Essential Scala

Noel Welsh and Dave Gurnell

Version 1.3, April 2017

underscore

Essential Scala

Version 1.3, April 2017

Published by Underscore Consulting LLP, Brighton, UK.

Copies of this, and related topics, can be found at http://underscore.io/training.

Team discounts, when available, may also be found at that address.

Contact the author regarding this text at: hello@underscore.io.

Our courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Underscore titles, please visit http://underscore.io/training.

Contents

Foreword

This book is aimed to programmer learning Scala for the first time. We assume you have some familiarity with an object-oriented programming language such as Java, but little or no experience with functional programming.

Our goal is to describe how to use Scala in-the-small. To this end our focus is on the core patterns used in idiomatic Scala code, and we introduce Scala's features in the context of the patterns they enable. We are not aiming for exhaustive coverage of Scala's features, and this text is not a reference manual.

Except for a few exercises we don't rely on any external libraries. You should be able to complete all the problems inside with only a text editor and Scala's REPL, or an IDE such as the Scala IDE for Eclipse or IntelliJ IDEA.

Essential Scala was created by Noel Welsh and Dave Gurnell of Underscore. It was built using Underscore's eBook Template, plain text, and a deep and profound love of functional programming.

Conventions Used in This Book

This book contains a lot of technical information and program code. We use the following typographical conventions to reduce ambiguity and highlight important concepts:

Typographical Conventions

New terms and phrases are introduced in *italics*. After their initial introduction they are written in normal roman font.

Terms from program code, filenames, and file contents, are written in `monospace font`. Note that we do not distinguish between singular and plural forms. For example, might write `String` or `Strings` to refer to the `java.util.String` class or objects of that type.

References to external resources are written as hyperlinks. References to API documentation are written using a combination of hyperlinks and monospace font, for example: `Option`.

Source Code

Source code blocks are written as follows. Syntax is highlighted appropriately where applicable:

```
object MyApp extends App {
  println("Hello world!") // Print a fine message to the user!
}
```

Some lines of program code are too wide to fit on the page. In these cases we use a *continuation character* (curly arrow) to indicate that longer code should all be written on one line. For example, the following code:

```
println("This code should all be written ⏎
  on one line.")
```

should actually be written as follows:

```
println("This code should all be written on one line.")
```

Callout Boxes

We use three types of *callout box* to highlight particular content:

> Tip callouts indicate handy summaries, recipes, or best practices.

> Advanced callouts provide additional information on corner cases or underlying mechanisms. Feel free to skip these on your first read-through—come back to them later for extra information.

> Warning callouts indicate common pitfalls and gotchas. Make sure you read these to avoid problems, and come back to them if you're having trouble getting your code to run.

Thanks

A big thanks to Richard Dallway and Jonathan Ferguson, who took on the herculean task of proof reading our early drafts and helped develop the rendering pipeline that produces the finished book.

Thanks also to Amir Aryanpour, Rebecca Grenier, Joe Halliwell, Jason Scott, Daniel Watford, N. Sriram, and Audrey Welsh who sent us corrections and suggestions. Knowing that our work was being read and used made the long haul of writing the book worthwhile.

Chapter 1

Getting Started

Throughout this book we will be working with short examples of Scala code. There are two recommended ways of doing this:

1. Using the *Scala console* (better for people who like command lines)
2. Using *Worksheets* feature of *Scala IDE* (better for people who like IDEs)

We'll walk through the setup for each process here.

1.1 Setting up the Scala Console

Follow the instructions on http://scala-lang.org to set Scala up on your computer. Once Scala is installed, you should be able to run an interactive console by typing `scala` at your command line prompt. Here's an example from OS X:

```
dave@Jade ~> scala
Welcome to Scala version 2.11.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_45).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

You can enter individual expressions at the `scala>` prompt and press *Enter* to compile and execute them:

```
scala> "Hello world!"
res0: String = Hello world!
```

1.1.1 Entering Single-Line Expressions

Let's try entering a simple expression:

```
scala> 1 + 2 + 3
res1: Int = 6
```

When we press Enter, the console responds with three things:

- an *identifier* `res1`;
- a *type* `Int`;
- a *value* 6.

As we will see in the next chapter, every expression in Scala has a *type* and a *value*. The type is determined at compile time and the value is determined by executing the expression. Both of these are reported here.

The identifier `res1` is a convenience provided by the console to allow us to refer to the result of the expression in future expressions. For example, we can multiply our result by two as folllows:

```
scala> res1 * 2
res2: Int = 12
```

If we enter an expression that doesn't yield a useful value, the console won't print anything in response:

```
scala> println("Hello world!")
Hello world!
```

Here, the output `"Hello world!"` is from our `println` statement—the expression we entered doesn't actually return a value. The console doesn't provide output similar to the output we saw above.

1.1.2 Entering Multi-Line Expressions

We can split long expressions across multiple lines quite simply. If we press enter before the end of an expression, the console will print a `|` character to indicate that we can continue on the next line:

```
scala> for(i <- 1 to 3) {
     |   println(i)
     | }
1
2
3
```

Sometimes we want to enter multiple expressions at once. In these cases we can use the `:paste` command. We simply type `:paste`, press Enter, and write (or copy-and-paste) our code. When we're done we press `Ctrl+D` to compile and execute the code as normal. The console prints output for every expression in one big block at the end of the input:

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val x = 1
val y = 2
x + y

// Exiting paste mode, now interpreting.

x: Int = 1
y: Int = 2
res6: Int = 3
```

