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`.

Using scripts

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
}
```