Node - Constructor Export

JmxBuilder **supports the explicit description and export of constructors** defined in the underlying bean. There are several options available when exporting constructors. You can combine them however you want to achieve the desired level of manageability.

Export all Constructors with "*"

You can use the builder's special "*" **notation to export all constructors** declared on the underlying bean. The builder will use default values to describe the MBean constructors.

```
def objName = new ObjectName("jmx.tutorial:type=Object")
def beans = jmx.export {
    bean(
        target: new RequestController(),
        name: objName,
        constructors: "*"
    )
}
```

Export Constructors using Parameter Descriptor

JmxBuilder lets you **target specific constructor** to export **by describing the parameter signature**. This is useful when you have several constructors with different parameter signature and you want to export specific constructors.

```
def objName = new ObjectName("jmx.tutorial:type=Object")
def beans = jmx.export {
    bean(
        target: new RequestController(),
        name: objName,
        constructors: [
            "RequestController": ["Object"]
        ]
    )
}
```

Here, JmxBuilder will **export a constructor that takes one parameter of type "Object"**. Again, JmxBuilder will use default values to fill in the description of the constructor and the parameters.

Export Constructor with Explicit Descriptors

JmxBuilder allows you to **fully-describe** the constructor that you want to target for export (see syntax above).

```
def objName = new ObjectName("jmx.tutorial:type=Object")
def beans = jmx.export {
    bean(target: new RequestController(), name: objName,
        constructors: [
            "RequestController": [
                desc: "Constructor takes param",
                params: ["Object" : [name: "Resource", desc: "Resource for
controller"]]
            ]
        ]
    )
}
```

In the code above, JmxBuilder will target a constructor that takes one parameter for export to the MBeanServer. Notice how the constructor can be fully-described using all optional descriptor keys including parameter descriptors.

Bean() Node - Operation Export

Similar to constructors, JmxBuilder supports the description and export of MBean operations using a flexible notation (see above for syntax). You can combine these notations however you want to achieve the level of operation manageability desired.

Export All Operations with "*"

You can use the builder's special "*" **notation to export all operations** defined on the bean to be exposed for management. The builder will use default descriptor values for the operations being exported.

```
def objName = new ObjectName("jmx.tutorial:type=Object")
def beans = jmx.export {
    bean(
        target: new RequestController(),
        name: objName,
        operations: "*"
    )
}
```

In this snippet, JmxBuilder will **export all bean operations** and will use default values to describe them in the MBeanServer.

Export Operation List

JmxBuilder has a shorthand notation that lets you quickly target operations to be exported by

providing a list of methods to export.

```
def objName = new ObjectName("jmx.tutorial:type=Object")
def beans = jmx.export {
    bean(
        target: new RequestController(),
        name: objName,
        operations: ["start", "stop"]
    )
}
```

In the snippet above, the **builder will only export methods start() and stop()**. All other methods will be ignored. JmxBuilder will use default descriptor values to describe the operations being exported.

Export Operations by Signature

Using JmxBuilder, you can target methods to export for management using the methods' parameter signature. This is useful when you want to distinguish methods with the same name that you want to export (i.e. stop() instead of stop(boolean)).

```
def objName = new ObjectName("jmx.tutorial:type=Object")
def beans = jmx.export {
    bean(
        target: new RequestController(),
        name: objName,
        operations: [
            "makeRequest": ["String"]
        ]
    )
}
```

In the snippet above, JmxBuilder would **select method makeRequest(String)** to be exported instead of the other version makeRequest() which takes no parameter. In this shorthand context, the signature is specified as a list of type (i.e. "String").

Export Operations with Explicit Descriptors

JmxBuilder supports detailed descriptors for bean operations. You can supply deep descriptor info about any operation on your bean including a name, description, method parameters, parameter type, and parameter description.

```
def objName = new ObjectName("jmx.tutorial:type=Object")
def beans = jmx.export {
    bean(target: new RequestController(), name: objName,
        operations: [
            "start": [desc: "Starts request controller"],
            "stop": [desc: "Stops the request controller"],
```

```
            "setResource": [params: ["Object"]],
            "makeRequest": [
                desc: "Executes the request.",
                params: [
                    "String": [name: "Resource", desc: "The resource to request"]
                ]
            ]
        ]
    )
}
```

The snippet above shows all the ways JmxBuilder allows you to describe an operation targeted for management:

- Operations **start() and stop()** are described by the "desc" key (this is enough since there are no params).
- In operation **setResource()** uses of a shorthand version of **params**: to describe the parameters for the method.
- **makeRequest()** uses the extended descriptor syntax to describe all aspects of the operation.

Embedding Descriptor

JmxBuilder supports the ability to **embed descriptors directly in your Groovy class**. So, instead of wrapping your description around the declared object (as we've seen here), you can embed your JMX descriptors directly in your class.

RequestControllerGroovy

```
class RequestControllerGroovy {
    // attributes
    boolean started
    int requestCount
    int resourceCount
    int requestLimit
    Map resources

    // operations
    void start() { }
    void stop(){ }
    void putResource(String name, Object resource) { }
    void makeRequest(String res) { }
    void makeRequest() { }

    static descriptor = [
        name: "jmx.builder:type=EmbeddedObject",
        operations: ["start", "stop", "putResource"],
        attributes: "*"
    ]
}
```

```
// export
jmx.export(
    bean(new RequestControllerGroovy())
)
```

There are two things going on in the code above:

- Groovy class RequestControllerGroovy is defined and includes a **static descriptor** member. That member is used to declare a JmxBuilder descriptor to describe member of the class targeted for JMX export.
- The second part of the code shows how to use JmxBuilder to export that class for management.

Timer Export

JMX standards mandate that the implementation of the API makes available a timer service. Since JMX is a component-based architecture, timers provide an excellent signalling mechanism to communicate to registered listener components in the MBeanServer. JmxBuilder supports the creation and export of timers using the same easy syntax we've seen so far.

Timer Node Syntax

```
timer(
    name:ObjectName,
    event:"...",
    message:"...",
    data:dataValue
    startDate:"now"|dateValue
    period:"99d"|"99h"|"99m"|"99s"|99
    occurrences:long
)
```

The timer() node supports several attributes:

- **name**: - Required The qualified JMX ObjectName instance (or String) for the timer.
- **event**: - The JMX event type string that will be broadcast with every timing signal (default **"jmx.builder.event"**).
- **message**: - An optional string value that can be sent to listeners.
- **data**: - An optional object that can be sent to listeners of timing signal.
- **startDate**: - When to start timer. Set of valid values ["now", date object]. Default is "now"
- **period**: - A timer's period expressed as either a number of millisecond or time unit (day, hour, minute, second). See description below.
- **occurrences**: - A number indicating the number of time to repeat timer. Default is forever.

Exporting a Timer

```
def timer = jmx.timer(name: "jmx.builder:type=Timer", event: "heartbeat", period:
"1s")
timer.start()
```

This snippet above **describes, creates, and exports a standard JMX Timer** component. Here, the **timer()** node **returns a GroovyMBean** that represents the registered timer MBean in the MBeanServer.

An **alternative way of exporting timers** is within the JmxBuilder.export() node.

```
def beans = jmx.export {
    timer(name: "jmx.builder:type=Timer1", event: "event.signal", period: "1s")
    timer(name: "jmx.builder:type=Timer2", event: "event.log", period: "1s")
}
beans[0].start()
beans[1].start()
```

Timer Period

The **timer() node supports a flexible notation** for specifying the **timer period values**. You can specify the time in second, minutes, hour, and day. The default is millisecond.

- timer(**period: 100**) = 100 millisecond
- timer(**period: "1s"**) = 1 second
- timer(**period: "1m"**) = 1 minute
- timer(**period: "1h"**) = 1 hour
- timer(**period: "1d"**) = 1 day

The node will automatically translate.

JmxBuilder and Events

An integral part of **JMX** is its **event model**. Registered management beans can **communicate with each other by broadcasting events** on the MBeanServer's event bus. **JmxBuilder provides several ways to easily listen and react to events** broadcasted on the MBeanServer's event bus. Developers can **capture any event on the bus or throw their own** to be consumed by other components registered on the MBeanServer.

Event Handling Closures

JmxBuilder leverages Groovy's use of closures to provide simple, yet elegant, mean of reacting to JMX events. JmxBuilder supports two closure signatures:

Parameterless

```
callback = { ->
    // event handling code here.
}
```

JmxBuilder executes the closure and passes no information about the event that was captured on the bus.

With Event Parameter

```
callback = { event ->
    // event handling code
}
```

JmxBuilder will pass an **"event" object to the closure** using this format. The event object contains information about the event was intercepted so that it can be handled by the handler. The parameter will contain different set of info depending on the event that was captured.

Handling Attribute onChange Event

When describing attributes (see bean() node section above), you can **provide a closure (or method pointer) for callback to be executed when the value of the attribute is updated** on the exported MBean. This gives developers an opportunity to listen to and react to state changes on the MBean.

```
jmx.export {
    bean(
        target: new RequestController(), name: "jmx.tutorial:type=Object",
        attributes: [
            "Resource": [
                readable: true, writable: true,
                onChange: { e ->
                    println e.oldValue
                    println e.newValue
                }
            ]
        ]
    )
}
```

The sample snippet above shows how to **specify an "onChange" callback closure** when describing MBean attributes. In this sample code, whenever attribute "Resource" is updated via the exported MBean, the **onChange event will be executed**.

Attribute onChange Event Object

When handling the attribute onChange event, the handler closure will receive an event object with

the following info:

- event.**oldValue** - the previous attribute value before the change event.
- event.**newValue** - the new value of the attribute after the change.
- event.**attribute** - the name of the attribute on which the event occurred.
- event.**attributeType** - the data type of the attribute that causes the event.
- event.**sequenceNumber** - a numeric value representing the sequence number of event.
- event.**timeStamp** - a time stamp for the event occurrence.

Handling Operation onCall Event

Similar to mbean attributes, JmxBuilder affords developers the **ability to listen for operation invocation** on an MBean registered in the MBeaServer. JmxBuilder accepts a **callback closure that will be executed after the MBean method has invoked**.

```
class EventHandler {
    void handleStart(e){
        println e
    }
}
```

```
def handler = new EventHandler()

def beans = jmx.export {
    bean(target: new RequestController(), name: "jmx.tutorial:type=Object",
        operations: [
            "start": [
                desc:"Starts request controller",
                onCall:handler.&handleStart
            ]
        ]
    )
}
```

The snippet above **shows how to declare an "onCall" closure to be used as listener** when operation "start()" is invoked on the MBean. This sample **uses the method pointer syntax** to illustrate the versatility of JmxBuilder.

Operation onCall Event Object

When handling the operation onCall event, the callback closure will receive an event object with the following info:

- event.**event** - the event type string that was broadcasted.
- event.**source** - The object on which the method was invoked.

- event.**data** - the data type of the attribute that causes the event.
- event.**sequenceNumber** - a numeric value representing the sequence number of event.
- event.**timeStamp** - a time stamp for the event occurrence.

Listener MBean

When you export an MBean with the bean() node, you can define events the MBean can listen and react to. The bean() node provides a "listeners:" attribute that lets you define event listeners that your bean can react to.

```
def beans = jmx.export {
    timer(name: "jmx.builder:type=Timer", event: "heartbeat", period: "1s").start()
    bean(target: new RequestController(), name: "jmx.tutorial:type=Object",
        operations: "*",
        listeners: [
            heartbeat: [
                from: "jmx.builder:type=Timer",
                call: { e ->
                    println e
                }
            ]
        ]
    )
}
```

In the sample above, we see the **syntax for adding listeners to an exported MBean**.

- First, a **timer is exported** and started.
- Then, an **MBean is declared that will listen to the timer** event and do something meaningful.
- The **"heartbeat:"** name is arbitrary and has no correlation to the timer declared above.
- The **source** of the event **is specified using the "from:" attribute**.

You can also specify an **event type** you are interested in receiving from a broadcaster (since a broadcaster can be emitting multiple events).

Listening to JMX Events

In some cases, you will want to create stand-alone event listeners (not attached to exported MBeans). JmxBuilder provides the Listener() node to let you create JMX listeners that can listen to MBeanServer events. This is useful when creating JMX client applications to monitor/manage JMX agents on remote JMX MBeanServers.

Listener Node Syntax

```
jmx.listener(
    event: "...",
    from: "object name" | ObjectName,
```

```
    call: { event-> }
)
```

Here is the description of the **listener()** node attributes:

- event: An optional string that identifies the JMX event type to listen for.
- from (required): The JMX ObjectName of the component to listen to. This can be specified as a string or an instance of ObjectName.
- call: The closure to execute when the event is captured. This can also be specified as a Groovy method pointer.

Here is an example of JmxBuilder's listener node:

```
jmx.timer(name: "jmx.builder:type=Timer", period: "1s").start()

jmx.listener(
    from: "jmx.builder:type=Timer",
    call: { e ->
        println "beep..."
    }
)
```

This example shows how you can use a stand-alone listener (outside an MBean export). Here, we **export a timer with a 1 second** resolution. Then, we specify a listener to that timer that will print "beep" every second.

Emitting JMX Events

JmxBuilder provides the **tools needed to broadcast your own events** on the MBeanServer's event bus. There are no restrictions on the event type you can broadcast. You simply **declare your emitter** and the event type that you want to send, then **broadcast your event** at any time. Any registered component in the MBeanServer can register themselves to listen to your events.

Emitter Syntax

```
jmx.emitter(name:"Object:Name", event:"type")
```

The attributes for the node Emitter() can be summarized as follows:

- name: an optional JMX ObjectName used to register your emitter in the MBeanServer. Default is jmx.builder:type=Emitter,name=Emitter@OBJECT_HASH_VALUE
- event: an option string value that describes the JMX event type. Default is **"jmx.builder.event.emitter"**.

Declare the Emitter

```
def emitter = jmx.emitter()
```

The snippet **declares the emitter using implicit descriptor syntax**. JmxBuilder will do the followings:

- Create and register an emitter MBean with a default ObjectName.
- Setup a **default event type** with value **"jmx.builder.event.emitter"**.
- Return a GroovyMBean representing the emitter.

As with other nodes in the builder, **you can override all keys in the emitter() node**. You can specify the **ObjectName** and the **event type**.

Broadcast Event

Once you have declared your emitter, you can broadcast your event.

```
emitter.send()
```

The sample above shows the **emitter sending an event**, once it has been declared. Any JMX component registered in the MBeanServer can register to receive message from this emitter.

Sending Event Objects

You can optionally pass data to the receiver when you send the message.

```
emitter.send("Hello!")
```

If you use an **event listener closure (see above) that accepts a parameter**, you can access that value.

Further JMX Information

- Monitoring the Java Virtual Machine
- Using Groovy for System Management
- Groovier jconsole!
- JMX Scripts with Eclipse Monkey
- Using JMX to monitor Apache ActiveMQ

Creating Swing UIs

Creating Swing UIs is made easy thanks to the use of SwingBuilder.

Security

Security is a complex and multi-faceted issue and needs to be addressed in a holistic way. Groovy offers some features to improve security, but organisations concerned about security should already be addressing other necessary aspects such as network security, file-system security, operating system security, database security, passwords and potentially encryption.

Also, since Groovy runs on the JDK and optionally uses other library dependencies, users should ensure their JDK and all dependencies are up-to-date with respect to the latest security fixes.

With regard to security issues that may affect the Groovy project itself, the project follows the Apache general guidelines for handling security vulnerabilities. See also the project's security policy and list of past vulnerabilities.

