

2

The Groovy Essentials

[Per our phone conversation, please find places where you can create diagrams to illustrate relationships between concepts. -Tom]

This chapter covers

- Introducing Groovy
- Differences between Java and Groovy
- Groovy's dynamic features

Grails is a great framework that removes much of the drudgery from the development of web applications. You have already seen how easy it is to get a usable application up and running from scratch. A large part of the productivity gain comes from using Groovy as the main language for Grails development.

This chapter aims to be a primer for Java developers so that you can easily follow the rest of the book. The pace is fast, but you don't have to understand everything in one go. In fact, we encourage you to refer back to this chapter as your experience grows and you need to dig a bit deeper into Groovy. Even if you are familiar with the language, we talk about some subtleties and pitfalls that you may not be aware of.

We begin by covering the many small differences between Groovy and Java, most of which amount to simple variations in the syntax. We then highlight the Java features that are missing from Groovy (there are only a few), before moving on to closures and the dynamic nature of the language. Even after a quick read through, you will know enough to follow the examples in the book and write your own code. After that, there is enough information for you to become a very competent Groovy (and Grails!) developer.

2.1 An introduction

We could go straight into explaining how Groovy and Java differ and show you some example code, but there is nothing like getting your hands dirty to get a real feel for a language and to learn more quickly. With that in mind, we will introduce you to a couple of Grails commands that allow you to experiment with Groovy easily and safely. We will follow that up with a few minor but nonetheless important differences between the two languages that will enable you to follow the rest of the examples in the chapter.

2.1.1 Let's play!

You have Grails installed and now you want to try out this outrageously named programming language called Groovy with a minimum of fuss. What can you do? Well, you saw in the first chapter how you can use "grails create-app" to create a Grails project. There are also two commands available that provide interactive environments for running Groovy code. In fact, as you will see in later chapters, they can be used to interactively play with your Grails application.

Grails and Groovy versions

. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=493>

This book targets Grails version 1.1, which comes with its own copy of Groovy 1.6, so that's the version of Groovy that this chapter is based on. Most of the information is also applicable to Groovy 1.5, but some code may not work.

The first command is “grails shell”. This starts an interactive shell or command prompt in which you can type Groovy expressions and see the results. Figure 2.1 shows an example session. As you can see, to exit the shell simply type “quit” on its own. The shell is great for expression evaluation and checking what the result of a method call might be, and it is even powerful enough that you can define new classes from within it. However, it is impossible to edit code that you have already written, so it can easily become frustrating to use.

Figure 2.1 A session with “grails shell”

```

pal20@sycamore: ~/dev/projects/scratch/test-console
Groovy Shell (1.5.6, JVM: 1.5.0_15-b04)
Type 'help' or '\h' for help.
-----
groovy:000> 1 + 1
=> 2
groovy:000> println 'Hello, world!'
Hello, world!
=> null
groovy:000> list = new ArrayList()
=> []
groovy:000> list.add(4)
=> true
groovy:000> list.add(7)
=> true
groovy:000> list
=> [4, 7]
groovy:000> list.reverse()
=> [7, 4]
groovy:000> quit
org.codehaus.groovy.tools.shell.Groovysh
pal20@sycamore ~/dev/projects/scratch/test-console$

```

“grails console” command, is a Swing application that makes it easy to write simple scripts and run them. It consists of two panes: the upper one contains the Groovy code you want to execute, while the one below shows the output from running the script. The console’s biggest advantage over the shell is that you can change a script and immediately run it with those changes. Figure 2.2 shows the console application in action with a script and the output from its execution. See how you get syntax highlighting for free! You can also save the script and load it back in at a later date.

Figure 2.2 The Grails console in action

```

class Test {
    private Set items

    boolean isEmpty() {
        return this.items
    }

    void addItem(Object item) {
        if (this.items == null) this.items = new HashSet()

        this.items.add(item)
    }
}

test = new Test()
assert test.isEmpty()

test.addItem("Hello, world")
assert !test.isEmpty()

println "Done!"

groovy> this.items.add(item)
groovy> ]
groovy> }
groovy> test = new Test()
groovy> assert test.isEmpty()
groovy> test.addItem('Hello, world')
groovy> assert test.isEmpty()
groovy> println "Done!"

Done!

Execution complete. Result was null. 13:2

```

Groovy and Java that we are about to cover, but before we do that we should explain what scripts are.

WHAT'S THIS SCRIPT BUSINESS, THEN?

In order to run Java code directly, you must create a class and define a static “main” method on it. This doesn't sound particularly onerous, but it is a surprisingly effective disincentive to writing simple applications. To remove this hurdle, Groovy supports scripts: a collection of statements in a text file that are executed in order. You can think of them as the body of a “main” method without the class and method definitions. Try this very simple script in the Grails console:

```
println "Hello world!"
```

Either use the “Script > Run” menu item or the “Ctrl-R” keyboard shortcut to execute the script. That's it! You have now written and executed a simple Groovy script. As you can see in figure 2.2, you can even include class definitions.

Now that you know how to use the console and are therefore ready to test the examples as we go, let's take a look at some of the basic language features that you need to be aware of. These are used throughout the rest of the chapter and book.

2.1.2 The first steps

With a few exceptions, valid Java code is also valid Groovy. However, writing *idiomatic* Groovy means taking advantage of some of its extra features. We start with some familiar concepts that vary slightly compared to Java. Fewer imports

As you know, all classes in the `java.lang` package are effectively automatically imported by Java. Groovy extends this behaviour to:

- `java.io`
- `java.math`
- `java.net`
- `java.util`
- `groovy.lang`
- `groovy.util`

This is a real god-send when writing scripts because you often end up using classes from these core packages extensively, and it gets pretty tedious running a script only to find you missed out one of those imports.

ASSERTIONS

Many of the examples in this chapter make use of the `assert` keyword:

```
assert 1 == 1
assert 'c' != 'z'
```

All it does is throw a `java.lang.AssertionError` if the corresponding expression evaluates to `false` (according to Groovy Truth, which we look at next). You should be familiar with it from Java, but be aware that although it has similar semantics to the Java version, it can not be switched on and off via the `-ea` argument to the JVM.

THE TRUTH...

With an expression like “Groovy Truth”, you may rightly be expecting the answer to life, the universe, and everything. Sadly, you will have to find that elsewhere. The actual meaning is far more prosaic and relates to what Groovy views as `true` or `false`. In Java, you can only use booleans and object references in conditions (the expression `!obj` evaluates to `true` if and only if `obj` is `null`). This makes testing for an empty set, for example, quite verbose:

```
if (mySet == null || mySet.isEmpty()) {
    ...
}
```

Groovy helps to reduce this verbosity and thus improve readability by coercing other objects to booleans. For example, a `null` reference and an empty set will both evaluate to `false`, and so the above code would look like this:

```
if (!mySet) {
```

. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=493>

```

    ...
}

```

Table 2.1 lists the coercions used by Groovy Truth. Note that the null object reference coercion applies to all the object types listed, so both a null map and an empty map will evaluate to `false`.