If we have Scala code in a file, we can use `:paste` to paste the contents of the file into the console. This is much more convenient than re-entering expressions in the console. For example, with a file named `example.txt` containing `1 + 2 + 3` we can use `:paste` like so:

```
scala> :paste example.scala
Pasting file example.scala...
res0: Int = 6
```

1.1.3 Printing the Type of an Expression

One final tip for using the console. Occasionally we want to know the *type* of an expression without actually running it. To do this we can use the `:type` command:

```
scala> :type println("Hello world!")
Unit
```

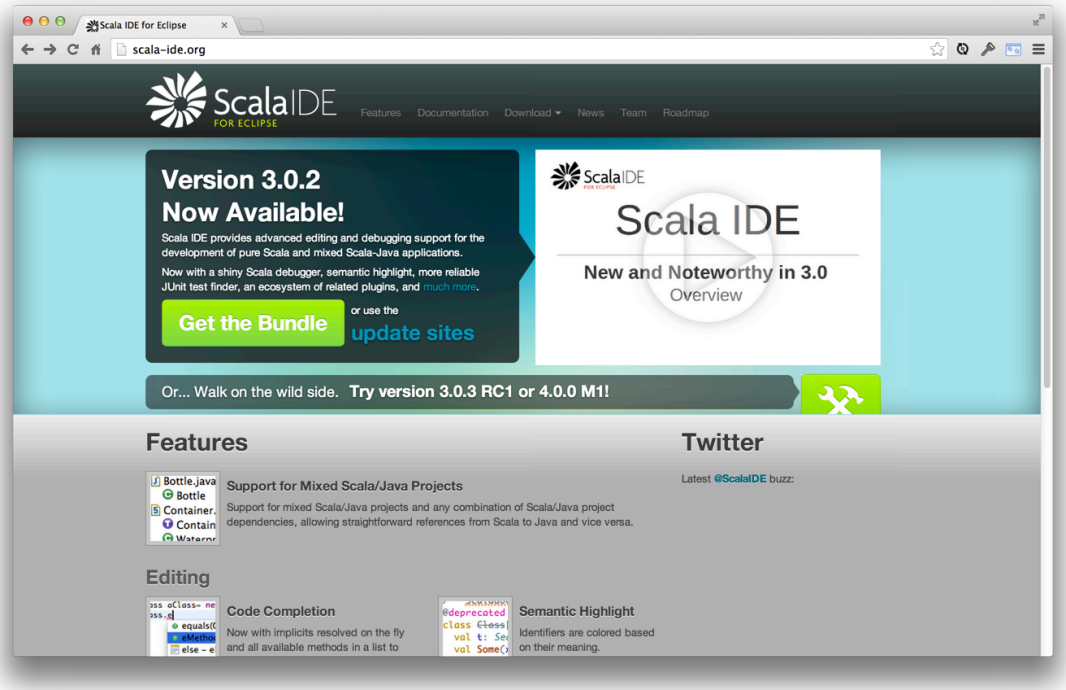
Notice that the console doesn't execute our `println` statement in this expression. It simply compiles it and prints out its type, which in this case is something called `Unit`.

`Unit` is Scala's equivalent of `void` from Java and C. Read Chapter 1 to find out more.

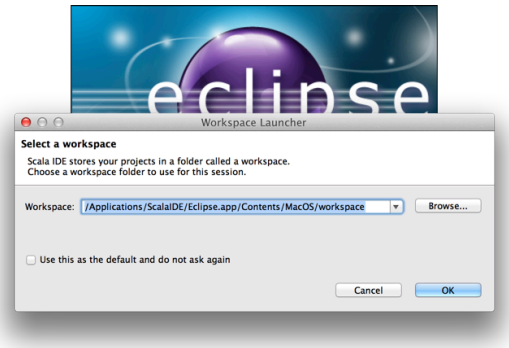
1.2 Setting up Scala IDE

Scala IDE is a plugin that adds Scala language support to Eclipse. A complete version of Scala IDE with Eclipse is also available as a standalone bundle from http://scala-ide.org. This is the easiest way to install the software so we recommend you install the standalone bundle for this course.

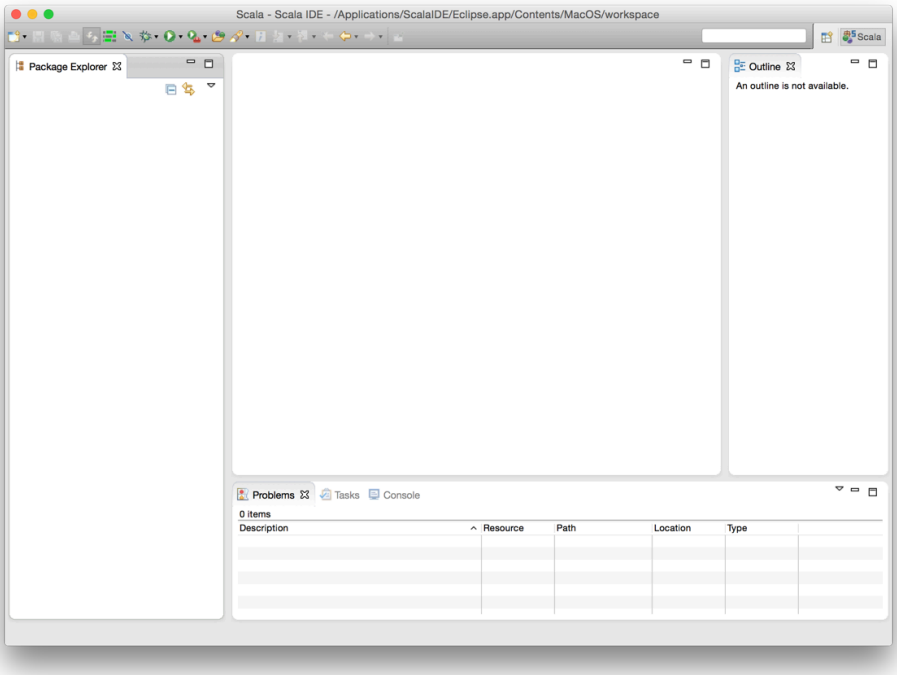
Go to http://scala-ide.org now, click the **Get the Bundle** button, and follow the on-screen instructions to download Scala IDE for your operating system:



Once you have downloaded and uncompressed the bundle you should find an application called **Eclipse**. Launch this. You will be asked to choose a folder for your *workspace*:



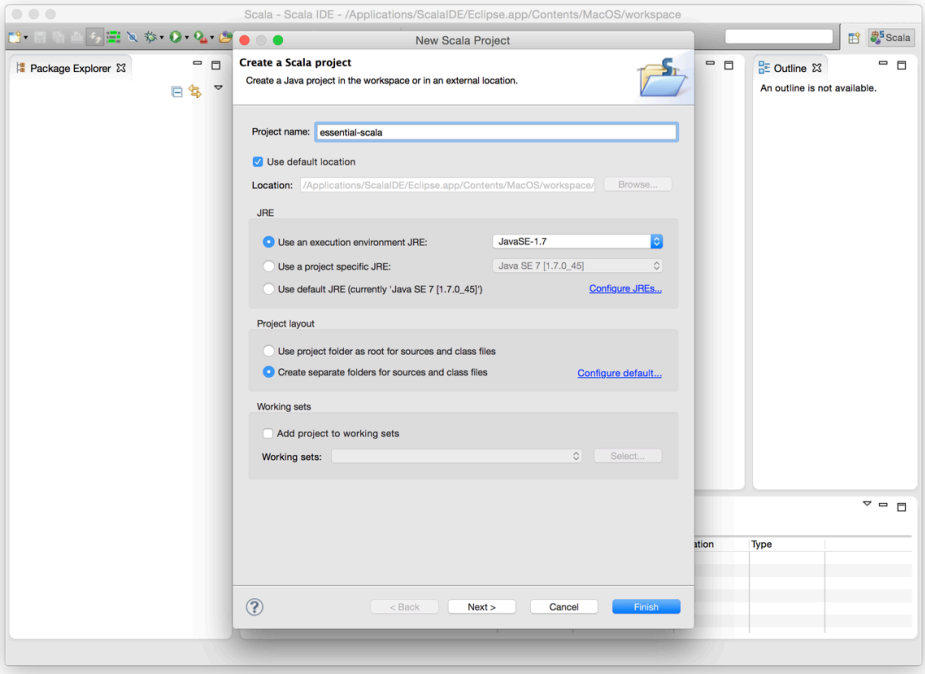
Accept the default location and you will see an empty main Eclipse window:



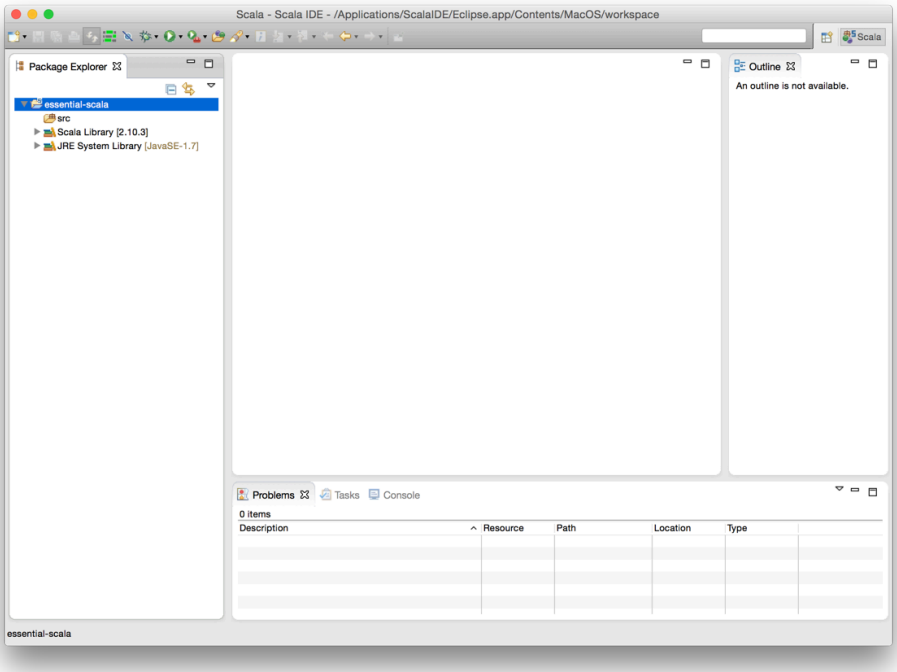
1.2.1 Creating your First Application

Your *Eclipse workspace* is two things: a folder containing files and settings, and a main window where you will be doing most of your Scala programming. In your workspace you can find *projects* for each Scala application you work on.