By virtue of running on the JVM and following various Java conventions, Groovy programs offer some of the same security features as Java programs, including:

- programs cannot access arbitrary memory locations
- final variables cannot be changed
- array bounds are checked
- class loaders perform bytecode verification when loading classes
- casting cannot be done to an incompatible class
- access is available to APIs for encryption and authentication

Special security support is provided through:

- org.codehaus.groovy.control.customizers.SecureASTCustomizer secures source code by controlling what code constructs are permitted or prohibited in a code base (or part of a code base)
- Default XML processing has secure processing enabled and doctype definitions disabled
- Groovy's SQL processing features provide support to guard against SQL injection
- Temporary directory creation protects against known security vulnerabilities such as privilege escalation if scripts are stored in operating system temp directories

Design patterns in Groovy

Using design patterns with Java is a well-established topic. Design patterns also apply to Groovy:

- some patterns carry over directly (and can make use of normal Groovy syntax improvements for greater readability)
- some patterns are no longer required because they are built right into the language or because Groovy supports a better way of achieving the intent of the pattern
- some patterns that have to be expressed at the design level in other languages can be implemented directly in Groovy (due to the way Groovy can blur the distinction between design and implementation)

Patterns

Abstract Factory Pattern

The Abstract Factory Pattern provides a way to encapsulate a group of individual factories that have a common theme. It embodies the intent of a normal factory, i.e. remove the need for code using an interface to know the concrete implementation behind the interface, but applies to a set of interfaces and selects an entire family of concrete classes which implement those interfaces.

As an example, I might have interfaces Button, TextField and Scrollbar. I might have WindowsButton, MacButton, FlashButton as concrete classes for Button. I might have WindowsScrollBar, MacScrollBar and FlashScrollBar as concrete implementations for ScrollBar. Using the Abstract Factory Pattern should allow me to select which windowing system (i.e. Windows, Mac, Flash) I want to use once and from then on should be able to write code that references the interfaces but is always using the appropriate concrete classes (all from the one windowing system) under the covers.

Example

Suppose we want to write a game system. We might note that many games have very similar features and control.

We decide to try to split the common and game-specific code into separate classes.

First let's look at the game-specific code for a Two-up game:

```
class TwoupMessages {
    def welcome = 'Welcome to the twoup game, you start with $1000'
    def done = 'Sorry, you have no money left, goodbye'
}

class TwoupInputConverter {
    def convert(input) { input.toInteger() }
}

class TwoupControl {
    private money = 1000
    private random = new Random()
    private tossWasHead() {
        def next = random.nextInt()
        return next % 2 == 0
    }
    def moreTurns() {
        if (money > 0) {
            println "You have $money, how much would you like to bet?"
            return true
        }

        false
    }
```

```
    def play(amount) {
        def coin1 = tossWasHead()
        def coin2 = tossWasHead()
        if (coin1 && coin2) {
            money += amount
            println 'You win'
        } else if (!coin1 && !coin2) {
            money -= amount
            println 'You lose'
        } else {
            println 'Draw'
        }
    }
}
```

Now, let's look at the game-specific code for a number guessing game:

```
class GuessGameMessages {
    def welcome = 'Welcome to the guessing game, my secret number is between 1 and
100'
    def done = 'Correct'
}

class GuessGameInputConverter {
    def convert(input) { input.toInteger() }
}

class GuessGameControl {
    private lower = 1
    private upper = 100
    private guess = new Random().nextInt(upper - lower) + lower
    def moreTurns() {
        def done = (lower == guess || upper == guess)
        if (!done) {
            println "Enter a number between $lower and $upper"
        }

        !done
    }
    def play(nextGuess) {
        if (nextGuess <= guess) {
            lower = [lower, nextGuess].max()
        }
        if (nextGuess >= guess) {
            upper = [upper, nextGuess].min()
        }
    }
}
```

Now, let's write our factory code:

```
def guessFactory = [messages: GuessGameMessages, control: GuessGameControl, converter:
GuessGameInputConverter]
def twoupFactory = [messages: TwoupMessages, control: TwoupControl, converter:
TwoupInputConverter]

class GameFactory {
    def static factory
    def static getMessages() { return factory.messages.newInstance() }
    def static getControl() { return factory.control.newInstance() }
    def static getConverter() { return factory.converter.newInstance() }
}
```

The important aspect of this factory is that it allows selection of an entire family of concrete classes.

Here is how we would use the factory:

```
GameFactory.factory = twoupFactory
def messages = GameFactory.messages
def control = GameFactory.control
def converter = GameFactory.converter
println messages.welcome
def reader = new BufferedReader(new InputStreamReader(System.in))
while (control.moreTurns()) {
    def input = reader.readLine().trim()
    control.play(converter.convert(input))
}
println messages.done
```

Note that the first line configures which family of concrete game classes we will use. It's not important that we selected which family to use by using the factory property as shown in the first line. Other ways would be equally valid examples of this pattern. For example, we may have asked the user which game they wanted to play or determined which game from an environment setting.

With the code as shown, the game might look like this when run:

```
Welcome to the twoup game, you start with $1000
You have 1000, how much would you like to bet?
300
Draw
You have 1000, how much would you like to bet?
700
You win
You have 1700, how much would you like to bet?
1700
You lose
```

```
Sorry, you have no money left, goodbye
```

If we change the first line of the script to GameFactory.factory = guessFactory, then the sample run might look like this:

```
Welcome to the guessing game, my secret number is between 1 and 100
Enter a number between 1 and 100
75
Enter a number between 1 and 75
35
Enter a number between 1 and 35
15
Enter a number between 1 and 15
5
Enter a number between 5 and 15
10
Correct
```

Adapter Pattern

The Adapter Pattern (sometimes called the wrapper pattern) allows objects satisfying one interface to be used where another type of interface is expected. There are two typical flavours of the pattern: the *delegation* flavour and the *inheritance* flavour.

Delegation Example

Suppose we have the following classes:

```
class SquarePeg {
    def width
}

class RoundPeg {
    def radius
}

class RoundHole {
    def radius
    def pegFits(peg) {
        peg.radius <= radius
    }
    String toString() { "RoundHole with radius $radius" }
}
```

We can ask the `RoundHole` class if a `RoundPeg` fits in it, but if we ask the same question for a `SquarePeg`, then it will fail because the `SquarePeg` class doesn't have a `radius` property (i.e. doesn't satisfy the required interface).

To get around this problem, we can create an adapter to make it appear to have the correct interface. It would look like this:

```
class SquarePegAdapter {
    def peg
    def getRadius() {
        Math.sqrt(((peg.width / 2) ** 2) * 2)
    }
    String toString() {
        "SquarePegAdapter with peg width $peg.width (and notional radius $radius)"
    }
}
```

We can use the adapter like this:

```
def hole = new RoundHole(radius: 4.0)
(4..7).each { w ->
    def peg = new SquarePegAdapter(peg: new SquarePeg(width: w))
    if (hole.pegFits(peg)) {
        println "peg $peg fits in hole $hole"
    } else {
        println "peg $peg does not fit in hole $hole"
    }
}
```

Which results in the following output:

```
peg SquarePegAdapter with peg width 4 (and notional radius 2.8284271247461903) fits in
hole RoundHole with radius 4.0
peg SquarePegAdapter with peg width 5 (and notional radius 3.5355339059327378) fits in
hole RoundHole with radius 4.0
peg SquarePegAdapter with peg width 6 (and notional radius 4.242640687119285) does not
fit in hole RoundHole with radius 4.0
peg SquarePegAdapter with peg width 7 (and notional radius 4.949747468305833) does not
fit in hole RoundHole with radius 4.0
```

Inheritance Example

Let's consider the same example again using inheritance. First, here are the original classes (unchanged):

```
class SquarePeg {
    def width
}

class RoundPeg {
    def radius
```

```
}

class RoundHole {
    def radius
    def pegFits(peg) {
        peg.radius <= radius
    }
    String toString() { "RoundHole with radius $radius" }
}
```

An adapter using inheritance:

```
class SquarePegAdapter extends SquarePeg {
    def getRadius() {
        Math.sqrt(((width / 2) ** 2) * 2)
    }
    String toString() {
        "SquarePegAdapter with width $width (and notional radius $radius)"
    }
}
```

Using the adapter:

```
def hole = new RoundHole(radius: 4.0)
(4..7).each { w ->
    def peg = new SquarePegAdapter(width: w)
    if (hole.pegFits(peg)) {
        println "peg $peg fits in hole $hole"
    } else {
        println "peg $peg does not fit in hole $hole"
    }
}
```

The output:

```
peg SquarePegAdapter with width 4 (and notional radius 2.8284271247461903) fits in
hole RoundHole with radius 4.0
peg SquarePegAdapter with width 5 (and notional radius 3.5355339059327378) fits in
hole RoundHole with radius 4.0
peg SquarePegAdapter with width 6 (and notional radius 4.242640687119285) does not fit
in hole RoundHole with radius 4.0
peg SquarePegAdapter with width 7 (and notional radius 4.949747468305833) does not fit
in hole RoundHole with radius 4.0
```

Adapting using Closures

As a variation of the previous examples, we could instead define the following interface:

```
interface RoundThing {
    def getRadius()
}
```

We can then define an adapter as a closure as follows:

```
def adapter = {
    p -> [getRadius: { Math.sqrt(((p.width / 2) ** 2) * 2) }] as RoundThing
}
```

And use it like this:

```
def peg = new SquarePeg(width: 4)
if (hole.pegFits(adapter(peg))) {
    // ... as before
}
```

Adapting using the ExpandoMetaClass

As of Groovy 1.1, there is a built-in MetaClass which can automatically add properties and methods dynamically.

Here is how the example would work using that feature:

```
def peg = new SquarePeg(width: 4)
peg.metaClass.radius = Math.sqrt(((peg.width / 2) ** 2) * 2)
```

After you create a peg object, you can simply add a property to it on the fly. No need to change the original class and no need for an adapter class.

Bouncer Pattern

The Bouncer Pattern describes usage of a method whose sole purpose is to either throw an exception (when particular conditions hold) or do nothing. Such methods are often used to defensively guard pre-conditions of a method.

When writing utility methods, you should always guard against faulty input arguments. When writing internal methods, you may be able to ensure that certain pre-conditions always hold by having sufficient unit tests in place. Under such circumstances, you may reduce the desirability to have guards on your methods.

Groovy differs from other languages in that you frequently use the `assert` method within your methods rather than having a large number of utility checker methods or classes.

Null Checking Example

We might have a utility method such as:

```
class NullChecker {
    static check(name, arg) {
        if (arg == null) {
            throw new IllegalArgumentException(name + ' is null')
        }
    }
}
```

And we would use it like this:

```
void doStuff(String name, Object value) {
    NullChecker.check('name', name)
    NullChecker.check('value', value)
    // do stuff
}
```

But a more Groovy way to do this would simply be like this:

```
void doStuff(String name, Object value) {
    assert name != null, 'name should not be null'
    assert value != null, 'value should not be null'
    // do stuff
}
```

Validation Example

As an alternative example, we might have this utility method:

```
class NumberChecker {
    static final String NUMBER_PATTERN = "\\\\d+(\\\\.\\\\d+(E-?\\\\d+)?)?"
    static isNumber(str) {
        if (!str ==~ NUMBER_PATTERN) {
            throw new IllegalArgumentException("Argument '$str' must be a number")
        }
    }
    static isNotZero(number) {
        if (number == 0) {
            throw new IllegalArgumentException('Argument must not be 0')
        }
    }
}
```

And we would use it like this:

```
def stringDivide(String dividendStr, String divisorStr) {
    NumberChecker.isNumber(dividendStr)
    NumberChecker.isNumber(divisorStr)
    def dividend = dividendStr.toDouble()
    def divisor = divisorStr.toDouble()
    NumberChecker.isNotZero(divisor)
    dividend / divisor
}

println stringDivide('1.2E2', '3.0')
// => 40.0
```

But with Groovy we could just as easily use:

```
def stringDivide(String dividendStr, String divisorStr) {
    assert dividendStr =~ NumberChecker.NUMBER_PATTERN
    assert divisorStr =~ NumberChecker.NUMBER_PATTERN
    def dividend = dividendStr.toDouble()
    def divisor = divisorStr.toDouble()
    assert divisor != 0, 'Divisor must not be 0'
    dividend / divisor
}
```

Chain of Responsibility Pattern

In the Chain of Responsibility Pattern, objects using and implementing an interface (one or more methods) are intentionally loosely coupled. A set of objects that *implement* the interface are organised in a list (or in rare cases a tree). Objects using the interface make requests from the first *implementor* object. It will decide whether to perform any action itself and whether to pass the request further down the line in the list (or tree). Sometimes a default implementation for some request is also coded into the pattern if none of the implementors respond to the request.

Example using traditional classes

In this example, the script sends requests to the `lister` object. The `lister` points to a `UnixLister` object. If it can't handle the request, it sends the request to the `WindowsLister`. If it can't handle the request, it sends the request to the `DefaultLister`.

```
class UnixLister {
    private nextInLine
    UnixLister(next) { nextInLine = next }
    def listFiles(dir) {
        if (System.getProperty('os.name') == 'Linux') {
            println "ls $dir".execute().text
        } else {
            nextInLine.listFiles(dir)
        }
    }
```

```
}

class WindowsLister {
    private nextInLine
    WindowsLister(next) { nextInLine = next }
    def listFiles(dir) {
        if (System.getProperty('os.name').startsWith('Windows')) {
            println "cmd.exe /c dir $dir".execute().text
        } else {
            nextInLine.listFiles(dir)
        }
    }
}

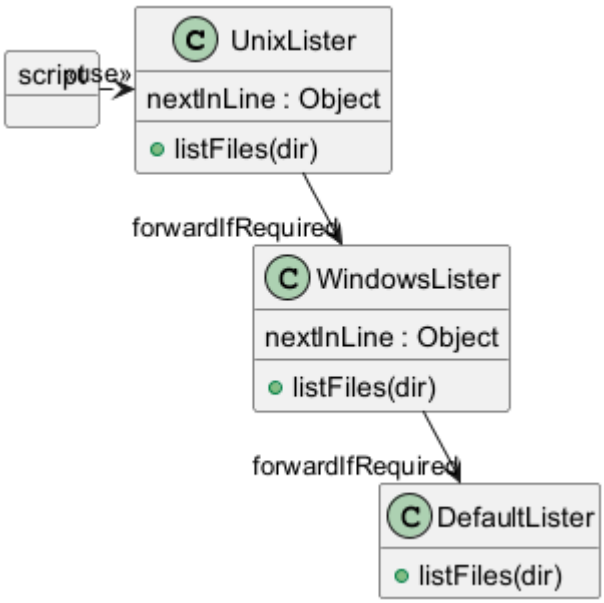
class DefaultLister {
    def listFiles(dir) {
        new File(dir).eachFile { f -> println f }
    }
}

def lister = new UnixLister(new WindowsLister(new DefaultLister()))

lister.listFiles('Downloads')
```

The output will be a list of files (with slightly different format depending on the operating system).

