Groovy Training

Author Mhmd Alhjaj

Table of Contents

Intro

What is Groovy?

Groovy is an optionally typed, dynamic language for the Java platform with many features that are inspired by languages like Python, Ruby, and Smalltalk, making them available to Java developers using a Java-like syntax.

Unlike other alternative languages,it's designed as a companion to , not a replacement for, Java.

Groovy is often referred to as a scripting language, and it works very well for script- ing. It's a mistake to label Groovy purely in those terms, though.
It can be precompiled into Java bytecode, integrated into Java applications, power web applications,add an extra degree of control within build files, and be the basis of whole applications on its own.

Groovy, obviously, is too flexible to be pigeonholed.

SEAMLESS INTEGRATION

The integration aspect of Groovy: it runs inside the JVM and makes use of Java's libraries (together called the Java Runtime Environment, or JRE). Groovy is only a new way of creating ordinary Java classes— from a runtime perspective, Groovy is Java with an additional JAR file as a dependency.

Consequently, calling Java from Groovy is a nonissue. When developing in Groovy, you end up doing this all the time without noticing. Every Groovy type is a subtype of `java.lang.Object` . Every Groovy object is an instance of a type in the normal way.

A Groovy date is a `java.util.Date` .You can call all methods on it that you know are available for a Date , and you can pass it as an argument to any method that expects a Date .

SYNTAX ALIGNMENT

Lets compare these snippets :

```
import java.util.*;
Date today = new Date();
```

```
def date = new Date()
```

```
require 'date'
today = Date.new
```

```
import java.util._
var today = new Date
```

```
(import '(java.util Date))
(def today (new Date))
(def today (Date.))
```

```
import java.util.*
var date=Date()
```

They are similar the only difference is the optional typing which you can actually use the explicit type if you want another difference is the import statement you don't need it in groovy because groovy import java.util.* by default.

FEATURE-RICH LANGUAGE

Groovy has two main enhancements over and above those of Java:

- language features
- libraries specific to Groovy, and additions to the existing Java standard classes (known as the Groovy Development Kit, or GDK)

here are a few examples :

LISTING A FILE : CLOSURES AND I/O ADDITIONS

in groovy

```
def number = 0
new File('data.txt').eachLine { line ->
    number++
    println "$number: $line"
}
```

in java

```
final int[] number = {0};
        try {
            Files.lines(Paths.get(filePath)).forEach(line -> {
                number[0]++;
                System.out.println(number[0] + ":" + line);
            });
        } catch (IOException e) {
            e.printStackTrace();
        }
```

PRINTING A LIST : COLLECTION LITERALS AND SIMPLIFIED PROPERTY ACCESS

in groovy

```
def classes = [String, List, File]

for (clazz in classes) {
    println clazz.package.name
}
```

in java

```
import java.io.File;
import java.util.List;

Class[] classes={String.class,List.class,File.class};
for(Class clazz : classes){
    System.out.println(clazz.getPackage().getName());
}
```

XML HANDLING THE GROOVY WAY : GPATH WITH DYNAMIC PROPERTIES

customers.xml

```
<?xml version="1.0" ?>
<customers>
    <corporate>
        <customer name="Bill Gates"
        <customer name="Tim Cook"
        <customer name="Larry Ellison"
    </corporate>
    <consumer>
        <customer name="John Doe" />
        <customer name="Jane Doe" />
    </consumer>
</customers>
```

in groovy

```
def customers = new XmlSlurper().parse(new File('customers.xml'))
for (customer in customers.corporate.customer) {
    println "${customer.@name} works for ${customer.@company}"
}
```

in java

```
try {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder documentBuilder = factory.newDocumentBuilder();
    Document document = documentBuilder.parse(new File("customers.xml"));
    NodeList customers = document.getElementsByTagName("customer");
    for (int i = 0; i < customers.getLength(); i++) {
        Node item = customers.item(i);
        System.out.println(item.getAttributes().getNamedItem("name")
                + " works for "
                + (item.getAttributes().getNamedItem("company")));
    }
} catch (ParserConfigurationException | IOException | SAXException e) {
    e.printStackTrace();
}
```

Running Groovy

Table 1. Commands to execute Groovy

Command	What it does
`groovy`	Starts the processor that executes Groovy scripts. Single-line Groovy scripts can be specified as command-line arguments.

`groovysh`	Starts the groovysh command-line shell, used to execute Groovy code inter- actively. By entering statements or whole scripts line by line into the shell, code is executed on the fly.
`groovyConsole`	Starts a graphical interface that's used to execute Groovy code interactively; moreover, groovyConsole loads and runs Groovy script files.

Compiling and running Groovy

```
groovyc -d classes HelloWorld.groovy
```

the `groovyc` compiler outputs Java class files to a directory named `classes` , which you told it to do with the `-d` flag. If the directory specified with `-d` doesn't exist, it's created. When you're running the compiler, the name of each gener- ated class file is printed to the console.

For each script, Groovy generates a class that extends `groovy.lang.Script` , which contains a main method so that Java can execute it. The name of the compiled class matches the name of the script being compiled. More classes may be generated, depending on the script code.

Running a compiled Groovy script with Java

Running a compiled Groovy program is identical to running a compiled Java pro- gram, with the added requirement of having the embeddable `groovy-all-*.jar` file in your JVM 's classpath, which will ensure that all of Groovy's third-party dependencies will be resolved automatically at runtime. Make sure you add the directory in which your compiled program resides to the classpath, too. You then run the program in the same way you'd run any other Java program, with the `java` command.

```
java –cp %GROOVY_HOME%/embeddable/groovy-all-2.4.0.jar;classes HelloWorld
```

BASICS

Commenting Groovy code

Single-line comments and multiline comments are exactly like those in Java, with an additional option for the first line of a script

```
#!/usr/bin/env groovy
// some line comment
/* some multi
line comment */
```

Java's syntax is part of the Groovy syntax

This applies to:

- The general packaging mechanism.
- Statements (including package and import statements).
- Class, interface, enum, field, and method definitions including nested classes,
- except for special cases with nested class definitions inside methods or other
- deeply nested blocks.
- Control structures.
- Operators, expressions, and assignments.
- Exception handling.
- Declaration of literals, with the exception of literal array initialization where the
- Java syntax would clash with Groovy's use of braces. Groovy uses a shorter
- bracket notation for declaring lists instead.
- Object instantiation, referencing and dereferencing objects, and calling methods.
- Declaration and use of generics and annotations.

The added value of Groovy's syntax includes the following:

- Ease access to Java objects through new expressions and operators.
- Allow more ways of creating objects using literals.
- Provide new control structures to allow advanced flow control.
- Use annotations to generate invisible code, the so-called AST transformations.
- Introduce new datatypes together with their operators and expressions.
- A backslash at the end of a line escapes the line feed so that the statement can proceed on the following line.
- Additional parentheses force Groovy to treat the enclosed content as an expression

Beauty through brevity

Groovy allows you to leave out some elements of syntax that are always required in Java. Omitting these elements often results in code that's shorter and more expressive. Compare the Java and Groovy code for encoding a string for use in a URL .

in java

```
java.net.URLEncoder.encode("a b", "UTF-8");
```

in groovy

```
URLEncoder.encode 'a b', 'UTF-8'
```

By leaving out the package prefix, parentheses, and semicolon, the code boils down to the bare minimum.

Although these rules are unambiguous, they're not always intuitive. Omitting parentheses can lead to misunderstandings, even though the compiler is happy with the code. We pre- fer to include the parentheses for all but the most trivial situations. The compiler doesn't try to judge your code for readability—you must do this yourself.

> Groovy automatically imports the packages `groovy.lang.\*` , `groovy.util.*`, `java.lang.\*` , `java.util.*` , `java.net.\*` , and `java.io.*` , as well as the classes `java.math.BigInteger` and `BigDecimal` . As a result, you can refer to the classes in these packages without specifying the package names. We'll use this feature through- out the book, and we'll use fully qualified class names only for disambiguation or for pointing out their origin. Note that Java automatically imports `java.lang.*` , but noth- ing else.

Probing the language with assertions

assertion is an old paradigm that has been around since before the advent of object-oriented languages.
An assertion is a statement that asserts that some condition is true, and causes an execution failure if the condition is not respected.
It's that simple. Failure can be translated into an exception, a runtime error, or even a runtime failure resulting in unstoppable application termination.

```
assert(true)
assert 1 == 1
def x = 1
assert x == 1
def y = 1; assert y == 1
```

What happens if an assertion fails?

```
def a = 5
def b = 9
assert b == a + a

Assertion failed:
assert b == a + a
| | | | |
9 | 5 | 5
|
10
false
```

Assertions serve multiple purposes:

- They can be used to reveal the current program state, as they're used in the examples in this book. The one-line assertion in the previous example reveals that the variable `y` now has the value `1` .
- They often make good replacements for line comments, because they reveal assumptions and verify them at the same time. The assertion reveals that, at this point, it's assumed that `y` has the value `1` . Comments may go out of date without anyone noticing—assertions are always checked for correctness. They're like tiny unit tests sitting inside the real code.

Declaring classes

Classes are the cornerstone of object-oriented programming (OOP), because they define the blueprints from which objects are created.

Book.groovy

```
class Book {
    private String title

    Book (String theTitle) {
        title = theTitle
    }

    String getTitle(){
        return title
    }
}
```

Everything looks much like Java, except there's no accessibility modifier: methods are `public by default`.

Scripts are text files, typically with an extension of *.groovy, that can be executed from the command shell like this:

```
> groovy myfile.groovy
```

This is very different from Java. In Groovy, you're executing the source code! An ordinary Java class is generated for you and executed behind the scenes. But from a user's perspective, it looks like you're executing plain Groovy source code.

Scripts contain Groovy statements without an enclosing class declaration. Scripts can even contain method definitions outside of class definitions to better structure the code.

myscript.groovy

```
Book gina = new Book('Groovy in Action')

assert gina.getTitle() == 'Groovy in Action'
assert getTitleBackwards(gina) == 'noitcA ni yvoorG'

String getTitleBackwards(book) {
    String title = book.getTitle()
    return title.reverse()
}
```

To run this snippet of code :

```
> groovy myscript.groovy
```

GroovyBeans

JavaBeans are ordinary Java 5 classes that expose properties. What is a property? That's not easy to explain, because it's not a single standalone concept. It's made up from a naming convention. If a class exposes methods with the naming scheme `getName()` and `setName(name)` , then the concept describes `name` as a property of that class. The get and set methods are called accessor methods.

Boolean properties can use an `is` prefix instead of get , leading to method names such as `isAdult` .

GroovyBean

is a JavaBean defined in Groovy.
In Groovy, working with beans is much easier than in Java. Groovy facilitates working with beans in three ways:

- Generating the accessor methods
- Allowing simplified access to all JavaBeans (including GroovyBeans)
- Simplifying registration of event handlers together with annotations that declare a property as bindable

Person.groovy

```
class Person{
    String name
    Integer Age
}

def person=new Person()

person.setName("Bob")            ①
assert person.getName() == "Bob" ①

person.age=24                    ②
assert person.age == 24          ②
```

① Property use with explicit getter calls

② Property use with Groovy shortcuts

NOTE `groovyBook.title` is not a field access. Instead, it's a shortcut for the corresponding accessor method. It'd work even if you'd explicitly declared the prop- erty longhand with a `getTitle()` method.

Annotations

In Groovy, you can define and use annotations just like in Java, which is a distinctive feature among JVM languages. Beyond that, Groovy also uses annotations to mark code structures for special compiler handling. Let's have a look at one of those anno- tations that comes with the Groovy distribution: `@Immutable` .

Person.groovy

```
import groovy.transform.Immutable

@Immutable class FixedBook { ①
    String title
}

def gina = new FixedBook('Groovy in Action') ②
def regina = new FixedBook(title:'Groovy in Action') ③

assert gina.title == 'Groovy in Action'
assert gina == regina

try {
    gina.title = "Oops!" ④
    assert false, "should not reach here"
} catch (ReadOnlyPropertyException expected) {
    println "Expected Error: '$expected.message'"
}
```

① AST annotation.

② Positional constructor.

③ Named-arg constructor.

④ Not Allowed Final member can't be changed

NOTE The annotation does actually much more than what you see it adds a correct `hashCode()` implementation and enforces defensive copying for access to all properties that aren't immutable by themselves.

Using grapes

The `@Grab` annotation is used to explicitly define your external library dependencies within a script. We sometimes use the term `grapes` as friendly shorthand for our external Groovy library dependencies. In the Java world, you might store your dependent libraries in a lib directory and add that to your classpath and IDE settings, or you might capture that information in an Ivy, Maven, or Gradle build file. Groovy provides an additional alternative that's very handy for making scripts self-contained.

Outer.groovy

```
@Grab('commons-lang:commons-lang:2.4')

import org.apache.commons.lang.ClassUtils

class Outer {
    class Inner {}
}

assert !ClassUtils.isInnerClass(Outer)
assert ClassUtils.isInnerClass(Outer.Inner)
```

Handling text

Just as in Java, character data is mostly handled using the `java.lang.String` class. But Groovy provides some tweaks to make that easier, with more options for string literals and some helpful operators.

GStrings

In Groovy, string literals can appear in single or double quotes. The double-quoted version allows the use of placeholders, which are automatically resolved as required. This is a `GString`, and that's also the name of the class involved. The following code demonstrates a simple variable expansion, although that's not all GStrings can do:

```
def nick = 'ReGina'
def msg = 'Hello'
assert "$nick says $msg" == 'ReGina says Hello'
```

REGULAR EXPRESSIONS

Groovy makes it easy to declare regular expression patterns, and provides operators for applying them.

```
assert '12345' =~ /\d+/ ①
assert 'xxxxx' == '12345'.replaceAll(/\d/, 'x') ②

assert java.lang.String == /foo/.class
assert ( /Count is \d/ == "Count is \\d" )

def name = "Ted Naleid"
assert ( /$name/ == "Ted Naleid" )
assert ( /$name/ == "$name" )

def shoutedWord = ~/\b[A-Z]+\b/
assert java.util.regex.Pattern == shoutedWord.class

def matcher = ("EUREKA" =~ shoutedWord)
assert matcher.matches() // TRUE

assert "1234" ==~ /\d+/    // TRUE
assert "FOO2" ==~ /\d+/    // FALSE

assert "Green Eggs and Spam" == "Spam Spam".replaceFirst(/Spam/, "Green Eggs and")

assert ["foobar", "bazbar"] == ["foobar", "bazbar", "barquux"].grep(~/.*bar$/)

assert ["foobar", "bazbar"] == ["foobar", "bazbar", "barquux"].findAll { it ==~
/.*bar$/ }
```

① find operator

② regular expression syntax

Numbers are objects

Hardly any program can do without numbers, whether for calculations or (more fre- quently) for counting and indexing. Groovy numbers have a familiar appearance, but unlike in Java, they're first-class objects rather than primitive types.

In Java, you cannot invoke methods on primitive types. If `x` is of primitive type `int` , you cannot write `x.toString()` . On the other hand, if `y` is an object, you cannot use `2*y` .

In Groovy, both are possible. You can use numbers with numeric operators, and you can also call methods on number instances.

```
def x = 1
def y = 2
assert x + y == 3
assert x.plus(y) == 3
assert x instanceof Integer
```

Using lists, maps, and ranges

Many languages, including Java, only have direct support for a single collection type— an array at the syntax level and have language features that only apply to that type. In practice, other collections are widely used, and there's no reason why the language should make it harder to use those collections than arrays.

Groovy makes collection handling simple, with added support for operators, literals, and extra methods beyond those provided by the Java standard libraries.

LISTS

Java supports indexing arrays with a square bracket syntax, which we'll call the `subscript operator`.

In Groovy the same syntax can be used with lists instances of `java.util.List` which allows adding and removing elements, changing the size of the list at runtime, and storing items that aren't necessarily of a uniform type.

In addition, Groovy allows lists to be indexed outside their current bounds ,which again can change the size of the list. Furthermore, lists can be specified as literals directly in your code.

```
def roman = ['', 'I', 'II', 'III', 'IV', 'V', 'VI', 'VII']
assert roman[4] == 'IV'
roman[8] = 'VIII'
assert roman.size() == 9
```

NOTE	there was no list item with index 8 when you assigned a value to it. You indexed the list outside the current bounds.

SIMPLE MAPS

A map is a storage type that associates a key with a value. Maps store and retrieve values by key; lists retrieve them by numeric index.

Unlike Java, Groovy supports maps at the language level, allowing them to be specified with literals and providing suitable operators to work with them. It does so with a clear and easy syntax.

The syntax for maps looks like an array of key–value pairs, where a colon separates keys and values. That's all it takes.

Http.groovy

```
def http = [
   100 : 'CONTINUE',
   200 : 'OK',
   400 : 'BAD REQUEST'
]
assert http[200] == 'OK'
http[500] = 'INTERNAL SERVER ERROR'
assert http.size() == 4
```

NOTE The syntax is consistent with that used to declare, access, and modify lists. The differences between using maps and lists are minimal, so it's easy to remember both. This is a good example of the Groovy language designers taking commonly required operations and making programmers' lives easier by providing a simple and consistent syntax.

RANGES

Although ranges don't appear in the standard Java libraries, most programmers have an intuitive idea of what a range is—effectively a start point and an end point, with an operation to move between the two in discrete steps. Again, Groovy provides literals to support this useful concept, along with other language features such as the for statement, which understands ranges.

Ranges.groovy

```
def x = 1..10
assert x.contains(5)
assert !x.contains(15)
assert x.size() == 10
assert x.from == 1
assert x.to == 10
assert x.reverse() == 10..1
```

Code as objects: closures

The concept of closures isn't a new one, but it has usually been associated with func- tional languages, allowing one piece of code to execute an arbitrary piece of code that has been specified elsewhere.

In object-oriented languages, the Method Object pattern has often been used to simulate the same kind of behavior by defining types, the sole purpose of which is to implement an appropriate single-method interface. The instances of those types can subsequently be passed as arguments to

methods, which then invoke the method on the interface.

A good example is the `java.io.File.list(FilenameFilter)` method.

The FilenameFilter interface specifies a single method, and its only purpose is to allow the list of files returned from the list method to be filtered while it's being generated.

Unfortunately, this approach leads to an unnecessary proliferation of types, and the code involved is often widely separated from the logical point of use. Java uses anonymous inner classes and, since Java 8, lambdas and method references to address these issues. Although similar in function, Groovy closures are much more versatile and powerful when it comes to reaching out to the caller's scope and putting closures in a dynamic execution context.