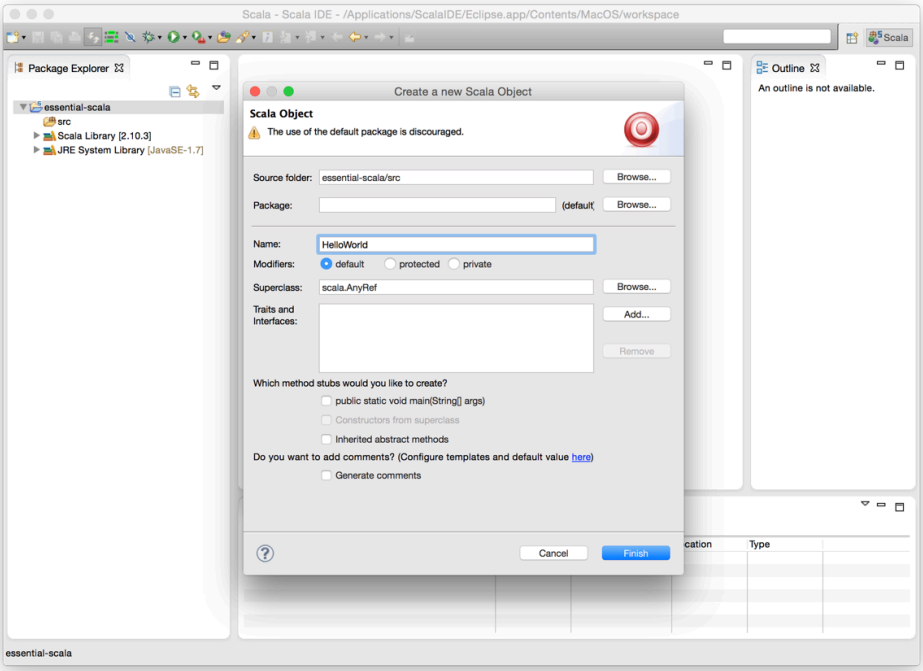
Let's create a project for the book exercises now. Select the **File menu** and choose **New > Scala Project**:



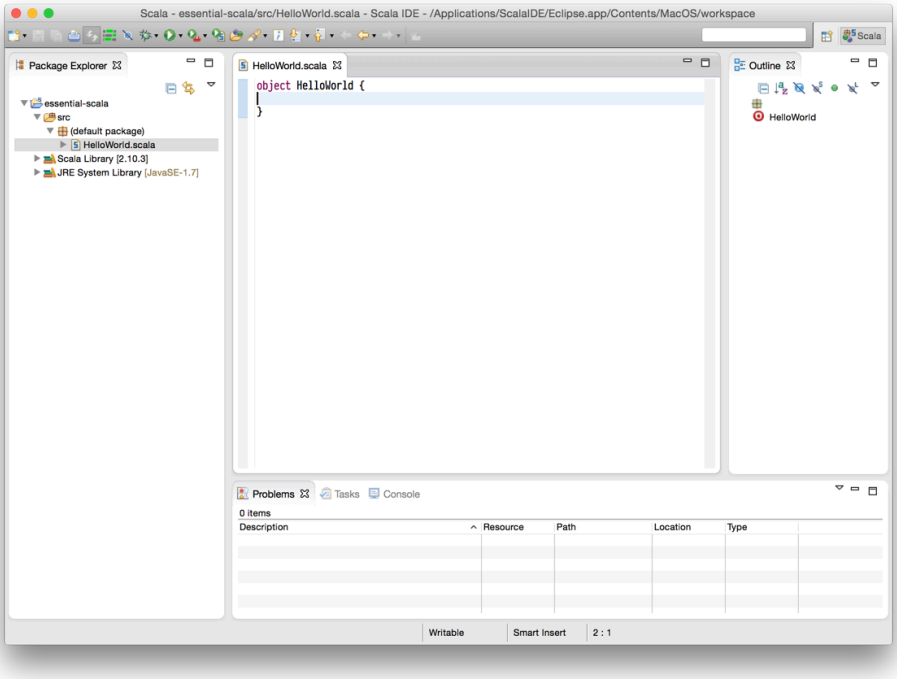
Enter a **Project name** of `essential-scala` and click **Finish**. The tree view on the left of your workspace should now contain an empty project:



A project is no good without code to run! Let's create our first simple Scala application - the obligatory *Hello World* app. Select the **File Menu** and choose **New > Scala Object**:



Name your object `HelloWorld` and click **Finish**. A new file called `HelloWorld.scala` should appear in the tree view on the left of the main window. Eclipse should open the new file in the main editor ready for you to start coding:



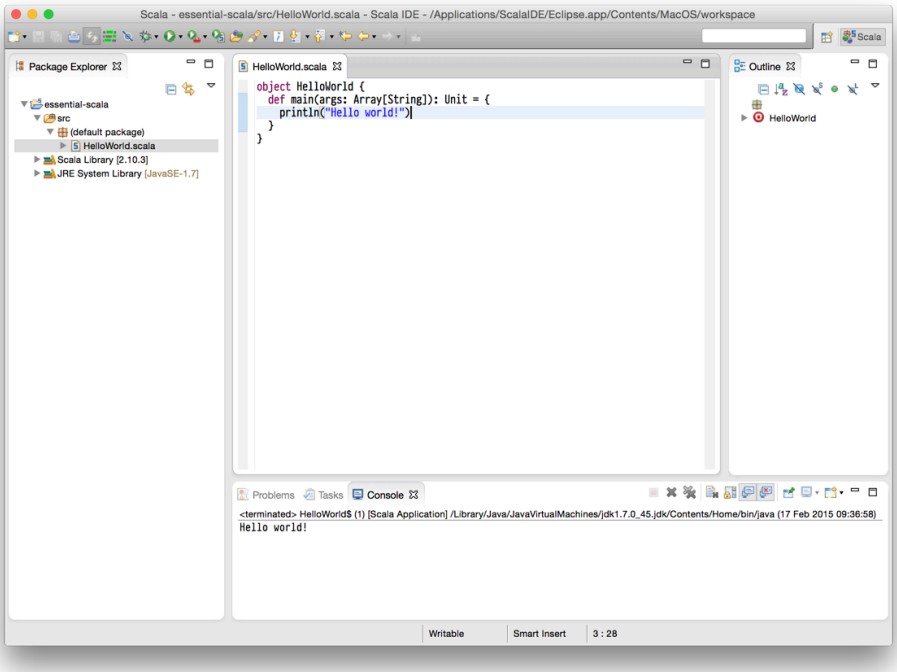
The content of the file should read as follows:

```
object HelloWorld {

}
```

Replace this text with the following minimalist application:

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello world!")
  }
}
```

Select the **Run Menu** and choose **Run**. This should execute the code in your application, resulting in the words `Hello world!` appearing in the *Console* pane at the bottom of the window. Congratulations - you just ran your first Scala application!



Developers with Java experience will notice the resemblance of the code above to the Java hello world app:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}
```

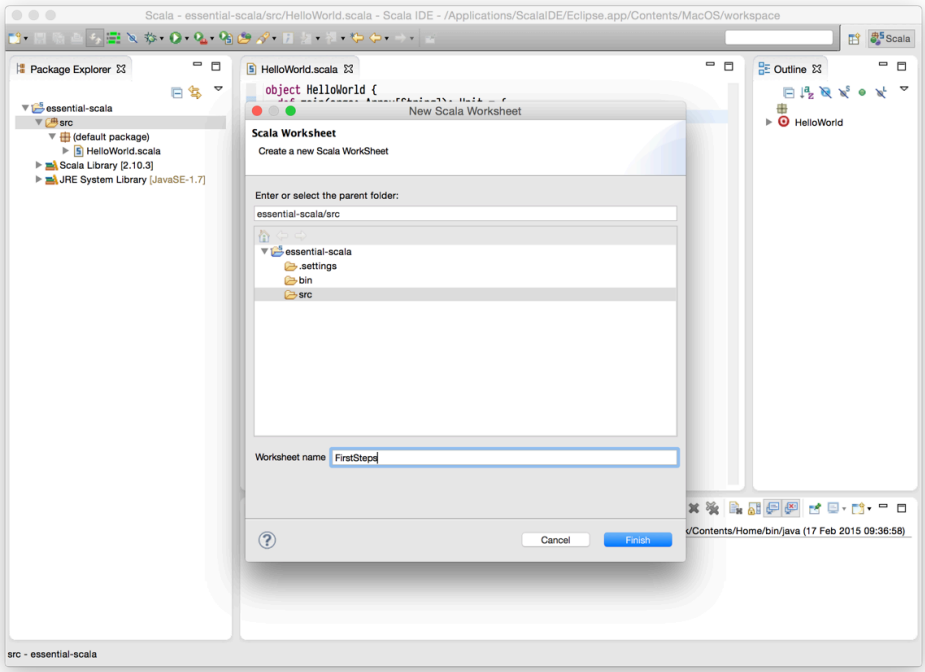
The resemblance is, of course, no coincidence. These two applications compile to more or less the same byte-code and have exactly the same semantics. We will learn much more about the similarities and differences between Scala and Java as the course continues.

1.2.2 Creating your First Worksheet

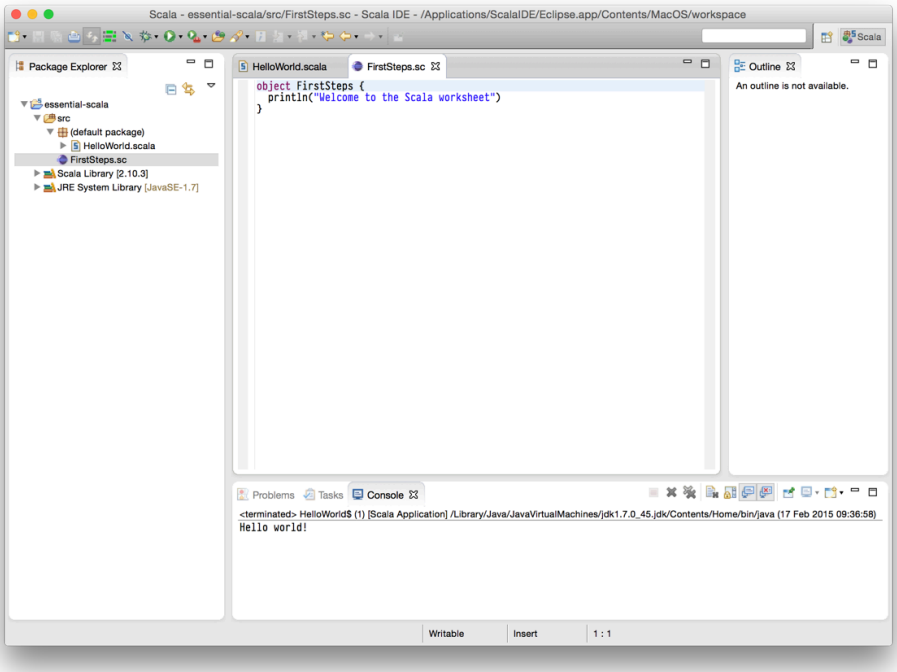
Compiling and running code whenever you make a change is a time consuming process that isn't particularly suitable to a learning environment.