Table 2.1 Groovy Truth

Type	Evaluates to false
Boolean	<code>false</code> , <code>Boolean.FALSE</code>
Object reference	<code>Null</code>
Number	<code>0</code>
String / GString	empty string
Collection	empty collection
Map	empty map
Iterator	<code>hasNext()</code> returns <code>false</code>
Enumeration	<code>hasMoreElements()</code> returns <code>false</code>
Regex Matcher	<code>find()</code> returns <code>false</code>

Be careful when relying on Groovy Truth. For example, if you know you are testing a collection and you want to treat a null reference the same as an empty collection, then using Groovy Truth is appropriate. However, if you are only interested in whether a reference is null or not, make the check explicit, i.e. `obj == null`. This has the advantage of making the intent clear to anyone reading your code. It also avoids problems when the code finds itself unexpectedly testing something like a collection or map: an empty collection is definitely not null but does evaluate to `false` as we have seen.

“PUBLIC” IS THE DEFAULT SCOPE

As you know, Java has four scopes: `private`, `protected`, `public`, and the default (package) scope. The designers of Groovy decided that the last of these is redundant, and so it only supports the first three. This in turn led to the decision to make “public” the default scope, since it is the most commonly used. In other words, if a class, field, or method has no explicit scope declared, it is automatically “public” rather than “package”.

CHECKED EXCEPTIONS DON'T HAVE TO BE CAUGHT

Error-handling is very nearly exactly the same in Groovy as it is in Java. Syntactically, exceptions are treated identically using `try...catch...finally` blocks. The difference is that you do not have to catch checked exceptions in Groovy. In other words, checked exceptions are for all intents and purposes the same as runtime exceptions.

On the whole this results in less boilerplate code, since you only catch the exceptions you know what to do with. However, there are occasions when it is important not to forget that you are dealing checked exceptions. As an example, you will see later that Spring's default transaction behaviour only works with runtime exceptions. If that doesn't mean anything to you at the moment, don't worry – just remember that checked exceptions are still checked exceptions even if you don't have to catch them!

That wasn't so bad, was it? We now have the foundations to move on to more substantial differences, all of which are geared towards making your life easier.

2.1.3 Some new operators

You are undoubtedly familiar with operators such as `+`, `-`, `.`, etc. Groovy supports all of the same operators as Java, but it also introduces a few new ones that genuinely make your life easier and your code easier to read.

. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=493>

? - NULL-SAFE OBJECT NAVIGATION

Do you ever get annoyed with having to check whether a reference is null before calling a method on it? The Groovy developers do, and so they added a null-safe `?.` operator to the language. Take this common idiom from Java which checks whether an object or one of its properties is null:

```
if (obj != null && obj.getValue() != null) {
    ...
}
```

In Groovy, you can write this as:

```
if (obj?.getValue() != null) {
    ...
}
```

The operator is `?.` and in the example, `getValue()` is called only if `obj` is not null. If `obj` is null, the expression `obj?.getValue()` itself evaluates to null – no `NullPointerException` will be thrown!

***. - THE “SPREAD-DOT” OPERATOR**

No operator is better disguised by its name than “spread-dot”. What does it do? Surprisingly, something very useful: it calls a method on every item in a collection and creates a new list containing the results of those methods. As an example, imagine you have a list of objects that all have a `getName()` method; the following code will create a new list containing the names of the original items:

```
List names = people*.getName()
```

The resulting list has the same order as the original, so in the above example the names will be in the same order as the people. The operator can also be used on other types of collection, but the result is always a list and its items are always in the same order as the iteration order of the original collection.

We could also use property notation, which is described in a later section:

```
List names = people*.name
```

This is probably the most common use of the operator you will see in both examples and real code.

<=> - COMPARISONS WITH A SPACESHIP

Java has plenty of operators for comparing two values, so why should we want another one? If you have ever implemented the `Comparable.compareTo()` or `Comparator.compare()` methods, you will know why. Here is a naïve implementation of a comparator for integers:

```
public int compare(int i1, int i2) {
    if (i1 == i2) return 0;
    else if (i1 < i2) return -1;
    else return 1;
}
```

Although you can reduce this to a one-liner for numbers, it’s not so easy for other types. So what would the method look like if we had an operator with identical behaviour to the `compare()` method? Let’s see:

```
public int compare(int i1, int i2) {
    return i1 <=> i2;
}
```

That’s the Groovy “spaceship” operator, which can be used on any types that implement `Comparable`. It is particularly useful for custom comparators and, in combination with some other features of Groovy that we will see later, sorting lists.

“==” MEANS “EQUALS()”

When you use `==` in Groovy, the generated code uses the `equals()` method. It is also null-safe, which means that it will work if either or both of the operands are null. The following example demonstrates the behaviour using asserts and strings:

```
String str = null
assert str == null
assert str != "test"

str = "test"
assert str != null
assert str == "test"

str += " string"
assert str == "test string"
```

This isn’t the whole story, and in fact there is an exception to this rule: if the object on the left-hand side implements `java.lang.Comparable`, then `==` uses the `compareTo()` method rather than

`equals()`. In this case, the result is `true` if the `compareTo()` method returns zero, or `false` otherwise.

In case you're wondering why Groovy does this, remember that `0.3` and `0.30` are *not* equivalent according to the `BigDecimal.equals()` method, whereas they are with the `compareTo()` method. Again, the principle of least surprise suggests that the expression `0.3 == 0.30` should evaluate to `true`, which is exactly what happens in Groovy.

Lastly, how do you get the Java behaviour? Use the `is()` method:

```
BigDecimal x = 0.234
BigDecimal y = x

assert y.is(x)
assert !y.is(0.234)
assert y == 0.234
```

As you can see, it will only return `true` if two objects are the same instance.

Although these operators may be unfamiliar now, it won't be long before you are confidently using them in your own code. So with that gentle introduction out of the way, it is time to look at the type system.

2.2 Types

As with most things in Groovy, if you start from the assumption that its type system is the same as Java's, you won't go far wrong. Nonetheless, there are some key differences between the two languages when it comes to types and their handling, which you need to be aware of the unlock the full potential of the language.

2.2.1 Basic types

ALL THE TYPES YOU EXPECT FROM JAVA ARE THERE IN BOTH PRIMITIVE AND OBJECT FORM: BOOLEAN, INT, DOUBLE, STRING ETC. THIS MAKES BOTH TRANSITION TO THE LANGUAGE AND INTEGRATION WITH JAVA RELATIVELY EASY. BUT THE TYPE SYSTEMS ARE NOT IDENTICAL. TAKE PRIMITIVE TYPES: EVEN IF YOU DECLARE A VARIABLE AS A PRIMITIVE TYPE, YOU CAN STILL TREAT IT AS AN OBJECT, FOR EXAMPLE BY CALLING METHODS ON IT. THIS IS MOST OBVIOUS IN THE HANDLING OF NUMBERS.