Here is a UML representation:



Example using simplifying strategies

For simple cases, consider simplifying your code by not requiring the chain of classes. Instead, use Groovy truth and the elvis operator as shown here:

```
String unixListFiles(dir) {
    if (System.getProperty('os.name') == 'Linux') {
        "ls $dir".execute().text
    }
}

String windowsListFiles(dir) {
    if (System.getProperty('os.name').startsWith('Windows')) {
        "cmd.exe /c dir $dir".execute().text
    }
}

String defaultListFiles(dir) {
    new File(dir).listFiles().collect{ f -> f.name }.join('\\n')
}

def dir = 'Downloads'
println unixListFiles(dir) ?: windowsListFiles(dir) ?: defaultListFiles(dir)
```

Or Groovy's switch as shown here:

```
String listFiles(dir) {
    switch(dir) {
    case { System.getProperty('os.name') == 'Linux' }:
        return "ls $dir".execute().text
    case { System.getProperty('os.name').startsWith('Windows') }:
        return "cmd.exe /c dir $dir".execute().text
    default:
        new File(dir).listFiles().collect{ f -> f.name }.join('\\n')
    }
}

println listFiles('Downloads')
```

Alternatively, for Groovy 3+, consider using streams of lambdas as shown here:

```
Optional<String> unixListFiles(String dir) {
    Optional.ofNullable(dir)
        .filter(d -> System.getProperty('os.name') == 'Linux')
        .map(d -> "ls $d".execute().text)
}

Optional<String> windowsListFiles(String dir) {
    Optional.ofNullable(dir)
        .filter(d -> System.getProperty('os.name').startsWith('Windows'))
        .map(d -> "cmd.exe /c dir $d".execute().text)
}
```

```
Optional<String> defaultListFiles(String dir) {
    Optional.ofNullable(dir)
        .map(d -> new File(d).listFiles().collect{ f -> f.name }.join('\\n'))
}

def dir = 'Downloads'
def handlers = [this::unixListFiles, this::windowsListFiles, this::defaultListFiles]
println handlers.stream()
    .map(f -> f(dir))
    .filter(Optional::isPresent)
    .map(Optional::get)
    .findFirst()
    .get()
```

When not to use

If your use of chain of responsibility involves frequent use of the `instanceof` operator, like here:

```
import static Math.PI as π
abstract class Shape {
    String name
}
class Polygon extends Shape {
    String name
    double lengthSide
    int numSides
}
class Circle extends Shape {
    double radius
}

class CircleAreaCalculator {
    def next
    def area(shape) {
        if (shape instanceof Circle) {            ①
            return shape.radius ** 2 * π
        } else {
            next.area(shape)
        }
    }
}
class SquareAreaCalculator {
    def next
    def area(shape) {
        if (shape instanceof Polygon && shape.numSides == 4) {      ①
            return shape.lengthSide ** 2
        } else {
            next.area(shape)
        }
    }
```

```
}
class DefaultAreaCalculator {
    def area(shape) {
        throw new IllegalArgumentException("Don't know how to calculate area for
$shape")
    }
}

def chain = new CircleAreaCalculator(next: new SquareAreaCalculator(next: new
DefaultAreaCalculator()))
def shapes = [
    new Circle(name: 'Circle', radius: 5.0),
    new Polygon(name: 'Square', lengthSide: 10.0, numSides: 4)
]
shapes.each { println chain.area(it) }
```

① instanceof code smell

It could indicate that instead of using the chain of responsibility pattern, you might consider using richer types, perhaps in combination with Groovy's multimethods. For example, perhaps this:

```
// ...
class Square extends Polygon {
    // ...
}

double area(Circle c) {
    c.radius ** 2 * π
}

double area(Square s) {
    s.lengthSide ** 2
}

def shapes = [
    new Circle(radius: 5.0),
    new Square(lengthSide: 10.0, numSides: 4)
]
shapes.each { println area(it) }
```

or using more traditional object-oriented style like this:

```
import static Math.PI as π
interface Shape {
    double area()
}
abstract class Polygon implements Shape {
    double lengthSide
    int numSides
```

```
    abstract double area()
}
class Circle implements Shape {
    double radius
    double area() {
        radius ** 2 * π
    }
}
class Square extends Polygon {
    // ...
    double area() {
        lengthSide ** 2
    }
}

def shapes = [
    new Circle(radius: 5.0),
    new Square(lengthSide: 10.0, numSides: 4)
]
shapes.each { println it.area() }
```

Going further

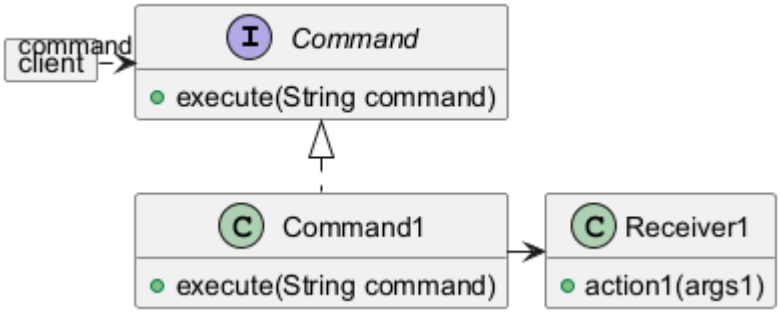
Other variations to this pattern:

- we could have an explicit interface in the traditional example, e.g. `Lister`, to statically type the implementations but because of *duck-typing* this is optional
- we could use a chain tree instead of a list, e.g. `if (animal.hasBackbone())` delegate to `VertebrateHandler` else delegate to `InvertebrateHandler`
- we could always pass down the chain even if we processed a request (no early return)
- we could decide at some point to not respond and not pass down the chain (pre-emptive abort)
- we could use Groovy's meta-programming capabilities to pass unknown methods down the chain, e.g. combine chain of responsibility with the use of `methodMissing`

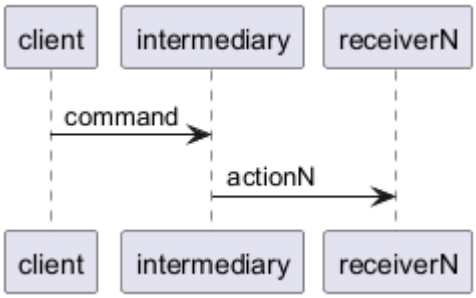
Command Pattern

The Command Pattern is a pattern for loosely coupling a client object which wants to execute a series of commands and receiver objects which enact those commands. Instead of talking to receivers directly, clients interact with an intermediary object which then relays the necessary commands to the receivers. The pattern is in common use within the JDK, for example the api:javax.swing.Action[] class in Swing decouples swing code from receivers like buttons, menu items and panels.

The class diagram showing the typical classes is:



The sequence of interactions is as shown below for an arbitrary receiver:



Example with traditional classes

The relevant classes required for turning a light on and off (see the example in the earlier wikipedia reference) would be as follows:

```
interface Command {
    void execute()
}

// invoker class
class Switch {
    private final Map<String, Command> commandMap = new HashMap<>()

    void register(String commandName, Command command) {
        commandMap[commandName] = command
    }

    void execute(String commandName) {
        Command command = commandMap[commandName]
        if (!command) {
            throw new IllegalStateException("no command registered for " +
commandName)
        }
        command.execute()
    }
}

// receiver class
class Light {
    void turnOn() {
        println "The light is on"
```

```
    }

    void turnOff() {
        println "The light is off"
    }
}

class SwitchOnCommand implements Command {
    Light light

    @Override // Command
    void execute() {
        light.turnOn()
    }
}

class SwitchOffCommand implements Command {
    Light light

    @Override // Command
    void execute() {
        light.turnOff()
    }
}

Light lamp = new Light()
Command switchOn = new SwitchOnCommand(light: lamp)
Command switchOff = new SwitchOffCommand(light: lamp)

Switch mySwitch = new Switch()
mySwitch.register("on", switchOn)
mySwitch.register("off", switchOff)

mySwitch.execute("on")
mySwitch.execute("off")
```

Our client scripts sends `execute` commands to an intermediary and knows nothing about any specific receivers, or any specific action method names and arguments.

Simplifying variations

Given that Groovy has first-class function support, we can do away with the actual command classes (like `SwitchOnCommand`) by instead using closures as shown here:

```
interface Command {
    void execute()
}

// invoker class
class Switch {
```

```
    private final Map<String, Command> commandMap = [:]

    void register(String commandName, Command command) {
        commandMap[commandName] = command
    }

    void execute(String commandName) {
        Command command = commandMap[commandName]
        if (!command) {
            throw new IllegalStateException("no command registered for $commandName")
        }
        command.execute()
    }
}

// receiver class
class Light {
    void turnOn() {
        println 'The light is on'
    }

    void turnOff() {
        println 'The light is off'
    }
}

Light lamp = new Light()

Switch mySwitch = new Switch()
mySwitch.register("on", lamp.&turnOn)       ①
mySwitch.register("off", lamp.&turnOff)     ①

mySwitch.execute("on")
mySwitch.execute("off")
```

① Command closures (here method closures) but could be lambdas/method references for Groovy 3+

We can simplify further using the JDK's existing `Runnable` interface and using a switch map rather than a separate `Switch` class as shown here:

```
class Light {
    void turnOn() {
        println 'The light is on'
    }

    void turnOff() {
        println 'The light is off'
    }
}
```

```
class Door {
    static void unlock() {
        println 'The door is unlocked'
    }
}

Light lamp = new Light()
Map<String, Runnable> mySwitch = [
    on: lamp::turnOn,
    off: lamp::turnOff,
    unlock: Door::unlock
]

mySwitch.on()
mySwitch.off()
mySwitch.unlock()
```

We have added an additional `Door` receiver to illustrate how to expand the original example. Running this script results in:

```
The light is on
The light is off
The door is unlocked
```

As a variation, if the command names aren't important to us, we can forgo using the switch map and just have a list of tasks to invoke as shown here:

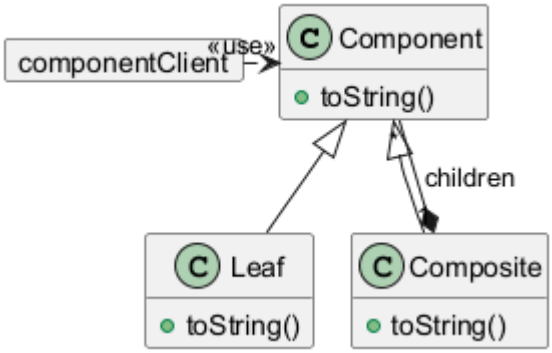
```
// ...
List<Runnable> tasks = [lamp::turnOn, lamp::turnOff, Door::unlock]
tasks.each{ it.run() }
```

Composite Pattern

The Composite Pattern allows you to treat single instances of an object the same way as a group of objects. The pattern is often used with hierarchies of objects. Typically, one or more methods should be callable in the same way for either *leaf* or *composite* nodes within the hierarchy. In such a case, composite nodes typically invoke the same named method for each of their children nodes.

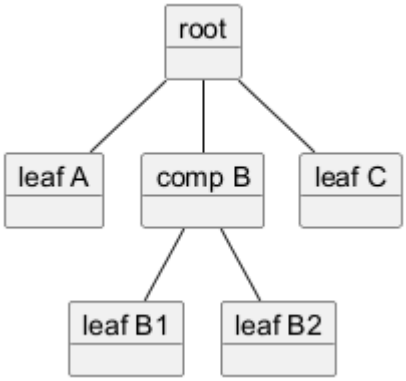
Example

Consider this usage of the composite pattern where we want to call `toString()` on either `Leaf` or `Composite` objects.



In Java, the `Component` class is essential as it provides the type used for both leaf and composite nodes. In Groovy, because of duck-typing, we don't need it for that purpose, however, it can still serve as a useful place to place common behaviour between the leaf and composite nodes.

For our purposes, we will assemble the following hierarchy of components.



Here is the code:

```
abstract class Component {
    def name
    def toString(indent) {
        ("-" * indent) + name
    }
}

class Composite extends Component {
    private children = []
    def toString(indent) {
        def s = super.toString(indent)
        children.each { child ->
            s += "\\n" + child.toString(indent + 1)
        }
        s
    }
    def leftShift(component) {
        children << component
    }
}
```

```
class Leaf extends Component { }

def root = new Composite(name: "root")
root << new Leaf(name: "leaf A")
def comp = new Composite(name: "comp B")
root << comp
root << new Leaf(name: "leaf C")
comp << new Leaf(name: "leaf B1")
comp << new Leaf(name: "leaf B2")
println root.toString(0)
```

Here is the resulting output:

```
root
-leaf A
-comp B
--leaf B1
--leaf B2
-leaf C
```

Decorator Pattern

The Decorator Pattern provides a mechanism to embellish the behaviour of an object without changing its essential interface. A decorated object should be able to be substituted wherever the original (non-decorated) object was expected. Decoration typically does not involve modifying the source code of the original object and decorators should be able to be combined in flexible ways to produce objects with several embellishments.

Traditional Example

Suppose we have the following `Logger` class.

```
class Logger {
    def log(String message) {
        println message
    }
}
```

There might be times when it is useful to timestamp a log message, or times when we might want to change the case of the message. We could try to build all of this functionality into our `Logger` class. If we did that, the `Logger` class would start to be very complex. Also, everyone would obtain all of the features even when they might want only a small subset of the features. Finally, feature interaction would become quite difficult to control.

To overcome these drawbacks, we instead define two decorator classes. Uses of the `Logger` class are free to embellish their base logger with zero or more decorator classes in whatever order they desire. The classes look like this:

```
class TimeStampingLogger extends Logger {
    private Logger logger
    TimeStampingLogger(logger) {
        this.logger = logger
    }
    def log(String message) {
        def now = Calendar.instance
        logger.log("$now.time: $message")
    }
}

class UpperLogger extends Logger {
    private Logger logger
    UpperLogger(logger) {
        this.logger = logger
    }
    def log(String message) {
        logger.log(message.toUpperCase())
    }
}
```

We can use the decorators like so:

```
def logger = new UpperLogger(new TimeStampingLogger(new Logger()))
logger.log("G'day Mate")
// => Tue May 22 07:13:50 EST 2007: G'DAY MATE
```

You can see that we embellish the logger behaviour with both decorators. Because of the order we chose to apply the decorators, our log message comes out capitalised and the timestamp is in normal case. If we swap the order around, let's see what happens:

```
logger = new TimeStampingLogger(new UpperLogger(new Logger()))
logger.log('Hi There')
// => TUE MAY 22 07:13:50 EST 2007: HI THERE
```

Now the timestamp itself has also been changed to be uppercase.

Simplifying with closures or lambdas

Closures make it easy to represent code. We can use that fact to make a general purpose logger class that accepts the decoration code as a closure. This saves us defining many decoration classes.

```
class DecoratingLogger {
    def decoration = Closure.IDENTITY

    def log(String message) {
        println decoration(message)
```

```
    }
}

def upper = { it.toUpperCase() }
def stamp = { "$Calendar.instance.time: $it" }
def logger = new DecoratingLogger(decoration: stamp << upper)
logger.log("G'day Mate")
// Sat Aug 29 15:28:29 AEST 2020: G'DAY MATE
```

We can use the same approach with lambdas:

```
import java.util.function.Function

class DecoratingLogger {
    Function<String, String> decoration = Function.identity()

    def log(String message) {
        println decoration.apply(message)
    }
}

Function<String, String> upper = s -> s.toUpperCase()
Function<String, String> stamp = s -> "$Calendar.instance.time: $s"
def logger = new DecoratingLogger(decoration: upper.andThen(stamp))
logger.log("G'day Mate")
// => Sat Aug 29 15:38:28 AEST 2020: G'DAY MATE
```

A touch of dynamic behaviour

Our previous decorators were specific to `Logger` objects. We can use Groovy's Meta-Object Programming capabilities to create a decorator which is far more general purpose in nature. Consider this class:

```
class GenericLowerDecorator {
    private delegate
    GenericLowerDecorator(delegate) {
        this.delegate = delegate
    }
    def invokeMethod(String name, args) {
        def newargs = args.collect { arg ->
            if (arg instanceof String) {
                return arg.toLowerCase()
            } else {
                return arg
            }
        }
        delegate.invokeMethod(name, newargs)
    }
```

```
}
```

It takes any class and decorates it so that any `String` method parameter will automatically be changed to lower case.

```
logger = new GenericLowerDecorator(new TimeStampingLogger(new Logger()))
logger.log('IMPORTANT Message')
// => Tue May 22 07:27:18 EST 2007: important message
```

Just be careful with ordering here. The original decorators were restricted to decorating `Logger` objects. This decorator works with any object type, so we can't swap the ordering around, i.e. this won't work:

```
// Can't mix and match Interface-Oriented and Generic decorators
// logger = new TimeStampingLogger(new GenericLowerDecorator(new Logger()))
```

We could overcome this limitation be generating an appropriate Proxy type at runtime but we won't complicate the example here.