Groovy allows closures to be specified in a concise, clean, and powerful way, effectively promoting the Method Object pattern to a first- class position in the language.

```
[1, 2, 3].each { entry -> println entry }
```

in groovy

```
def totalClinks = 0
def partyPeople = 100

1.upto(partyPeople) { guestNumber ->
            clinksWithGuest = guestNumber-1
            totalClinks += clinksWithGuest
}
assert totalClinks == (partyPeople * (partyPeople-1)) / 2
```

in java

```
int totalClinks = 0;
int partyPeople = 100;
for(int guestNumber = 1; guestNumber <= partyPeople;guestNumber++) {
    int clinksWithGuest = guestNumber-1;
    totalClinks += clinksWithGuest;
}
```

Groovy control structures

Control structures allow a programming language to control the flow of execution through code. There are simple versions of everyday control structures like if-else , while , switch , and try-catch-finally in Groovy, just like in Java. In conditionals, null is treated like false , and so are empty strings, collections, and maps.

```
for(i in x) { body }
```

x can be anything that Groovy knows how to iterate through, such as an iterator, an enumeration, a collection, a range, a map—or literally any object

In Groovy, the for loop is often replaced by iteration methods that take a closure argument. The following listing gives an overview.

```
if (false) assert false

if (null){
    assert false
}
else {
        assert true
}

def i = 0
while (i < 10) {
    i++
}
assert i == 10

def clinks = 0
for (remainingGuests in 0..9) {
    clinks += remainingGuests
}
assert clinks == (10*9)/2

def list = [0, 1, 2, 3]
for (j in list) {
    assert j == list[j]
}

list.each() { item ->
    assert item == list[item]
}

switch(3) {
    case 1 : assert false; break
    case 3 : assert true; break
    default: assert false
}
```

Simple Groovy datatypes

Java's type system: primitives and references

Java distinguishes between primitive types (such as boolean , short , int , double , float , char , and byte) and reference types (such as Object and String).

You cannot call methods on values of primitive types, and you cannot use them where Java expects objects of type `java.lang.Object` . For each primitive type, Java has a wrapper type a reference type that stores a value of the primitive type in an object.

The wrapper for `int` , for example, is `java.lang.Integer` . Conversely, operators such as * in `3 * 2` or `a * b` aren't supported for arbitrary 1 reference types in Java, but only for primitive types (with the notable exception of + , which is also supported for strings).

```
(60 * 60 * 24 * 365).toString(); // invalid Java
int secondsPerYear = 60 * 60 * 24 * 365;
secondsPerYear.toString(); // invalid Java
new Integer(secondsPerYear).toString();
assert "abc" - "a" == "bc" // invalid Java
```

Groovy's answer: everything's an object

To make Groovy fully object-oriented, and because at the JVM level Java doesn't sup- port object-oriented operations such as method calls on primitive types, the Groovy designers decided to do away with primitive types. When Groovy needs to store values that would have used Java's primitive types, Groovy uses the wrapper classes already provided by the Java platform.

Table 2. Java's primitive datatypes and their wrappers

Primitive type	Wrapper type	Description
byte	java.lang.Byte	8-bit signed integer
short	java.lang.Short	16-bit signed integer
int	java.lang.Integer	32-bit signed integer
long	java.lang.Long	64-bit signed integer
float	java.lang.Float	Single-precision (32-bit) floating-point value
double	java.lang.Double	Double-precision (64-bit) floating-point value
char	java.lang.Character	16-bit Unicode character
boolean	java.lang.Boolean	Boolean value (true or false)

Any time you see what looks like a primitive literal value (the number 5 , for example, or the Boolean value true) in Groovy source code, that's a reference to an instance of the appropriate wrapper class.

Table 3. Numeric literals in Groovy

Type	Example literals
java.lang.Integer	15, 0x1234ffff, 0b00110011, 100_000_000
java.lang.Long	100L, 200l
java.lang.Float	1.23f, 4.56F
java.lang.Double	1.23d, 4.56D
java.math.BigInteger	123g, 456G
java.math.BigDecimal	1.23, 4.56, 1.4E4, 2.8e4, 1.23g, 1.23G

Interoperating with Java: automatic boxing and unboxing

Groovy performs these operations automatically for you where necessary. This is primarily the case when you call a Java method from Groovy. This automatic boxing and unboxing is known as autoboxing.

```
assert 'ABCDE'.indexOf(67) == 2
```

From Groovy's point of view, you're passing an Integer containing the value 67 (the Unicode value for the letter C), even though the method expects a parameter of primitive type int .

Groovy takes care of the unboxing. The method returns a primitive type int that's boxed into an Integer as soon as it enters the world of Groovy. That way, you can compare it to the Integer with value 2 back in the Groovy script.

No intermediate unboxing

If in 1+1 both numbers are objects of type Integer , you may be wondering whether those Integer objects are unboxed to execute the plus operation on primitive types.

The answer is no: Groovy is more object-oriented than Java. It executes this expres- sion as 1.plus(1) , calling the plus() method of the first Integer object, and passing 2 the second Integer object as an argument. The method call returns an Integer object of value 2 .

This is a powerful model. Calling methods on objects is what object-oriented lan- guages should do. It opens the door for applying the full range of object-oriented capabilities to those operators

Optional typing

Assigning types

Groovy offers the choice of explicitly specifying variable types just as you do in Java, And also offer

the choice of implicit typing which is done by using The def keyword.

```
def a = 1       //java.lang.Integer
def b = 1.0f    //java.lang.Float
int c = 1       //java.lang.Integer
float d = 1     //java.lang.Float
Integer e = 1   //java.lang.Integer
String f = '1'  //java.lang.String
```

Regardless of whether a variable's type is explic- itly declared, the system is type safe.

Unlike untyped languages, Groovy doesn't allow you to treat an object of one type as an instance of a different type without a well- defined conversion being available.

You could never assign a `java.util.Date` to a reference of type `java.lang.Number` , in the hope that you'd end up with an object that you could use for calculation. That sort of behavior would be dangerous, which is why

Groovy doesn't allow it any more than Java does.

Dynamic Groovy is type safe

Static is often associated with the appearance of type markers in the code. For instance, code such as

```
String greeting = readFromConsole()
```

is often considered static because of the String type marker, while unmarked code like

```
def greeting = readFromConsole()
```

is usually deemed dynamic.

By default, Groovy is very much a dynamic language. You can safely leave out type markers (and also type casts) in most scenarios and know that Groovy will do the appropriate runtime checks to ensure type safety when required.

Because type markers are optional in Groovy, that concept is often called **optional typing**.

Groovy uses type markers to enforce the Java type system at runtime.But it only does so at runtime, where Java does so with a mixture of compile-time and runtime checks.

This explains why the Groovy compiler 6 takes no issue with

```
Integer myInt = new Object() // ClassCastException At runtime
println myInt
```

In fact, this is the same effect you see if you write a typecast on the right-hand side of the assignment in Java. Consider this Java code:

```
Integer myInt = (Integer) returnsObject(); // Java!
```

Groovy types aren't dynamic, they never change

The word "dynamic" doesn't mean that the type of a reference, once declared, can ever change. Once you've declared Integer `myInt`, you cannot execute `myInt = new Object()`. This will throw a `GroovyCastException`.

You can only assign a value, which Groovy can cast to an Integer.

Casting

Groovy actually applies convenience logic when casting, which is mainly concerned with casting primitive types to their wrapper classes and vice versa, arrays to lists, characters to integers, Java's type widening for numeric types, applying the "Groovy truth"for casts to boolean , calling toString() for casts to string , and so on. The exhaustive list can be looked up in `DefaultTypeTransformation.castToType` .

```
import java.awt.*

Point topLeft = new Point(0, 0) // classic

Point botRight = [100, 100] // List cast

Point center = [x:50, y:50] // Map cast

assert botRight instanceof Point
assert center instanceof Point

def rect = new Rectangle()
rect.location = [0, 0] // Point
rect.size = [width:100, height:100] // Dimension
```

Implicit runtime casting can lead to very readable code,especially in cases like property assignments where Groovy knows that `rect.size` is of type java.awt.Dimension and can cast your list or map of constructor arguments onto that.

You don't have to worry about it: Groovy infers the type for you.

Optional typing

Groovy is an "optionally" typed language, and that distinction is an important one to grasp when

understanding the fundamentals of the language. Groovy's nearest ancestor, Java, is said to be a "strongly" typed language, whereby the compiler knows all of the types for every variable and can understand and honor contracts at compile time. This means that method calls are able to be determined at compile time, and therefore take the onus of their resolution off of the runtime system.

```
class UserHibernateDAO {
    def sessionFactory

    def getByFirstName(String name) {
        List<User> users = sessionFactory.createQuery("select * from user where
firstName = :name")
                            .setParameter("name", name)
                            .list()
        users.size() == 1 ? users[0] : users
    }
}
```

Duck typing

If it walks like a duck and quacks like a duck, it must be a duck.

Duck typing
: implies that as long as an object has a certain set of method signatures, it's interchangeable with any other object that has the same set of methods, regard- less of whether the two have a related inheritance hierarchy.

NOTE Experienced Groovy programmers tend to follow this rule of thumb: as soon as you think about the type of a reference, declare it; if you're thinking of it as "just an object," leave the type out.

Overriding operators

When a language bases its operators on method calls and allows these methods to be overridden, the approach is called operator overriding.

Table 4. Method-based operators

Operator	Name
+	a.plus(b)
-	a.minus(b)

Operator	Name
*	a.multiply(b)
/	a.div(b)
%	a.mod(b)
<<	a.leftShift(b)
>>	a.rightShift(b)
>>>	a.rightShiftUnsigned(b)
**	a.power(b)
|	a.or(b)
&	a.and(b)
^	a.xor(b)
++	a.next()
—	a.previous()
~a	a.bitwiseNegate()
-a	a.negative()
a[b]	a.getAt(b)
a[b] = c	a.putAt(b, c)
a in b	b.isCase(a)
+a	a.positive()
as	a.asType(b)
a()	a.call()
switch(a){case b:}	b.isCase(a)
a in	b.isCase(a)
a ==	if (a implements Comparable) { a.compareTo(b) == 0 } else { a.equals(b) }
a != b	!(a==b)
a <⇒	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
a as b	a.asType(b)

Example 1. Overridden operators

```
import groovy.transform.Immutable;

@Immutable
class Money {
    int value

    Money plus(Money other) {
        new Money(this.value + other.value) ①
    }
}

def tenDollar = new Money(10)
def fiveDollar = new Money(5)

assert (tenDollar + fiveDollar).value == 15
```

① Implicit return statement

NOTE Our plus operation on the Money class returns Money objects in both cases. We describe this by saying that Money 's plus operation is closed under its type. Whatever operation you perform on an instance of Money , you end up with another instance of Money .

coercion

```
1 + 1.0
```

What is the return type? the issue is more general.

One of the two arguments needs to be promoted to the more general type. This is called `coercion`.

When implementing operators, there are three main issues to consider as part of coercion:

Supported argument types

You need to decide which argument types and values will be allowed. If an operator must take a potentially inappropriate type, throw an IllegalArgumentException where necessary.

Promoting more specific arguments

If the argument type is a more specific one than your own type, promote it to your type and return an object of your type.
`Integer` is more specific than `BigDecimal` : every `Integer` value can be expressed as a `BigDecimal` , but the reverse isn't true. So for the `BigDecimal`.plus(Integer) operator, you'd consider promoting the `Integer` to `BigDecimal` , performing the addi- tion, and then returning another `BigDecimal`

—even if the result could accurately be expressed as an `Integer` .

Handling more general arguments with double dispatch

If the argument type is more general, call its operator method with yourself as an argument. Let it promote you. This is also called double dispatch,and it helps to avoid duplicated , asymmetric, possibly inconsistent code.

NOTE Groovy's general strategy of coercion is to return the most general type.

Strings

Table 5. String literal styles available in Groovy

Start/end characters	Example	Placeholder resolved?	Backslash escapes?
Single quote	*hello Dierk*	No	Yes
Double quote	"hello $name"	Yes	Yes
Triple single quote	' ' ' ======= Total: $0.02 ======= ' ' '	No	Yes
Triple double quote	"""first $line second $line third $line"""	Yes	Yes
Forward slash	/x(\d*)y/	Yes	Occasionally
Dollar slash	$/x(\d*)y/$	Yes	Occasionally

The single-quoted

never pays any attention to placeholders. This is closely equivalent to Java string literals.

The double-quoted

is the equivalent of the single-quoted form, except that if the text contains unescaped dollar signs, the dollar sign introduces a placeholder, and the string will be treated as a GS tring instead of a plain string. GStrings are covered in more detail in the next section.

The triple-quoted (or multiline string literal)

allows the literal to span sev- eral lines. New lines are always treated as \n regardless of the platform, but all other whitespace is preserved as it appears in the text file. Multiline string literals may also be GS trings, depending on whether single quotes or double quotes are used. Multiline string literals act similar to Ruby or Perl.

The slashy

is also multiline but allows strings with back- slashes to be specified simply without having to escape all the backslashes. This is particularly useful with regular expressions, as you'll see later. There are only a few exceptions and limitations. Slashes are escaped with a backslash. A backslash can't appear as the last character of a slashy string. Dollar symbols that could introduce a placeholder but aren't meant to also need to be escaped. If you want to create a

string with a backslash followed by a u, the backslash needs to be escaped so as not to be interpreted as a Unicode character, which happens in the earliest stages of parsing.

The dollar slashy

allows strings with backslashes to be specified without having to escape all the backslashes. Only Unicode characters are escaped with a backslash. Dollar signs and slashes are escaped with a dollar sign. The other restrictions on backslashes you saw for normal slashy strings don't apply.

Table 6. Escaped characters as known to Groovy

Escaped special character	Meaning
\b	Backspace
\t	Tab
\r	Carriage return
\n	Linefeed
\f	Form feed
\\	Backslash
\$	Dollar sign
\uabcd	Unicode character u + abcd (where a, b, c, and d are hex digits)
\abc	Unicode character u + abc (where a, b, and c are octal digits, and b and c are optional)
\'	Single quote
\"	Double quote

```
char a = 'x'
Character b = 'x'

'x' as char
'x'.toCharacter()
```

GStrings

`GStrings` are like strings with additional capabilities.They're literally declared in double quotes. What makes a double-quoted string literally a GS tring is the appearance of placeholders.

Placeholders may appear in a full `${expression}` syntax or an abbreviated `$reference` syntax.

```
import java.time.LocalDateTime

def name = "Jone"
def meetingName = "Simple"
def template = "Dear $name, $meetingName meeting starts at
${LocalDateTime.now().hour}pm tonight."
assert template == 'Dear Jone, Simple meeting starts at 14pm tonight.'

TimeZone.default = TimeZone.getTimeZone('GMT')
def date = new Date(0)
def dateMap = [y:date[YEAR]-1900
             , m:date[MONTH]
             , d:date[DAY_OF_MONTH]]

def out = "Year $dateMap.y Month $dateMap.m Day $dateMap.d"
assert out == 'Year 70 Month 0 Day 1'
Extended
def timeZone = TimeZone.getTimeZone('GMT')
def format = 'd MMM YYYY HH:mm:SS z'
out = "Date is ${date.format(format, timeZone)} !"
assert out == 'Date is 1 Jan 1970 00:00:00 GMT !'

def sql = """
SELECT FROM MyTable
WHERE Year = $dateMap.y
"""

assert sql == """
SELECT FROM MyTable
WHERE Year = 70
"""

out = "my 0.02\$"
assert out == 'my 0.02$'
```

Although GS trings behave like java.lang.String objects for all operations that a programmer is usually concerned with, they’re implemented differently to capture the fixed and dynamic parts (the so-called values) separately. This is revealed by the following code:

```
def me = 'Tarzan'
def you = 'Jane'
def line = "me $me - you $you"

assert line == 'me Tarzan - you Jane'
assert line instanceof GString

assert line.strings[0] == 'me '
assert line.strings[1] == ' - you '

assert line.values[0] == 'Tarzan'
assert line.values[1] == 'Jane' 14
```

Placeholder evaluation time

Each placeholder inside a GString is evaluated at declaration time and the result is stored in the GString object.

By the time the GString value is converted into a java.lang.String (by calling its toString method or casting it to a string), each value gets written 14 to the string. Because the logic of how to write a value can be elaborate for certain types (most notably closures), this behavior can be used in advanced ways that make the evaluation of such placeholders appear to be lazy.

```
// Eager
def value=1
def gstring="$value"
value=2
assert gstring=="1"

//Lazy
def value=1
def gstring="${-> value}"
value=2
assert gstring=="2"
```

Extra Groovy Capabillity in String

A miscellany of string operations

```
String greeting = 'Hello Groovy!'

assert greeting.startsWith('Hello')

assert greeting.getAt(0) == 'H'

assert greeting[0] == 'H'

assert greeting.indexOf('Groovy') >= 0

assert greeting.contains('Groovy')

assert greeting[6..11] == 'Groovy'

assert 'Hi' + greeting - 'Hello' == 'Hi Groovy!'

assert greeting.count('o') == 3

assert 'x'.padLeft(3) == '  x'

assert 'x'.padRight(3,'_') == 'x__'

assert 'x'.center(3) == ' x '

assert 'x' * 3 == 'xxx'

def greeting = 'Hello'

greeting <<= ' Groovy'
assert greeting instanceof java.lang.StringBuffer

greeting << '!'
assert greeting.toString() == 'Hello Groovy!'

greeting[1..4] = 'i'
assert greeting.toString() == 'Hi Groovy!'
```

NOTE	Although the expression stringRef << string returns a StringBuffer , note that StringBuffer isn't automatically assigned to the stringRef . When used on a String , it needs explicit assignment; on StringBuffer it doesn't. With a StringBuffer , the data in the existing object is changed with a String you can't change the existing data, so you have to return a new object instead. You might also note that a greeting was explicitly typed. It's effectively of type Object and can reference both String and StringBuffer values.

Regular expressions

Regular expressions are prominent in scripting languages and have also been avail- able in the Java library since JDK 1.4.