Not only can primitive variables be treated as objects, but this behaviour also extends to numeric literals. So `1` is an instance of `Integer` and `546L` is an instance of `Long`. Here we show how you can call a method on an integer literal:

```
assert (-12345).abs() == 12345
```

A subtle but notable feature in Groovy is that floating point arithmetic is based on `BigDecimal` rather than `double` or `float`. At first this may seem strange, but Groovy aims for the principle of least surprise and one of the most surprising things in Java is that `0.1D * 3` is `0.300...04` rather than `0.3`. Since Groovy is not targeted at high-performance floating-point calculations, it prefers to use `BigDecimal` so that `0.1 * 3` is `0.3`.

You may be worried about having to use the `BigDecimal` API, but fear not! Groovy helpfully makes the normal arithmetic operators work transparently with the class. Here's a quick example:

```
assert 0.25 instanceof BigDecimal
assert 0.1 * 3 == 0.3
assert 1.1 + 0.1 == 1.2
assert 1 / 0.25 == 4
```

Anyone that has tried to perform `BigDecimal` division in Java will truly appreciate the simplicity on display.

GROOVY STRINGS

You have just seen that Java string literals also work in Groovy. However, there are significant differences. First, Groovy has no character literal like Java. Single-quotes (') delimit string not character literals. So what is the difference between single-quotes and double-quotes? The latter allow you to embed Groovy expressions in the string. This means that rather than manually concatenating strings as in Java, you can use code like this:

```
String name = "Ben"
String greeting = "Good morning, ${name}"
assert greeting == 'Good morning, Ben'

String output = "The result of 2 + 2 is: ${2 + 2}"
assert greeting == "The result of 2 + 2 is: 4"
```

. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=493>

Anything inside the `${}` will be evaluated, converted to a string via the `toString()` method, and inserted into that position. If your string contains dollar symbols, then you can either use the single-quote form, or escape them using a backslash (“\`$`”). This type of string is known as a GString.

Both forms of string literal have corresponding *multi-line* versions. These allow you to easily create string literals that include line breaks. If you have ever tried to build these strings in Java, you will understand just how big a deal this is. Take for example a simple method that returns the body of an e-mail given a person’s name:

```
String getEmailBody(String name) {
    return "" "Dear ${name},
```

```
    Thank you for your recent enquiry. One of our team members
    will process it shortly and get back to you. Some time in
    the next decade. Probably.
```

```
    Warmest and best regards,
```

```
    Customer Services
    ""
}
```

See how we have embedded the variable “name” inside the string? Simple and elegant. One word of warning though: any whitespace that appears between the triple-quotes will be included in the string. In other words, make sure that each line starts in column one of the file unless you actually *want* the text indented in the resulting string.

That’s pretty much all you need to know about strings in Groovy. There is one more thing, though. By now, you are probably wondering how on earth you get a char literal. There are two options: either assign a single character string to a char variable, or use the `as` keyword. Here we provide examples of both:

```
char ch = 'D'
assert ch instanceof Character

String str = "Good morning Ben"
str = str.replace(' ' as char, '+' as char)
assert str == "Good+morning+Ben"
```

Although we use single-quotes for our character literals in the above example, you can equally use double-quotes. It comes down to user preference, but see how the above example is almost valid Java? Remove the two “as char” elements and you have yourself pure Java code! The other advantage of using single-quotes for character literals and double-quotes for string literals is that plain Java developers will be more comfortable with the code – a benefit not to be sneezed at.

So what does that `as` keyword do?

YOUR FLEXIBLE FRIEND: “AS”

In Java, you can cast an object to a type as long as the object is actually an instance of that type. Groovy supports the Java cast, but as you will find out it’s redundant in a dynamically typed language. Something that *would* be useful is a means of easily converting between types.

Imagine you have the string “12345”. It looks like a number, but as far as Java and Groovy are concerned, it’s just a string. To convert it to a number in Java, you have to use `Integer.parseInt()` or a similar method. That’s all well and good, but what if you want to convert an array to a list? For that you can use `Arrays.asList()`. There are other conversions that you can perform in Java, but you have to find the appropriate method – something that’s not always trivial.

Like a knight in shining armour, Groovy comes to the rescue with its “as” operator. This allows you perform conversions like string to number or array to list using a consistent syntax. As an added bonus, your code will tend to be easier to read because “as” is implemented as an operator. Here are some examples of its syntax:

```
String numString = 543667 as String
List characters = "Good morning!".toCharArray() as List

assert 1234.compareTo("34749397" as int) < 0
```

. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=493>

Groovy supports a collection of conversions by default, a subset of which we have listed in table 2.2. You may not understand all of them at this stage, but we will come across many of them later in the book.

Table 2.2 Conversions supported by “as”

Source Type	Target Type	Description
String	Number (int, double, Long, BigDecimal, etc.)	Parses a string as if it were a number.
List	Array	Groovy does not support Java's array literals, so “as” can be used to convert a list to an array, e.g. <code>myList as String[]</code> .
Anything	boolean	Uses Groovy Truth to convert a value to either true or false.
Collection	List	Creates a list that is a copy of the original collection. The items in the list are in the iteration order of the original collection.
String	List	Treats a string as a sequence of characters and thus turns it into a list.

You can also add custom conversions by implementing the `asType()` method on a class. We won't go into the details here, but Grails adds some extra conversions itself and it's useful to know how it does it.

That's pretty much all there is as far as the basic types go. Hopefully you will find GStrings and type coercion as useful as we do. Certainly they are more elegant alternatives to string concatenation and casting. Yet these improvements are minor compared to Groovy's collection and map support.

2.2.2 Collections, maps, and ranges

These types are so common that some other languages have them built into their type system. And yet Java only has limited support for them - they seem to be almost an afterthought. Groovy goes some way towards rectifying this situation by effectively treating them as first class citizens at the language level.

LITERALS

If you have ever tried to build up a list or map of reference data in Java, you know how painful the experience is. You first have to create an instance of the collection, and then add each item explicitly. Groovy makes life much easier in this respect with its list and map literals. A list literal is a comma-separated sequence of items between square brackets, and a map literal is the same, but in this case each “item” is a key and value separated by a colon. Listing 2.1 compares the Groovy and Java syntax for creating lists and maps, while at the same time showing the list and map literals in action.

Listing 2.1 Initializing lists and maps

```

List myList = new ArrayList()           #A
myList.add("apple")                     #A
myList.add("orange")                    #A
myList.add("lemon")                     #A
                                         #A
Map myMap = new HashMap()               #A
myMap.put(3, "three")                    #A
myMap.put(6, "six")                      #A
myMap.put(2, "two")                      #A

List myList = [ "apple", "orange", "lemon" ] #B
Map myMap = [ 3: "three", 6: "six", 2: "two" ] #B

List l = []                              #C
Map m = [:]                              #C
#A Java-style lists and maps

```

. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=493>

#B Groovy list and map literals**#C Empty list and map**

These literals are actually just instances of `java.util.List` and `java.util.Map`, so they have all the methods and properties you would expect. For example:

```
assert 3 == [ 5, 6, 7 ].size()
```

ARRAY-STYLE NOTATION FOR LISTS AND MAPS

Java has a rather clunky syntax for accessing items in lists and maps because it relies on methods. Groovy improves the situation dramatically by extending Java's array notation to both types. This is best understood by example and listing 2.2 demonstrates getting and setting values with both types.