Runtime behaviour embellishment

You can also consider using the `ExpandoMetaClass` from Groovy 1.1 to dynamically embellish a class with behaviour. This isn't the normal style of usage of the decorator pattern (it certainly isn't nearly as flexible) but may help you to achieve similar results in some cases without creating a new class.

Here's what the code looks like:

```
// current mechanism to enable ExpandoMetaClass
GroovySystem.metaClassRegistry.metaClassCreationHandle = new
ExpandoMetaClassCreationHandle()

def logger = new Logger()
logger.metaClass.log = { String m -> println 'message: ' + m.toUpperCase() }
logger.log('x')
// => message: X
```

This achieves a similar result to applying a single decorator but we have no way to easily apply and remove embellishments on the fly.

More dynamic decorating

Suppose we have a calculator class (Actually any class would do).

```
class Calc {
    def add(a, b) { a + b }
}
```

We might be interested in observing usage of the class over time. If it is buried deep within our codebase, it might be hard to determine when it is being called and with what parameters. Also, it might be hard to know if it is performing well. We can easily make a generic tracing decorator that prints out tracing information whenever any method on the `Calc` class is called and also provide timing information about how long it took to execute. Here is the code for the tracing decorator:

```
class TracingDecorator {
    private delegate
    TracingDecorator(delegate) {
        this.delegate = delegate
    }
    def invokeMethod(String name, args) {
        println "Calling $name$args"
        def before = System.currentTimeMillis()
        def result = delegate.invokeMethod(name, args)
        println "Got $result in ${System.currentTimeMillis()-before} ms"
        result
    }
}
```

Here is how to use the class in a script:

```
def tracedCalc = new TracingDecorator(new Calc())
assert 15 == tracedCalc.add(3, 12)
```

And here is what you would see after running this script:

```
Calling add{3, 12}
Got 15 in 31 ms
```

Decorating with an Interceptor

The above timing example hooks into the lifecycle of Groovy objects (via `invokeMethod`). This is such an important style performing meta-programming that Groovy has special support for this style of decorating using *interceptors*.

Groovy even comes with a built-in `TracingInterceptor`. We can extend the built-in class like this:

```
class TimingInterceptor extends TracingInterceptor {
    private beforeTime
    def beforeInvoke(object, String methodName, Object[] arguments) {
        super.beforeInvoke(object, methodName, arguments)
        beforeTime = System.currentTimeMillis()
    }
    Object afterInvoke(Object object, String methodName, Object[] arguments, Object
result) {
        super.afterInvoke(object, methodName, arguments, result)
```

```
        def duration = System.currentTimeMillis() - beforeTime
        writer.write("Duration: $duration ms\\n")
        writer.flush()
        result
    }
}
```

Here is an example of using this new class:

```
def proxy = ProxyMetaClass.getInstance(Calc)
proxy.interceptor = new TimingInterceptor()
proxy.use {
    assert 7 == new Calc().add(1, 6)
}
```

And here is the output:

```
before Calc.ctor()
after  Calc.ctor()
Duration: 0 ms
before Calc.add(java.lang.Integer, java.lang.Integer)
after  Calc.add(java.lang.Integer, java.lang.Integer)
Duration: 2 ms
```

Decorating with java.lang.reflect.Proxy

If you are trying to decorate an object (i.e. just a particular instance of the class, not the class generally), then you can use Java's `java.lang.reflect.Proxy`. Groovy makes working with this easier than just Java. Below is a code sample taken out of a grails project that wraps a `java.sql.Connection` so that it's close method is a no-op:

```
protected Sql getGroovySql() {
    final Connection con = session.connection()
    def invoker = { object, method, args ->
        if (method.name == "close") {
            log.debug("ignoring call to Connection.close() for use by groovy.sql.Sql")
        } else {
            log.trace("delegating $method")
            return con.invokeMethod(method.name, args)
        }
    } as InvocationHandler;
    def proxy = Proxy.newProxyInstance( getClass().getClassLoader(), [Connection] as
Class[], invoker )
    return new Sql(proxy)
}
```

If there were many methods to intercept, then this approach could be modified to look up closure

in a map by method name and invoke it.

Decorating with Spring

The Spring Framework allows decorators to be applied with *interceptors* (you may have heard the terms *advice* or *aspect*). You can leverage this mechanism from Groovy as well.

First define a class that you want to decorate (we'll also use an interface as is normal Spring practice):

Here's the interface:

```
interface Calc {
    def add(a, b)
}
```

Here's the class:

```
class CalcImpl implements Calc {
    def add(a, b) { a + b }
}
```

Now, we define our wiring in a file called `beans.xml` as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:lang="http://www.springframework.org/schema/lang"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/lang
https://www.springframework.org/schema/lang/spring-lang.xsd">

    <bean id="performanceInterceptor" autowire="no"
          class="org.springframework.aop.interceptor.PerformanceMonitorInterceptor">
        <property name="loggerName" value="performance"/>
    </bean>
    <bean id="calc" class="util.CalcImpl"/>
    <bean class=
"org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
        <property name="beanNames" value="calc"/>
        <property name="interceptorNames" value="performanceInterceptor"/>
    </bean>
</beans>
```

Now, our script looks like this:

```
@Grab('org.springframework:spring-context:5.2.8.RELEASE')
import org.springframework.context.support.ClassPathXmlApplicationContext

def ctx = new ClassPathXmlApplicationContext('beans.xml')
def calc = ctx.getBean('calc')
println calc.add(3, 25)
```

And when we run it, we see the results:

```
21/05/2007 23:02:35 org.springframework.aop.interceptor.PerformanceMonitorInterceptor
invokeUnderTrace
FINEST: StopWatch 'util.Calc.add': running time (millis) = 16
```

You may have to adjust your `logging.properties` file for messages at log level `FINEST` to be displayed.

Asynchronous Decorators using GPars

The following example is inspired by some of the early example code for the Panini programming language. These days, you'll see this style used with async functions in JavaScript.

```
import groovy.concurrent.Awaitable

interface Document {
    void print()
    String getText()
}

class DocumentImpl implements Document {
    def document
    void print() { println document }
    String getText() { document }
}

def words(String text) {
    text.replaceAll('[^a-zA-Z]', ' ').trim().split("\\\\s+")*.toLowerCase()
}

def avgWordLength = {
    def words = words(it.text)
    sprintf "Avg Word Length: %4.2f", words*.size().sum() / words.size()
}
def modeWord = {
    def wordGroups = words(it.text).groupBy {it}.collectEntries { k, v -> [k, v.
size()] }
    def maxSize = wordGroups*.value.max()
    def maxWords = wordGroups.findAll { it.value == maxSize }
    "Mode Word(s): ${maxWords*.key.join(', ')} ($maxSize occurrences)"
}
```

```
def wordCount = { d -> "Word Count: " + words(d.text).size() }

def asyncDecorator(Document d, Closure c) {
    ProxyGenerator.INSTANCE.instantiateDelegate([print: {
        def result = Awaitable.go { c(d) }
        d.print()
        println await(result)
    }], [Document], d)
}

Document d = asyncDecorator(asyncDecorator(asyncDecorator(
        new DocumentImpl(document:"This is the file with the words in it\\n\\t\\nDo
you see the words?\\n"),
//        new DocumentImpl(document: new File('AsyncDecorator.groovy').text),
        wordCount), modeWord), avgWordLength)
d.print()
```

Delegation Pattern

The Delegation Pattern is a technique where an object's behavior (public methods) is implemented by delegating responsibility to one or more associated objects.

Groovy allows the traditional style of applying the delegation pattern, e.g. see Replace Inheritance with Delegation.

Implement Delegation Pattern using ExpandoMetaClass

The groovy.lang.ExpandoMetaClass allows usage of this pattern to be encapsulated in a library. This allows Groovy to emulate similar libraries available for the Ruby language.

Consider the following library class:

```
class Delegator {
    private targetClass
    private delegate
    Delegator(targetClass, delegate) {
        this.targetClass = targetClass
        this.delegate = delegate
    }
    def delegate(String methodName) {
        delegate(methodName, methodName)
    }
    def delegate(String methodName, String asMethodName) {
        targetClass.metaClass."$asMethodName" = delegate.&"$methodName"
    }
    def delegateAll(String[] names) {
        names.each { delegate(it) }
    }
    def delegateAll(Map names) {
        names.each { k, v -> delegate(k, v) }
```

```
    }
    def delegateAll() {
        delegate.class.methods*.name.each { delegate(it) }
    }
}
```

With this in your classpath, you can now apply the delegation pattern dynamically as shown in the following examples. First, consider we have the following classes:

```
class Person {
    String name
}

class MortgageLender {
    def borrowAmount(amount) {
       "borrow \\$$amount"
    }
    def borrowFor(thing) {
       "buy \\$thing"
    }
}

def lender = new MortgageLender()

def delegator = new Delegator(Person, lender)
```

We can now use the *delegator* to automatically borrow methods from the *lender* object to extend the *Person* class. We can borrow the methods as is or with a rename:

```
delegator.delegate 'borrowFor'
delegator.delegate 'borrowAmount', 'getMoney'

def p = new Person()

println p.borrowFor('present')   // => buy present
println p.getMoney(50)
```

The first line above, adds the *borrowFor* method to the *Person* class by delegating to the *lender* object. The second line adds a *getMoney* method to the *Person* class by delegating to the *lender* object's *borrowAmount* method.

Alternatively, we could borrow multiple methods like this:

```
delegator.delegateAll 'borrowFor', 'borrowAmount'
```

Which adds these two methods to the *Person* class.

Or if we want all the methods, like this:

```
delegator.delegateAll()
```

Which will make all the methods in the delegate object available in the *Person* class.

Alternatively, we can use a map notation to rename multiple methods:

```
delegator.delegateAll borrowAmount:'getMoney', borrowFor:'getThing'
```

Implement Delegation Pattern using @Delegate annotation

Since version 1.6 you can use the built-in delegation mechanism which is based on AST transformation.

This make delegation even easier:

```
class Person {
    def name
    @Delegate MortgageLender mortgageLender = new MortgageLender()
}

class MortgageLender {
    def borrowAmount(amount) {
       "borrow \\$$amount"
    }
    def borrowFor(thing) {
       "buy $thing"
    }
}

def p = new Person()

assert "buy present" == p.borrowFor('present')
assert "borrow \\$50" == p.borrowAmount(50)
```

Flyweight Pattern

The Flyweight Pattern is a pattern for greatly reducing memory requirements by not requiring that heavy-weight objects be created in large numbers when dealing with systems that contain many things that are mostly the same. If for instance, a document was modelled using a complex character class that knew about unicode, fonts, positioning, etc., then the memory requirements could be quite large for large documents if each physical character in the document required its own character class instance. Instead, characters themselves might be kept within Strings and we might have one character class (or a small number such as one character class for each font type) that knew the specifics of how to deal with characters.

In such circumstances, we call the state that is shared with many other things (e.g. the character type) *intrinsic* state. It is captured within the heavy-weight class. The state which distinguishes the physical character (maybe just its ASCII code or Unicode) is called its *extrinsic* state.

Example

First we are going to model some complex aircraft (the first being a hoax competitor of the second - though that is not relevant to the example).

```
class Boeing797 {
    def wingspan = '80.8 m'
    def capacity = 1000
    def speed = '1046 km/h'
    def range = '14400 km'
    // ...
}
```



```
class Airbus380 {
    def wingspan = '79.8 m'
    def capacity = 555
    def speed = '912 km/h'
    def range = '10370 km'
    // ...
}
```



If we want to model our fleet, our first attempt might involve using many instances of these heavy-weight objects. It turns out though that only a few small pieces of state (our extrinsic state) change for each aircraft, so we will have singletons for the heavy-weight objects and capture the extrinsic state (bought date and asset number in the code below) separately.

```
class FlyweightFactory {
    static instances = [797: new Boeing797(), 380: new Airbus380()]
}

class Aircraft {
```

```
    private type         // intrinsic state
    private assetNumber  // extrinsic state
    private bought       // extrinsic state
    Aircraft(typeCode, assetNumber, bought) {
        type = FlyweightFactory.instances[typeCode]
        this.assetNumber = assetNumber
        this.bought = bought
    }
    def describe() {
        println """
        Asset Number: $assetNumber
        Capacity: $type.capacity people
        Speed: $type.speed
        Range: $type.range
        Bought: $bought
        """
    }
}

def fleet = [
    new Aircraft(380, 1001, '10-May-2007'),
    new Aircraft(380, 1002, '10-Nov-2007'),
    new Aircraft(797, 1003, '10-May-2008'),
    new Aircraft(797, 1004, '10-Nov-2008')
]

fleet.each { p -> p.describe() }
```

So here, even if our fleet contained hundreds of planes, we would only have one heavy-weight object for each type of aircraft.

As a further efficiency measure, we might use lazy creation of the flyweight objects rather than create the initial map up front as in the above example.

Running this script results in:

```
Asset Number: 1001
Capacity: 555 people
Speed: 912 km/h
Range: 10370 km
Bought: 10-May-2007

Asset Number: 1002
Capacity: 555 people
Speed: 912 km/h
Range: 10370 km
Bought: 10-Nov-2007

Asset Number: 1003
Capacity: 1000 people
```

```
Speed: 1046 km/h
Range: 14400 km
Bought: 10-May-2008

Asset Number: 1004
Capacity: 1000 people
Speed: 1046 km/h
Range: 14400 km
Bought: 10-Nov-2008
```

Iterator Pattern

The Iterator Pattern allows sequential access to the elements of an aggregate object without exposing its underlying representation.

Groovy has the iterator pattern built right in to many of its closure operators, e.g. `each` and `eachWithIndex` as well as the `for .. in` loop.

For example:

```
def printAll(container) {
    for (item in container) { println item }
}

def numbers = [ 1,2,3,4 ]
def months = [ Mar:31, Apr:30, May:31 ]
def colors = [ java.awt.Color.BLACK, java.awt.Color.WHITE ]
printAll numbers
printAll months
printAll colors
```

Results in the output:

```
1
2
3
4
May=31
Mar=31
Apr=30
java.awt.Color[r=0,g=0,b=0]
java.awt.Color[r=255,g=255,b=255]
```

Another example:

```
colors.eachWithIndex { item, pos ->
    println "Position $pos contains '$item'"
```

```
}
```

Results in:

```
Position 0 contains 'java.awt.Color[r=0,g=0,b=0]'
Position 1 contains 'java.awt.Color[r=255,g=255,b=255]'
```

The iterator pattern is also built in to other special operators such as the `eachByte`, `eachFile`, `eachDir`, `eachLine`, `eachObject`, `eachMatch` operators for working with streams, URLs, files, directories and regular expressions matches.

Loan my Resource Pattern

The Loan my Resource pattern ensures that a resource is deterministically disposed of once it goes out of scope.

This pattern is built in to many Groovy helper methods. You should consider using it yourself if you need to work with resources in ways beyond what Groovy supports.

Example

Consider the following code which works with a file. First we might write some line to the file and then print its size:

```
def f = new File('junk.txt')
f.withPrintWriter { pw ->
    pw.println(new Date())
    pw.println(this.class.name)
}
println f.size()
// => 42
```

We could also read back the contents of the file a line at a time and print each line out:

```
f.eachLine { line ->
    println line
}
// =>
// Mon Jun 18 22:38:17 EST 2007
// RunPattern
```

Note that normal Java `Reader` and `PrintWriter` objects were used under the covers by Groovy but the code writer did not have to worry about explicitly creating or closing those resources. The built-in Groovy methods loan the respective reader or writer to the closure code and then tidy up after themselves. So, you are using this pattern without having to do any work.

Sometimes however, you wish to do things slightly differently to what you can get for free using

Groovy's built-in mechanisms. You should consider utilising this pattern within your own resource-handling operations.

Consider how you might process the list of words on each line within the file. We could actually do this one too using Groovy's built-in functions, but bear with us and assume we have to do some resource handling ourselves. Here is how we might write the code without using this pattern:

```
def reader = f.newReader()
reader.splitEachLine(' ') { wordList ->
    println wordList
}
reader.close()
// =>
// [ "Mon", "Jun", "18", "22:38:17", "EST", "2007" ]
// [ "RunPattern" ]
```

Notice that we now have an explicit call to `close()` in our code. If we didn't code it just right (here we didn't surround the code in a `try ··· finally` block, we run the risk of leaving the file handle open.

Let's now apply the loan pattern. First, we'll write a helper method:

```
def withListOfWordsForEachLine(File f, Closure c) {
    def r = f.newReader()
    try {
        r.splitEachLine(' ', c)
    } finally {
        r?.close()
    }
}
```

Now, we can re-write our code as follows:

```
withListOfWordsForEachLine(f) { wordList ->
    println wordList
}
// =>
// [ "Mon", "Jun", "18", "22:38:17", "EST", "2007" ]
// [ "RunPattern" ]
```

This is much simpler and has removed the explicit `close()`. This is now catered for in one spot so we can apply the appropriate level of testing or reviewing in just one spot to be sure we have no problems.

Using Monoids

Monoids allow the mechanics of an aggregation algorithm to be separated from the algorithm-

specific logic associated with that aggregation. It is often thought to be a functional design pattern.

Perhaps, it is easiest seen with an example. Consider the code for integer sum, integer product and string concatenation. We might note various similarities:

```
def nums = [1, 2, 3, 4]

def sum = 0    ①
for (num in nums) { sum += num }    ②
assert sum == 10

def product = 1    ①
for (num in nums) { product *= num }    ②
assert product == 24

def letters = ['a', 'b', 'c']

def concat = ''    ①
for (letter in letters) { concat += letter }    ②
assert concat == 'abc'
```

① Initialize an aggregate counter

② Loop throw elements with for/while/iteration adjusting counter

We can remove the duplicate aggregation coding and the tease out the important differences for each algorithm. We might instead use Groovy's `inject` method. This is a *fold* operation in functional programming jargon.

```
assert nums.inject(0){ total, next -> total + next } == 10
assert nums.inject(1){ total, next -> total * next } == 24
assert letters.inject(''){ total, next -> total + next } == 'abc'
```

Here the first parameter is the initial value, and the supplied closure contains the algorithm-specific logic.

Similarly, for Groovy 3+, we can use the JDK stream API and lambda syntax as follows:

```
assert nums.stream().reduce(0, (total, next) -> total + next) == 10
assert nums.stream().reduce(1, (total, next) -> total * next) == 24
assert letters.stream().reduce('', (total, next) -> total + next) == 'abc'
```

A touch of formalism

Looking at these examples, we might think all aggregation can be supported this way. In fact, we look for certain characteristics to ensure that this aggregation pattern will apply:

- Closure: performing the aggregation step should produce a result of the same type as the

elements being aggregated.

Examples: `1L + 3L` produces a `Long`, and `'foo' + 'bar'` produces a `String`.
Non-monoid examples: `'foo'.size() + 'bar'.size()` (takes strings, returns an integer), the type *odd numbers* with respect to addition, algorithms which don't handle null arguments if such arguments are possible.



When using the term *closure* here, we simply mean closed under the operation, not the Groovy `Closure` class.

- Associativity: the order in which we apply the aggregation step should not matter.

Examples: `(1 + 3) + 5` is the same as `1 + (3 + 5)`, and `('a' + 'b') + 'c'` is the same as `'a' + ('b' + 'c')`.
Non-monoid example: `(10 - 5) - 3` is not equal to `10 - (5 - 3)` therefore integers are not a monoid with respect to subtraction.

- Identity element (sometimes also called a 'zero' element): there should be an element which aggregated with any element returns the original element.

Examples: `0 + 42 == 42`, `42 + 0 == 42`, `1 * 42 == 42`, and `'' + 'foo' == 'foo'`.
Non-monoid example: the type *non-empty strings* is not a monoid with respect to concatenation.

If your algorithm doesn't satisfy all the monoid properties, that doesn't mean aggregation isn't possible. It just means that you won't get all the benefits from monoids, which we'll cover shortly, or you might have a little more work to do. Also, you might be able to convert your data structures slightly to turn your problem into one involving monoids. We'll cover that topic a little later in this section.

Benefits of monoids

Consider adding the integers 10 through 16. Because the operation of addition for integers is a monoid, we already know that we can save writing code and instead use the approach we saw in the earlier `inject` examples. There are some other nice properties.

Because of the *closure* property, if we have a pairwise method like `sum(Integer a, Integer b)`, then for a monoid, we can always extend that method to work with a list, e.g. `sum(List<Integer> nums)` or `sum(Integer first, Integer… rest)`.

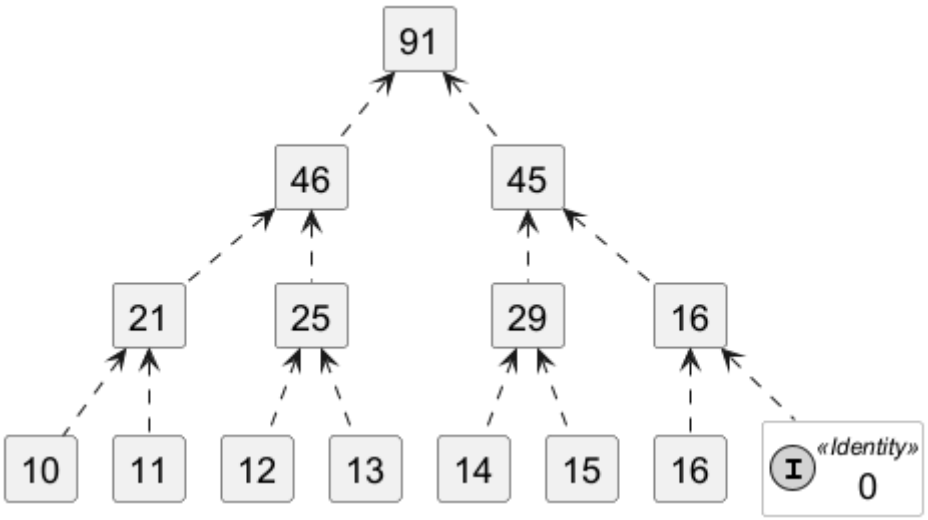
Because of *associativity*, we can employ some interesting ways to solve the aggregation including:

- Divide and conquer algorithms which break the problem into smaller pieces
- Various incremental algorithms for example memoization would allow summing from 1..5 to potentially start part way through be reusing a cached value of summing 1..4 if that had been

calculated earlier

- Inherent parallelization can make use of multiple cores

Let's just look at the first of these in more detail. With a multicore processor, one core could add 10 plus 11, another core 12 plus 13, and so on. We'd use the *identity* element if needed (shown being added to 16 in our example). Then the intermediate results could also be added together concurrently and so on until the result was reached.



We have reduced the amount of code we need to write, and we also have potential performance gains.

Here is how we might code the previous example using Groovy's parallel collections:

```
def nums = 10..16
ParallelScope.withPool(Pool.cpu()) {
    assert 91 == nums.toList().injectParallel(0){ total, next -> total + next }
}
```

Working with Non-monoids

Suppose we want to find the average of the numbers 1..10. Groovy has a built-in method for this:

```
assert (1..10).average() == 5.5
```

Now, suppose we want to build our own monoid solution instead of using the built-in version. It might seem difficult to find the *identity* element. After all:

```
assert (0..10).average() == 5
```

Similarly, if we are tempted to write the pairwise aggregation closure it might be something like:

```
def avg = { a, b -> (a + b) / 2 }
```

What `b` can we use for the *identity* element here so that our equation returns the original? We need to use `a`, but that isn't a fixed value, so there is no *identity*.

Also, associativity doesn't hold for this initial attempt at defining `avg` as these examples show:

```
assert 6 == avg(avg(10, 2), 6)
assert 7 == avg(10, avg(2, 6))
```

Also, what about our *closure* property? Our original numbers were integers, but our average (`5.5`) is not. We can solve this by making our average work for any `Number` instances, but it might not always be this easy.

It might appear that this problem is not amenable to a monoidal solution. However, there are numerous ways to bring monoids into the solution.

We can split it into two parts:

```
def nums = 1..10
def total = nums.sum()
def avg = total / nums.size()
assert avg == 5.5
```

The calculation of `sum()` can follow monoid rules and then our last step can calculate the average. We can even do a parallel version:

```
ParallelScope.withPool(Pool.cpu()) {
    assert 5.5 == nums.toList().sumParallel{ a, b -> a + b } / nums.size()
}
```

Here, we were using the built-in `sum()` method (and `sumParallel()` for the parallel example), but if you were doing it by hand, the monoid nature of that part of your calculation would make it easier to write your own code for that step.

Alternatively, we can introduce a helper data structure that reworks the problem to be a monoid. Instead of just keeping the total, let's keep a list containing the total and count of numbers. The code could look something like this:

```
def holder = nums
    .collect{ [it, 1] }
    .inject{ a, b -> [a[0] + b[0], a[1] + b[1]] }
def avg = holder[0] / holder[1]
assert avg == 5.5
```

Or, to be a little fancier, we could introduce a class for our data structure and even calculate in parallel:

```
class AverageHolder {
    int total
    int count
    AverageHolder plus(AverageHolder other) {
        return new AverageHolder(total: total + other.total,
                                 count: count + other.count)
    }
    static final AverageHolder ZERO =
        new AverageHolder(total: 0, count: 0)
}

def asHolder = {
    it instanceof Integer ? new AverageHolder(total: it, count : 1) : it
}
def pairwiseAggregate = { aggregate, next ->
    asHolder(aggregate) + asHolder(next)
}
ParallelScope.withPool(Pool.cpu()) {
    def holder = nums.toList().injectParallel(AverageHolder.ZERO, pairwiseAggregate)
    def avg = holder.with{ total / count }
    assert avg == 5.5
}
```

Null Object Pattern

The Null Object Pattern involves using a special object place-marker object representing null. Typically, if you have a reference to null, you can't invoke `reference.field` or `reference.method()` You receive the dreaded `NullPointerException`. The null object pattern uses a special object representing null, instead of using an actual `null`. This allows you to invoke field and method references on the null object. The result of using the null object should semantically be equivalent to *doing nothing*.

Simple Example

Suppose we have the following system:

```
class Job {
    def salary
}

class Person {
    def name
    def Job job
}

def people = [
```

```
    new Person(name: 'Tom', job: new Job(salary: 1000)),
    new Person(name: 'Dick', job: new Job(salary: 1200)),
]

def biggestSalary = people.collect { p -> p.job.salary }.max()
println biggestSalary
```

When run, this prints out `1200`. Suppose now that we now invoke:

```
people << new Person(name: 'Harry')
```

If we now try to calculate `biggestSalary` again, we receive a null pointer exception.

To overcome this problem, we can introduce a `NullJob` class and change the above statement to become:

```
class NullJob extends Job { def salary = 0 }

people << new Person(name: 'Harry', job: new NullJob())
biggestSalary = people.collect { p -> p.job.salary }.max()
println biggestSalary
```

This works as we require but it's not always the best way to do this with Groovy. Groovy's safe-dereference operator (`?.`) operator and null aware closures often allow Groovy to avoid the need to create a special null object or null class. This is illustrated by examining a groovier way to write the above example:

```
people << new Person(name:'Harry')
biggestSalary = people.collect { p -> p.job?.salary }.max()
println biggestSalary
```

Two things are going on here to allow this to work. First of all, `max()` is *'null aware'* so that [300, null, 400].max() == 400. Secondly, with the `?.` operator, an expression like `p?.job?.salary` will be equal to null if `salary` is equal to null, or if `job` is equal ` null or if `p` is equal to null. You don't need to code a complex nested if ... then ... else to avoid a `NullPointerException`.

Tree Example

Consider the following example where we want to calculate size, cumulative sum and cumulative product of all the values in a tree structure.

Our first attempt has special logic within the calculation methods to handle null values.

```
class NullHandlingTree {
    def left, right, value
```

```
    def size() {
        1 + (left ? left.size() : 0) + (right ? right.size() : 0)
    }

    def sum() {
       value + (left ? left.sum() : 0) + (right ? right.sum() : 0)
    }

    def product() {
       value * (left ? left.product() : 1) * (right ? right.product() : 1)
    }
}

def root = new NullHandlingTree(
    value: 2,
    left: new NullHandlingTree(
        value: 3,
        right: new NullHandlingTree(value: 4),
        left: new NullHandlingTree(value: 5)
    )
)

println root.size()
println root.sum()
println root.product()
```

If we introduce the null object pattern (here by defining the `NullTree` class), we can now simplify the logic in the `size()`, `sum()` and`product()` methods. These methods now much more clearly represent the logic for the normal (and now universal) case. Each of the methods within `NullTree` returns a value which represents doing nothing.