Groovy relies on Java's regex (regular expression) support and adds three operators for convenience:

- The regex find operator, `=~`
- The regex match operator, `==~`
- The regex pattern operator, `~string`

```
import java.util.regex.Matcher
import java.util.regex.Pattern

def pattern = ~/\d+/
assert pattern instanceof Pattern

def matcher = 123 =~/\d+/
assert matcher instanceof Matcher

def result = 123==~/\d+/
assert result instanceof Boolean

assert result == matcher.matches()
```

Regular expression GStrings

```
def reference = "hello"
assert reference == /$reference/
```

TIP	Sometimes the slashy syntax interferes with other valid Groovy expressions such as line comments or numerical expressions with multiple slashes for divi- sion. When in doubt, put parentheses around your pattern like (/pattern/). Parentheses force the parser to interpret the content as an expression.

Table 7. Regular expression symbols

Symbol	Meaning
.	Any character
^	Start of line (or start of document, when in single-line mode)
$	End of line (or end of document, when in single-line mode)
\d	Digit character
\D	Any character except digits
\s	Whitespace character
\S	Any character except whitespace
\w	Word character
\W	Any character except word characters
\b	Word boundary
()	Grouping
(x | y)	x or y , as in (Groovy,Java,Ruby)
\1	Backmatch to group one; for example, find doubled characters with (.)\1
x *	Zero or more occurrences of x
x	One or more occurrences of x
x ?	Zero or one occurrence of x
x { m , n }	At least m and at most n occurrences of x
x { m }	Exactly m occurrences of x
[a-f]	Character class containing the characters a , b , c , d , e , f
[^a]	Character class containing any character except a
(?is:x)	Switches mode when evaluating x ; i turns on ignoreCase , s means single- line mode

TIP Symbols tend to have the same first letter as what they represent; for example, digit, space, word, and boundary. Uppercase symbols define the com- plement; think of them as a warning sign for no.

```
def twister = 'she sells sea shells at the sea shore of seychelles'
// twister must contain a substring of size 3
// that starts with s and ends with a
assert twister =~ /s.a/

def finder = (twister =~ /s.a/)
assert finder instanceof java.util.regex.Matcher

// twister must contain only words delimited by single spaces
assert twister ==~ /(\w+ \w+)*/

def WORD = /\w+/
matches = (twister ==~ /($WORD $WORD)*/)
assert matches instanceof java.lang.Boolean
assert !(twister ==~ /s.e/)

def wordsByX = twister.replaceAll(WORD, 'x')
assert wordsByX == 'x x x x x x x x x'

def words = twister.split(/ /)
assert words.size() == 10
assert words[0] == 'she'
```

TIP

To remember the difference between the =~ find operator and the ==~ match operator (it looks like a burning match), recall that match is more restrictive, because the pattern needs to cover the whole string. The demanded coverage is “longer” just like the operator itself.

What is a match?

A match is the occurrence of a regular expression pattern in a string.

It’s therefore a string: a substring of the original string. When the pattern contains groupings like in /begin(.*?)end/, you need to know more information: not just the string matching the whole pattern, but also what part of that string matched each group.

Therefore,the match becomes a list of strings, containing the whole match at position 0 with group matches being available as match[n] where n is group number n. Groups are numbered by the sequence of their opening parentheses.

Working on each match of a pattern

```
def myFairStringy = 'The rain in Spain stays mainly in the plain!'
// words that end with 'ain': \b\w*ain\b
def wordEnding = /\w*ain/
def rhyme = /\b$wordEnding\b/
def found = ''
myFairStringy.eachMatch(rhyme) { match ->
found += match + ' '
}
assert found == 'rain Spain plain '
found = ''
(myFairStringy =~ rhyme).each { match ->
found += match + ' '
}
assert found == 'rain Spain plain '

def cloze = myFairStringy.replaceAll(rhyme){ it-'ain'+'___' }
assert cloze == 'The r___ in Sp___ stays mainly in the pl___!'
```

The GDK enhances the Matcher class with simplified array-like access to this information.

In Groovy, you can think about a matcher as if it was a list of all its matches.

matches all nonwhitespace characters

```
def matcher = 'a b c' =~ /\S/
assert matcher[0] == 'a'
assert matcher[1..2] == ['b','c']
assert matcher.size() == 3
```

This use case comes with an interesting variant that uses Groovy's parallel assignment feature that allows you to directly assign each match to its own reference.

parallel assignment

```
def (a,b,c) = 'a b c' =~ /\S/
assert a == 'a'
assert b == 'b'
assert c == 'c'
```

Example 2. groupings in the match.

If the pattern contains parentheses to define groups, then the result of asking for a particular match is an array of strings rather than a single one: the same behavior as we mentioned for `eachMatch` . Again, the first result (at index 0) is the match for the whole pattern. Consider this example, where each match finds pairs of strings that are separated by a colon. For later processing, the match is split into two groups, for the left and the right string:

```
def matcher = 'a:1 b:2 c:3' =~ /(\S+):(\S+)/
assert matcher.hasGroup()
assert matcher[0] == ['a:1', 'a', '1'] // 1st match
assert matcher[1][2] == '2' // 2nd match, 2nd group
```

This also applies to the matcher's `each` method

```
def matcher = 'a:1 b:2 c:3' =~ /(\S+):(\S+)/
matcher.each { full, key, value ->
    assert full.size() == 3
    assert key.size() == 1 // a,b,c
    assert value.size() == 1 // 1,2,3
}
```

IMPORTANT	Groovy internally stores the most recently used matcher (per thread). It can be retrieved with the static property `Matcher.lastMatcher` .

You can also set the index property of a matcher to make it look at the respective match with `matcher.index = x` . Both can be useful in some exotic corner cases.

Patterns and performance

The rationale behind this construction is that patterns are internally backed by a finite-state machine that does all the high-performance magic.

This machine is compiled when the pattern object is created.

The more complicated the pattern, the longer the creation takes. In contrast, the matching process as performed by the machine is extremely fast.

Increasing performance with pattern reuse

```
def twister = 'she sells sea shells at the sea shore of seychelles'
// some more complicated regex:
// word that starts and ends with same letter
def regex = /\b(\w)\w*\1\b/
def many = 100 * 1000

start = System.nanoTime()
many.times{
    twister =~ regex
}
timeImplicit = System.nanoTime() - start
start = System.nanoTime()
pattern = ~regex
many.times{
    pattern.matcher(twister)
}
timePredef = System.nanoTime() - start
assert timeImplicit > timePredef * 2
```

NOTE To find words that start and end with the same character, the `\1` backmatch is used to refer to that character.

Patterns for classification

The Pattern object, as returned from the pattern operator, implements an `isCase(String)` method that's equivalent to a full match of that pattern with the string. This classification method is a prerequisite for using patterns conveniently with the in operator, the `grep` method, and `in` switch cases.

```
def fourLetters = ~/\w{4}/

assert fourLetters.isCase('work')

assert 'love' in fourLetters

switch('beer'){
    case fourLetters: assert true; break
    default: assert false
}
beasts = ['bear','wolf','tiger','regex']
assert beasts.grep(fourLetters) == ['bear','wolf']
```

TIP Classifications read nicely with in , switch , and grep . It's rare to call classifier.isCase(candidate) directly, but when you see such a call, it's easiest to read it from right to left: "candidate is a case of classifier."

Numbers

Table 8. Numerical coercion

+ - *	B	S	I	C	L	BI	BD	F	D
Byte	I	I	I	I	L	BI	BD	D	D
Short	I	I	I	I	L	BI	BD	D	D
Integer	I	I	I	I	L	BI	BD	D	D
Character	I	I	I	I	L	BI	BD	D	D
Long	L	L	L	L	L	BI	BD	D	D
BigInteger	BI	BI	BI	BI	BI	BI	BD	D	D
BigDecimal	BD	BD	BD	BD	BD	BD	BD	D	D
Float	D	D	D	D	D	D	D	D	D
Double	D	D	D	D	D	D	D	D	D

GDK methods for numbers

```
assert 1 == (-1).abs()
assert 2 == 2.5.toInteger() // conversion
assert 2 == 2.5 as Integer // enforced coercion
assert 2 == (int) 2.5 // cast
assert 3 == 2.5f.round()
assert 3.142 == Math.PI.round(3)
assert 4 == 4.5f.trunc()
assert 2.718 == Math.E.trunc(3)
assert '2.718'.isNumber() // String methods
assert 5 == '5'.toInteger()
assert 5 == '5' as Integer
assert 53 == (int) '5' // gotcha!
assert '6 times' == 6 + ' times' // Number + String
```

WARNING

Don’t cast strings to numbers! In Groovy, you can cast a string of length 1 directly to a char . But char and int are essentially the same thing on the Java platform. This leads to the gotcha where *5* is cast to its Unicode value 53 . Instead, use the type conversion methods.

GDK methods on numbers

```
def store = ''
10.times{
    store += 'x'
}
assert store == 'xxxxxxxxxx'

store = ''
1.upto(5) { number ->
    store += number
}
assert store == '12345'

store = ''
2.downto(-2) { number ->
    store += number + ' '
}
assert store == '2 1 0 -1 -2 '

store = ''
0.step(0.5, 0.1){ number ->
    store += number + ' '
}
assert store == '0 0.1 0.2 0.3 0.4 '
```

Collective Groovy datatypes

Working with ranges

Think about how often you've written a loop like this

```
for (int i=0; i<upperBound; i++){
// do something with i
}
```

Next, consider how often you've written a conditional like this:

```
if (x >= 0 && x <= upperBound) {
// do something with x
}
```

Ranges

are specified using the double-dot range operator `( .. )` between the left and right bounds.

This operator has a low precedence, so you often need to enclose the declaration in parentheses. Ranges can also be declared using their respective constructors.

The `..<` range operator specifies a half-exclusive range—that is, the value on the right isn't part of the range:

```
left..right
(left..right)
(left..<right)
```

Ranges usually have a lower left bound and a higher right bound. When this is switched it's called a ***reverse*** range.

```
assert (0..10).contains(0)
assert (0..<10).contains(9)

def a = 0..10
assert a instanceof Range
assert a.contains(5)

a = new IntRange(0,10)
assert a.contains(5)

assert (0.0..1.0).contains(1.0)
assert (0.0..1.0).containsWithinBounds(0.5)

def today = new Date()
def yesterday = today - 1
assert (yesterday..today).size() == 2

assert ('a'..'c').contains('b')
```

TIP	You can walk through a range with the each method

Ranges are objects

Because every range is an object, you can pass a range around and call its methods. The most prominent methods are `each` , which executes a specified closure for each element in the range, and `contains` , which specifies whether or not a value is within a range.

Ranges are objects

```
def result = ''
(5..9).each { element ->
        result += element
}
assert result == '56789'

assert 5 in 0..10
assert (0..10).isCase(5)

def age = 36
switch(age){
      case 16..20 : insuranceRate = 0.05 ; break
      case 21..50 : insuranceRate = 0.06 ; break
      case 51..65 : insuranceRate = 0.07 ; break
      default: throw new IllegalArgumentException()
}
assert insuranceRate == 0.06

def ages = [20, 36, 42, 56]
def midage = 21..50
assert ages.grep(midage) == [36, 42]
```

Working with lists

Specifying lists

```
List myList = [1, 2, 3]
assert myList.size() == 3
assert myList[0]== 1

assert myList instanceof ArrayList
List emptyList = []
assert emptyList.size() == 0
List longList = (0..1000).toList()
assert longList[555] == 555

List explicitList = new ArrayList()
explicitList.addAll(myList)
assert explicitList.size() == 3
explicitList[0] = 10
assert explicitList[0] == 10

explicitList = new LinkedList(myList)
assert explicitList.size() == 3
explicitList[0] = 10
assert explicitList[0] == 10
```

Using list operators

lists override `getAt` and `putAt` methods to implement the subscript operator.

Accessing parts of a list with an overloaded subscript operator

```
myList = ['a','b','c','d','e','f']

assert myList[0..2] == ['a','b','c']
assert myList[0,2,4] == ['a','c','e']

myList[0..2] = ['x','y','z']
assert myList == ['x','y','z','d','e','f']

myList[3..5] = []
assert myList == ['x','y','z']
myList[1..1] = [0, 1, 2]
assert myList == ['x', 0, 1, 2, 'z']
```

Subscript assignments with ranges

don't need to be of identical size. When the assigned list of values is smaller than the range or even empty, the list shrinks . When the assigned list of values is bigger, the list grows

Adding and removing items

```
def myList = []
myList += 'a'
assert myList == ['a']

myList += ['b','c']
assert myList == ['a','b','c']

def myList = []
myList << 'a' << 'b'
assert myList == ['a','b']

assert myList - ['b'] == ['a']
assert myList * 2 == ['a','b','a','b']
```

Control structures

Groovy lists are more than flexible storage places.

They also play a major role in organizing the execution flow of Groovy programs.

```
myList = ['a', 'b', 'c']
assert myList.isCase('a')
assert 'b' in myList

def candidate = 'c'
switch(candidate){
    case myList : assert true; break
    default: assert false
}
assert ['x','a','z'].grep(myList) == ['a']

myList = []
if (myList) assert false

// Lists can be iterated with a 'for' loop
def expr = ''
for (i in [1,'*',5]){
    expr += i
}
assert expr == '1*5'
```

Using list methods

```
assert [1,[2,3]].flatten() == [1,2,3]
assert [1,2,3].intersect([4,3,1])== [3,1]
assert [1,2,3].disjoint([4,5,6])

list =[1,2,3]
popped = list.pop()
assert popped == 3
assert list == [1,2]

assert [1,2].reverse() == [2,1]
assert [3,1,2].sort() == [1,2,3]

def list = [ [1,0], [0,1,2] ]
list = list.sort { a,b -> a[0] <=> b[0] }
assert list == [ [0,1,2], [1,0] ]

list = list.sort { item -> item.size() }
assert list == [ [1,0], [0,1,2] ]

list = ['a','b','c']
list.remove(2)
assert list == ['a','b']
list.remove('b')
assert list == ['a']
list = ['a','b','b','c']
list.removeAll(['b','c'])
assert list == ['a']

def doubled = [1,2,3].collect{ item ->
    item*2
}
assert doubled == [2,4,6]
def odd = [1,2,3].findAll{ item ->
    item % 2 == 1
}
assert odd == [1,3]

def x = [1,1,1]
assert [1] == new HashSet(x).toList()
assert [1] == x.unique()

def x = [1,null,1]
assert [1,1] == x.findAll{it != null}
assert [1,1] == x.grep{it}
```

Accessing list content

```
// querying
def list = [1, 2, 3]
```

```
assert list.first()== 1
assert list.head()== 1
assert list.tail()== [2, 3]

assert list.last()=3
assert list.count(2)=1
assert list.max()=3
assert list.min()=1

def even = list.find { item ->
    item % 2 == 0
}
assert even == 2

assert list.every { item -> item < 5 }
assert list.any { item -> item < 2 }

// iterating

def store = ''
list.each { item ->
    store += item
}

assert store == '123'
store = ''
list.reverseEach { item ->
    store += item
}
assert store == '321'

store = ''
list.eachWithIndex { item, index ->
    store += "$index:$item "
}
assert store == '0:1 1:2 2:3 '

// accumulating
assert list.join('-') == '1-2-3'
result = list.inject(0) { clinks, guests ->
    clinks + guests
}
assert result == 0 + 1 + 2 + 3
assert list.sum() == 6

factorial = list.inject(1) { fac, item ->
    fac * item
}
assert factorial == 1 * 1 * 2 * 3
```

Working with maps

```
def myMap = [a:1, b:2, c:3]
assert myMap instanceof LinkedHashMap
assert myMap.size() == 3
assert myMap['a'] == 1

def emptyMap = [:]
assert emptyMap.size() == 0
def explicitMap = new TreeMap()
explicitMap.putAll(myMap)
assert explicitMap['a'] == 1

def composed = [x:'y', *:myMap] //Spread operator
assert composed == [x:'y', a:1, b:2, c:3]

assert ['a':1] == [a:1]

def x = 'a'
assert ['x':1] == [x:1]
assert ['a':1] == [(x):1]
```

Using map operators

```
def myMap = [a:1, b:2, c:3]

// Retrieves existing elements
assert myMap['a'] == 1
assert myMap.a == 1
assert myMap.get('a') == 1
assert myMap.get('a',0) == 1

// Attempts to retrieve missing elements
assert myMap['d'] == null
assert myMap.d == null
assert myMap.get('d') == null

// Default value
assert myMap.get('d',0) == 0
assert myMap.d == 0

//Single putAt
myMap['d'] = 1
assert myMap.d == 1
myMap.d = 2
assert myMap.d == 2

//Equals
def myMap = [a:1, b:2, c:3]
def other = [b:2, c:3, a:1]
assert myMap == other

// JDK Methods
assert !myMap.isEmpty()
assert myMap.size() == 3
assert myMap.containsKey('a')
assert myMap.containsValue(1)
assert myMap.entrySet() instanceof Collection

//GDK methods
assert myMap.any {entry -> entry.value > 2 }
assert myMap.every {entry -> entry.key < 'd'}
assert myMap.keySet() == ['a','b','c'] as Set

//Equals List
assert myMap.values().toList() == [1, 2, 3]
```

Assignments to maps can be done using the subscript operator or via the dot-key syn- tax. If the key in the dot-key syntax contains special characters, it can be put into string markers, like so:

```
myMap = ['a.b':1]
assert myMap.'a.b' == 1
```

Just writing `myMap.a.b` wouldn't work here—that would be the equivalent of calling `myMap.getA().getB()`.

Iterating over maps (GDK)

```
def myMap = [a:1, b:2, c:3]

//Iterates over entries
def store = ''
myMap.each { entry ->
    store += entry.key
    store += entry.value
}
assert store == 'a1b2c3'

//Iterates over keys and values
def store = ''
myMap.each { key,value ->
    store += key
    store += value
}
assert store == 'a1b2c3'

//Iterates over just keys
store = ''
for (key in myMap.keySet()) {
    store += key
}
assert store == 'abc'

//Iterates over just values
store = ''
for (value in myMap.values()) {
    store += value
}
assert store == '123'
```

Changing map content and building new objects from it

```
def myMap = [a:1, b:2, c:3]
myMap.clear()
assert myMap.isEmpty()

myMap = [a:1, b:2, c:3]
myMap.remove('a')
assert myMap.size() == 2
assert [a:1] + [b:2] == [a:1, b:2]

myMap = [a:1, b:2, c:3]
def abMap = myMap.subMap(['a', 'b'])
assert abMap.size() == 2

abMap = myMap.findAll { entry -> entry.value < 3 }
assert abMap.size() == 2
assert abMap.a == 1

def found = myMap.find { entry -> entry.value < 2 }
assert found.key == 'a'
assert found.value == 1

def doubled = myMap.collect { entry -> entry.value *= 2 }
assert doubled instanceof List
assert doubled.every  { item -> item % 2 == 0 }

def addTo = []
myMap.collect(addTo) { entry -> entry.value *= 2 }
assert addTo instanceof List
assert addTo.every { item -> item % 2 == 0 }
```

Notes on Groovy collections

One of the typical peculiarities of the Java collections is that you shouldn't try to structurally change one while iterating through it. A structural change is one that adds an entry, removes an entry, or changes the sequence of entries when the collection is sequence-aware. This applies even when iterating through a view onto the collection, such as using list[range] .