Listing 2.2 Accessing list and map elements

```
List numbers = [ 5, 10, 15, 20, 25 ]
assert numbers[0] == 5 #A
assert numbers[3] == 20 #A

assert numbers[-1] == 25 #1
assert numbers[-3] == 15 #1

numbers[2] = 3 #B
assert numbers[2] == 3

numbers << 30 #2
assert numbers[5] == 30

Map items = [ "one": "apple",
             "two": "orange",
             "three": "pear",
             "four": "cherry" ]
assert items["two"] == "orange" #C
assert items["four"] == "cherry" #C

items["one"] = "banana" #D
assert items["one"] == "banana"

items["five"] = "grape" #E
assert items["five"] == "grape"
```

Cueballs in code and text**#1 Index from end of list****#2 Add item to list****No cueball****#A Get item from list****#B Replace item in list****#C Get item from map****#D Replace item in map****#E Add item to map**

As you can see, Groovy makes using lists and maps pleasant and intuitive. Worthy of particular note is the ability to index lists (and arrays!) from the end rather than the beginning, as demonstrated by #1. The example also shows the “left-shift” operator (#2) in action. This is the standard way of appending items to a list in Groovy.

A NEW TYPE: RANGE

Somewhat related to lists is Groovy's *range* type. In fact, all range objects implement the `java.util.List` interface. A full account of the capabilities of this type is outside the scope of this book, but it is important that you can recognize integer range literals and understand some of their uses.

In essence, a range is the combination of a lower and an upper bound, which in the case of an integer range represents all numbers between the two bounds. The lower bound is always inclusive, but you can

. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=493>

specify ranges with either an inclusive or exclusive upper bound. Listing 2.3 shows code that iterates over a list of strings, printing each item in the list to the console, and then printing just a sub-set of the items.

Listing 2.3 Ranges

```
List fruit = [
    "apple",
    "pear",
    "lemon",
    "orange",
    "cherry" ]

for (int i in 0..<fruit.size()) {
    println "Fruit number $i is '${fruit[i]}'"
} #1

List subList = fruit[1..3]
for (int i in 0..<fruit.size()) {
    println "Sub list item $i is '${subList[i]}'"
} #2
```

Cueballs in code and text

#1 Iterate through an exclusive range

#2 Extract a list slice

The listing demonstrates the two most common uses for ranges: as something to iterate over (#1), and as an argument to the array accessor (#2). You may not be familiar with the `for` loop shown at #1, but don't worry. The code is simply iterating over the integers from 0 to 4 inclusive (there are 5 items in the list), assigning the number on each iteration to the variable `i`. In this case the "exclusive upper bound" notation is being used, i.e. `lower..<upper`. Also note that you can use variables and expressions as range bounds.

#2 shows how you can get a "slice" of a list, or in Java terms a sub-list. Here we have used the "inclusive upper bound" notation to extract items two through four from the list of fruit. Note that any changes to the original list will be reflected in the slice! In other words, if you were to change the third fruit from "lemon" to "lime", then the second item of the slice would also be "lime".

If all that hasn't got you excited, then either you are coming from a language that already has rich support for collections and maps or nothing will! Those of you coming from Java will wonder just how you managed without these features. And so we end our coverage of the type system. We still have substantial matters to deal with, but first we look at some features that are difficult to pigeon-hole.

2.3 Tying up loose ends

Sometimes you always end up with pesky information that is difficult to categorise. So before we move on to the most significant features of Groovy, let's take a look at some that are less significant but still noteworthy. We start with the seemingly innocuous semi-colons at the end of statements.

2.3.1 Who needs semi-colons?

One of the most obvious differences between Java and Groovy is that the latter does not require semi-colons at the end of lines, or to put it more accurately, at the end of statements. If you want to put more than one statement on a single line, you must still separate them with semi-colons as in Java. You should also be aware that this flexibility comes at a cost. You must be careful how you break statements across multiple lines, as some code arrangements will cause compiler errors. Listing 2.4 shows some examples of invalid and valid line breaks:

Listing 2.4 Valid and invalid line breaks

```
private void invalidMethod #A
    (String name) {
    String fruit = "orange, apple, pear, " #B
        + "banana, cherry, nectarine"
```

. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=493>

```

}

private int validMethod(                               #C
    String name) {
    String furniture = "table, chair, sofa, bed, " +   #D
        "cupboard, wardrobe, desk"

    String fruit = "orange, apple, pear, " \         #1
        + "banana, cherry, nectarine"

    int i = 0; i++; return i;                         #E
}

```

Cueballs in code and text

#1 Line continuation

No cueball

#A Invalid line break in method
#B Invalid line break in concatenation
#C Valid line break in method
#D Valid break in concatenation
#E Multiple statements

You can get round these limitations by using a line continuation (#1) that effectively says “this line hasn’t ended yet”. You may be tempted to insert semi-colons to achieve the same result, but that won’t work. For multi-line statements, Groovy parses the end of lines before it ever “sees” the relevant statement terminator so it will still throw an error.

At this stage you are probably wondering what the underlying rules are that determine whether a given formatting is valid or not. It comes down to Groovy making an educated guess as to whether a line is a valid statement or not. If you are interested, you can find out when and why these errors occur, but we think it is easier to be aware of the problem and simply fix any compilation errors as and when you come across them.

...OR PARENTHESES?

You will soon notice that method calls don’t seem to need parentheses. The classic example is the `println()` method provided by the Groovy JDK (see later):

```
println "Hello, world!"
```

Method calls that take a closure as a single argument (again, see later) also usually lack parentheses due to the improved readability. However, you *must* use parentheses for zero-argument method calls or if the first argument is a list or map literal:

```
println()
println([1, 2, 3, 4])
```

There are probably other instances where parentheses are required by Groovy, so we recommend that you use them in most cases, leaving them out only in a small set of common cases (such as `println()` and with closures). This has the added advantage that your code will be easier for Java developers to read.

YOU DON’T NEED RETURN STATEMENTS EITHER?

It may seem strange, but the `return` statement is unnecessary in Groovy. If no `return` is present, a method will return the result of the last expression evaluated. For example, the following method returns the result of `value + 1`:

```
int addOne(int value) { value + 1 }
```

You may be wondering why this feature exists. Simple: conciseness. It is particularly useful when inlining closures, which you will see later. This doesn’t prevent you in any way from using `return` if you wish, and sometimes including it is actually more readable for Java-attuned eyes.

Optional elements like these need to be discussed, but they’re hardly satisfying are they? You need something to really sink your teeth into and we have just the thing.

. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=493>

2.3.2 Native regular expressions

Regular expression support was introduced into the core Java API with version 1.4. Groovy builds on this and provides support for regular expressions at the language level. It provides new literals, new operators for regular expression matching, and a nice syntax for extracting groups, all of which make working with regular expressions a more productive experience.

LITERALS

Let's start with the new form of string literal that uses slashes ('/'). Why do we need or even want another type of string literal? Regular expressions use backslash ('\') as an escape character, for example '\b' represents a word boundary. Unfortunately, Java's string literals also use backslash as an escape character, so regular expressions in Java can quickly enter "backslash hell". As an example, take this regular expression that simply matches a backslash followed by a word character: "\\w".

The slash-based strings introduced by Groovy are a solution to this problem. They do not use backslash as an escape character except when you need a literal slash, thus making regular expressions much easier to write. You should also be aware that these string literals are GStrings, so you can embed Groovy expressions. For example, the regular expression /\${name}: \w+\/ matches the value of the name variable followed by a colon and then a sequence of word characters, e.g. "Peter: /test/".

If you prefer to deal explicitly with `java.util.regex.Pattern` instances, you can create them by prefixing any string literal with a `~`:

```
assert ~"London" instanceof java.util.regex.Pattern
assert ~/\w+/ instanceof java.util.regex.Pattern
```

MATCHING

Literal patterns aren't the only language level support Groovy provides for regular expressions. It also introduces two new operators for matching that should be familiar to you if you know Perl. The first of these is `==~` which returns a `java.util.regex.Matcher` that matches the string on the left-hand-side against the pattern on the right-hand-side. In other words:

```
myString ==~ /\w+/
```

is equivalent to:

```
Pattern.compile("\\w+").matcher(myString)
```

The second operator, `==~`, returns a boolean whose value is true if and only if the string on the left-hand-side matches the given pattern *exactly*. This begs the question: how do you easily check whether *any part* of the string matches? The Perl version of `==~` does exactly this and Groovy has a neat trick for mimicking the behaviour. It will automatically coerce a `Matcher` to a boolean if required, for example if the expression is used as a condition. This coercion involves calling the `find()` method on the matcher, which returns true if the pattern matches any part of the target string. The assertion in this example forces the coercion to a boolean:

```
String str = "The rain in Spain falls mainly on the plain"
assert str ==~ /[\\w\\s]+ plain$/
```

GROUPS

Some regular expressions include capturing groups. Groovy treats matches with groups as multi-dimensional arrays, so you can use array notation. The example in listing 2.5 extracts all words that contain "ain", printing out the prefix and suffix of each match. When run, you will see this printed to the console:

```
Found: 'rain' - prefix: 'r', suffix: "
Found: 'Spain' - prefix: 'Sp', suffix: "
Found: 'mainly' - prefix: 'm', suffix: 'ly'
Found: 'plain' - prefix: 'pl', suffix: "
```

List 2.5 Regular expression capture groups

```
static void main(String args) {
    String str = "The rain in Spain falls mainly on the plain"
    Matcher m = str ==~ /\b(\w*)ain(\w*)\b/           #A

    if (m) {                                         #B
        for (int i = 0; i < m.count; i++) {         #C
```

. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=493>

```

        println "Found: '${m[i][0]}' - prefix: '${m[i][1]}' +      #1
              ", suffix: '${m[i][2]}'"
    }
}
}

```

Cueballs in code and text

#1 Access whole match and sub-groups

No cueball

#A Match with two capture groups

#B Coerce Matcher to boolean

#C Iterate through all matches

You can see how the slash-literals, match operator, and groups come together to make using regular expressions concise and almost trivial. The only tricky part is understanding the array-notation (#1).

The first array index refers to the match itself. In the listing you can see that the pattern has four matches: "rain", "Spain", "mainly", and "plain". So, if the first array index has a value of 1, we are dealing with the match "Spain". The second index allows you to access specific groups within a match, but be careful. An index value of zero refers to the whole of the match, e.g. "Spain" or "mainly". The capture groups can be accessed by index values starting with 1, which is why we use 1 to get the prefix and 2 to get the suffix (#1).

That's it for the regular expression support. We recommend that you take full advantage of it in your own code if you can, but remember that you can always fall back to using the Java classes and methods if you aren't comfortable with the feature. We certainly think this feature is a true blessing for day-to-day development. In fact, the main thing you have to be wary of is using regular expressions too much!

We only have one more minor feature left to discuss, but it is an important one because almost all Groovy code in the whole universe uses it. OK, maybe that's an exaggeration - but not much of one.

2.3.3 Property notation

The JavaBeans specification introduced Java programmers to the concept of an *object property*, but left us with an unwieldy syntax: "getter" and "setter" methods. Groovy rectifies this problem by allowing us to use field access notation for JavaBean properties.

Take the `java.util.Date` class. It has the methods `long getTime()` and `void setTime(long milliseconds)`, which conform to the JavaBeans specification and represent a long property named "time". In Groovy you can access this property as if it were a field:

```

Date now = new Date()
println "Current time in milliseconds: ${ now.time }"

```

```

now.time = 103467843L
assert now.time == 103467843L

```

Once you start using this syntax, you won't look back. It is far more natural than calling the methods, requires less typing, and is more readable. What more can we say? Of course, using properties is all well and good, but you still have to define those "getter" and "setter" methods don't you?

DEFINING PROPERTIES

Writing those "getter" and "setter" methods is so laborious that IDEs will auto-generate the code for you. Even so, your class still ends up polluted with boilerplate methods that don't contain any real logic. Thankfully, Groovy provides a shortcut. If you declare a field without any scope, Groovy will automatically make the field private and generate the "getter" and "setter" methods behind the scenes. You can even override those methods if you wish. All of the fields in the following example are properties except for the last two:

```

class MyProperties {
    static String classVar

    final String constant

```

. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=493>

```

String name

public String publicField
private String privateField
}

```

That's it - nothing could be easier. Note that for boolean properties, Groovy will even generate an `is...()` method. All that's left now is a trick for instantiating and initialising beans

INSTANTIATING BEANS

When you define a bean, it must have a default constructor. Groovy uses that requirement to provide a shortcut for initialising instances of such classes. Simply provide a list of *named arguments* matching the properties that you wish to set. The next example creates a new instance of `java.text.SimpleDateFormat` using this approach:

```

DateFormat format = new SimpleDateFormat(
    lenient: false,
    numberFormat: NumberFormat.getIntegerInstance(),
    timeZone: TimeZone.getTimeZone("EST"))

```

The named arguments look like a map without the square brackets, in which the key is the name of a property on the class. In fact, behind the scenes Groovy does use a map and so you could even pass one yourself to the constructor. This is a common idiom when creating domain classes from web request parameters, as you will see.

2.3.4 Is anything missing?

As we have already stated, the majority of Java code is also valid Groovy. A few constructs are missing, though, so it is important to be aware of them.