```
class Tree {
    def left = new NullTree(), right = new NullTree(), value

    def size() {
        1 + left.size() + right.size()
    }

    def sum() {
       value + left.sum() + right.sum()
    }

    def product() {
       value * left.product() * right.product()
    }
}

class NullTree {
    def size() { 0 }
    def sum() { 0 }
```

```
    def product() { 1 }
}

def root = new Tree(
    value: 2,
    left: new Tree(
        value: 3,
        right: new Tree(value: 4),
        left: new Tree(value: 5)
    )
)

println root.size()
println root.sum()
println root.product()
```

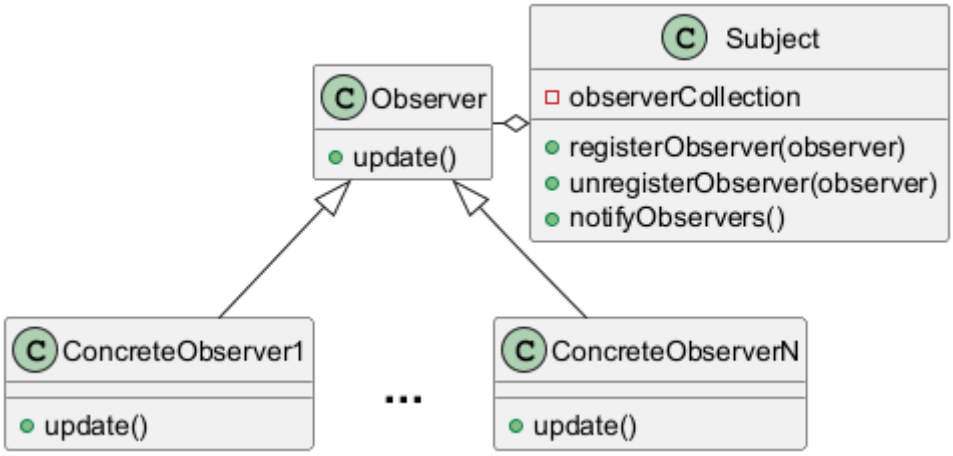
The result of running either of these examples is:

```
4
14
120
```

Note: a slight variation with the null object pattern is to combine it with the singleton pattern. So, we wouldn't write new NullTree() wherever we needed a null object as shown above. Instead we would have a single null object instance which we would place within our data structures as needed.

Observer Pattern

The Observer Pattern allows one or more *observers* to be notified about changes or events from a *subject* object.



Example

Here is a typical implementation of the classic pattern:

```
interface Observer {
    void update(message)
}

class Subject {
    private List observers = []
    void register(observer) {
        observers << observer
    }
    void unregister(observer) {
        observers -= observer
    }
    void notifyAll(message) {
        observers.each{ it.update(message) }
    }
}

class ConcreteObserver1 implements Observer {
    def messages = []
    void update(message) {
        messages << message
    }
}

class ConcreteObserver2 implements Observer {
    def messages = []
    void update(message) {
        messages << message.toUpperCase()
    }
}

def o1a = new ConcreteObserver1()
def o1b = new ConcreteObserver1()
def o2 = new ConcreteObserver2()
def observers = [o1a, o1b, o2]
new Subject().with {
    register(o1a)
    register(o2)
    notifyAll('one')
}
new Subject().with {
    register(o1b)
    register(o2)
    notifyAll('two')
}
def expected = [['one'], ['two'], ['ONE', 'TWO']]
assert observers*.messages == expected
```

Using Closures, we can avoid creating the concrete observer classes as shown below:

```
interface Observer {
    void update(message)
}

class Subject {
    private List observers = []
    void register(Observer observer) {
        observers << observer
    }
    void unregister(observer) {
        observers -= observer
    }
    void notifyAll(message) {
        observers.each{ it.update(message) }
    }
}

def messages1a = [], messages1b = [], messages2 = []
def o2 = { messages2 << it.toUpperCase() }
new Subject().with {
    register{ messages1a << it }
    register(o2)
    notifyAll('one')
}
new Subject().with {
    register{ messages1b << it }
    register(o2)
    notifyAll('two')
}
def expected = [['one'], ['two'], ['ONE', 'TWO']]
assert [messages1a, messages1b, messages2] == expected
```

As a variation for Groovy 3+, let's consider dropping the `Observer` interface and using lambdas as shown below:

```
import java.util.function.Consumer

class Subject {
    private List<Consumer> observers = []
    void register(Consumer observer) {
        observers << observer
    }
    void unregister(observer) {
        observers -= observer
    }
    void notifyAll(message) {
        observers.each{ it.accept(message) }
    }
}
```

```
def messages1a = [], messages1b = [], messages2 = []
def o2 = { messages2 << it.toUpperCase() }
new Subject().with {
    register(s -> messages1a << s)
    register(s -> messages2 << s.toUpperCase())
    notifyAll('one')
}
new Subject().with {
    register(s -> messages1b << s)
    register(s -> messages2 << s.toUpperCase())
    notifyAll('two')
}
def expected = [['one'], ['two'], ['ONE', 'TWO']]
assert [messages1a, messages1b, messages2] == expected
```

We are now calling the `accept` method from `Consumer` rather than the `update` method from `Observer`.

@Bindable and @Vetoable

The JDK has some built-in classes which follow the observer pattern. The `java.util.Observer` and `java.util.Observable` classes are deprecated from JDK 9 due to various limitations. Instead, you are recommended to use various more powerful classes in the `java.beans` package such as `java.beans.PropertyChangeListener`. Luckily, Groovy has some built-in transforms (groovy.beans.Bindable and groovy.beans.Vetoable) which support for some key classes from that package.

```
import groovy.beans.*
import java.beans.*

class PersonBean {
    @Bindable String first
    @Bindable String last
    @Vetoable Integer age
}

def messages = [:].withDefault{[]}
new PersonBean().with {
    addPropertyChangeListener{ PropertyChangeEvent ev ->
        messages[ev.propertyName] << "prop: $ev.newValue"
    }
    addVetoableChangeListener{ PropertyChangeEvent ev ->
        def name = ev.propertyName
        if (name == 'age' && ev.newValue > 40)
            throw new PropertyVetoException()
        messages[name] << "veto: $ev.newValue"
    }
    first = 'John'
    age = 35
    last = 'Smith'
```

```
    first = 'Jane'
    age = 42
}

def expected = [
    first:['prop: John', 'prop: Jane'],
    age:['veto: 35'],
    last:['prop: Smith']
]
assert messages == expected
```

Here, methods like `addPropertyChangeListener` perform the same role as `registerObserver` in previous examples. There is a `firePropertyChange` method corresponding to `notifyAll`/`notifyObservers` in previous examples but Groovy adds that automatically here, so it isn't visible in the source code. There is also a `propertyChange` method that corresponds to the `update` method in previous examples, though again, that isn't visible here.

Pimp my Library Pattern

The Pimp my Library Pattern suggests an approach for extending a library that nearly does everything that you need but just needs a little more. It assumes that you do not have source code for the library of interest.

Example

Suppose we want to make use of the built-in Integer facilities in Groovy (which build upon the features already in Java). Those libraries have nearly all of the features we want but not quite everything. We may not have all of the source code to the Groovy and Java libraries so we can't just change the library. Instead we augment the library. Groovy has a number of ways to do this. One way is to use a Category.

First, we'll define a suitable category.

```
class EnhancedInteger {
    static boolean greaterThanAll(Integer self, Object[] others) {
        greaterThanAll(self, others)
    }
    static boolean greaterThanAll(Integer self, others) {
        others.every { self > it }
    }
}
```

We have added two methods which augment the Integer methods by providing the `greaterThanAll` method. Categories follow conventions where they are defined as static methods with a special first parameter representing the class we wish to extend. The greaterThanAll(Integer self, others) static method becomes the `greaterThanAll(other)` instance method.

We defined two versions of `greaterThanAll`. One which works for collections, ranges etc. The other which works with a variable number of `Integer` arguments.

Here is how you would use the category.

```
use(EnhancedInteger) {
    assert 4.greaterThanAll(1, 2, 3)
    assert !5.greaterThanAll(2, 4, 6)
    assert 5.greaterThanAll(-4..4)
    assert 5.greaterThanAll([])
    assert !5.greaterThanAll([4, 5])
}
```

As you can see, using this technique you can effectively enrich an original class without having access to its source code. Moreover, you can apply different enrichments in different parts of the system as well as work with un-enriched objects if we need to.

Proxy Pattern

The Proxy Pattern allows one object to act as a pretend replacement for some other object. In general, whoever is using the proxy, doesn't realise that they are not using the real thing. The pattern is useful when the real object is hard to create or use: it may exist over a network connection, or be a large object in memory, or be a file, database or some other resource that is expensive or impossible to duplicate.

Example

One common use of the proxy pattern is when talking to remote objects in a different JVM. Here is the client code for creating a proxy that talks via sockets to a server object as well as an example usage:

```
class AccumulatorProxy {
    def accumulate(args) {
        def result
        def s = new Socket("localhost", 54321)
        s.withObjectStreams { ois, oos ->
            oos << args
            result = ois.readObject()
        }
        s.close()
        return result
    }
}

println new AccumulatorProxy().accumulate([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
// => 55
```

Here is what your server code might look like (start this first):

```
class Accumulator {
    def accumulate(args) {
```

```
        args.inject(0) { total, arg -> total += arg }
    }
}

def port = 54321
def accumulator = new Accumulator()
def server = new ServerSocket(port)
println "Starting server on port $port"
while(true) {
    server.accept() { socket ->
        socket.withObjectStreams { ois, oos ->
            def args = ois.readObject()
            oos << accumulator.accumulate(args)
        }
    }
}
```

Singleton Pattern

The Singleton Pattern is used to make sure only one object of a particular class is ever created. This can be useful when exactly one object is needed to coordinate actions across a system; perhaps for efficiency where creating lots of identical objects would be wasteful, perhaps because a particular algorithm needing a single point of control is required or perhaps when an object is used to interact with a non-shareable resource.

Weaknesses of the Singleton pattern include:

- It can reduce reuse. For instance, there are issues if you want to use inheritance with Singletons. If `SingletonB` extends `SingletonA`, should there be exactly (at most) one instance of each or should the creation of an object from one of the classes prohibit creation from the other. Also, if you decide both classes can have an instance, how do you override the `getInstance()` method which is static?
- It is also hard to test singletons in general because of the static methods but Groovy can support that if required.

Example: The Classic Java Singleton

Suppose we wish to create a class for collecting votes. Because getting the right number of votes may be very important, we decide to use the singleton pattern. There will only ever be one `VoteCollector` object, so it makes it easier for us to reason about that objects creation and use.

```
class VoteCollector {
    def votes = 0
    private static final INSTANCE = new VoteCollector()
    static getInstance() { return INSTANCE }
    private VoteCollector() { }
    def display() { println "Collector:${hashCode()}, Votes:$votes" }
}
```

Some points of interest in this code:

- it has a private constructor, so no `VoteCollector` objects can be created in our system (except for the `INSTANCE` we create)
- the `INSTANCE` is also private, so it can't be changed once set
- we haven't made the updating of votes thread-safe at this point (it doesn't add to this example)
- the vote collector instance is not lazily created (if we never reference the class, the instance won't be created; however, as soon as we reference the class, the instance will be created even if not needed initially)

We can use this singleton class in some script code as follows:

```
def collector = VoteCollector.instance
collector.display()
collector.votes++
collector = null

Thread.start{
    def collector2 = VoteCollector.instance
    collector2.display()
    collector2.votes++
    collector2 = null
}.join()

def collector3 = VoteCollector.instance
collector3.display()
```

Here we used the instance 3 times. The second usage was even in a different thread (but don't try this in a scenario with a new class loader).

Running this script yields (your hashcode value will vary):

```
Collector:15959960, Votes:0
Collector:15959960, Votes:1
Collector:15959960, Votes:2
```

Variations to this pattern:

- To support lazy-loading and multi-threading, we could just use the `synchronized` keyword with the `getInstance()` method. This has a performance hit but will work.
- We can consider variations involving double-checked locking and the `volatile` keyword, but see the limitations of this approach here.

Example: Singleton via MetaProgramming

Groovy's meta-programming capabilities allow concepts like the singleton pattern to be enacted in a far more fundamental way. This example illustrates a simple way to use Groovy's meta-

programming capabilities to achieve the singleton pattern but not necessarily the most efficient way.

Suppose we want to keep track of the total number of calculations that a calculator performs. One way to do that is to use a singleton for the calculator class and keep a variable in the class with the count.

First we define some base classes. A `Calculator` class which performs calculations and records how many such calculations it performs and a `Client` class which acts as a facade to the calculator.

```
class Calculator {
    private total = 0
    def add(a, b) { total++; a + b }
    def getTotalCalculations() { 'Total Calculations: ' + total }
    String toString() { 'Calc: ' + hashCode() }
}

class Client {
    def calc = new Calculator()
    def executeCalc(a, b) { calc.add(a, b) }
    String toString() { 'Client: ' + hashCode() }
}
```

Now we can define and register a *MetaClass* which intercepts all attempts to create a `Calculator` object and always provides a pre-created instance instead. We also register this MetaClass with the Groovy system:

```
class CalculatorMetaClass extends MetaClassImpl {
    private static final INSTANCE = new Calculator()
    CalculatorMetaClass() { super(Calculator) }
    def invokeConstructor(Object[] arguments) { return INSTANCE }
}

def registry = GroovySystem.metaClassRegistry
registry.setMetaClass(Calculator, new CalculatorMetaClass())
```

Now we use instances of our `Client` class from within a script. The client class will attempt to create new instances of the calculator but will always get the singleton.

```
def client = new Client()
assert 3 == client.executeCalc(1, 2)
println "$client, $client.calc, $client.calc.totalCalculations"

client = new Client()
assert 4 == client.executeCalc(2, 2)
println "$client, $client.calc, $client.calc.totalCalculations"
```

Here is the result of running this script (your hashcode values may vary):

```
Client: 7306473, Calc: 24230857, Total Calculations: 1
Client: 31436753, Calc: 24230857, Total Calculations: 2
```

Guice Example

We can also implement the Singleton Pattern using Guice.

Consider the Calculator example again.

Guice is a Java-oriented framework that supports Interface-Oriented design. Hence we create a `Calculator` interface first. We can then create our `CalculatorImpl` implementation and a `Client` object which our script will interact with. The `Client` class isn't strictly needed for this example but allows us to show that non-singleton instances are the default. Here is the code:

```
@Grapes([@Grab('aopalliance:aopalliance:1.0'), @Grab(
'com.google.code.guice:guice:1.0')])
import com.google.inject.*

interface Calculator {
    def add(a, b)
}

class CalculatorImpl implements Calculator {
    private total = 0
    def add(a, b) { total++; a + b }
    def getTotalCalculations() { 'Total Calculations: ' + total }
    String toString() { 'Calc: ' + hashCode() }
}

class Client {
    @Inject Calculator calc
    def executeCalc(a, b) { calc.add(a, b) }
    String toString() { 'Client: ' + hashCode() }
}

def injector = Guice.createInjector (
    [configure: { binding ->
        binding.bind(Calculator)
               .to(CalculatorImpl)
               .asEagerSingleton() } ] as Module
)

def client = injector.getInstance(Client)
assert 3 == client.executeCalc(1, 2)
println "$client, $client.calc, $client.calc.totalCalculations"

client = injector.getInstance(Client)
```

```
assert 4 == client.executeCalc(2, 2)
println "$client, $client.calc, $client.calc.totalCalculations"
```

Note the `@Inject` annotation in the `Client` class. We can always tell right in the source code which fields will be injected.

In this example we chose to use an *explicit* binding. All of our dependencies (ok, only one in this example at the moment) are configured in the binding. The Guide injector knows about the binding and injects the dependencies as required when we create objects. For the singleton pattern to hold, you must always use Guice to create your instances. Nothing shown so far would stop you creating another instance of the calculator manually using new CalculatorImpl() which would of course violate the desired singleton behaviour.

In other scenarios (though probably not in large systems), we could choose to express dependencies using annotations, such as the following example shows:

```
@Grapes([@Grab('aopalliance:aopalliance:1.0'), @Grab(
'com.google.code.guice:guice:1.0')])
import com.google.inject.*

@ImplementedBy(CalculatorImpl)
interface Calculator {
    // as before ...
}

@Singleton
class CalculatorImpl implements Calculator {
    // as before ...
}

class Client {
    // as before ...
}

def injector = Guice.createInjector()

// ...
```

Note the `@Singleton` annotation on the `CalculatorImpl` class and the `@ImplementedBy` annotation in the `Calculator` interface.

When run, the above example (using either approach) yields (your hashcode values will vary):

```
Client: 8897128, Calc: 17431955, Total Calculations: 1
Client: 21145613, Calc: 17431955, Total Calculations: 2
```

You can see that we obtained a new client object whenever we asked for an instance but it was injected with the same calculator object.

Spring Example

We can do the Calculator example again using Spring as follows:

```
@Grapes([@Grab('org.springframework:spring-core:5.2.8.RELEASE'), @Grab
('org.springframework:spring-beans:5.2.8.RELEASE')])
import org.springframework.beans.factory.support.*

interface Calculator {
    def add(a, b)
}

class CalculatorImpl implements Calculator {
    private total = 0
    def add(a, b) { total++; a + b }
    def getTotalCalculations() { 'Total Calculations: ' + total }
    String toString() { 'Calc: ' + hashCode() }
}

class Client {
    Client(Calculator calc) { this.calc = calc }
    def calc
    def executeCalc(a, b) { calc.add(a, b) }
    String toString() { 'Client: ' + hashCode() }
}