Understanding concurrent modification

If you fail to meet this constraint, you'll see a ConcurrentModificationException . For example, you cannot remove all elements from a list by iterating through it and removing the first element at each step:

```
def list = [1, 2, 3, 4]
list.each{ list.remove(0) }
// throws ConcurrentModificationException !!
```

In this case, the correct solution is to use the clear method. The Java Collections API has lots of such specialized methods. When searching for alternatives, consider collect , addAll , removeAll , findAll , and grep .

Distinguishing between copy and modify semantics

Groovy tries to adapt to Java and follow the heuristics that you can spot when looking through the Java Collections API :

- Methods that modify the receiver typically don't return a collection. Examples: add , addAll , remove , removeAll , and retainAll . Counterexample: sort .
- Methods that return a collection typically don't modify the receiver. Examples: grep , findAll , and collect . Counterexample: sort (though we recommend using toSorted in that case). And yes, sort is a counterexample for both, because it returns a collection and modifies the receiver.
- Methods that modify the receiver have imperative names. They sound like there could be an exclamation mark behind them. (Indeed, this is Ruby's naming convention for such methods.) Examples: add , addAll , remove , removeAll , retainAll , and sort . Counterexamples: collect , grep , and findAll , which are imperative but don't modify the receiver and return a modified copy.

The preceding rules can be mapped to operators, by applying them to the names of their method counterparts: `<< leftShift` is imperative and modifies the receiver (on lists, unfortunately not on strings—doing so would break Java's invariant of strings being immutable); `plus` isn't imperative and returns a copy.

NOTE The convention in Groovy is that any method that implements an arithmetic operator (`plus, minus, multiply, divide`) doesn't modify the receiver but returns a copy.

Closures

A gentle introduction to closures

A Closure

is a piece of code wrapped up as an object. It acts like a method in that it can take parameters and return a value. It's a normal object in that you can pass a reference to it just as you can to any other object.

```
Closure envelope = { person -> new Letter(person).send() }
addressBook.each (envelope)

addressBook.each { new Letter(it).send() }
```

Using iterators

```
// Java 5
for (ItemType item : list) {
    // do something with item
}

// Java 8 with lambda
list.stream().forEach( (item) -> {
    // do something with item
} );
```

```
// Groovy object iteration
list.each { item -> /* do something with item */ }

// Groovy closures with Java 8
list.stream().forEach { println it}
```

Handling resources with a protocol

How many times have you seen code that opens a stream but calls close at the end of the method, overlooking the fact that the close statement may never be reached when an exception occurs while processing? So, it needs to be protected with a `try-catch` block. No—wait—that should be `try-finally` , or should it? And inside the finally block, close can throw another exception that needs to be handled. There are too many details to remember, and so resource handling is often implemented incorrectly. With Groovy's closure support, you can put that logic in one place and use it like this:

```
new File('myfile.txt').eachLine { println it }
```

Let's see what resource-handling solutions Java provides and why they're not used often, and

then we'll show you the corresponding Groovy solutions.

A COMMON JAVA APPROACH : USE INNER CLASSES

Java's limitations get in the way too much to make it an elegant solution. The fol- lowing example uses a Resource that it gets from a ResourceHandler , which is respon- sible for its proper construction and destruction. Only the boldface code is needed for doing the job:

```
// Java
interface ResourceUser { // a @FunctionalInterface in Java 8
    void use(Resource resource)
}
resourceHandler.handle(new ResourceUser(){
    public void use (Resource resource) {
        resource.doSomething()
    }
});
```

```
// groovy
resourceHandler.handle { resource -> resource.doSomething() }
```

AN ALTERNATIVE JAVA APPROACH : THE TEMPLATE METHOD PATTERN

Another strategy to centralize resource handling in Java is to do it in a superclass and let the resource-using code live in a subclass. This is the typical implementation of the Template Method (Gang of Four) pattern.

If there were only one interface that could be used for the purpose of passing logic around, like the imaginary `ResourceUser` interface from the previous example, then things wouldn't be too bad. But in Java, there's no such beast—that is, no single ResourceUser interface that serves all purposes. The signature of the callback method use needs to adapt to the purpose: the number and type of parameters, the number and type of declared exceptions, and the return type.

Declaring closures

Simple declaration

```
log = ''
(1..10).each{ counter -> log += counter }
assert log == '12345678910'
log = ''
(1..10).each{ log += it }
assert log == '12345678910'
```

TIP Think of the arrow as an indication that parameters are passed from the method on the left into the closure body on the right.

Using assignments for declaration

```
def printer = { line -> println line }
```

TIP Whenever you see the braces of a closure, think: `new Closure(){}` .

```
def Closure getPrinter() {
    return { line -> println line }
}
```

Referring to methods as closures

```
class SizeFilter {
    Integer limit
    boolean sizeUpTo(String value) {
        return value.size() <= limit
    }
}

SizeFilter filter6 = new SizeFilter(limit:6)
SizeFilter filter5 = new SizeFilter(limit:5)

Closure sizeUpTo6 = filter6.&sizeUpTo
def words = ['long string', 'medium', 'short', 'tiny']

assert 'medium' == words.find (sizeUpTo6)
assert 'short' == words.find (filter5.&sizeUpTo)
```

Multimethod

```
class MultiMethodSample {
    int mysteryMethod (String value) {
        return value.length()
    }
    int mysteryMethod (List list) {
        return list.size()
    }
    int mysteryMethod (int x, int y) {
        return x+y
    }
}

MultiMethodSample instance = new MultiMethodSample()
Closure multi = instance.&mysteryMethod
assert 10 == multi ('string arg')
assert 3 == multi (['list', 'of', 'values'])
assert 14 == multi (6, 8)
```

Comparing the available options

```
Map map = ['a':1, 'b':2]
map.each{ key, value -> map[key] = value * 2 }
assert map == ['a':2, 'b':4]

Closure doubler = {key, value -> map[key] = value * 2 }
map.each(doubler)
assert map == ['a':4, 'b':8]

def doubleMethod (entry){
    entry.value = entry.value * 2
}
doubler = this.&doubleMethod
map.each(doubler)
assert map == ['a':8, 'b':16]
```

Using closures

Calling a closure

Suppose you have a reference x pointing to a closure; you can call it with `x.call()` or simply `x()` . You've probably guessed that any arguments to the closure call go between the parentheses.

Calling Closure

```
def adder = { x, y -> return x+y }
assert adder(4, 3) == 7
assert adder.call(2, 6) == 8
```

Calling a closure from within a method body

```
def benchmark(int repeat, Closure worker) {
    def start = System.nanoTime()
    repeat.times { worker(it) }
    def stop = System.nanoTime()
    return stop - start
}
def slow = benchmark(10000) { (int) it / 2 }
def fast = benchmark(10000) { it.intdiv(2) }
assert fast * 2 < slow
```

More closure capabilities

The class `groovy.lang.Closure` is an ordinary class, albeit one with extraordinary power and extra language support. It has a number of methods available beyond `call` . We'll present the most important ones. Even though you'll usually just declare and call closures, it's nice to know there's extra power available when you need it.

REACTING ON THE PARAMETER COUNT OR TYPE

`Map's each` method. It passes either a `Map.Entry` object or key and value separately into a supplied closure, depending on whether the closure takes one or two arguments. The each method adapts its behavior depending on the number of arguments that the closure that it receives was built with. You can do this in your own methods by retrieving the expected parameter count (and types, if declared) by calling Closure 's `getMaximumNumberOfParameters` and `getParameterTypes` methods:

```
def numParams (Closure closure){
closure.getMaximumNumberOfParameters()
}
assert numParams { one -> } == 1
assert numParams { one, two -> } == 2
def paramTypes (Closure closure){
    closure.getParameterTypes()
}
assert paramTypes { String s -> } == [String]
assert paramTypes { Number n, Date d -> } == [Number, Date]
```

HOW TO CURRY FAVOR WITH A CLOSURE

> **Currying** is a technique invented by Moses Schönfinkel and Gottlob Frege, and named after the logician Haskell Brooks Curry (1900–1982), a pioneer in functional program- ming.

```
The basic idea is to take a function with multiple parameters and transform it into a
function with fewer parameters by fixing some of the values.
```

In Groovy, Closure 's curry method returns a clone of the current closure, having bound one or more parameters to a given value. Parameters are bound to curry 's arguments from left to right. The following listing gives an implementation.

```
def mult = { x, y -> return x * y }
def twoTimes = mult.curry(2)
assert twoTimes(5) == 10

def configurator = { format, filter, line ->
    filter(line) ? format(line) : null
}
def appender = { config, append, line ->
    def out = config(line)
    if (out) append(out)
}

def dateFormatter = { line -> "${new Date()}: $line" }
def debugFilter = { line -> line.contains('debug') }
def consoleAppender = { line -> println line }
def myConf = configurator.curry(dateFormatter, debugFilter)
def myLog = appender.curry(myConf, consoleAppender)

myLog('here is some debug message')
myLog('this will not be printed')
```

CLOSURE COMPOSITION

```
def fourTimes = twoTimes >> twoTimes
def eightTimes = twoTimes << fourTimes
assert eightTimes(1) == twoTimes(fourTimes(1))
```

MEMOIZATION

When you have a closure that's called much too often with the same arguments or the execution of the closure is very expensive, then you may want to cache the results.

```
def fib
fib = { it < 2 ? 1 : fib(it-1) + fib(it-2) }
fib = fib.memoize()
assert fib(40) == 165_580_141
```

JUMPING ON THE TRAMPOLINE

Our fib closure included a recursive call. Such calls can easily lead to a stack overflow, and because the JVM has no tail call elimination, this is difficult to overcome. Groovy offers two approaches. The first follows the trampoline 9 algorithm, and we'll use the respective method for very inefficiently (but functionally) finding the last element of anything that has at least a size, a head, and a tail

```
def last
last = { it.size() == 1 ? it.head() : last.trampoline(it.tail()) }
last = last.trampoline()
assert last(0..10_000) == 10_000
```

Without trampoline , the code goes into a stack overflow before 2,000 iterations.

CLASSIFICATION VIA THE IS CASE METHOD

Closures implement the isCase method to make them work as classifiers in grep and switch .

```
def odd = { it % 2 == 1}
assert [1,2,3].grep(odd) == [1, 3]
switch(10) {
    case odd : assert false
}
if (2 in odd) assert false
```

The `asWriteable` method returns a clone of the current closure that has an addi- tional writeTo(Writer) method to write the result of a closure call directly into the given Writer .

Understanding closure scope

you cannot directly set this to a different value, you can set a so-called delegate , which will be used when resolving free variables. Per default, the delegate refers to the owner .

```
class Mother {
    def prop = 'prop'

    def method() { 'method' }

    Closure birth(param) {
        def local = 'local'
        def closure = {
            [this, prop, method(), local, param]
        }
        return closure
    }
}

Mother julia = new Mother()
def closure = julia.birth('param')

def context = closure.call()
assert context[0] == julia
assert context[1, 2] == ['prop', 'method']
assert context[3, 4] == ['local', 'param']

assert closure.thisObject == julia
assert closure.owner == julia

assert closure.delegate == julia
assert closure.resolveStrategy == Closure.OWNER_FIRST
```

The GDK with method does exactly that: executing a closure by first setting the delegate to the receiver of the with method:

```
def map = [:]
map.with { //delegate is now map
     a = 1
     b = 2
}
assert map == [a:1, b:2]
```

NOTE `ResolveStrategy` possible options `OWNER_ONLY` , `OWNER_FIRST (default)` , `DELEGATE_ONLY` , `DELEGATE_FIRST` , or `SELF_ONLY` .

The classic accumulator test

Paul Graham first proposed this test in his excellent article “Revenge of the Nerds"

> We want to write a function that generates accumulators—a function that takes a number `n` , and returns a function that takes another number `i` and returns `n` incremented by `i` .

lisp

```
(defun foo (n)
(lambda (i) (incf n i)))
```

groovy

```
def foo(n) {
    return {n += it}
}
```

Control Structures

Groovy truth

Evaluating Boolean tests

The expression of a Boolean test can be of any (nonvoid) type. It can apply to any object. Groovy decides whether to consider the expression as being true or false by applying the rules shown in table

Table 9. Sequence of rules used to evaluate a Boolean test

Runtime type	**Evaluation criterion required for truth**
Boolean	Corresponding Boolean value is true
Matcher	Matcher has a match
Collection	Collection is nonempty
Map	Map is nonempty
String , GString	String is nonempty
Number , Character	Value is nonzero
None of the above	Object reference is non-null

```
//Boolean values are trivial
assert true
assert !false

//Matchers must match
assert ('a' =~ /./)
assert !('a' =~ /b/)

//Collections must be nonempty
assert [1]
assert ![]

//Iterators must have next element
Iterator iter = [1].iterator()
assert iter
iter.next()
assert !iter

//Maps must be nonempty
assert ['a':1]
assert ![:]

//Strings must be nonempty
assert 'a'
assert !''

//Numbers (any type) must be nonzero
assert 1
assert 1.1
assert 1.2f
assert 1.3g
assert 2L
assert 3G
assert !0

//Objects must be non-null
assert ! null
assert new Object()

//Custom truth
class AlwaysFalse {
    boolean asBoolean() { false }
}
assert ! new AlwaysFalse()
```

if statement

```
if (true) assert true
else assert false

if (1) assert true
else if (2) assert true
else assert false

if ('nonEmpty') assert true
else if (['x']) assert true
else
    if (0) assert false
    else if ([]) assert false
```

The conditional ?: operator and Elvis

Groovy also supports the ternary conditional operator ?: for small inline tests.

```
def result = (1==1) ? 'ok' : 'failed'
assert result == 'ok'
result = 'some string' ? 10 : ['x']
assert result == 10

// default value
def value = argument ?: standard
```

The switch statement and the in operator

```
switch (10) {
    case 0 : assert false ; break;
    case 0..9 : assert false ; break;
    case [8,9,11] : assert false ; break;
    case Float : assert false ; break;
    case {it%3 == 0}: assert false ; break;
    case ~/../ : assert true ; break;
    default : assert false
}
```

Table 10. Standard implementations of isCase for switch , grep , and in

Class	**a.isCase(b) implemented as**
Object	a.equals(b)
Class	a.isInstance(b)
Collection	a.contains(b)
Range	a.contains(b)
Pattern	a.matcher(b.toString()).matches()

Class	a.isCase(b) implemented as
String	(a==null && b==null)
	a.equals(b)
Closure	a.call(b)

NOTE The isCase method is also used with grep on collections such that `collection.grep(classifier)` returns a collection of all items that are a case of that classifier.

THE IN OPERATOR

The isCase logic is actually used three times: for switch cases for grep classification and for the in operator as used for conditionals like the following assertion:

```
def okValues = [1, 2, 3]
def value = 2
assert value in okValues
```

TIP It's possible to overload the isCase method to support different kinds of classification logic depending on the candidate type. If you provide both methods, `isCase(String candidate)` and `isCase(Integer candidate)` , then `switch ('1')` can behave differently than switch(1) with your object as the classifier.

PRODUCING INFORMATIVE FAILURE MESSAGES

```
a = 1
assert a==1

//with messages
input = new File('no such file')
assert input.exists() , "cannot find '$input.name'"
assert input.canRead() , "cannot read '$input.canonicalPath'"
println input.text
```

ENSURE CODE WITH INLINE UNIT TESTS

```
def host = /\/\/([a-zA-Z0-9-]+(\.[a-zA-Z0-9-])*?)(:|\/)/
assertHost 'http://a.b.c:8080/bla', host ,'a.b.c'
assertHost 'http://a.b.c/bla', host ,'a.b.c'
assertHost 'http://127.0.0.1:8080/bla', host ,'127.0.0.1'
assertHost 'http://t-online.de/bla', host ,'t-online.de'
assertHost 'http://T-online.de/bla', host ,'T-online.de'

def assertHost (candidate, regex, expected){
    candidate.eachMatch(regex){ assert it[1] == expected
}
```

Looping

Looping with while

```
def list = [1,2,3]
while (list) {
    list.remove(0)
}
assert list == []
while (list.size() < 3) list << list.size() + 1
assert list == [1,2,3]
```

NOTE there are no `do {} while(condition)` or `repeat {} until (condition)` loops in Groovy.

Looping with for

```
def store = ''
for (String s in 'a'..'c') store += s
assert store == 'abc'
store = ''
for (i in [1, 2, 3]) {
    store += i
}
assert store == '123'

def myString = 'Old school Java'
store = ''
for (int i=0; i < myString.size(); i++) {
    store += myString[i]
}
assert store == myString
myString = 'Java range index'
store = ''
for (int i : 0 ..< myString.size()) {
```

```
    store += myString[i]
}
assert store == myString

myString = 'Groovy range index'
store = ''
for (i in 0 ..< myString.size()) {
    store += myString[i]
}
assert store == myString

myString = 'Java string Iterable'
store = ''
for (String s : myString) {
    store += s
}
assert store == myString
myString = 'Groovy iterator'
store = ''
for (s in myString) {
    store += s
}
assert store == myString


def file = new File('myFileName.txt')
for (line in file) println line

def matcher = '12xy3'=~/\d/
for (match in matcher) println match


for (x in 0..9) { println x }
```

Normal termination: return/break/continue

```
def a = 1
while (true) {
    a++
    break
}
assert a == 2

for (i in 0..10) {
    if (i == 0) continue
    a++
    if (i > 0) break
}
assert a == 3
```

Exceptions: throw/try-catch-finally

```
def myMethod() {
    throw new IllegalArgumentException()
}
def log = []
try {
    myMethod()
} catch (Exception e) {
    log << e.toString()
} finally {
    log << 'finally'
}
assert log.size() == 2


try {
if (Math.random() < 0.5) 1 / 0
    else null.hashCode()
} catch (ArithmeticException | NullPointerException exception) {
    println exception.class.name
}
```

NOTE	Java 7 introduced a try-with-resources mechanism. At the time of writing, Groovy doesn't support that syntax. try-with-resources isn't needed in Groovy, where we have full closure support.