CHARACTER LITERALS

We showed in the previous section that single-quotes are used as string delimiters, and so Groovy has no character literal. However, you can coerce single character strings to `chars` using the `"as"` keyword.

JAVA "FOR" LOOP

Groovy has limited support for the standard Java `"for"` loop. You can not use the `","` operator, so this works:

```
for (int i = 0; i < 10; i++) { ... }
```

but this does not:

```
for (int i = 0, j = 0; i < 10; i++, j++) { ... }
```

DO...WHILE

There is no `do...while` loop, but we doubt that anyone will mourn its passing. We have rarely, if ever, seen it in the wild and you can always use `while` and `for` loops to get the same effect.

INNER AND ANONYMOUS CLASSES

You can not declare inner or anonymous classes. Support for these is coming, but on the whole both closures and the ability to declare more than one class in a Groovy file mean that this is only a real issue with libraries and frameworks that heavily depend on them.

2.4 The tricky stuff

You should now be fairly confident in your ability to understand and write Groovy code. So far, the majority of differences between Groovy and Java have fallen under the heading of *"syntactic sugar"*. We now move on to two very important concepts that separate Groovy from Java on a more fundamental level: closures and Groovy's dynamic nature.

2.4.1 Closures

You are probably familiar with the `sort()` method in `java.util.Collections` for sorting a list using a given comparator. Listing 2.6 shows a simple fragment of code that will sort a list of strings, ignoring case. You should see this in the console:

```
Sorted fruit: apple, Avocado, cherry, Orange, pear
```

Listing 2.6 Java sorting

. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=493>

```

List<String> fruit = Arrays.asList(
    "apple", "Orange", "Avocado",
    "pear", "cherry");

Collections.sort(fruit, new Comparator<String>() {
    public int compare(String str1, String str2) {
        return str1.compareToIgnoreCase(str2);
    }
})

System.out.println("Sorted fruit: " + fruit);

```

Notice how the comparator is a single method implementation? In fact, all we really want to do is pass a two-argument function to the `sort()` method. This just isn't possible in Java, so you need an interface to "host" the function and an object to implement it. Fortunately for us, the JDK already provides the `java.util.Comparator` interface and anonymous classes; but what if you were to write an algorithm that traverses a tree and performs a function on each leaf node? You would have to write an interface to "host" the function and pass an object that implements that interface to the algorithm.

This seems like more work than it should be and even anonymous classes leave one aching for a more elegant alternative. As usual, Groovy offers a solution: *closures*. You can think of these as anonymous functions that can be assigned to variables, passed to methods, or even returned from methods. Getting used to the syntax can be difficult at first, so we will introduce it by example. Listing 2.7 does the same thing as the Java code in listing 2.6, but it uses a `sort()` method that compares items using a closure. You may be wondering where this `sort()` method has come from – don't worry, Groovy adds it to `List` implementations via the Groovy JDK, which we will cover later in the chapter.

Listing 2.7 Groovy sorting

```

List fruit = [ "apple", "Orange", "Avocado", "pear", "cherry" ]

fruit.sort { String a, String b ->                               #1
    a.compareToIgnoreCase(b)                                     #2
}

println "Sorted fruit: ${fruit}"

```

Cueballs in code and text

#1 Method call with closure #2 Implicit return value

In this example, we have not only declared a closure but also passed it as an argument to the `sort()` method (#1). See how the closure looks like a method body without a name? The major difference is that the arguments are declared *within* the curly braces.

Before we move on, we should point out two things in the listing. The first is that the `sort()` method is called without parentheses (#1). This is common when a method is called with a closure as its sole argument. The second is that the closure uses an implicit return value to help keep the code compact.

The standard syntax for a closure is:

```
{ <arguments> -> <body> }
```

where `<arguments>` is the list of typed or untyped parameters to the closure, and `<body>` is the equivalent of a method body. A zero-argument closure can be expressed using `{-> ... }`, i.e. no argument declarations before the arrow. There is also a special case that you need to be aware of, which is demonstrated by this example:

```
[ "apple", "pear", "cherry" ].each { println it }
```

When a closure definition has no `->`, it is created with a single implicit argument that you can access via the special variable "it". So in the above example, `each()` calls the closure for each item in the list, passing the current item in as an argument. Therefore "it" contains the current list item (the name of a fruit) and the code prints out each item in the list to the console.

You have seen closures declared at the point of use, i.e. as method arguments, but we also mentioned that you could assign them to variables and pass them around. Listing 2.8 modifies the example from listing 2.7 so that you can see how a reference to a closure can be used.

Listing 2.8 Closure assigned to a variable

```
Closure comparator = { String a, String b ->           #A
    a.compareToIgnoreCase(b)
}

List fruit = [ "apple", "Orange", "Avocado", "pear", "cherry" ]

fruit.sort(comparator)                                #1

println "Sorted fruit: ${fruit}"

assert comparator("banana", "Lemon") < 0            #2
```

Cueballs in code and text

#1 Closure reference as argument
#2 Calling a closure directly

No cueball

#A Assign closure to variable

Passing a closure to a method via a reference looks just like a normal method call (#1). We have also included an example of how you can call a closure directly as if it were a method (#2). This notation isn't particularly surprising when you consider that closures are effectively anonymous functions. When you assign a closure to a variable, you bind it to a name so it effectively becomes a named function!

We have now covered enough ground that you will be able to follow the many examples in this book where closures are used. However, to become truly comfortable with them in the context of Grails, you need to know a few more things. The first of these is that single argument closures are unique: they can be called with or without an argument, e.g. `myClosure(1)` or `myClosure()`. In the latter case, the closure argument has a value of `null`.

Second, there is a simple restriction on the special syntax used in the early examples of this section. Take the following example, which sums the items in a list. Don't worry, you don't have to understand how it works. We are only interested in the syntax at the moment.

```
List l = [ 1, 3, 5, 6 ]
assert 15 == l.inject(0) { runningTotal, value -> runningTotal + value }
```

The closure is declared *after* the method's closing parenthesis, and yet it is passed to the `inject()` method as the second argument. This is one of those syntactic tricks that Groovy provides to make code easier to read. That method call could equally be written as:

```
l.inject(0, { runningTotal, value -> runningTotal + value })
```

but this form doesn't work well with larger, multi-line closures. So what is the restriction we mentioned? You can only use the former syntax if the closure is the last argument to the method, whereas the latter form will work irrespective of the argument position.

The final point is possibly the most important: closures are not methods! The way in which they are used means that they often look like methods, but in the end they are objects whereas proper methods in Groovy actually compile to normal Java methods. In many cases the distinction is unimportant, but you should understand that the normal rules for overriding methods do not apply. In other words, if you are extending a class like this:

```
Class SuperClass {
    Closure action = {
        ...
    }
}
```

. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=493>

we recommend that you do not declare an “action” property in your sub-class. If you want polymorphic behaviour, use methods.