Fortunately, Scala IDE allows us to create special files called *Scala Worksheets* that are specifically designed for training and experimentation. Every time you save a Worksheet, Eclipse automatically compiles and runs your code and displays the output on the right-hand side of your editor. This provides instant feedback, which is exactly what we need when investigating new concepts!

Create your first Scala Worksheet by selecting the **File Menu** and choosing **New > Scala Worksheet**:



Enter a **Worksheet name** of `FirstSteps` and click **Finish**. A new file called `FirstSteps.sc` should appear in the tree view on the left of the main window, and should open it in the main editor in the middle:



Note that the object on the left contains a single line of Scala code:

```
println("Welcome to the Scala worksheet")
```

for which Eclipse is displaying the corresponding output on the right:

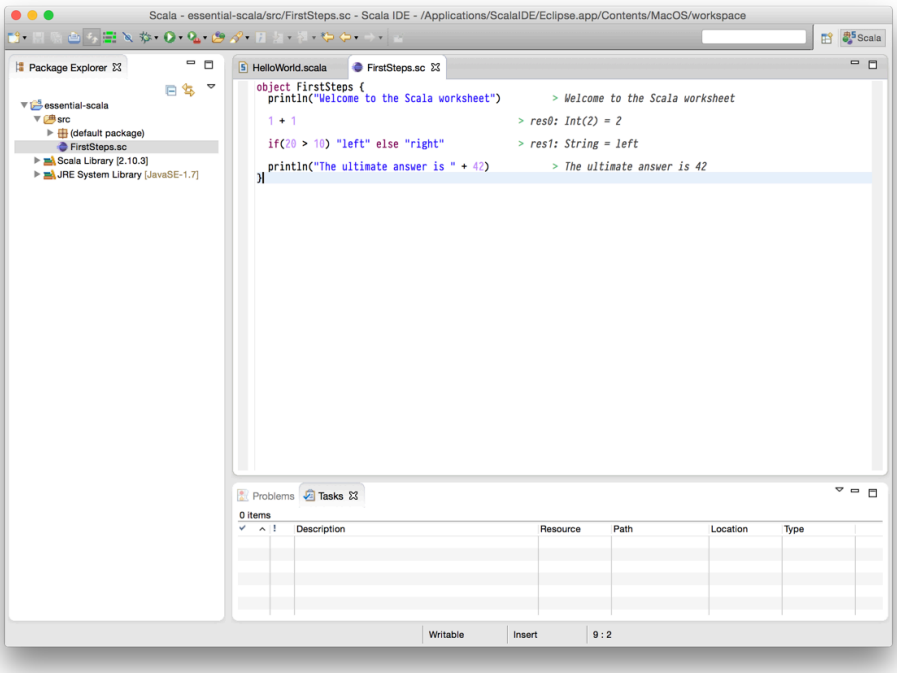
```
Welcome to the Scala worksheet
```

Any expression you add to the left of the editor is evaluated and printed on the right. To demonstrate this, change the text in the editor to the following:

```
object FirstSteps {
  println("Welcome to the Scala worksheet")

  1 + 1

  if(20 > 10) "left" else "right"

  println("The ultimate answer is " + 42)
}
```

Save your work by selecting the **File Menu** and choosing **Save** (or better still by pressing **Ctrl+S**). Eclipse should automatically evaluate each line of code and print the results on the right of the editor:

```
object FirstSteps {
  println("Welcome to the Scala worksheet")   //> Welcome to the Scala worksheet

  1 + 1                                       //> res0: Int(2) = 2

  if(20 > 10) "left" else "right"             //> res1: String = left

  println("The ultimate answer is " + 42)     //> The ultimate answer is 42
}
```



We'll dive into what all of the text on the right means as we proceed with the course ahead. For now you're all set to start honing your Scala skills!

Chapter 2

Expressions, Types, and Values

In this chapter we look at the fundamental building blocks of Scala programs: *expressions*, *types*, and *values*. Understanding these concepts is necessary to build a mental model of how Scala programs work.

2.1 Your First Program

In the Scala console or worksheet enter `"Hello world!"` and press return (in the console) or save the worksheet. You should see an interaction similar to this:

```
"Hello world!"
// res: String = Hello world!
```

There is a lot to say about this program. It consists of a single expression, and in particular a *literal expression* or literal for short.

Scala runs, or *evaluates*, our program. When we evaluate a program in the Scala console or worksheet we are told two pieces of information: the *type* of the program, and the *value* it evaluates to. In this case the type is `String` and the value is `"Hello world!"`.

Although the output value "Hello world!" looks the same as the program that created it, there is a difference between the two. The literal expression is the program text we entered, while what the console prints is the result of evaluating that program. (Literals are so-named because they literally look like what they evaluate to.)

Let's look at a slightly more complex program

```
"Hello world!".toUpperCase
// res: String = HELLO WORLD!
```

This program extends our first example by adding a *method call*. Evaluation in Scala proceeds left to right. First the literal `"Hello world!"` is evaluated, as in the first example. Then the method `toUpperCase` is called on the result. This method transforms a string value to its upper case equivalent and returns this new string. This is the final value we see printed by the console.