Object Oriented

Defining classes and scripts

Defining fields and local variables

DECLARING VARIABLES

Groovy uses Java's modifiers—the keywords private , protected , and public for modifying visibility 2 ; final for disallowing reassignment; and static to denote class variables. A nonstatic field is also known as an instance variable. These modifiers all have the same meaning as in Java.

The default visibility for fields has a special meaning in Groovy. When no visibility modifier is attached to a field declaration, a property is generated for the respective name.

Defining the type of a variable is optional. But the identifier must not stand alone in the declaration. When no type and modifier are given, the def keyword must be used as a replacement, effectively indicating that the field or variable can be assigned an object of any type at runtime.

```
class ClassWithTypedAndUntypedFieldsAndProperties {
    public fieldWithModifier
    String typedField
    def untypedField
    protected field1, field2, field3
    private assignedField = new Date ( )
    static classField
    public static final String CONSTA = 'a', CONSTB = 'b'

    def someMethod() {
        def localUntypedMethodVar = 1
        int localTypedMethodVar = 1
        def localVarWithoutAssignment, andAnotherOne
    }
}

def localvar = 1
boundvar1 = 1

def someMethod() {
    def localMethodVar = 1
    boundvar2 = 1
}

someMethod()
```

NOTE Java's default package-wide visibility is supported via the `@PackageScope` annotation.

Assignments to typed references must conform to the type.

Groovy provides autoboxing and coercion when it makes sense. All other cases are type-breaking assignments and lead to a ClassCastException at runtime,

```
final String PI = '3.14'
assert PI.class.name == 'java.lang.String'
assert PI.size() == 4
GroovyAssert.shouldFail(ClassCastException){
    Float areaOfCircleRadiusOne = PI
}
```

REFERENCING AND DEREFERENCING FIELDS

In addition to referring to fields by name with the obj.fieldname syntax, they can also be referenced with the subscript operator. This allows you to access fields using a dynamically determined name.

Referencing fields with the subscript operator

```
class Counter {
    public count = 0
}

def counter = new Counter()
counter.count = 1
assert counter.count == 1
def fieldName = 'count'
counter[fieldName] = 2
assert counter['count'] == 2
```

Can I override the subscript operator?

Sure you can, and you'll extend but not override the general field-access mechanism that way. But you can do even better and extend the field-access operator!

Extending the general field-access mechanism

```
class PretendFieldCounter {
    public count = 0

    Object get(String name) {
        return 'pretend value'
    }

    void set(String name, Object value) {
        count++
    }
}

def pretender = new PretendFieldCounter()
assert pretender.isNoField == 'pretend value'
assert pretender.count == 0
pretender.isNoFieldEither = 'just to increase counter'
assert pretender.count == 1
```

Methods and parameters

```
class ClassWithTypedAndUntypedMethods {
    static void main(args) {
        def some = new ClassWithTypedAndUntypedMethods()
        some.publicVoidMethod()
        assert 'hi' == some.publicUntypedMethod()
        assert 'ho' == some.publicTypedMethod()
        combinedMethod()
    }

    void publicVoidMethod() {}

    def publicUntypedMethod() {
        return 'hi'
    }

    String publicTypedMethod() {
        return 'ho'
    }

    private static final void combinedMethod() {}
}
```

```
//Java
public static void main (String[] args)
//groovy
static main (args)
```

NOTE The Java compiler fails on missing return statements when a return type is declared for the method. In Groovy, return statements are optional, there- fore it's impossible for the compiler to detect "accidentally" missing returns.

```
class ClassWithTypedAndUntypedMethodParams {
    static void main(args) {
        assert 'untyped' == method(1)
        assert 'typed' == method('whatever')
        assert 'two args' == method(1, 2)
    }

    static method(arg) {
        return 'untyped'
    }

    static method(String arg) {
        return 'typed'
    }

    static method(arg1, Number arg2) {
        return 'two args'
    }
}
```

Advanced parameter uses

```
class Summer {
    def sumWithDefaults(a, b, c = 0) {
        a + b + c
    }

    def sumWithList(List args) {
        args.inject(0) { sum, i -> sum += i }
    }

    def sumWithOptionals(a, b, Object[] optionals) {
        return a + b + sumWithList(optionals.toList())
    }

    def sumNamed(Map args) {
        ['a', 'b', 'c'].each { args.get(it, 0) }
        return args.a + args.b + args.c
    }

}

def summer = new Summer()
assert 2 == summer.sumWithDefaults(1, 1)
assert 3 == summer.sumWithDefaults(1, 1, 1)
assert 2 == summer.sumWithList([1, 1])
assert 3 == summer.sumWithList([1, 1, 1])
assert 2 == summer.sumWithOptionals(1, 1)
assert 3 == summer.sumWithOptionals(1, 1, 1)
assert 2 == summer.sumNamed(a: 1, b: 1)
assert 3 == summer.sumNamed(a: 1, b: 1, c: 1)
assert 1 == summer.sumNamed(c: 1)
```

NOTE There are more ways of implementing parameter lists of variable length. You can use varargs with the method(args...) or method(Type[] args) nota- tion or even hook into Groovy's method dispatch by overriding the invoke- Method(name, params[]) that every GroovyObject provides.

ADVANCED NAMING

```
objectReference.methodName()
objectReference.'method Name'()
```

NOTE Where there's a string, you can generally also use a GString. So how about obj."${var}"() ? Yes, this is also possible, and the GString will be resolved to determine the name of the method that's called on the object!

Safe dereferencing with the ?. operator

When a reference doesn't point to any specific object, its value is null . When calling a method or accessing a field on a null reference, a NullPointerException (NPE) is thrown. This is useful to protect code from working on undefined preconditions, but it can easily get in the way of "best-effort" code that should be executed for valid refer- ences and just be silent otherwise.

```
def map = [a: [b: [c: 1]]]
assert map.a.b.c == 1
if (map && map.a && map.a.x) {
    assert map.a.x.c == null
}
try {
    assert map.a.x.c == null
} catch (NullPointerException ignore) {
}
assert map?.a?.x?.c == null
```

Constructors

Objects are instantiated from their classes via constructors. If no constructor is given, an implicit constructor without arguments is supplied by the compiler. This appears to be exactly like in Java, but because this is Groovy, it should not be surprising that addi- tional features are available.

Positional Parameters

```
class VendorWithCtor {
    String name, product

    VendorWithCtor(name, product) {
        this.name = name
        this.product = product
    }
}

def first = new VendorWithCtor('Canoo','ULC')
def second = ['Canoo','ULC'] as VendorWithCtor
VendorWithCtor third = ['Canoo','ULC']
```

Named Parameters

```
class SimpleVendor {
    String name, product
}

new SimpleVendor()
new SimpleVendor(name: 'Canoo')
new SimpleVendor(product: 'ULC')
new SimpleVendor(name: 'Canoo', product: 'ULC')

def vendor = new SimpleVendor(name: 'Canoo')
assert 'Canoo' == vendor.name
```

Implicit Constructors

```
java.awt.Dimension area = [200, 100]
assert area.width == 200
assert area.height == 100
```

Organizing classes and scripts

Groovy classes are Java classes at the bytecode level, and consequently, Groovy objects are Java objects in memory. At the source-code level, Groovy class and object handling is for all practical purposes a superset of the Java syntax.

File to class relationship

The relationship between files and class declarations isn't as fixed as in Java. Groovy files can contain any number of public class declarations according to the following rules:

- If a Groovy file contains no class declaration, it's handled as a script; that is, it's transparently wrapped into a class of type Script . This automatically generated class has the same name as the source script filename 7 (without the extension). The content of the file is wrapped into a run method, and an additional main method is constructed for easily starting the script.
- If a Groovy file contains exactly one class declaration with the same name as the file (without the extension), then there's the same one-to-one relationship as in Java.
- A Groovy file may contain multiple class declarations of any visibility, and there's no enforced rule that any of them must match the filename. The groovyc compiler happily creates *.class files for all declared classes in such a file. If you wish to invoke your script directly—for example, using groovy on the command line or within an IDE —then the first class within your file should have a main method. 8
- A Groovy file may mix class declarations and scripting code. In this case, the scripting code will become the main class to be executed, so don't declare a class yourself having the same name as the source filename.

```
class Vendor {
    public String name
    public String product
    public Address address = new Address()
}

class Address {
    public String street, town, state
    public int zip
}

def canoo = new Vendor()
canoo.name = 'Canoo Engineering AG'
canoo.product = 'UltraLightClient (ULC)'
canoo.address.street = 'Kirschgartenst. 7'
canoo.address.zip = 4051
canoo.address.town = 'Basel'
canoo.address.state = 'Switzerland'
assert canoo.dump() =~ /ULC/
assert canoo.address.dump() =~ /Basel/
```

Vendor and Address are simple data storage classes. They're roughly equivalent to struct in C or record in Pascal.

Organizing classes in packages

Groovy follows Java's approach of organizing files in packages of hierarchical struc- ture. The package structure is used to find the corresponding class files in the filesys- tem's directories.

Because *.groovy source files aren't necessarily compiled to *.class files, there's also a need to look up *.groovy files. When doing so, the same strategy is used: the compiler looks for a Groovy class Vendor in the business package in the file busi- ness/Vendor.groovy .

CLASSPATH

The lookup has to start somewhere, and Java uses its classpath for this purpose. The classpath is a list of possible starting points for the lookup of *.class files. Groovy reuses the classpath for looking up *.groovy files.

When looking for a given class, if Groovy finds both a *.class and a *.groovy file, it uses whichever is newer; that is, it'll recompile source files into *.class files if they've changed since the previous class file was compiled.

PACKAGES

Exactly like in Java, Groovy classes must specify their package before the class defini- tion. When no package declaration is given, the default package is assumed.

```
package business

class Vendor {
    public String name
    public String product
    public Address address = new Address()
}

class Address {
    public String street, town, state
    public int zip
}
```

To reference Vendor in the business package, you can either use business.Vendor within the code or use import statements for abbreviation.

IMPORTS

Groovy follows Java's notion of allowing import statements before any class declaration to abbreviate class references.

NOTE	Please keep in mind that unlike in some other scripting languages, an import statement has nothing to do with literal inclusion of the imported class or file. It merely informs the compiler how to resolve references.

```
import business.*
def canoo = new Vendor()
canoo.name = 'Canoo Engineering AG'
canoo.product = 'UltraLightClient (ULC)'
assert canoo.dump() =~ /ULC/
```

Default import statements

By default, Groovy imports six packages and two classes, making it seem like every Groovy code program contains the following initial statements:

```
imoprt java.lang.*
imoprt java.util.*
imoprt java.io.*
imoprt java.net.*
imoprt groovy.lang.*
imoprt groovy.util.*
imoprt java.math.BigInteger
imoprt java.math.BigDecimal
```

TYPE ALIASING

An import statement has another nice twist: together with the as keyword, it can be used for type aliasing. Whereas a normal import statement allows a fully qualified class to be referred to by its base name, a type alias allows a fully qualified class to be referred to by a name of your choosing. This feature resolves naming conflicts and supports local changes or bug fixes to a third-party library.

```
package thirdparty

class MathLib {
    Integer twice(Integer value) {
        return value * 3
// intentionally wrong!
    }

    Integer half(Integer value) {
        return value / 2
    }
}

package logic

import thirdparty.MathLib as OrigMathLib

class MathLib extends OrigMathLib {
    Integer twice(Integer value) {
        return value * 2
    }
}
// nothing changes below here
def mathlib = new MathLib()
assert 10 == mathlib.twice(5)
assert 2 == mathlib.half(5)
```

avoid names clashes

```
import thirdparty.MathLib as TwiceHalfMathLib
import thirdparty2.MathLib as IncMathLib
def math1 = new TwiceHalfMathLib()
def math2 = new IncMathLib()
assert 3 == math1.half(math2.increment(5))
```

Advanced object-oriented features

Inheritance

Groovy classes can extend Groovy and Java classes and interfaces alike. Java classes can also extend

Groovy classes and interfaces. You need to compile your Java and Groovy classes in a particular order for this to work The only other thing you need to be aware of is that Groovy is more dynamic than Java when it selects which methods to invoke for you.

Using interfaces

In Java, you'd normally write an interface for the plug-in mechanism and then an implementation class for each plug-in that implements that interface. In Groovy, dynamic typing allows you to more easily create and use implementations that meet a certain need. You're likely to be able to create just two classes as part of developing two plug-in implementations. In general, you have a lot less scaffolding code and a lot less typing.

Implementing interfaces and SAM types

If you decide to make heavy use of interfaces, Groovy provides ways to make them more dynamic. If you have an interface, MyInterface, with a single method and a clo- sure, myClosure, you can use the as keyword to coerce the closure to be of type MyInterface. In fact from Groovy 2.2, you don't even need the as keyword. Groovy does implicit closure coercion into single abstract method types as shown in this exam- ple, where the addListener method would normally require an ActionListener:

```
import java.awt.event.ActionListener
listeners = []
def addListener(ActionListener al) { listeners << al }
addListener { println "I heard that!" }
listeners*.actionPerformed()
```

Alternatively, if you have an interface with several methods, you can create a map of closures keyed on the method names and coerce the map to your interface type.

```
interface CrudRepository {
    def find(def id)

    def findAll()

    def save(def entity)

    def delete(def id)
}

CrudRepository crudRepository = [
        find   : { "find $it" },
        findAll: { "findAll" },
        save   : { "$it saved"; true },
        delete : { "deleted $it"; true }
] as CrudRepository

assert crudRepository.save("John")
assert crudRepository.delete(7L)
assert crudRepository.find(70L)
assert crudRepository.findAll()
```

Multimethods

Remember that Groovy's mechanics of method lookup take the dynamic type of method arguments into account, whereas Java relies on the static type. This Groovy feature is called multimethods.

```
def oracle(Object o) { return 'object' }

def oracle(String o) { return 'string' }

Object x = 1
Object y = 'foo'
assert 'object' == oracle(x)
assert 'string' == oracle(y)
```

Because Groovy dispatches by the runtime type, the specialized implementation of oracle(String) is used in the second case.

With this capability in place, you can better avoid duplicated code by being able to override behavior more selectively. Consider the equals implementation in the fol- lowing listing that overrides Object ’s default equals method only for the argument type Equalizer .

```
class Equalizer {
boolean equals(Equalizer e){
    return true
    }
}
Object same = new Equalizer()
Object other = new Object()
assert
new Equalizer().equals( same )
assert ! new Equalizer().equals( other )
```

Using traits

> Java’s decision to adopt single inheritance of implementation greatly simplified the language at the expense of making it more difficult to support certain kinds of reuse. We’ve all heard the mantra “prefer delegation over inheritance.” It’s arguable that this is a direct consequence of Java’s restrictions.
>
> A programmer might have the desire to share code capabilities within their classes without duplication, but given Java’s restrictions, they create inappropriate subtype relationships. Default methods in Java 8 interfaces lift this restriction somewhat but still don’t allow a full “design by capability” that includes state.

Groovy traits support composition of capabilities. Capabilities that are designed to be shared are implemented in traits. Your classes can then implement those traits to indi- cate that they provide that capability. 14 They “inherit” the implementation from the trait but can override it if they wish. If this sounds like Java 8 default methods, you’re on the right track, but Groovy traits also support state.

```
trait HasId {
    long id
}

trait HasVersion {
    long version
}

trait Persistent {
    boolean save() { println "saving ${this.dump()}" }
}

trait Entity implements Persistent, HasId, HasVersion {
    boolean save() {
        version++
        Persistent.super.save()
    }
}

class Publication implements Entity {  ①
    String title
}

class Book extends Publication {
    String isbn
}

Entity gina = new Book(id:1, version:1, title:"gina", isbn:"111111")
gina.save()
assert gina.version == 2
```

At (1) we make Publication an Entity . This is what we call the intrusive way of apply- ing traits. There's an even more flexible one: applying them nonintrusively at runtime. Publications stay totally agnostic of persistency:

```
class Publication {
    String title
}

Entity gina = new Book(title:"gina", isbn:"111111") as Entity
gina.id = 1
gina.version = 1
```

NOTE that gina is no longer of type Book as it was before. That's the price we pay for flexibility. But this nonintrusive way of extending a class independent from its inheri- tance in a type-safe manner is a great way of developing incrementally.

Working with GroovyBeans

Declaring beans

java

```
public class MyBean implements java.io.Serializable {
    private String myprop;
    public String getMyprop(){
        return myprop;
    }
    public void setMyprop(String value){
        myprop = value;
    }
}
```

groovy

```
class MyBean implements Serializable {
    String myprop
}
```

Declaring properties in GroovyBeans

```
class MyBean implements Serializable {
    def untyped
    String typed
    def item1, item2
    def assigned = 'default value'
}
def bean = new MyBean()
assert 'default value' == bean.getAssigned()
bean.setUntyped('some value')
assert 'some value' == bean.getUntyped()
bean = new MyBean(typed:'another value')
assert 'another value' == bean.getTyped()
```

Table 11. Groovy accessor method to property mappings

Java	Groovy
x.getProperty()	x.property
x.setProperty(y)	x.property=y

```
class MrBean {
    String firstname, lastname

    String getName(){
        return "$firstname $lastname"
    }
}

def bean = new MrBean(firstname: 'Rowan')
bean.lastname = 'Atkinson'

assert 'Rowan Atkinson' == bean.name
```

Advanced accessors with Groovy

```
class DoublerBean {
    public value

    void setValue(value){
        this.value=value
    }

    def getValue(){
        value * 2
    }
}

def bean = new DoublerBean(value: 100)
assert 200 == bean.value
assert 100 == bean.@value
```

IMPORTANT	Inside the lexical scope of a field, references to fieldname or this.fieldname are resolved as field access, not as property access. The same effect can be achieved from outside the scope using the reference.@field- name syntax.

BEAN-STYLE EVENT HANDLING

Groovy supports event listeners in a simple but powerful way.

java

```
// Java
final JButton button = new JButton("Push me!");
button.addActionListener(new IActionListener(){
    public void actionPerformed(ActionEvent event){
        System.out.println(button.getText());
    }
});
```

A Groovy programmer only has to attach a closure to the button as if it were a field named by the respective callback method:

```
button = new JButton('Push me!')
button.actionPerformed = { event ->
    println button.text
}
```

NOTE	Groovy uses bean introspection to determine whether a field setter refers to a callback method of a listener that's supported by the bean. If so, a Closure- Listener is transparently added that calls the closure when notified. A ClosureListener is a proxy implementation of the required listener interface.