// Here we 'wire' up our dependencies through the API. Alternatively,
// we could use XML-based configuration or the Grails Bean Builder DSL.
def factory = new DefaultListableBeanFactory()
factory.registerBeanDefinition('calc', new RootBeanDefinition(CalculatorImpl))
def beanDef = new RootBeanDefinition(Client, false)
beanDef.setAutowireMode(AbstractBeanDefinition.AUTOWIRE_AUTODETECT)
factory.registerBeanDefinition('client', beanDef)

def client = factory.getBean('client')
assert 3 == client.executeCalc(1, 2)
println "$client, $client.calc, $client.calc.totalCalculations"

client = factory.getBean('client')
assert 4 == client.executeCalc(2, 2)
println "$client, $client.calc, $client.calc.totalCalculations"
```

And here is the result (your hashcode values will vary):

```
Client: 29418586, Calc: 10580099, Total Calculations: 1
Client: 14800362, Calc: 10580099, Total Calculations: 2
```

Further information

- Simply Singleton
- Use your singletons wisely
- Double-checked locking and the Singleton pattern
- Lazy Loading Singletons
- Implementing the Singleton Pattern in C#

State Pattern

The State Pattern provides a structured approach to partitioning the behaviour within complex systems. The overall behaviour of a system is partitioned into well-defined states. Typically, each state is implemented by a class. The overall system behaviour can be determined firstly by knowing the *current state* of the system; secondly, by understanding the behaviour possible while in that state (as embodied in the methods of the class corresponding to that state).

Example

Here is an example:

```
class Client {
    def context = new Context()
    def connect() {
        context.state.connect()
    }
    def disconnect() {
        context.state.disconnect()
    }
    def send_message(message) {
        context.state.send_message(message)
    }
    def receive_message() {
        context.state.receive_message()
    }
}

class Context {
    def state = new Offline(this)
}

class ClientState {
    def context
    ClientState(context) {
        this.context = context
        inform()
    }
}

class Offline extends ClientState {
```

```
    Offline(context) {
        super(context)
    }
    def inform() {
        println "offline"
    }
    def connect() {
        context.state = new Online(context)
    }
    def disconnect() {
        println "error: not connected"
    }
    def send_message(message) {
        println "error: not connected"
    }
    def receive_message() {
        println "error: not connected"
    }
}

class Online extends ClientState {
    Online(context) {
        super(context)
    }
    def inform() {
        println "connected"
    }
    def connect() {
        println "error: already connected"
    }
    def disconnect() {
        context.state = new Offline(context)
    }
    def send_message(message) {
        println "\"$message\" sent"
    }
    def receive_message() {
        println "message received"
    }
}

client = new Client()
client.send_message("Hello")
client.connect()
client.send_message("Hello")
client.connect()
client.receive_message()
client.disconnect()
```

Here is the output:

```
offline
error: not connected
connected
"Hello" sent
error: already connected
message received
offline
```

One of the great things about a dynamic language like Groovy though is that we can take this example and express it in many different ways depending on our particular needs. Some potential variations for this example are shown below.

Variation 1: Leveraging Interface-Oriented Design

One approach we could take is to leverage Interface-Oriented Design. To do this, we could introduce the following interface:

```
interface State {
    def connect()
    def disconnect()
    def send_message(message)
    def receive_message()
}
```

Then our `Client`, `Online` and 'Offline ` classes could be modified to implement that interface, e.g.:

```
class Client implements State {
  // ... as before ...
}

class Online implements State {
  // ... as before ...
}

class Offline implements State {
  // ... as before ...
}
```

You might ask: Haven't we just introduced additional boilerplate code? Can't we rely on duck-typing for this? The answer is 'yes' and 'no'. We can get away with duck-typing but one of the key intentions of the state pattern is to partition complexity. If we know that the *client* class and each *state* class all satisfy one interface, then we have placed some key boundaries around the complexity. We can look at any state class in isolation and know the bounds of behaviour possible for that state.

We don't have to use interfaces for this, but it helps express the intent of this particular style of partitioning and it helps reduce the size of our unit tests (we would have to have additional tests in

place to express this intent in languages which have less support for interface-oriented design).

Variation 2: Extract State Pattern Logic

Alternatively, or in combination with other variations, we might decide to extract some of our State Pattern logic into helper classes. For example, we could define the following classes in a state pattern package/jar/script:

```
abstract class InstanceProvider {
    static def registry = GroovySystem.metaClassRegistry
    static def create(objectClass, param) {
        registry.getMetaClass(objectClass).invokeConstructor([param] as Object[])
    }
}

abstract class Context {
    private context
    protected setContext(context) {
        this.context = context
    }
    def invokeMethod(String name, Object arg) {
        context.invokeMethod(name, arg)
    }
    def startFrom(initialState) {
        setContext(InstanceProvider.create(initialState, this))
    }
}

abstract class State {
    private client

    State(client) { this.client = client }

    def transitionTo(nextState) {
        client.setContext(InstanceProvider.create(nextState, client))
    }
}
```

This is all quite generic and can be used wherever we want to introduce the state pattern. Here is what our code would look like now:

```
class Client extends Context {
    Client() {
        startFrom(Offline)
    }
}

class Offline extends State {
    Offline(client) {
```

```
        super(client)
        println "offline"
    }
    def connect() {
        transitionTo(Online)
    }
    def disconnect() {
        println "error: not connected"
    }
    def send_message(message) {
        println "error: not connected"
    }
    def receive_message() {
        println "error: not connected"
    }
}

class Online extends State {
    Online(client) {
        super(client)
        println "connected"
    }
    def connect() {
        println "error: already connected"
    }
    def disconnect() {
        transitionTo(Offline)
    }
    def send_message(message) {
        println "\"$message\" sent"
    }
    def receive_message() {
        println "message received"
    }
}

client = new Client()
client.send_message("Hello")
client.connect()
client.send_message("Hello")
client.connect()
client.receive_message()
client.disconnect()
```

You can see here the `startFrom` and `transitionTo` methods begin to give our example code a DSL feel.

Variation 3: Bring on the DSL

Alternatively, or in combination with other variations, we might decide to fully embrace a Domain Specific Language (DSL) approach to this example.

We can define the following generic helper functions (first discussed here):

```
class Grammar {
    def fsm

    def event
    def fromState
    def toState

    Grammar(a_fsm) {
        fsm = a_fsm
    }

    def on(a_event) {
        event = a_event
        this
    }

    def on(a_event, a_transitioner) {
        on(a_event)
        a_transitioner.delegate = this
        a_transitioner.call()
        this
    }

    def from(a_fromState) {
        fromState = a_fromState
        this
    }

    def to(a_toState) {
        assert a_toState, "Invalid toState: $a_toState"
        toState = a_toState
        fsm.registerTransition(this)
        this
    }

    def isValid() {
        event && fromState && toState
    }

    public String toString() {
        "$event: $fromState=>$toState"
    }
}
```

```
class FiniteStateMachine {
    def transitions = [:]
```

```
    def initialState
    def currentState

    FiniteStateMachine(a_initialState) {
        assert a_initialState, "You need to provide an initial state"
        initialState = a_initialState
        currentState = a_initialState
    }

    def record() {
        Grammar.newInstance(this)
    }

    def reset() {
        currentState = initialState
    }

    def isState(a_state) {
        currentState == a_state
    }

    def registerTransition(a_grammar) {
        assert a_grammar.isValid(), "Invalid transition ($a_grammar)"
        def transition
        def event = a_grammar.event
        def fromState = a_grammar.fromState
        def toState = a_grammar.toState

        if (!transitions[event]) {
            transitions[event] = [:]
        }

        transition = transitions[event]
        assert !transition[fromState], "Duplicate fromState $fromState for transition
$a_grammar"
        transition[fromState] = toState
    }

    def fire(a_event) {
        assert currentState, "Invalid current state '$currentState': passed into
constructor"
        assert transitions.containsKey(a_event), "Invalid event '$a_event', should be
one of ${transitions.keySet()}"
        def transition = transitions[a_event]
        def nextState = transition[currentState]
        assert nextState, "There is no transition from '$currentState' to any other
state"
        currentState = nextState
        currentState
    }
```

```
}
```

Now we can define and test our state machine like this:

```
class StatePatternDslTest {
    private fsm

    void setUp() {
        fsm = FiniteStateMachine.newInstance('offline')
        def recorder = fsm.record()
        recorder.on('connect').from('offline').to('online')
        recorder.on('disconnect').from('online').to('offline')
        recorder.on('send_message').from('online').to('online')
        recorder.on('receive_message').from('online').to('online')
    }

    void testInitialState() {
        assert fsm.isState('offline')
    }

    void testOfflineState() {
        shouldFail{
            fsm.fire('send_message')
        }
        shouldFail{
            fsm.fire('receive_message')
        }
        shouldFail{
            fsm.fire('disconnect')
        }
        assert 'online' == fsm.fire('connect')
    }

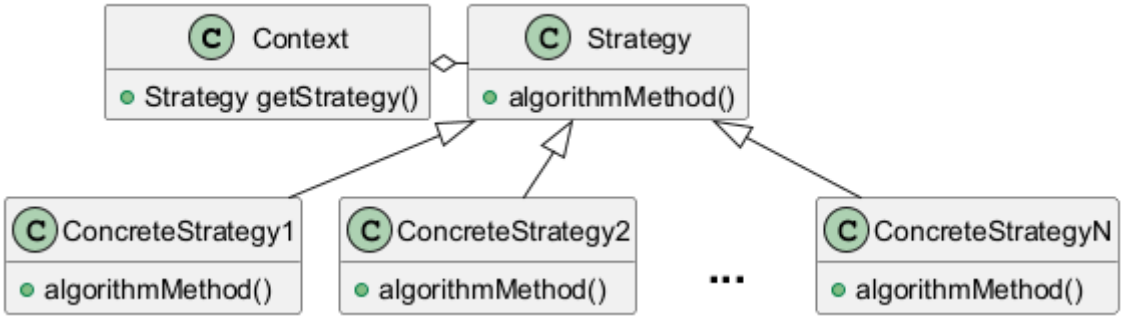
    void testOnlineState() {
        fsm.fire('connect')
        fsm.fire('send_message')
        fsm.fire('receive_message')
        shouldFail{
            fsm.fire('connect')
        }
        assert 'offline' == fsm.fire('disconnect')
    }
}
```

This example isn't an exact equivalent of the others. It doesn't use predefined `Online` and `Offline` classes. Instead, it defines the entire state machine on the fly as needed. See the previous reference for more elaborate examples of this style.

See also: Model-based testing using ModelJUnit

Strategy Pattern

The Strategy Pattern allows you to abstract away particular algorithms from their usage. This allows you to easily swap the algorithm being used without having to change the calling code. The general form of the pattern is:



In Groovy, because of its ability to treat code as a first class object using anonymous methods (which we loosely call *Closures*), the need for the strategy pattern is greatly reduced. You can simply place algorithms inside Closures.

Example using traditional class hierarchy

First let's look at the traditional way of encapsulating the Strategy Pattern.

```
interface Calc {
    def execute(n, m)
}

class CalcByMult implements Calc {
    def execute(n, m) { n * m }
}

class CalcByManyAdds implements Calc {
    def execute(n, m) {
        def result = 0
        n.times{
            result += m
        }

        result
    }
}

def sampleData = [
    [3, 4, 12],
    [5, -5, -25]
]

Calc[] multiplicationStrategies = [
    new CalcByMult(),
    new CalcByManyAdds()
```

```
]

sampleData.each{ data ->
    multiplicationStrategies.each { calc ->
        assert data[2] == calc.execute(data[0], data[1])
    }
}
```

Here we have defined an interface `Calc` which our concrete strategy classes will implement (we could also have used an abstract class). We then defined two algorithms for doing simple multiplication: `CalcByMult` the normal way, and CalcByManyAdds using only addition (don't try this one using negative numbers - yes we could fix this but it would just make the example longer). We then use normal polymorphism to invoke the algorithms.

Example using closures

Here is the Groovier way to achieve the same thing using Closures:

```
def multiplicationStrategies = [
    { n, m -> n * m },
    { n, m -> def result = 0; n.times{ result += m }; result }
]

def sampleData = [
    [3, 4, 12],
    [5, -5, -25]
]

sampleData.each{ data ->
    multiplicationStrategies.each { calc ->
        assert data[2] == calc(data[0], data[1])
    }
}
```

Example using lambdas

For Groovy 3+, we can leverage lambda syntax:

```
interface Calc {
    def execute(n, m)
}

List<Calc> multiplicationStrategies = [
    (n, m) -> n * m,
    (n, m) -> { def result = 0; n.times{ result += m }; result }
]

def sampleData = [
    [3, 4, 12],
```

```
    [5, -5, -25]
]

sampleData.each{ data ->
    multiplicationStrategies.each { calc ->
        assert data[2] == calc(data[0], data[1])
    }
}
```

Or we can use the built-in JDK `BiFunction` class:

```
import java.util.function.BiFunction

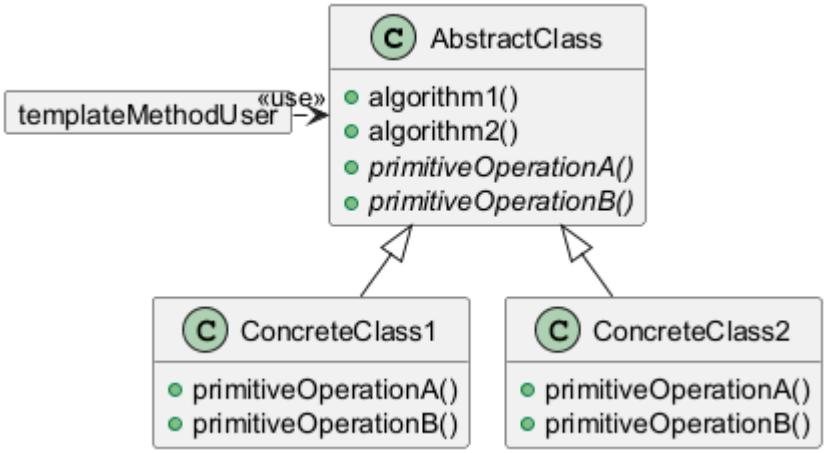
List<BiFunction<Integer, Integer, Integer>> multiplicationStrategies = [
    (n, m) -> n * m,
    (n, m) -> { def result = 0; n.times{ result += m }; result }
]

def sampleData = [
    [3, 4, 12],
    [5, -5, -25]
]

sampleData.each{ data ->
    multiplicationStrategies.each { calc ->
        assert data[2] == calc(data[0], data[1])
    }
}
```

Template Method Pattern

The Template Method Pattern abstracts away the details of several algorithms. The generic part of an algorithm is contained within a base class. Particular implementation details are captured within child classes. The generic pattern of classes involved looks like this:



Example with traditional classes

In this example, the base `Accumulator` class captures the essence of the accumulation algorithm. The child classes `Sum` and `Product` provide particular customised ways to use the generic accumulation algorithm.