Once again the type of this program is `String`, but in this case it evaluates to `"HELLO WORLD!"`

2.1.1 Compile-time and Run-time

There are two distinct stages that a Scala program goes through: first it is *compiled*, and if it compiles successfully it can then be *run* or evaluated. We refer to the first stage as *compile-time* and the latter as *run-time*.

When using the Scala console our programs are evaluated as soon as they compile, which gives the appearance that there is only one stage. It is important to understand that compile- and run-time really are distinct, as it is this distinction that allows us to properly understand the difference between types and values.

Compilation is a process of checking that a program makes sense. There are two ways in which a program must "make sense":

1. It must be *syntactically correct*, meaning the parts of the program must be arranged according to the grammar of the language. An example English sentence that is not syntactically correct is "on cat mat sat the". An example syntactically incorrect Scala program is

```
toUpperCase."Hello world!"
// error: identifier expected but string literal found.
//       toUpperCase."Hello world!"
//                   ^
```

2. It must *type check*, meaning it must obey certain constraints on what a sensible program is. An example English sentence that is syntactically correct but fails to make sense is "the mat sat on the cat". A simple program that would fail to type check is trying to convert a number to uppercase.

```
2.toUpperCase
// error: value toUpperCase is not a member of Int
//             2.toUpperCase
//               ^
```

The concept of upper and lowercase doesn't make sense for numbers, and the type system will catch this error.

If a program passes the checks at compile-time it may then be run. This is the process of the computer performing the instructions in the program.

Even though a program successfully compiles it can still fail at run-time. Dividing an integer by zero causes a run-time error in Scala.

```
2 / 0
// java.lang.ArithmeticException: / by zero
```

The type of integers, `Int`, allows division so the program type checks. At run-time the program fails because there is no `Int` that can represent the result of the division.

2.1.2 Expressions, Types, and Values

So what exactly are expressions, types, and values?

Expressions are part of a program's text—what we type into a file, or the console or worksheet. They are the main components of a Scala program. We will see other components, namely *definitions* and *statements*, in due course. Expressions exist at compile-time.

The defining characteristic of an expression is that it evaluates to a value. A value is information stored in the computer's memory. It exists at run-time. For example, the expression 2 evaluates to a particular sequence of bits in a particular location in the computer's memory.

We compute with values. They are entities that our programs can pass around and manipulate. For example, to compute the minimum of two numbers we might write a program like

```
2.min(3)
// res: Int = 2
```

Here we have two values, 2 and 3, and we combine them into a larger program that evaluates to 2.

In Scala all values are *objects*, which has a particular meaning we will see shortly.

Now let's turn to types. Types are restrictions on our programs that limit how we can manipulate objects. We have already seen two types, `String` and `Int`, and seen that we can perform different operations depending on the type.

At this stage, the most important point about types is that *expressions have types but values do not*. We cannot inspect an arbitrary piece of the computer's memory and divine how to interpret it without knowing the program that created it. For example, in Scala the `Int` and `Float` types are both represented by 32-bits of memory. There are no tags or other indications that a given 32-bits should be interpreted as an `Int` or a `Float`.

We can show that types exist at compile-time by asking the Scala console to tell us the type of an expression that causes a run-time error.

```
:type 2 / 0
// Int

2 / 0
// java.lang.ArithmeticException: / by zero
```

We see that the expression `2 / 0` has type `Int` even though this expression fails when we evaluate it.

Types, which exist at compile-time, restrict us to writing programs that give a consistent interpretation to values. We cannot claim that a particular 32-bits is at one point an `Int` and another a `Float`. When a program type checks, Scala guarantees that all values are used consistently and thus it does not need to record type information in a value's representation. This process of removing type information is called *type erasure*[1].

Types necessarily do not contain all possible information about the values that conform to the type. If they did, type checking would be equivalent to running the program. We have already seen that the type system will not prevent us from dividing an `Int` by zero, which causes a run-time error.

An key part of designing Scala code is deciding which error cases we wish to rule out using the type system. We will see that we can express many useful constraints in the type system, improving the reliability of our programs. We could implement a division operator that used the type system to express the possibility of error, if we decided this was important enough in our program. Using the type system well is one of the main themes of this book.

2.1.3 Take Home Points

We must build a mental model of Scala programs if we are to use Scala. Three fundamental components of this model are *expressions*, *types*, and *values*.

Expressions are the parts of a program that evaluate to a value. They are the major part of a Scala program.

Expressions have types, which express some restrictions on programs. During *compile-time* the types of our programs are checked. If they are inconsistent then compilation fails and we cannot evaluate, or run, our program.

[1]This is not entirely true. The Java Virtual Machine, the program that runs Scala code, distinguishes between two kinds of objects. Primitive types don't store any type information along with the value they represent. Object types do store type information. However this type information is not complete and there are occasions where it is lost. Blurring the distinction between compile- and run-time is thus dangerous. If we never rely on type information being around at run-time (and the patterns we will show you do not) we will never run into these issues.

Values exist in the computer's memory, and are what a running program manipulates. All values in Scala are *objects*, the meaning of which we will discuss soon.

2.1.4 Exercises

2.1.4.1 Type and Value

Using the Scala console or worksheet, determine the type and value of the following expressions:

```
1 + 2
```

See the solution

```
"3".toInt
```

See the solution

```
"foo".toInt
```

See the solution

2.2 Interacting with Objects

In the previous section we saw the fundamental components of Scala programs: expressions, types, and values. We learned that *all values are objects*. In this section we will learn more about objects and how we can interact with them.