Using bean methods for any object

```
class ClassWithProperties {
    def someProperty
    public someField
    private somePrivateField
}

def obj = new ClassWithProperties()
def store = []
obj.properties.each { property ->
    store += property.key
    store += property.value
}
assert store.contains('someProperty')
assert store.contains('someField') == false
assert store.contains('somePrivateField') == false
assert store.contains('class')
assert obj.properties.size() == 2
```

Fields, accessors, maps, and Expando

`object.name` Here's what happens when Groovy resolves this reference:

- If object refers to a map, object.name refers to the value corresponding to the name key that's stored in the map. Otherwise, if name is a property of object , the property is referenced .
- Every Groovy object has the opportunity to implement its own getProperty (name) and setProperty(name, value) methods. When it does, these imple- mentations are used to control the property access. Maps, for example, use this mechanism to expose keys as properties.
- Field access can be intercepted by providing the object.get(name) method, This is a last resort as far as the Groovy runtime is con- cerned: it's used only when there's no appropriate JavaBeans property available and when getProperty isn't implemented.

Expando

An Expando can be thought of as an expandable alternative to a bean.

```
def boxer = new Expando()
assert null == boxer.takeThis
boxer.takeThis = 'ouch!'
assert 'ouch!' == boxer.takeThis
boxer.fightBack = {times -> delegate.takeThis * times}
assert 'ouch!ouch!ouch!' == boxer.fightBack(3)
```

Using advanced syntax features

```
this.class.methods

this.class.methods.name

this.class.methods.name.grep(~/get.*/).sort()

list.property == list.collect{ item -> item?.property }
list*.member == list.collect{ item -> item?.member }
```

examples using Gpath

```
class Invoice {
    List  items
    Date date
}

class LineItem {
    Product product
    int count

    int total() {
        return product.dollar * count
    }
}

class Product {
    String name
    def dollar
}

def ulcDate =Date.parse('yyyy-MM-dd', '2015-01-01')
def otherDate= Date.parse('yyyy-MM-dd', '2015-02-02')
def ulc = new Product(dollar:1499, name:'ULC')
def ve = new Product(dollar:499, name:'Visual Editor')


def invoices = [
        new Invoice(date:ulcDate, items: [
        new LineItem(count:5, product:ulc),
        new LineItem(count:1, product:ve)
    ]),
        new Invoice(date:otherDate, items: [
        new LineItem(count:4, product:ve)
    ])
]

def allItems = invoices.items.flatten()

assert [5*1499, 499, 4*499] == allItems*.total()

assert ['ULC'] == allItems.grep{it.total() > 7000}.product.name

def searchDates = invoices.grep{
    it.items.any{it.product == ulc}
}.date*.toString()

assert [ulcDate.toString()] == searchDates
```

Injecting the spread operator

Groovy provides a * (spread) operator that's connected to the spread-dot operator in that it deals with tearing a list apart. It can be seen as the reverse counterpart of the subscript operator that creates a list from a sequence of comma-separated objects. The spread operator distributes all items of a list to a receiver that can take this sequence. Such a receiver can be a method that takes a sequence of arguments or a list constructor.

```
def getList() {
    return [1, 2, 3]
}

def sum(a, b, c) {
    return a + b + c
}

assert 6 == sum(*list)


def range = (1..3)
assert [0,1,2,3] == [0,*range]

def map = [a:1,b:2]
assert [a:1, b:2, c:3] == [c:3, *:map]
```

Concise syntax with command chains

Command chains are such a feature that's based on a very simple idea: one can omit dots and parentheses in chain-of-method calls.

```
link(producer).to(consumer)
link producer to consumer

move(10, forward).painting(color:blue)
move 10, forward painting color:blue
```

IMPORTANT It works with all methods that have at least one argument.

Dynamic Programming

What is dynamic programming?

In classic object-oriented systems, every class has a well-known set of states, captured in the fields of that class, and well-known behavior, defined by its methods. Neither the set of states nor the behavior ever changes after compilation, and it's identical for all instances of a class.

> **Dynamic programming**
>
> breaks this limitation by allowing the introduction of a new state, or even more importantly, allowing the addition of a new behavior or modi- fication of an existing one.

What is "meta"?

```
Meta means applying a concept onto itself—for example, metainformation is information
about information.
Likewise, because programming is "writing code," metaprogramming means writing code
that writes code.
```

Meta Object Protocol

```
println 'Hello' // Groovy
InvokerHelper.invokeMethod(this, "println", {"Hello"}); // Java
```

NOTE	Every innocent method call that you write in Groovy is really a call into the MOP , regardless of whether the call target has been compiled with Groovy or Java. This applies equally to static and instance method calls, constructor calls, and property access, even if the target is the same object as the caller.

Customizing the MOP with hook methods

MethodMissing

```
Object methodMissing(String name, Object arguments)

class Pretender {
    def methodMissing(String name, Object args) {
        "called $name with $args"
    }
}
def bounce = new Pretender()
assert bounce.hello('world') == 'called hello with [world]'
```

Using methodMissing to simulate a miniature GORM

```
class MiniGorm {
def db = []
    def methodMissing(String name, Object args) {
        db.find { it[name.toLowerCase()-'findby'] == args[0] }
    }
}

def people=new MiniGorm()
def dierk=[first:'Dierk',last:'Agerm']
def paul=[first:'Paul',last:'King']
assert people.findByFirst('Dierk') == dierk
assert people.findByLast('King') == paul
```

PropertyMissing

```
Object propertyMissing(String name)

class PropPretender {
    def propertyMissing(String name) {
        "accessed $name"
    }
}

def bounce = new PropPretender()
assert bounce.hello == 'accessed hello'
```

```
def propertyMissing(String name) {
    int result = 0
    name.each {
        result <<= 1
        if (it == 'I') result++
    }
    return result
}
IIOI + IOI == IOOIO
```

Using closures for dynamic hooks

```
class DynamicPretender {
    Closure whatToDo = { name -> "accessed $name"}
    def propertyMissing(String name) {
    whatToDo(name)
    }
}

def one = new DynamicPretender()
assert one.hello == 'accessed hello'
one.whatToDo = { name -> name.size() }
assert one.hello == 5
```

Customizing GroovyObject methods

```
public interface GroovyObject {
    Object invokeMethod(String methodName, Object args);
    Object getProperty(String propertyName);
    void setProperty(String propertyName, Object newValue);
    MetaClass getMetaClass();
    void setMetaClass(MetaClass metaClass);
}

public abstract class GroovyObjectSupport implements GroovyObject {
    public Object invokeMethod(String name, Object args) {
        return getMetaClass().invokeMethod(this, name, args);
    }
    public Object getProperty(String property) {
        return getMetaClass().getProperty(this, property);
    }
    public void setProperty(String property, Object newValue) {
        getMetaClass().setProperty(this, property, newValue);
    }
// more here...
}
```

NOTE	You can fool the MOP into thinking that a class that was actually com- piled by Java was compiled by Groovy. You only need to implement the Groovy- Object interface or, more conveniently, extend GroovyObjectSupport .

As soon as a class implements GroovyObject , the following rules apply:

- Every access to a property calls the `getProperty()` method.
- Every modification of a property calls the `setProperty()` method.
- Every call to an unknown method calls `invokeMethod()` . If the method is known, `invokeMethod()` is only called if the class implements `GroovyObject` and the marker interface `GroovyInterceptable`.

Using getProperty to call parameterless methods without parentheses

```
class NoParens {
    def getProperty(String propertyName) {
        if (metaClass.hasProperty(this, propertyName)) {
            return metaClass.getProperty(this, propertyName)
        }
        invokeMethod propertyName, null
    }
}

class PropUser extends NoParens {
    boolean existingProperty = true
}
def user = new PropUser()
assert user.existingProperty
assert user.toString() == user.toString
```

NOTE Once you've implemented getProperty() , every property will be found and thus propertyMissing() will no longer be called.

Modifying behavior through the metaclass

MetaClass knows it all

```
MetaClass mc = String.metaClass
final Object[] NO_ARGS = []
assert 1 == mc.respondsTo("toString", NO_ARGS).size()
assert 3 == mc.properties.size()
assert 76 == mc.methods.size()
assert 177 == mc.metaMethods.size()
assert "" == mc.invokeMethod("","toString", NO_ARGS)
assert null == mc.invokeStaticMethod(String, "println", NO_ARGS)
assert "" == mc.invokeConstructor(NO_ARGS)
```

Calling a method means calling the metaclass

You can assume that Groovy never calls methods directly in the bytecode but always through the object's metaclass. At least, this is how it looks to you as a programmer.

Behind the scenes there are optimizations going on that technically circumvent the metaclass, but only when it's safe to do so.

How to find the metaclass and invoke methods

Objects that don't inherit from GroovyObject aren't asked for the metaClass property. Their

metaclass is retrieved from the MetaClassRegistry .

IMPORTANT	The default metaclass can be changed from the outside with- out touching any application code. Let's assume you have a class Custom in package custom . Then you can change its default metaclass by putting a meta- class with the name groovy.runtime.metaclass.custom.CustomMetaClass on the classpath. This device has been proven useful when inspecting large Groovy codebases in production.

```
// MOP pseudo code
def mopInvoke(Object obj, String method, Object args) {
    if (obj instanceof GroovyObject) {
        return groovyObjectInvoke(obj, method, args)
    }
    registry.getMetaClass(obj.class).invokeMethod(obj, method, args)
}
```

```
def groovyObjectInvoke(Object obj, String method, Object args) {

    if (obj instanceof GroovyInterceptable) {
        return obj.metaClass.invokeMethod(method, args)
    }
    if (!obj.metaClass.respondsTo(method, args))
        return obj.metaClass.invokeMethod(method, args)

    obj.metaClass.invokeMethod(obj, method, args)
}
```

```
// Default meta class pseudo code
def invokeMethod(Object obj, String method, Object args) {
    if (obj.metaClass.respondsTo(method, args)) {
        return methodCall(obj, method, args)
    }
    if (methodMissingAvailable(obj)) {
        return obj.metaClass.methodMissing(method, args)
    }
    throw new MissingMethodException()
}
```

Setting other metaclasses

Groovy comes with a number of metaclasses:

- The default metaclass MetaClassImpl , which is used in the vast majority of cases
- The ExpandoMetaClass , which can expand the state and behavior

- A ProxyMetaClass , which can decorate a metaclass with interception capabilities
- Additional metaclasses that are used internally and for testing purposes

Assigning a ProxyMetaClass to a GroovyObject for tracing method calls

```
class InspectMe {
int outer(){
return inner()
}
private int inner(){
return 1
}
}
def tracer = new TracingInterceptor(writer: new StringWriter())
def proxyMetaClass = ProxyMetaClass.getInstance(InspectMe)
proxyMetaClass.interceptor = tracer
InspectMe inspectMe = new InspectMe()
inspectMe.metaClass = proxyMetaClass

assert 1 == inspectMe.outer()
assert "\n" + tracer.writer.toString() == """
before InspectMe.outer()
before InspectMe.inner()
after InspectMe.inner()
after InspectMe.outer()
"""
```

Interceptors are more than aspects

Interceptors may remind one or the other reader of aspect-oriented programming (AOP) and the TracingInterceptor suggests this connotation. But interceptors can do much more: they can redirect to a different method, change the arguments, sup- press the method call, and even change the return value!

```
def boxer = new Expando()
boxer.takeThis = 'ouch!'
boxer.fightBack = { times -> takeThis * times}
assert boxer.fightBack(3) == 'ouch!ouch!ouch!'
```

Adding low() to java.lang.String via ExpandoMetaClass

```
assert String.metaClass =~ /MetaClassImpl/
String.metaClass.low = {-> delegate.toLowerCase() }
assert String.metaClass =~ /ExpandoMetaClass/
assert "DiErK".low() == "dierk"
```

Modifying the metaclass of a class (Groovy and Java)

```
class MyGroovy1 { }
def before = new MyGroovy1()
MyGroovy1.metaClass.myProp = "MyGroovy prop"
MyGroovy1.metaClass.test = {-> myProp }

try {
before.test()
assert false, "should throw MME"
} catch(mme) { }

assert new MyGroovy1().test() == "MyGroovy prop"
```

Modifying the metaclass of a Groovy instance

```
class MyGroovy2 {}

def myGroovy = new MyGroovy2()
myGroovy.metaClass.myProp = "MyGroovy prop"
myGroovy.metaClass.test = { -> myProp }
try {
    new MyGroovy2().test()
    assert false, "should throw MME"
} catch (mme) {
}
```

Modifying the metaclass of a Java instance

```
def myJava = new String()
myJava.metaClass.myProp = "MyJava prop"
myJava.metaClass.test = { -> myProp }
try {
    new String().test()
    assert false, "should throw MME"
} catch (mme) {
}

assert myJava.test() == "MyJava prop"
```

Decoding A Space Odyssey with a metaclass builder

```
def move(string, distance) {
    string.collect { (it as char) + distance as char }.join()
}
String.metaClass {
    shift = -1
    encode {-> move delegate, shift }
    decode {-> move delegate, -shift }
    getCode {-> encode() }
    getOrig {-> decode() }
}
assert "IBM".encode() == "HAL"
assert "HAL".orig == "IBM"
def ibm = "IBM"
ibm.shift = 7
assert ibm.code == "PIT"
```

NOTE Modifying the metaclass of the String class will affect all future String instances.

Adding a static method to a class

```
Integer.metaClass.static.answer = {-> 42}
assert Integer.answer() == 42
```

Metaclass changes for superclasses and interfaces

```
class MySuperGroovy { }
class MySubGroovy extends MySuperGroovy { }
MySuperGroovy.metaClass.added = {-> true }
assert new MySubGroovy().added()
Map.metaClass.toTable = {->
    delegate.collect{ [it.key, it.value] }
}
assert [a:1, b:2].toTable() == [
        ['a', 1],
        ['b', 2]
]
```

Metaclass injection of operator and MOP hook methods

```
String.metaClass {
    rightShiftUnsigned = { prefix ->
        delegate.replaceAll(~/\w+/) { prefix + it }
    }
    methodMissing = { String name, args->
        delegate.replaceAll name, args[0]
    }
}

def people = "Dierk,Guillaume,Paul,Hamlet,Jon"
people >>>= "\n"
people = people.Dierk('Mittie').Guillaume('Mr.G')
assert people == '''
Mittie,
Mr.G,
Paul,
Hamlet,
Jon'''
```

Some takeaways and rules of thumb for metaclasses:

- All method calls from Groovy code go through a metaclass.
- Metaclasses can change for all instances of a class or per a single instance.
- Metaclass changes affect all future instances in all running threads.
- Metaclasses allow nonintrusive changes to both Groovy and Java code as long as the caller is Groovy. We can even change access to final classes like `java.lang.String` .
- Metaclass changes can take the form of property accessors (pretending prop- erty access), operator methods, GroovyObject methods, or MOP hook methods.
- ExpandoMetaClass makes metaclass modifications more convenient.
- Metaclass changes are best applied only once, preferably at application startup time.

Temporary MOP modifications using category classes

The last point directly leads us to another concept of dynamic programming in Groovy. ExpandoMetaClass isn't designed for easily removing a once dynamically added method or undoing any other change. For such temporary changes, Groovy provides category classes.

Using a category class is trivial. Groovy adds a use method to java.lang.Object that takes two parameters: a category class (or any number thereof) and a closure:

```
use CategoryClass, {
    // new methods are available
}
// new methods are no longer available
```

How to use existing categories like TimeCategory and Collections

```
import groovy.time.TimeCategory

def janFirst1970 = new Date(0)
use TimeCategory, {
    Date
    xmas = janFirst1970 + 1.year - 7.days
    assert xmas.month == Calendar.DECEMBER
    assert xmas.date == 25
}
use Collections, {
    def list = [0, 1, 2, 3]
    list.rotate 1
    assert list == [3, 0, 1, 2]
}
```

Category classes are by no means special. Neither do they implement a certain interface nor do they inherit from a certain class. They aren't configured or registered anywhere! They just happen to contain static methods with at least one parameter.

When a class is used as an argument to the use method, it becomes a category class and every static method like

```
static ReturnType methodName(Receiver self, optionalArgs) {...}
```

becomes available on the receiver as if the Receiver had an instance method like

```
ReturnType methodName(optionalArgs) {...}
```

Running a category to marshal and unMarshal integers to/from strings

```
class Marshal {
    static String marshal(Integer self) {
        self.toString()
    }
    static Integer unMarshal(String self) {
        self.toInteger()
    }
}

use Marshal, {
    assert 1.marshal() == "1"
    assert "1".unMarshal() == 1
    [Integer.MIN_VALUE, -1, 0, Integer.MAX_VALUE].each {
        assert it.marshal().unMarshal() == it
    }
}
```

Naming the receiver object self is just a convention. You can use any name you want. Groovy's design decision of using static methods to implement category behavior has a few beneficial effects.

- You're much less likely to run into concurrency issues, because there's less shared state.
- You can use a plethora of classes as categories even if they've been implemented without knowing about Groovy. Collections was just an example of many classes with static methods that reside in widely used helper libraries.
- They can easily be created in Groovy, Java, or any other JVM language that pro- duces classes and static methods.

Category classes are a good place to collect methods that work conjointly on different types, such as Integer and String , to accomplish a feature like marshaling. Key characteristics of using category classes are:

- The use method applies categories to the runtime scope of the closure (as opposed to the lexical scope). That means you can extract code from the clo- sure into a method and call the method from inside the closure.
- Category use is confined to the current thread.
- Category use is nonintrusive.
- If the receiver type refers to a superclass or even an interface, then the method will be available in all subclasses/implementors without further configuration.
- Category method names can well take the form of property accessors (pretend- ing property access), operator methods, and GroovyObject methods. MOP hook methods cannot be added through a category class. 10
- Category methods can override method definitions in the metaclass.
- Where performance is crucial, use categories with care and measure their influence.
- Categories cannot introduce a new state in the receiver object; they cannot add new properties

with a backing field.

Writing extension modules

Extension modules can be seen as categories that are always visible: you don't need to call use to enable the methods. Just like Groovy enriches the JDK classes with custom methods, you can make your categories globally visible and make them behave like methods from the GDK .

One of the most interesting use cases for this is that you can bundle such extension modules into their own JAR file and make them available to other programs just by adding the JAR file to your classpath.