```
abstract class Accumulator {
    protected initial
    abstract doAccumulate(total, v)
    def accumulate(values) {
        def total = initial
        values.each { v -> total = doAccumulate(total, v) }
        total
    }
}

class Sum extends Accumulator {
    def Sum() { initial = 0 }
    def doAccumulate(total, v) { total + v }
}

class Product extends Accumulator {
    def Product() { initial = 1 }
    def doAccumulate(total, v) { total * v }
}

assert 10 == new Sum().accumulate([1,2,3,4])
assert 24 == new Product().accumulate([1,2,3,4])
```

Example with simplifying strategies

In this particular case, you could use Groovy's inject method to achieve a similar result using Closures:

```
Closure addAll = { total, item -> total += item }
def accumulated = [1, 2, 3, 4].inject(0, addAll)
assert accumulated == 10
```

Thanks to duck-typing, this would also work with other objects which support an add (`plus()` in Groovy) method, e.g.:

```
accumulated = [ "1", "2", "3", "4" ].inject("", addAll)
assert accumulated == "1234"
```

We could also do the multiplication case as follows (re-writing as a one-liner):

```
assert 24 == [1, 2, 3, 4].inject(1) { total, item -> total *= item }
```

Using closures this way looks like the Strategy Pattern, but if we realise that Groovy's `inject` method is the generic part of the algorithm for our template method, then the Closures become the customised parts of the template method pattern.

For Groovy 3+, we can use lambda syntax as an alternative to the closure syntax:

```
assert 10 == [1, 2, 3, 4].stream().reduce(0, (l, r) -> l + r)
assert 24 == [1, 2, 3, 4].stream().reduce(1, (l, r) -> l * r)
assert '1234' == ['1', '2', '3', '4'].stream().reduce('', (l, r) -> l + r)
```

Here the stream api's `reduce` method is the generic part of the algorithm for our template method, and the lambdas are the customised parts of the template method pattern.

Visitor Pattern

The Visitor Pattern is one of those well-known but not often used patterns. Perhaps this is because it seems a little complex at first. But once you become familiar with it, it becomes a powerful way to evolve your code and as we'll see, Groovy provides ways to reduce some to the complexity, so there is no reason not to consider using this pattern.

The goal of the pattern is to separate an algorithm from an object structure. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures.

Simple Example

This example considers how to calculate the bounds of shapes (or collections of shapes). Our first attempt uses the traditional visitor pattern. We will see a more Groovy way to do this shortly.

```
abstract class Shape { }

@ToString(includeNames=true)
class Rectangle extends Shape {
    def x, y, w, h

    Rectangle(x, y, w, h) {
        this.x = x; this.y = y; this.w = w; this.h = h
    }

    def union(rect) {
        if (!rect) return this
        def minx = [rect.x, x].min()
        def maxx = [rect.x + rect.w, x + w].max()
        def miny = [rect.y, y].min()
        def maxy = [rect.y + rect.h, y + h].max()
        new Rectangle(minx, miny, maxx - minx, maxy - miny)
```

```
    }

    def accept(visitor) {
        visitor.visit_rectangle(this)
    }
}

class Line extends Shape {
    def x1, y1, x2, y2

    Line(x1, y1, x2, y2) {
        this.x1 = x1; this.y1 = y1; this.x2 = x2; this.y2 = y2
    }

    def accept(visitor){
        visitor.visit_line(this)
    }
}

class Group extends Shape {
    def shapes = []
    def add(shape) { shapes += shape }
    def remove(shape) { shapes -= shape }
    def accept(visitor) {
        visitor.visit_group(this)
    }
}

class BoundingRectangleVisitor {
    def bounds

    def visit_rectangle(rectangle) {
        if (bounds)
            bounds = bounds.union(rectangle)
        else
            bounds = rectangle
    }

    def visit_line(line) {
        def line_bounds = new Rectangle([line.x1, line.x2].min(),
                                        [line.y1, line.y2].min(),
                                        line.x2 - line.y1,
                                        line.x2 - line.y2)
        if (bounds)
            bounds = bounds.union(line_bounds)
        else
            bounds = line_bounds
    }

    def visit_group(group) {
        group.shapes.each { shape -> shape.accept(this) }
```

```
    }
}

def group = new Group()
group.add(new Rectangle(100, 40, 10, 5))
group.add(new Rectangle(100, 70, 10, 5))
group.add(new Line(90, 30, 60, 5))
def visitor = new BoundingRectangleVisitor()
group.accept(visitor)
bounding_box = visitor.bounds
assert bounding_box.toString() == 'Rectangle(x:60, y:5, w:50, h:70)'
```

That took quite a bit of code, but the idea now is that we could add further algorithms just by adding new visitors with our shape classes remaining unchanged, e.g. we could add a total area visitor or a collision detection visitor.

We can improve the clarity of our code (and shrink it to about half the size) by making use of Groovy Closures as follows:

```
abstract class Shape {
    def accept(Closure yield) { yield(this) }
}

@ToString(includeNames=true)
class Rectangle extends Shape {
    def x, y, w, h
    def bounds() { this }
    def union(rect) {
        if (!rect) return this
        def minx = [ rect.x, x ].min()
        def maxx = [ rect.x + rect.w, x + w ].max()
        def miny = [ rect.y, y ].min()
        def maxy = [ rect.y + rect.h, y + h ].max()
        new Rectangle(x:minx, y:miny, w:maxx - minx, h:maxy - miny)
    }
}

class Line extends Shape {
    def x1, y1, x2, y2
    def bounds() {
        new Rectangle(x:[x1, x2].min(), y:[y1, y2].min(),
                      w:(x2 - x1).abs(), h:(y2 - y1).abs())
    }
}

class Group {
    def shapes = []
    def leftShift(shape) { shapes += shape }
    def accept(Closure yield) { shapes.each{it.accept(yield)} }
}
```

```
def group = new Group()
group << new Rectangle(x:100, y:40, w:10, h:5)
group << new Rectangle(x:100, y:70, w:10, h:5)
group << new Line(x1:90, y1:30, x2:60, y2:5)
def bounds
group.accept{ bounds = it.bounds().union(bounds) }
assert bounds.toString() == 'Rectangle(x:60, y:5, w:50, h:70)'
```

Or, using lambdas as follows:

```
abstract class Shape {
    def accept(Function<Shape, Shape> yield) { yield.apply(this) }
}

@ToString(includeNames=true)
class Rectangle extends Shape {
    /* ... same as with Closures ... */
}

class Line extends Shape {
    /* ... same as with Closures ... */
}

class Group {
    def shapes = []
    def leftShift(shape) { shapes += shape }
    def accept(Function<Shape, Shape> yield) {
        shapes.stream().forEach(s -> s.accept(yield))
    }
}

def group = new Group()
group << new Rectangle(x:100, y:40, w:10, h:5)
group << new Rectangle(x:100, y:70, w:10, h:5)
group << new Line(x1:90, y1:30, x2:60, y2:5)
def bounds
group.accept(s -> { bounds = s.bounds().union(bounds) })
assert bounds.toString() == 'Rectangle(x:60, y:5, w:50, h:70)'
```

Advanced Example

Let's consider another example to illustrate some more points about this pattern.

```
interface Visitor {
    void visit(NodeType1 n1)
    void visit(NodeType2 n2)
}
```

```
interface Visitable {
    void accept(Visitor visitor)
}

class NodeType1 implements Visitable {
    Visitable[] children = new Visitable[0]
    void accept(Visitor visitor) {
        visitor.visit(this)
        for(int i = 0; i < children.length; ++i) {
            children[i].accept(visitor)
        }
    }
}

class NodeType2 implements Visitable {
    Visitable[] children = new Visitable[0]
    void accept(Visitor visitor) {
        visitor.visit(this)
        for(int i = 0; i < children.length; ++i) {
            children[i].accept(visitor)
        }
    }
}

class NodeType1Counter implements Visitor {
    int count = 0
    void visit(NodeType1 n1) {
        count++
    }
    void visit(NodeType2 n2){}
}
```

If we now use `NodeType1Counter` on a tree like this:

```
NodeType1 root = new NodeType1()
root.children = new Visitable[]{new NodeType1(), new NodeType2()}

def counter = new NodeType1Counter()
root.accept(counter)
assert counter.count == 2
```

Then we have one `NodeType1` object as root and one of the children is also a `NodeType1` instance. The other child is a `NodeType2` instance. That means using `NodeType1Counter` here should count 2 `NodeType1` objects as the last statement verifies.

When to use this

This example illustrates some of the advantages of the visitor pattern. For example, while our visitor has state (the count of `NodeType1` objects), the tree of objects itself is not changed. Similarly, if

we wanted to have a visitor counting all node types, or one that counts how many different types are used, or one that gathers information using methods special to the node types, again, the visitor alone is all that would need to be written.

What happens if we add a new type?

In this case we might have a fair bit of work to do. We probably have to change the `Visitor` interface to accept the new type, and change potentially most existing visitors based on that interface change, and we have to write the new type itself. A better approach is to write a default implementation of the visitor which all concrete visitors will extend. We'll see this approach in use shortly.

What if we want to have different iteration patterns?

Then you have a problem. Since the node describes how to iterate, you have no influence and stop iteration at a point or change the order. So maybe we should change this a little to this:

```
interface Visitor {
    void visit(NodeType1 n1)
    void visit(NodeType2 n2)
}

class DefaultVisitor implements Visitor{
    void visit(NodeType1 n1) {
        for(int i = 0; i < n1.children.length; ++i) {
            n1.children[i].accept(this)
        }
    }
    void visit(NodeType2 n2) {
        for(int i = 0; i < n2.children.length; ++i) {
            n2.children[i].accept(this)
        }
    }
}

interface Visitable {
    void accept(Visitor visitor)
}

class NodeType1 implements Visitable {
    Visitable[] children = new Visitable[0]
    void accept(Visitor visitor) {
        visitor.visit(this)
    }
}

class NodeType2 implements Visitable {
    Visitable[] children = new Visitable[0];
    void accept(Visitor visitor) {
        visitor.visit(this)
```

```
    }
}

class NodeType1Counter extends DefaultVisitor {
    int count = 0
    void visit(NodeType1 n1) {
        count++
        super.visit(n1)
    }
}
```

Some small changes but with big effect. The visitor is now recursive and tells me how to iterate. The implementation in the Nodes is minimized to `visitor.visit(this)`, `DefaultVisitor` is now able to catch the new types, we can stop iteration by not delegating to super. Of course the big disadvantage now is that it is no longer iterative, but you can't get all the benefits.

Make it Groovy

The question now is how to make that a bit more Groovy. Didn't you find this `visitor.visit(this)` strange? Why is it there? The answer is to simulate double dispatch. In Java, the compile time type is used, so for `visitor.visit(children[i])` the compiler won't be able to find the correct method, because `Visitor` does not contain a method `visit(Visitable)`. And even if it would, we would like to visit the more special methods with `NodeType1` or `NodeType2`.

Now Groovy is not using the static type, Groovy uses the runtime type. This means we can use `visitor.visit(children[i])` without any problem. Since we minimized the accept method to just do the double dispatch part and since the runtime type system of Groovy will already cover that, do we need the accept method? Not really, but we can do even more. We had the disadvantage of not knowing how to handle unknown tree elements. We had to *extend* the interface `Visitor` for that, resulting in changes to `DefaultVisitor` and then we have the task to provide a useful default like iterating the node or not doing anything at all. Now with Groovy we can catch that case by adding a `visit(Visitable)` method that does nothing. That would be the same in Java btw.

But don't let us stop here. Do we need the `Visitor` interface? If we don't have the accept method, then we don't need the `Visitor` interface at all. So the new code would be:

```
class DefaultVisitor {
    void visit(NodeType1 n1) {
        n1.children.each { visit(it) }
    }
    void visit(NodeType2 n2) {
        n2.children.each { visit(it) }
    }
    void visit(Visitable v) { }
}

interface Visitable { }

class NodeType1 implements Visitable {
```

```
    Visitable[] children = []
}

class NodeType2 implements Visitable {
    Visitable[] children = []
}

class NodeType1Counter extends DefaultVisitor {
    int count = 0
    void visit(NodeType1 n1) {
        count++
        super.visit(n1)
    }
}
```

Looks like we saved a few lines of code here, but we made more. The `Visitable` nodes now do not refer to any `Visitor` class or interface. This is about the best level of separation you might expect here, but we can go further. Let's change the `Visitable` interface a little and let it return the children we want to visit next. This allows us a general iteration method.

```
class DefaultVisitor {
    void visit(Visitable v) {
        doIteration(v)
    }
    void doIteration(Visitable v) {
        v.children.each {
            visit(it)
        }
    }
}

interface Visitable {
    Visitable[] getChildren()
}

class NodeType1 implements Visitable {
    Visitable[] children = []
}

class NodeType2 implements Visitable {
    Visitable[] children = []
}

class NodeType1Counter extends DefaultVisitor {
    int count = 0
    void visit(NodeType1 n1) {
        count++
        super.visit(n1)
    }
```

```
}
```

`DefaultVisitor` now looks a bit different. It has a `doIteration` method that will get the children it should iterate over and then call visit on each element. Per default this will call `visit(Visitable)` which then iterates over the children of this child. `Visitable` has also changed to ensure that any node will be able to return children (even if empty). We didn't have to change the `NodeType1` and `NodeType2` class, because the way the children field was defined already made them a property, which means Groovy is so nice to generate a get method for us. Now the really interesting part is `NodeType1Counter`, it is interesting because we have not changed it. `super.visit(n1)` will now call `visit(Visitable)` which will call `doIteration` which will start the next level of iteration. So no change. But `visit(it)` will call `visit(NodeType1)` if it is of type `NodeType1`. In fact, we don't need the `doIteration` method, we could do that in `visit(Visitable)` too, but this variant has some benefits. It allows us to write a new `Visitor` that overwrites visit(`Visitable`) for error cases which of course means we must not do `super.visit(n1)` but `doIteration(n1)`.

Summary

In the end we got ~40% less code, a robust and stable architecture, and we completely removed the Visitor from the Visitable. To achieve the same in Java, you would probably need to resort to reflection.

The visitor pattern has sometimes been described as a poor fit for extreme programming techniques because you need to make changes to so many classes all the time. With our design, if we add new types we don't need to change anything. So, the pattern is a good fit for agile approaches when using Groovy.

There are variants of the Visitor pattern, like the acyclic visitor pattern, that try to solve the problem of adding new node types with special visitors. The implementations of these visitors have their own code smells, like using casts, overuse of `instanceof`, and other tricks. What's more the problems such approaches are trying to solve don't occur within the Groovy version. We recommend avoiding that variant of this pattern.

Finally, in case it isn't obvious, `NodeType1Counter` could be implemented in Java as well. Groovy will recognize the visit methods and call them as needed because `DefaultVisitor` is still Groovy and does all the magic.

Further Information

- Componentization: the Visitor example

References

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.
    - *The canonical reference of design patterns.*
2. Martin Fowler (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley. ISBN 0-201-48567-2.
3. Joshua Kerievsky (2004). *Refactoring To Patterns*. Addison-Wesley. ISBN 0-321-21335-1.

4. Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates (2004). *Head First Design Patterns*. O'Reilly. ISBN 0-596-00712-4.
    - *A great book to read, informative as well as amusing.*
5. Dierk Koenig with Andrew Glover, Paul King, Guillaume Laforge and Jon Skeet (2007). *Groovy in Action*. Manning. ISBN 1-932394-84-2.
    - *Discusses Visitor, Builder and other Patterns.*
6. Brad Appleton (1999). Pizza Inversion - a Pattern for Efficient Resource Consumption.
    - *One of the most frequently used patterns by many software engineers!*
7. *Design Patterns in Dynamic Languages* by Neil Ford. Design Patterns in Dynamic Languages.

Acknowledgements

Contributors

The Groovy team would like to thank the contributors of this documentation (in alphabetical order of last/surname):

- Dan Allen
- Dmitry Andreychuk
- Hamlet D'Arcy
- Aseem Bansal
- Andrey Bloschetsov
- J Brown
- Jeff Scott Brown
- Cédric Champeau
- Tobia Conforto
- Dimitar Dimitrov
- Andrew Eisenberg
- Marcin Erdmann
- Christoph Frick
- Mario García
- David Michael Karr
- Paul King
- Guillaume Laforge
- Peter Ledbrook
- Grant McConnaughey
- Eric Milles
- David Nahodil
- James Northrop
- Marc Paquette
- Michael Schuenck
- Pascal Schumacher
- Shil Sinha
- Maksym Stavytskyi
- André Steingreß
- Daniel Sun

- Edinson Padrón Urdaneta
- Keegan Witt

License

This work is licensed under the Apache License, Version 2.0.