Inheritance is not the only area where using closures can cause problems. You may not be aware of or familiar with Aspect-oriented programming (AOP), but it has been growing in popularity in the last few years. Many AOP libraries and tools work with methods, so if you start using closures everywhere you will lose easy AOP integration.

Hopefully you will take these as words of caution rather than a reason never to use closures. They are a powerful concept that will make it easier to formulate solutions to certain problems. However, as with any new toy there is a danger of using them to the exclusion of other techniques.

2.4.2 Dynamic types, properties, and methods

Dynamic languages sound exciting, don't they? It's built into their name: “dynamic”. But what do we actually mean by that term? Throughout this chapter, we have assumed that you already know Java. In that language all variables, properties, method arguments, and method returns must have declared types. This allows the compiler to perform type-checking, eliminating a potential source of bugs before the application is ever run.

Now imagine that we have a Person class with the properties “givenName”, “familyName”, “age”, and “country”. In Java, we can easily sort a list of Person objects by one of these properties like so:

```
public void sortPeopleByGivenName(List<Person> personList) {
    Collections.sort(personList, new Comparator<Person>() {
        public int compare(Person p1, Person p2) {
            return p1.getFamilyName().compareTo(p2.getFamilyName());
        }
    })
}
```

But what if you didn't know which property to sort on until runtime? This might happen if a user can select which property to sort by in a user interface. Life suddenly becomes much trickier in Java. You either write a sort method like the one above for each of the properties and link them to the user interface somehow, or you use reflection in the comparator. The first option means duplication of the comparator code, which is both tedious and error prone. The second option would involve some pretty nasty code, so we will save you the pain of having to look at it.

Instead, let's look at a Groovy version in listing 2.9. This will introduce you to both untyped variables and “dynamic dispatch”. As required, the method simply sorts a list of people given the name of a property to sort on.

Listing 2.9 Sorting by an object property

```
def sortPeople(people, property) { #1
    return people.sort { p1, p2 -> #2
        p1."${property}" <=> p2."${property}"
    }
}
```

Cueballs in code and text

#1 Untyped arguments and return

#2 Dynamic property access

OK, so we have thrown you into the deep end with this example, but trust us – you'll be swimming in no time! Let's break the example down into bits, starting with the method signature (#1). This looks like any other method signature apart from the “def” keyword and the lack of types on the method arguments. The “def” keyword defines both untyped variables and methods. In other words, a variable declared using “def” can be assigned any value. You can even assign it a string on one line and then an integer on the next. When used on methods, it means that the method can return any type of value. Effectively, “def” is like using `java.lang.Object`.

Untyped method and closure arguments do not even need the “def” keyword. Simply leave out the type and they will be treated as untyped variables. Although Groovy is almost (if not actually) unique in

dynamic languages in supporting declared types, most developers seem to quickly move to using untyped variables, arguments, and methods simply because the code is more concise and easier on the eye.

Line #2 contains two new features. The simpler one to explain is the `<=>` operator. It is a comparison operator that effectively maps to a `compareTo()` call, i.e. it returns a negative number if the left-hand side is less than the right-hand side, zero if the operands are equal, and a positive number otherwise.

Which leaves us with the strange notation involving a `GString`. The first thing to understand is that Groovy will accept strings as property and method names. To see what we mean, here is an example in which the two lines do exactly the same thing:

```
peopleList.sort()
peopleList."sort"()
```

You will see later that the string notation can be particularly useful with builders, but in this case it's the ability to specify a property or method at runtime that's important. So if the "property" argument contains the string "age", line #2 becomes:

```
p1."age" <=> p2."age"
```

and so the method sorts on the "age" property. The exact same code can be used to sort on any of Person's properties. In fact, you will notice that the method has no dependency at all on Person, which means that the method can sort any list of objects so long as they have the given property! You have just seen how powerful a combination of untyped variables and dynamic dispatch can be. This "pattern" has a name.

DUCK TYPING

No discussion of a dynamic language is complete without a mention of the phrase "duck typing". It comes from the saying "if it walks like a duck and talks like a duck, it's probably a duck". Applied to dynamic languages, it means that if an object has a particular property or method signature, then it doesn't matter what type the object is, you can still call that method or access that property.

This is only possible in dynamic languages because properties and methods are resolved at *runtime*. This behaviour also allows you to add methods and properties to classes without modifying the class itself. This is how Groovy itself extends some of the JDK classes - see the section on the Groovy JDK to see some of the extra methods and properties it provides.

So, whenever you see references in examples to methods or properties that have no declaration, the chances are that Grails has injected them into the class. Don't worry, we will mention in the text when this happens and now you know where they come from!

THE DANGERS

Remember, with great power comes great responsibility.

Uncle Ben, *Spiderman*, 2002

Using Groovy for the first time can often feel like a straitjacket has been removed. It gives you great freedom in your coding and makes it easier to express your ideas and designs in code. With this power come pitfalls that you need to be aware of. First, the compiler can not pick up type violations or missing properties and methods, simply because these are all resolved at runtime. You are likely to become quite familiar with `MissingPropertyException` and its relative `MissingMethodException` early on in your Groovy experience, but you will be surprised at how rarely they appear later on. Even so, you should get into the habit of writing tests for your code at an early stage and we cover this in chapter 6.

Second, the application is more difficult to debug, particularly with dynamically injected properties and methods. In particular, using the "step into" feature of a debugger is likely to cause plenty of pain. Instead, make judicious use of breakpoints and the "run to cursor" feature.

Finally, over-use of dynamic injection and even untyped variables can make it difficult to understand code. Consider using conventions in a similar way to Grails so that people can find the implementations of dynamic properties and methods more easily. Also consider using declared types for methods and closure arguments: not only will people find it easier to use those methods, but IDEs can highlight potential type errors in the calling code.

. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=493>

Used judiciously, dynamically injected properties and methods can improve your productivity by enhancing classes in a non-invasive way. Probably the best example of this is the Groovy JDK, which is a set of enhancements to the core JDK classes.

2.5 The Groovy JDK

Have you ever bemoaned the lack of a method in the JDK or wondered why you have to use helper classes like `java.util.Collections`? Groovy tries to fill in any gaps it sees in the JDK and make code more uniformly object-oriented than Java through the Groovy JDK, a set of enhancements to some of the more commonly used JDK classes. We can not cover all the extra properties and methods because there are too many, but we will look at some of the more useful ones. For a full run down see the Groovy JDK reference online:

<http://groovy.codehaus.org/groovy-jdk/>

Probably the most useful additions relate to collections. Most of these also apply to arrays and strings, the latter of which are generally treated as lists of characters. There are also some nice enhancements to `java.io.File`. We then round off with some miscellaneous improvements.

COLLECTION/ARRAY/STRING.SIZE()

In the interests of consistency, Groovy provides the method `size()` for both arrays and strings. Of course, its behaviour matches the `length` property for arrays, and the `length()` method on `String`. No longer do you have to remember which property or method to use for a particular type!