Converting a category into an extension module is straightforward. Imagine that you want to use the Marshal category defined in previous listing without having to explicitly use the category. To achieve that, you only need two steps:

- Write the Marshal class into its own source file .
- Write an extension module descriptor and make it available on a classpath.

You need to create a file named org.codehaus.groovy.runtime.ExtensionModule and ensure it's found in the META-INF/services folder of your JAR . This file is used internally by Groovy to load your extension module and make the category transparently available. The descriptor file consists of four entries:

org.codehaus.groovy.runtime.ExtensionModule

```
moduleName=regina-marshal
moduleVersion=1.0
extensionClasses=regina.Marshal
staticExtensionClasses=
```

The moduleName and moduleVersion entries are used by Groovy when the runtime is initialized. If two versions of a module of the same name are found on the classpath, the module will not be loaded and an error will be thrown. The extensionClasses entry is a comma-separated list of category-like classes. This means that you can define multiple categories in a single extension module.

StringUtils

```
moduleName=apache-commons-stringutils
moduleVersion=3.2
extensionClasses=org.apache.commons.lang3.StringUtils
```

Using the @Category annotation

With @Category , you write your class as if it were an instance class but the annotation adjusts it to have the required format needed for categories, meaning, methods are made static and the self parameter:

```
@Category(Integer)
class IntegerMarshal {
    String marshal() {
        toString()
    }
}
@Category(String)
class StringMarshal {
    Integer unMarshal() {
        this.toInteger()
    }
}

use ([IntegerMarshal, StringMarshal]) {
    assert1.marshal() == "1"
    assert "1".unMarshal() == 1
}
```

The @Category annotation can only be used for creating categories associated with a single class; therefore we split our category into two.

Merging classes with Mixins

If you have a superclass A with a subclass B then any object of class B isn't only a B , it also is an A ! The definitions of A and B typically reside in different files. 12 The situation looks as if A and B would be merged when constructing an instance of B . They share both state and behavior. This class merging by inheritance is pretty restricted in Java.

- You cannot use it when inheritance has already been used for other purposes.
- You cannot merge (inherit from) more than one class.
- It's intrusive. You have to change the class definition.
- You cannot do it with final classes.

```
@Mixin(MessageFeature)
class FirstTest extends GroovyTestCase {
    void testWithMixinUsage() {
        message = "Called from Test"
        assertMessage "Called from Test"
    }
}

class MessageFeature {
    def message
    void assertMessage(String msg) {
        assertEquals msg, message
    }
}
```

Mixing in multiple sieve features nonintrusively

```
class EvenSieve {
    def getNo2() {
        removeAll { it % 2 == 0}
        return this
    }
}
class MinusSieve {
    def minus(int num) {
        removeAll { it % num == 0}
        return this
    }
}
ArrayList.mixin EvenSieve, MinusSieve
assert (0..10).toList().no2 - 3 - 5 == [1, 7]
```

Mixins are often compared with multiple inheritance but they're of a different nature.

In the first place, our ArrayList doesn't become a subtype of MinusSieve .

Any instanceof test will fail. There's no is-a relationship and no polymorphism. You can use enforced type coercion with the as operator, though.

Unlike many models of multiple inheritance, the mixing in of new features always happens in traceable sequence and, in case of conflicts, the latest addition wins. Mixins work like metaclass changes in that respect.

Mixins are designed for sharing features while not modifying any existing behavior of the receiver. Features can build on top of each other and merge and blend with the receiver

MOP priorities

It's always good advice to keep things simple. With dynamic programming one can easily go overboard by doing too much, such as using category classes, metaclass changes, and Mixins in combination. If you do anyway, then categories are looked at first, then the metaclass, and finally the Mixins:

`category class > meta class > mixin`

But this only applies to methods that are defined for the same class and have the same parameter types. Otherwise, the rules for method dispatch by class/super- class/interface take precedence.

NOTE In case of multiple method definitions, a category class shadows a pre- viously applied category class. Changes to an ExpandoMetaClass override previously added methods in that metaclass. Later applied Mixins shadow previously applied Mixins.

Real-world dynamic programming in action

Replacing constructors with factory methods

```
import java.awt.Dimension

Class.metaClass.make = { Object[] args ->
    delegate.metaClass.invokeConstructor(*args)
}

assert new HashMap() == HashMap.make()
assert new Integer(42) == Integer.make(42)
assert new Dimension(2, 3) == Dimension.make(2, 3)
```

Fooling IDEs for fun and profit

```
interface ChannelComponent {}
class Producer implements ChannelComponent {
    List<Integer> outChannel
}
class Adaptor implements ChannelComponent {
    List<Integer> inChannel
    List<String> outChannel
}
class Printer implements ChannelComponent {
    List<String> inChannel
}
class WiringCategory {
    static connections = []
    static setInChannel(ChannelComponent self, value){
        connections << [target:self, source:value]
    }
    static getOutChannel(ChannelComponent self){
        self
    }
}

Producer producer = new Producer()
Adaptor adaptor = new Adaptor()
Printer printer = new Printer()
use WiringCategory, {
    adaptor.inChannel = producer.outChannel
    printer.inChannel = adaptor.outChannel
}

assert WiringCategory.connections == [
        [source: producer, target: adaptor],
        [source: adaptor, target: printer]
]
```

Method aliasing and undoing metaclass modifications

```
MetaClass oldMetaClass = String.metaClass
MetaMethod alias = String.metaClass.metaMethods
        .find { it.name == 'size' }
String.metaClass {
    oldSize = { -> alias.invoke delegate }
    size = { -> oldSize() * 2 }
}
assert "abc".size() == 6
assert "abc".oldSize() == 3

if (oldMetaClass.is(String.metaClass)){
    String.metaClass {
        size = { -> alias.invoke delegate }
        oldSize = { -> throw new UnsupportedOperationException() }
    }
}else {
    String.metaClass = oldMetaClass
}

assert "abc".size() == 3
```

The Intercept/Cache/Invoke pattern

```
ArrayList.metaClass.methodMissing = { String name, Object args ->
    assert name.startsWith("findBy")
    assert args.size() == 1
    Object.metaClass."$name" = { value ->
        delegate.find { it[name.toLowerCase() - 'findby'] == value }
    }
    delegate."$name"(args[0])
}

def data = [
        [name: 'moon', au  : 0.0025],
        [name: 'sun', au  : 1],
        [name: 'neptune', au: 30]
]
assert data.findByName('moon')
assert data.findByName('sun')
assert data.findByAu(1)
```

compile-time_metaprogramming_and_ast_transformations = Compile-time metaprogramming and AST transformations

Making Groovy cleaner and leaner

Groovy ships with many AST transformations that you can use today to get rid of those annoying

bits of repetitive code in your classes. When applied properly, the annotations described here make your code less verbose, so that the bulk of the code expresses meaningful business logic to the reader instead of meaningful code templates to the compiler. AST transformations cover a wide range of functionality, from generating standard toString() methods, to easing object delegation, to clean- ing up Java synchronization constructs, and more.

Code-generation transformations

@GROOVY.TRANSFORM.TOSTRING

```
import groovy.transform.ToString
@ToString
class Detective {
    String firstName, lastName
}
def sherlock = new Detective(firstName: 'Sherlock', lastName: 'Holmes')
assert sherlock.toString() == 'Detective(Sherlock, Holmes)'
```

Using @ToString with annotation parameters

```
import groovy.transform.ToString
@ToString(includeNames = true, ignoreNulls = true)
class Sleuth {
    String firstName, lastName
}

def nancy = new Sleuth(firstName: 'Nancy', lastName: 'Drew')
assert nancy.toString() == 'Sleuth(firstName:Nancy, lastName:Drew)'
nancy.lastName = null
assert nancy.toString() == 'Sleuth(firstName:Nancy)'
```

@GROOVY.TRANSFORM.EQUALSANDHASHCODE

```
import groovy.transform.EqualsAndHashCode
@EqualsAndHashCode
class Actor {
    String firstName, lastName
}
def magneto = new Actor(firstName:'Ian', lastName: 'McKellen')
def gandalf = new Actor(firstName:'Ian', lastName: 'McKellen')
assert magneto == gandalf
```

@GROOVY.TRANSFORM.TUPLECONSTRUCTOR

```
import groovy.transform.TupleConstructor
@TupleConstructor
class Athlete {
    String firstName, lastName
}
def a1 = new Athlete('Michael', 'Jordan')
def a2 = new Athlete('Michael')
assert a1.firstName == a2.firstName
```

@GROOVY.TRANSFORM.LAZY

Lazy instantiation is a common idiom in Java. If a field is expensive to create, such as a database connection, then the field is initialized to null , and the actual connec- tion is created only the first time that field is used. Typical in this idiom is a null check and instantiation within a getter method. But not only is this boilerplate code, there are numerous tricky scenarios, such as correctly handling creation in a multi- threaded environment, which are error-prone. The @Lazy field annotation correctly delays field instantiation until the time when that field is first used and correctly handles numerous tricky special cases.

```
class Resource {
    private static alive = 0
    private static used = 0

    Resource() { alive++ }

    def use() { used++ }

    static stats() { "$alive alive, $used used" }
}

class ResourceMain {
    def res1 = new Resource()
    @Lazy res2 = new Resource()
    @Lazy static res3 = { new Resource() }()
    @Lazy(soft=true) volatile Resource res4
}

new ResourceMain().with {
    assert Resource.stats() == '1 alive, 0 used'
    res2.use()
    res3.use()
    res4.use()
    assert Resource.stats() == '4 alive, 3 used'
    assert res4 instanceof Resource
    def expected = 'res4=java.lang.ref.SoftReference'
    assert it.dump().contains(expected)
}
```

@GROOVY.TRANSFORM.INDEXEDPROPERTY

Using @IndexedProperty to generate index-based setters and getters

```
import groovy.transform.IndexedProperty

class Author {
    String name
    @IndexedProperty List<String> books
}
def books = ['The Mysterious Affair at Styles',
             'The Murder at the Vicarage']
new Author(name: 'Agatha Christie', books: books).with {
    books[0] = 'Murder on the Orient Express'
    setBooks(0, 'Death on the Nile')
    assert getBooks(0) == 'Death on the Nile'
}
```

@GROOVY.TRANSFORM.INHERITCONSTRUCTORS

The @InheritConstructors annotation removes the boilerplate of writing match- ing constructors for a superclass.

```
import groovy.transform.InheritConstructors

@InheritConstructors
class MyPrintWriter extends PrintWriter { }

def pw1 = new MyPrintWriter(new File('out1.txt'))
def pw2 = new MyPrintWriter('out2.txt', 'US-ASCII')

[pw1, pw2].each {
    it << 'foo'
    it.close()
}

assert new File('out1.txt').text == new File('out2.txt').text
['out1.txt', 'out2.txt'].each{ new File(it).delete() }
```

@GROOVY.TRANSFORM.SORTABLE

Using @Sortable to generate Comparable / Comparator methods

```
import groovy.transform.Sortable
@Sortable(includes = 'last,initial')
class Politician {
    String first
    Character initial
    String last
    String initials() { first[0] + initial + last[0] }
}
def politicians = [
        new Politician(first: 'Margaret', initial: 'H', last: 'Thatcher'),
        new Politician(first: 'George', initial: 'W', last: 'Bush')
]
def sorted = politicians.toSorted()
assert sorted*.initials() == ['GWB', 'MHT']
def byInitial = Politician.comparatorByInitial()
sorted = politicians.toSorted(byInitial)
assert sorted*.initials() == ['MHT', 'GWB']
```

@GROOVY.TRANSFORM.BUILDER

```
import groovy.transform.builder.Builder
@Builder
class Chemist {
    String first
    String last
    int born
}

def builder = Chemist.builder()
def c = builder.first("Marie").last("Curie").born(1867).build()
assert c.first == "Marie"
assert c.last == "Curie"
assert c.born == 1867
```

Table 12. Built-in @Builder strategies

Strategy	Description
DefaultStrategy	Creates a nested helper class for instance creation. Each method in the helper class returns the helper until finally a build() method is called, which returns a created instance.
SimpleStrategy	Creates chainable setters, where each setter returns the object itself after updating the appropriate property.

Strategy	Description
ExternalStrategy	Allows you to annotate an explicit builder class while leaving some builder class being built untouched. This is appropriate when you want to create a builder for a class you don't have control over such as from a library or another team in your organization.
InitializerStrategy	Creates a nested helper class for instance creation that when used with `@CompileStatic` allows type-safe object creation. Compatible with `@Immutable` .

Class design and design pattern annotations

@GROOVY.TRANSFORM.CANONICAL

```
import groovy.transform.Canonical
@Canonical
class Inventor {
    String firstName, lastName
}
def i1 = new Inventor('Thomas', 'Edison')
def i2 = new Inventor('Thomas')
assert i1 != i2
assert i1.firstName == i2.firstName
assert i1.toString() == 'Inventor(Thomas, Edison)'
```

@GROOVY.TRANSFORM.IMMUTABLE

```
import groovy.transform.Immutable
import static groovy.test.GroovyAssert.shouldFail
@Immutable
class Genius {
    String firstName, lastName
}

def g1 = new Genius(firstName: 'Albert', lastName: "Einstein")
assert g1.toString() == 'Genius(Albert, Einstein)'

def g2 = new Genius('Leonardo', "da Vinci")
assert g2.firstName == 'Leonardo'
assert g1 != g2
shouldFail(ReadOnlyPropertyException) {
    g2.lastName = 'DiCaprio'
}
```

@GROOVY.LANG.DELEGATE

```
class NoisySet extends HashSet {
    @Override
    boolean add(item) {
        println "adding $item"
        super.add(item)
    }
    @Override
    boolean addAll(Collection items) {
        items.each { println "adding $it" }
        super.addAll(items)
    }
}

class NoisySet implements Set {
    private Set delegate = new HashSet()

    @Override
    boolean add(item) {
        println "adding $item"
        delegate.add(item)
    }

    @Override
    boolean addAll(Collection items) {
        items.each { println "adding $it" }
        delegate.addAll(items)
    }
    @Override
    boolean isEmpty() {
        return delegate.isEmpty()
    }
    @Override
    boolean contains(Object o) {
        return delegate.contains(o)
    }
// ... ditto for size, iterator, toArray, remove,
// containsAll, retainAll, removeAll, clear ...

}

class NoisySet {
    @Delegate
    Set delegate = new HashSet()
    @Override
    boolean add(item) {
        println "adding $item"
        delegate.add(item)
    }
    @Override
```

```
    boolean addAll(Collection items) {
        items.each { println "adding $it" }
        delegate.addAll(items)
    }
}

Set ns = new NoisySet()
ns.add(1)
ns.addAll([2, 3])
assert ns.size() == 3
```

@GROOVY.LANG.SINGLETON

```
class Zeus {
    static final Zeus instance = new Zeus()
    private Zeus() { }
}
assert Zeus.instance
```

using @Singlton

```
import static groovy.test.GroovyAssert.shouldFail
@Singleton class Zeus { }
assert Zeus.instance
def ex = shouldFail(RuntimeException) { new Zeus() }
assert ex.message ==
        "Can't instantiate singleton Zeus. Use Zeus.instance"
```

@GROOVY.TRANSFORM.MEMOIZED

```
import groovy.transform.Memoized

class Calc {
    def log = []

    @Memoized
    int sum(int a, int b) {
        log << "$a+$b"
        a + b
    }
}

new Calc().with {
    assert sum(3, 4) == 7
    assert sum(4, 4) == 8
    assert sum(3, 4) == 7
    assert log.join(' ') == '3+4 4+4'
}
```

@GROOVY.TRANSFORM.TAILRECURSIVE

```
class ListUtil {
    static List reverse(List list) {
        if (list.isEmpty()) list
        else reverse(list.tail()) + list.head()
    }
}
assert ListUtil.reverse(['a', 'b', 'c']) == ['c', 'b','a']
```

```
import groovy.transform.TailRecursive
class ListUtil {
    static reverse(List list) {
        doReverse(list, [])
    }
    @TailRecursive
    private static doReverse(List todo, List done) {
        if (todo.isEmpty()) done
        else doReverse(todo.tail(), [todo.head()] + done)
    }
}
assert ListUtil.reverse(['a', 'b', 'c']) == ['c', 'b', 'a']
```

NOTE	The `Closure.trampoline()` method This method wraps the closure into a TrampolineClosure , which, instead of doing a recursive call to the closure, returns a new closure, which is called during the next step of the computation. This turns a recursive execution into a sequential one, thus helping avoiding the stack overflow, albeit at some performance cost.

Logging improvements

Using @Log to inject a Logger object into an object

```
import groovy.util.logging.Log

@Log
class Database {
    def search() {
        log.fine(runLongDatabaseQuery())
    }

    def runLongDatabaseQuery() {
        println 'Calling database'
        /* ... */
        return 'query result'
    }
}

new Database().search()
```

Table 13. Five @Log annotations

Name	Description
@Log	Injects a static final java.util.logging.Logger into your class and initializes it using Logger.getLogger(class.name) .
@Commons	Injects an Apache Commons logger as a static final org.apache.commons.logging.Log into your class and initializes it using LogFactory.getLog(class) .
@Log4j	Injects a Log4j logger as a static final org.apache.log4j.Logger into your class and initializes it using Logger.getLogger(class) .
@Log4j2	Injects a Log4j2 logger as a static final org.apache.log4j.Logger into your class and initializes it using Logger.getLogger(class) .
@Slf4j	Injects an Slf4j logger as a static final org.slf4j.Logger into your class and initializes it using org.slf4j.LoggerFactory.getLogger(class) . The

LogBack	framework uses SLF4J as the underlying logger, so LogBack users should use @Slf4j .