COLLECTION/ARRAY/STRING.EACH(CLOSURE)

You have already seen the `each()` method in action in some of the examples earlier in the chapter. It simply iterates over all elements of a collection, applying the given closure to each one in turn. Remember, a string is treated as a sequence of characters, and so this code:

```
"Test".each { println it }
```

will print this to the console:

```
T
e
s
t
```

COLLECTION/ARRAY/STRING.FIND(CLOSURE)

Once you begin to use this method (and its sibling, `findAll()`), you will wonder how you ever managed to live without it. It does exactly what it says on the tin: finds an element within a sequence. More specifically, it returns the first element for which the given closure returns `true`, or `null` if there is no such element. The closure effectively acts as the criteria. In this example, we retrieve the first object in a list for which the `firstName` property is "Glen":

```
def glen = personList.find { it.firstName == "Glen" }
```

Whereas `find()` returns the first element that matches the criteria, `findAll()` will return all matching elements as a list. If there are no matching elements, it will return an empty list rather than `null`.

COLLECTION/ARRAY/STRING.COLLECT(CLOSURE)

This is one of those methods that take some getting used to, but become invaluable once you are. In some other languages it is called the "map" function. Quite simply, it iterates through the collection, applying the given closure to each element and adding the return value to a new list. For example, say we have a list of strings and we want the lengths of those strings as a list. We can do that like so:

```
def names = [ "Glen", "Peter", "Alice", "Graham", "Fiona" ]
assert [ 4, 5, 5, 6, 5 ] == names.collect { it.size() }
```

Note that the new list has the same size and same order as the original.

COLLECTION/ARRAY.SORT(CLOSURE)

Sorts a collection using the given closure as a comparator. The closure can either take a single argument, in which case it should return a value that is itself comparable, for example a string, or it can take two arguments (like the `compareTo()` method) and return an integer representing the relative order of the two values.

To demonstrate what this means in practice, we will sort a list of names based on their lengths using both approaches. First, using a single argument closure:

. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=493>

```
def names = [ "Glen", "Peter", "Ann", "Graham", "Veronica" ]
def sortedNames = names.sort { it.size() }
assert [ "Ann", "Glen", "Peter", "Graham", "Veronica" ] == sortedNames
```

Next we use a two-argument closure:

```
def names = [ "Glen", "Peter", "Ann", "Graham", "Veronica" ]
def sortedNames = names.sort { name1, name2 ->
    name1.size() <=> name2.size()
}

assert [ "Ann", "Glen", "Peter", "Graham", "Veronica" ] == sortedNames
```

COLLECTION/ARRAY.JOIN(STRING)

Creates a string by concatenating the elements of the collection together in order, using the given string as a separator. If the elements are not strings themselves, then `toString()` is called before performing the concatenation. One of the most common uses for this method is to convert a real list of strings to a comma-separated “list” like so:

```
def names = [ "Glen", "Peter", "Alice", "Fiona" ]
assert "Glen, Peter, Alice, Fiona" == names.join(", ")
```

FILE.TEXT

Reads the given file and returns the contents as a string. Simple.

FILE.SIZE()

Simply returns the size of the file in bytes. It corresponds to the `File.length()` method.

FILE.WITHWRITER(CLOSURE)

There are various “with...()” methods for streams and files, but we will only cover this one here. See the Groovy JDK reference for more.

This method conveniently creates a `java.io.Writer` from the file and passes it to the closure. Once the closure is done, the underlying output stream is closed automatically and safely. No more “try...catch” blocks just to write to a file!

MATCHER.COUNT

Returns the number of matches found.

NUMBER.ABS()

Finally! You can now call the `abs()` method directly on numbers.

NUMBER.TIMES(CLOSURE)

This method calls the given closure n number of times, where n is the number the method is called on. Note that this only makes sense with integers. The closure is passed the current iteration number, which starts from zero.

We have only shown a fraction of the available properties and methods in the Groovy JDK, but these are some of the most common and they are used extensively throughout this book. As you become more experienced with Groovy, we recommend that you get into the habit of checking the Groovy JDK for any other useful properties or methods on the classes you are working with. You never know what gem might be hidden in there!

2.6 Summary

We have covered Groovy at a rather rapid pace, so it’s now time to stop and take stock. From the early sections and examples, you should be feeling pretty confident and hopefully excited about programming in Groovy. Even if you feel overwhelmed at this stage, don’t worry. The chapter includes more information than you actually need just to get started. As you progress through the rest of the book’s chapters, just flip back here whenever you see something in an example you don’t quite understand.

At first, you will just need to recognise some of the syntax and constructs that are unique to Groovy. Later on, a firm grasp of the dynamic nature of Groovy will help your confidence levels no end and help you avoid common pitfalls. From the start, we feel that you will soon enjoy Groovy’s expressivity and power. Just don’t blame us if you no longer want to program in plain Java!

. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=493>

Before we move on to the Grails fundamentals, remember that Groovy is not just a language for Grails. It can be used for a variety of purposes: writing scripts; as an embedded scripting language for a java application; build tools; rapid prototyping; and more! For a more comprehensive look at the language itself, check out Groovy in Action.

You have seen how quickly an application can be built from scratch. You should be able to understand and write Groovy without fear. It is now time to introduce you to the fundamental concepts of a Grails application, starting with domain classes.

3 Life after SQL, this ain't your Grandma's OUTER JOIN

What you'll learn:

- What GORM is and how it works
- Define Domain Model classes
- Validation and Constraining Fields
- Domain Class Relationships (1:1, 1:M, M:N)
- Basic Querying

In this chapter we'll explore Grails support for the Data Model portion of your applications – and if you think we'll be digging deep into OUTER JOINS, then you'll be pleasantly surprised. In fact, we won't be writing a line of SQL, and you won't find any Hibernate XML mappings in here either. We'll be taking full advantage of the “Convention over Configuration” paradigm we introduced to you in chapter 1 – which means less time configuring, and more time getting actual work done.

We'll be spending most of our time exploring the basics of how Grails persists your Domain model classes using GORM – the Grails Object Relational Mapping library. And a ton of things about how GORM handles various modeling relationships (you know, 1 to many, many to many, and all that jazz). We'll tour our way through validating user input, displaying custom error messages, interacting with HTML forms, and querying our data for fun and profit.

But knowledge works best in application, and we're practitioners not theory boys, so we'll start building the heart of our sample application, *hubhub*. You'll learn how to write a “user registration” module, so we can start signing up users, and then once we've signed them up and we'll learn a little about querying by writing a simple login module. Take a deep breath, there's lots to learn. But without further ado, it's time to introduce you to our sample application.

3.1 Reader, meet Hubhub: Starting Our Example Application

Our goal for this book is to take you to the stage where you would work full time as a productive Grails developer – to give you all the skills you need to produce world-class applications in record time. And that means taking you a lot further than just a bunch of facts, figures and tables – we want to give you a thorough mentoring in the core of Grails productivity. The best way we know to do that is let you look over our shoulder as we develop a real application.

. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=493>