Declarative concurrency

@Synchronized

> **Avoid low-level synchronization**
>
> Java contains many fine primitives for working with concurrent code, such as the synchronized keyword and the contents of the java.util.concurrent package. But these are mostly primitives and not abstractions. The tools are low level and meant to serve as a foundation.

```
import groovy.transform.Synchronized

class PhoneBook1 {
    private final phoneNumbers = [:]

    @Synchronized
    def getNumber(key) {
        phoneNumbers[key]
    }

    @Synchronized
    void addNumber(key, value) {
        phoneNumbers[key] = value
    }
}

def p1 = new PhoneBook1()
(0..99).collect { num ->
    Thread.start{
        p1.addNumber('Number' + num, '98765' + num)
    }
}*.join()
assert p1.getNumber('Number43') == '9876543'
```

Mixing @Synchronized with custom synchronized block

```
import groovy.transform.Synchronized
import groovy.util.logging.Log

@Log
class PhoneBook2 {
    private final phoneNumbers = [:]
    private final lock = new Object[0]

    @Synchronized('lock')
    def getNumber(key) {
        phoneNumbers[key]
    }

    def addNumber(key, value) {
        log.info("Adding phone number $value")
        synchronized (lock) {
            phoneNumbers[key] = value
        }
    }
}

def p2 = new PhoneBook2()
(0..99).collect { num ->
    Thread.start {
        p2.addNumber('Number' + num, '98765' + num)
    }
}*.join()
assert p2.getNumber('Number43') == '9876543'
```

@GROOVY.TRANSFORM.WITHREADLOCK AND @GROOVY.TRANSFORM.WITHWRITELOCK

```
import java.util.concurrent.locks.ReentrantReadWriteLock
class PhoneBook3 {
    private final phoneNumbers = [:]
    final private lock = new ReentrantReadWriteLock()
    def getNumber(key) {
        lock.readLock().lock()
        try {
            phoneNumbers[key]
        } finally {
            lock.readLock().unlock()
        }
    }
    def addNumber(key, value) {
        lock.writeLock().lock()
        try {
            phoneNumbers[key] = value
        } finally {
            lock.writeLock().unlock()
        }
    }
}
```

with @ReadLock and @WriteLock

```
import groovy.transform.*
class PhoneBook3 {
    private final phoneNumbers = [:]
    @WithReadLock
    def getNumber(key) {
        phoneNumbers[key]
    }
    @WithWriteLock
    def addNumber(key, value) {
        phoneNumbers[key] = value
    }
}
```

Using @WithReadLock and @WithWriteLock for efficient concurrency

```
class PhoneBook3 {
    private final phoneNumbers = dummyNums()

    private dummyNums() {
        (1..8).collectEntries {
            ['Number' + it, '765432' + it]
        }
    }

    @groovy.transform.WithReadLock
    def getNumber(key) {
        println "Reading started for $key"
        phoneNumbers[key]
        sleep 80
        println "Reading done for $key"
    }

    @groovy.transform.WithWriteLock
    def addNumber(key, value) {
        println "Writing started for $key"
        phoneNumbers[key] = value
        sleep 100
        println "Writing done for $key"
    }
}

def p3 = new PhoneBook3()
(3..4).each { count ->
    Thread.start {
        sleep 100 * count
        p3.addNumber('Number' + count, '9876543')
    }
}
(2..6).collect { count ->
    Thread.start {
        sleep 100 * count
        p3.getNumber('Number' + count)
    }
}*.join()
```

Easier cloning and externalizing

@GROOVY.TRANSFORM.AUTOCLONE

```
import groovy.transform.AutoClone
@AutoClone
class Chef1 {
    String name
    List<String> recipes
    Date born
}
def name = 'Heston Blumenthal'
def recipes = ['Snail porridge', 'Bacon & egg ice cream']
def born = Date.parse('yyyy-MM-dd', '1966-05-27')
def c1 = new Chef1(name: name, recipes: recipes, born: born)
def c2 = c1.clone()
assert c2.recipes == recipes
```

the class in source code

```
class Chef1 implements Cloneable {
    ...
    Chef1 clone() throws CloneNotSupportedException {
        Chef1 _result = (Chef1) super.clone()
        if (recipes instanceof Cloneable) {
            _result.recipes = (List<String>) recipes.clone()
        }
        _result.born = (Date) born.clone()
        return _result
    }
}
```

Table 14. Four @AutoClone styles

Name	Description
CLONE	Adds a `clone()` method to your class. The `clone()` method will call `super.clone()` before calling `clone()` on each Cloneable property of the class. Doesn't provide deep cloning. Not suitable if you have final properties. This is the default cloning style if no style attribute is provided.
SIMPLE	Adds a `clone()` method to your class that calls the no-arg constructor then copies each property calling `clone()` for each Cloneable property. Handles inheritance hierarchies. Not suitable if you have final properties. Doesn't provide deep cloning.

COPY_CONSTRUCTOR	Adds a copy constructor, which takes your class as its parameter, and a `clone()` method to your class. The copy constructor method copies each property calling `clone()` for each Cloneable property. The `clone()` method creates a new instance making use of the copy constructor. Suitable if you have final properties. Handles inheritance hierarchies. Doesn't provide deep cloning.
SERIALIZATION	Adds a `clone()` method to your class that uses serialization to copy your class. Suitable if your class already implements the Serializable or Externalizable interface. Automatically performs deep cloning. Not as time or memory efficient. Not suitable if you have final properties.

Using the COPY_CONSTRUCTOR style with @AutoClone

```
import groovy.transform.*
import static groovy.transform.AutoCloneStyle.*
@TupleConstructor
@AutoClone(style=COPY_CONSTRUCTOR)
class Person {
    final String name
    final Date born
}
@TupleConstructor(includeSuperProperties=true,
        callSuper=true)
@AutoClone(style=COPY_CONSTRUCTOR)
class Chef2 extends Person {
    final List<String> recipes
}
def name = 'Jamie Oliver'
def recipes = ['Lentil Soup', 'Crispy Duck']
def born = Date.parse('yyyy-MM-dd', '1975-05-27')
def c1 = new Chef2(name, born, recipes)
def c2 = c1.clone()
assert c2.name == name
assert c2.born == born
assert c2.recipes == recipes
```

The added methods generated for the Chef2 class look roughly like this:

```
protected Chef2(Chef2 other) {
    super(other)
    if (other.recipes instanceof Cloneable) {
        this.recipes = (List<String>) other.recipes.clone()
    } else {
        this.recipes = other.recipes
    }
}
public Chef2 clone() throws CloneNotSupportedException {
    new Chef2(this)
}
```

@GROOVY.TRANSFORM.AUTOEXTERNALIZE

Using @AutoExternalize for easier serialization

```
import groovy.transform.*
@AutoExternalize
@ToString
class Composer {
    String name
    int born
    boolean married
}
def c = new Composer(name: 'Wolfgang Amadeus Mozart',
        born: 1756, married: true)
def baos = new ByteArrayOutputStream()
baos.withObjectOutputStream{ os -> os.writeObject(c) }
def bais = new ByteArrayInputStream(baos.toByteArray())
def loader = getClass().classLoader
def result
bais.withObjectInputStream(loader) {
    result = it.readObject().toString()
}
assert result == 'Composer(Wolfgang Amadeus Mozart, 1756, true)'
```

Scripting support

@GROOVY.TRANSFORM.TIMEDINTERRUPT

Annotating a class with @TimedInterrupt sets a maximum time the script or instances of the class are allowed to exist. If the maximum time is exceeded then a Timeout- Exception is thrown. This annotation is designed to guard against runaway processes, infinite loops, or a maliciously long-running user script.

Using @TimedInterrupt to guard against slow scripts

```
import groovy.transform.TimedInterrupt
import java.util.concurrent.TimeoutException
import static java.util.concurrent.TimeUnit.MILLISECONDS
@TimedInterrupt(value = 480L, unit = MILLISECONDS)
class BlastOff1 {
    def log = []
    def countdown(n) {
        sleep 100
        log << n
        if (n == 0) log << 'ignition'
        else countdown(n - 1)
    }
}
def b = new BlastOff1()
Thread.start {
    try {
        b.countdown(10)
    } catch (TimeoutException ignore) {
        b.log << 'aborted'
    }
}.join()
assert b.log.join(' ') == '10 9 8 7 6 aborted'
```

@GROOVY.TRANSFORM.THREADINTERRUPT

For timely responsiveness, long-running user scripts should periodically check the Thread.currentThread().isInterrupted() status and throw an Interrupted- Exception when an interrupt is detected. But in practice, scripts are almost never written this way. An easy way to properly respect the interrupted flag is to use the @ThreadInterrupt annotation.

Using @ThreadInterrupt to detect interruptions

```
import groovy.transform.ThreadInterrupt
@ThreadInterrupt
class BlastOff2 {
    def log = []
    def countdown(n) {
        Thread.sleep 100
        log << n
        if (n == 0) log << 'ignition'
        else countdown(n - 1)
    }
}
def b = new BlastOff2()
def t1 = Thread.start {
    try {
        b.countdown(10)
    } catch (InterruptedException ignore) {
        b.log << 'aborted'
    }
}
sleep 590
t1.interrupt()
t1.join()
assert b.log.join(' ') == '10 9 8 7 6 aborted'
```

@GROOVY.TRANSFORM.CONDITIONALINTERRUPT

This annotation allows you to specify your own custom interrupt logic to be woven into a class. Like the others, the interrupt check occurs at the start of every method, the start of every closure, and each loop iteration.

Using @ConditionalInterrupt to set an automatic error threshold

```
import groovy.transform.ConditionalInterrupt

@ConditionalInterrupt({ count <= 5 })
class BlastOff3 {
    def log = []
    def count = 10

    def countdown() {
        while (count != 0) {
            log << count
            count--
        }
        log << 'ignition'
    }
}

def b = new BlastOff3()
try {
    b.countdown()
} catch (InterruptedException ignore) {
    b.log << 'aborted'
}
assert b.log.join(' ') == '10 9 8 7 6 aborted'
```

@GROOVY.TRANSFORM.FIELD

Using @Field for class-level instance variables in a script

```
import groovy.transform.Field
@Field List awe = [1, 2, 3]
def awesum() { awe.sum() }
assert awesum() == 6
```

The equivalent generated code would look like this:

```
class ScriptYYYYY extends Script {
    List awe = [1, 2, 3]
    public static void main(String[] args) {
        new ScriptYYYYY().run()
    }
    public awesum() {
        awe.sum()
    }
    public run() {
        assert awesum() == 6
    }
}
```

@GROOVY.TRANSFORM.BASESCRIPT

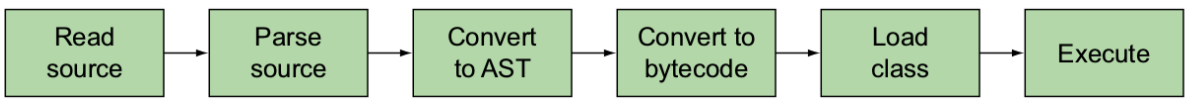
Annotating a script with @BaseScript lets you customize a script's parent class. Sup- pose you wanted all your scripts to save all printed lines to a log.

Using @BaseScript to customize a script's parent class

```
@BaseScript(LoggingScript)
import groovy.transform.BaseScript
abstract class LoggingScript extends Script {
    def log = []
    void println(args) {
        log << args
        System.out.println args
    }
}
println 'hello'
println 3 * 5
assert log.join(' ') == 'hello 15'
```

Exploring AST

An AST is a representation of your program in tree form. The tree has nodes that can have leaves and branches, and there's a single root node. Many compilers, not just Groovy, create an AST as a step toward a compiled program. In general and simplified terms, running a Groovy script is a multistep process:



AST by example: creating ASTs

Create By Hand

Creating AST objects by hand

```
import org.codehaus.groovy.ast.ClassHelper
import org.codehaus.groovy.ast.expr.*
import org.codehaus.groovy.ast.stmt.ReturnStatement
def ast = new ReturnStatement(
        new ConstructorCallExpression(
                ClassHelper.make(Date),
                ArgumentListExpression.EMPTY_ARGUMENTS
        )
)
assert ast instanceof ReturnStatement
```

Using the GeneralUtils helper class

```
import org.codehaus.groovy.ast.stmt.ReturnStatement
import static org.codehaus.groovy.ast.ClassHelper.make
import static org.codehaus.groovy.ast.tools.GeneralUtils.*
def ast = returnS(ctorX(make(Date)))
assert ast instanceof ReturnStatement
```

AstBuilder.buildFromSpec

Creating AST objects using buildFromSpec

```
import org.codehaus.groovy.ast.builder.AstBuilder
import org.codehaus.groovy.ast.stmt.ReturnStatement
def ast = new AstBuilder().buildFromSpec {
    returnStatement {
        constructorCall(Date) {
            argumentList {}
        }
    }
}
assert ast[0] instanceof ReturnStatement
```

AstBuilder.buildFromString

Creating AST objects using buildFromString

```
import org.codehaus.groovy.ast.builder.AstBuilder
import org.codehaus.groovy.ast.stmt.BlockStatement
import org.codehaus.groovy.ast.stmt.ReturnStatement

def ast = new AstBuilder().buildFromString('new Date()')
assert ast[0] instanceof BlockStatement
assert ast[0].statements[0] instanceof ReturnStatement
```

NOTE	The only knowledge required is that a script is a BlockStatement and that BlockStatement has a ReturnStatement in its statement list.

Trying to mix dynamic code with buildFromString

```
import org.codehaus.groovy.ast.builder.AstBuilder
import org.codehaus.groovy.control.CompilePhase
import org.codehaus.groovy.ast.*
def approxPI = 3.14G
def ast = new AstBuilder().buildFromString(
        CompilePhase.CLASS_GENERATION,
        false,
        'static double getTwoPI() { def pi = ' + approxPI + '; pi * 2 }'
)
assert ast[1] instanceof ClassNode
def method = ast[1].methods.find { it.name == 'getTwoPI' }
assert method instanceof MethodNode
```

AstBuilder.buildFromCode

Creating AST objects using buildFromCode

```
import org.codehaus.groovy.ast.builder.AstBuilder
import org.codehaus.groovy.ast.stmt.ReturnStatement
def ast = new AstBuilder().buildFromCode {
    new Date()
}
assert ast[0].statements[0] instanceof ReturnStatement
```

AST by example: local transformations

A local transformation relies on annotations to rewrite Groovy code.

```
class Greeter {
    @Main
    def greet() {
        println "Hello from the greet() method!"
    }
}
```

```
class Greeter {
    def greet() {
        println "Hello from the greet() method!"
    }
    public static void main(String[] args) {
        new Greeter().greet()
    }
}
```

Main.groovy

```
@Retention(RetentionPolicy.SOURCE)
@Target([ElementType.METHOD])
@GroovyASTTransformationClass(classes = [MainTransformation])
@interface Main {}
```

MainTransformation.groovy

```
@GroovyASTTransformation(phase = CompilePhase.INSTRUCTION_SELECTION)
class MainTransformation implements ASTTransformation {
    void visit(ASTNode[] astNodes, SourceUnit sourceUnit) {
        // perform any checks
        // construct appropriate main method
        // add main method to class
    }
}
```

IMPORTANT	You only need to use the ASTNode[] parameter. Element 0 contains the annotation that triggered the transformation and element 1 contains the ASTNode that was annotated.

Implementing the ASTTransformation for the @Main annotation

```
import static groovyjarjarasm.asm.Opcodes.*
import static org.codehaus.groovy.ast.ClassHelper.VOID_TYPE
import static org.codehaus.groovy.ast.tools.GeneralUtils.*

@GroovyASTTransformation(phase = CompilePhase.INSTRUCTION_SELECTION)
class MainTransformation implements ASTTransformation {
    private NO_EXCEPTIONS = ClassNode.EMPTY_ARRAY
    private STRING_ARRAY = ClassHelper.STRING_TYPE.makeArray()

    void visit(ASTNode[] astNodes, SourceUnit sourceUnit) {
        if (astNodes?.size() != 2) return
        if (!(astNodes[0] instanceof AnnotationNode)) return
        if (astNodes[0].classNode.name != Main.name) return
        if (!(astNodes[1] instanceof MethodNode)) return
        def targetMethod = astNodes[1]
        def targetClass = targetMethod.declaringClass
        def targetInstance = ctorX(targetClass)
        def callTarget = callX(targetInstance, targetMethod.name)
        def mainBody = block(stmt(callTarget))
        def visibility = ACC_STATIC | ACC_PUBLIC
        def parameters = params(param(STRING_ARRAY, 'args'))
        targetClass.addMethod('main', visibility,
                VOID_TYPE, parameters, NO_EXCEPTIONS, mainBody)
    }
}

new GroovyShell(getClass().classLoader).evaluate '''
class Greeter {
    @Main
    def greet() {
        println "Hello from the greet() method!"
    }
}
'''
```

AST by example: global transformations

Global transformations are similar to local transformations except that no annotation is required to wire-in a visitor. Instead of having the end user specify when your trans- formation is applied, global transformations are simply applied to every single source unit in the compilation.

```
println 'script compiled at: ' + compiledTime
class MyClass { }
println 'script class compiled at: ' + MyClass.compiledTime
```

CompiledAtASTTransformation

```
package transform

import org.codehaus.groovy.ast.ASTNode
import org.codehaus.groovy.ast.ClassNode
import org.codehaus.groovy.ast.MethodNode
import org.codehaus.groovy.ast.builder.AstBuilder
import org.codehaus.groovy.control.CompilePhase
import org.codehaus.groovy.control.SourceUnit
import org.codehaus.groovy.transform.ASTTransformation
import org.codehaus.groovy.transform.GroovyASTTransformation

import static groovyjarjarasm.asm.Opcodes.ACC_PUBLIC
import static groovyjarjarasm.asm.Opcodes.ACC_STATIC

@GroovyASTTransformation(phase=CompilePhase.CONVERSION)

class CompiledTimeAstTransformation implements ASTTransformation {
    private static final compileTime = new Date().toString()
    void visit(ASTNode[] astNodes, SourceUnit sourceUnit) {
        List classes = sourceUnit.ast?.classes
        classes.each { ClassNode clazz ->
            clazz.addMethod(makeMethod())
        }
    }
    MethodNode makeMethod() {
        def ast = new AstBuilder().buildFromSpec {
            method('getCompiledTime', ACC_PUBLIC | ACC_STATIC, String) {
                parameters {}
                exceptions {}
                block {
                    returnStatement {
                        constant(compileTime)
                    }
                }
                annotations {}
            }
        }
        ast[0]
    }
}
```

Test

```
package transform

class CompiledTimeAstTransformationTest extends GroovyTestCase {
    // matches format: EEE MMM dd HH:mm:ss zzz yyyy
    static DATE_FMT = /\w{3} \w{3} \d\d \d\d:\d\d:\d\d \S{3,9} \d{4}/

    @Override
    protected void setUp() throws Exception {
        super.setUp()
    }

    void testShouldApplyToThisTest() {
        assert compiledTime.toString() =~ DATE_FMT
    }

    void testShouldApplyToScriptAndScriptClasses() {
        assertScript '''
import static transform.CompiledTimeAstTransformationTest.*
assert compiledTime.toString() =~ DATE_FMT
class MyClass { }
assert MyClass.compiledTime.toString() =~ DATE_FMT
'''
    }
}
```