2.2.1 Objects

An object is a grouping of data and operations on that data. For example, 2 is an object. The data is the integer 2, and the operations on that data are familiar operations like +, -, and so on.

We have some special terminology for the data and operations of an object. The operations are known as *methods*. The data is stored in *fields*.

2.2.2 Method Calls

We interact with objects by *calling* methods[2]. We have already seen some examples of calling methods. For example, we have seen we can get the uppercase version of a `String` by calling its `toUpperCase` method.

```
"hello".toUpperCase
// res: String = HELLO
```

Some methods accept *parameters* or *arguments*, which control how the method works. The `take` method, for example, takes characters from a `String`. We must pass a parameter to `take` to specify how many characters we want.

[2]There is another way of interacting with objects, called pattern matching. We will introduce pattern matching later.

```
"abcdef".take(3)
// res: String = abc

"abcdef".take(2)
// res: String = ab
```

Method Call Syntax

The syntax for a method call is

```
anExpression.methodName(param1, ...)
```

or

```
anExpression.methodName
```

where

- `anExpression` is any expression (which evaluates to an object)
- `methodName` is the name of the method
- the optional `param1, ...` is one or more expressions evaluating to the parameters to the method.

A method call is an expression, and thus evaluates to an object. This means we can chain method calls together to make more complex programs:

```
"hello".toUpperCase.toLowerCase
// res: String = hello
```

In what order are the various expressions in a method call evaluated? Method parameters are evaluated left-to-right, before the method is called. So in the expression

```
"Hello world!".take(2 + 3)
```

the expression `"Hello world!"` is evaluated first, then `2 + 3` (which requires evaluating 2 and then 3 first), then finally `"Hello world!".take(5)`.

2.2.3 Operators

Because every value in Scala is an object we can also call methods on primitive types such as `Int` and `Boolean`. This is in contrast to Java where `int` and `boolean` are not objects:

```
123.toShort // this is how we define a `Short` in Scala
// res: Short = 123

123.toByte // and this is how we define a `Byte`
// res: Byte = 123
```

But if an `Int` is an object, what are the basic methematical operators such as `+` and `-`? Are they also methods? Yes—Scala methods can have symbolic names as well as alphanumeric ones!

```
43 - 3 + 2
// res: Int = 42

43.-(3).+(2)
// res: Int = 42
```

(Note that in Scala 2.10 and earlier you would have to write `(43).-(3).+(2)` to prevent `43.` being interpreted as a `Double`.)

> **Infix Operator Notation**
>
> Any Scala expression written `a.b(c)` can also be written `a b c`.
>
> Note that `a b c d e` is equivalent to `a.b(c).d(e)`, not `a.b(c, d, e)`.

We can use *infix operator notation* with any method that takes one parameter, regardless of whether it has a symbolic or alphanumeric name:

```
"the quick brown fox" split " "
// res: Array[String] = Array(the, quick, brown, fox)
```

Infix notation is one of several syntactic shorthands that allow us to write simple operator expressions instead of verbose method calls. There are also notations for *prefix*, *postfix*, *right-associative*, and *assignment-style operators*, but there are much less common than infix notation.

A question poses itself—what precedence rules should we associate with infix operators? Scala uses a set of precedence rules derived from the identifiers we use as method names that follow our intuitive understanding from mathematics and logic:

```
2 * 3 + 4 * 5
// res: Int = 26

(2 * 3) + (4 * 5)
// res: Int = 26

2 * (3 + 4) * 5
// res: Int = 70
```

2.2.4 Take home points

All Scala values are objects. We *interact with objects by calling methods* on them. If you come from a Java background note we can call methods on `Int` or any other primitive value.

The syntax for a method call is

```
anExpression.methodName(parameter, ...)
```

or

```
anExpression methodName parameter
```

Scala has very few operators - almost everything is a method call. We use syntactic conventions like infix operator notation to keep our code simple and readable, but we can always fall back to standard method notation where it makes sense.

As we will see, Scala's focus on programming with expressions allows us to write much shorter code than we can in Java. It also allows us to reason about code in a very intuitive way using values and types.

2.2.5 Exercises

2.2.5.1 Operator Style

Rewrite in operator-style

```
"foo".take(1)
```

See the solution

Rewrite in method call style

```
1 + 2 + 3
```

See the solution

2.2.5.2 Substitution

What is the difference between the following expressions? What are the similarities?

```
1 + 2 + 3

6
```

See the solution

2.3 Literal Objects

We have already covered some of Scala's basic types. In this section we're going to round out that knowledge by covering all of Scala's *literal expressions*. A literal expression represents a fixed value that stands "for itself". Here's an example:

```
42
// res: Int = 42
```

This interaction at the REPL shows us that the literal `42` evaluates to the `Int` `42`.

Don't confuse a literal with the value it evaluates to! The literal expression is the representation in the program text before the program is run, and the value is the representation in the computer's memory after the program has run.

If you have prior programming experience, particularly Java experience, the literals in Scala should be familiar to you.

2.3.1 Numbers

Numbers share the same types available in Java: `Int` for 32-bit integers, `Double` for 64-bit floating point, `Float` for 32-bit floating point, and `Long` for 64-bit integers.

```
42
// res: Int = 42

42.0
// res: Double = 42.0

42.0f
// res: Float = 42.0

42L
// res: Long = 42
```

Scala also has 16-bit `Short` integers and 8-bit `Bytes`, but there is no literal syntax for creating them. Instead, we create them using methods called `toShort` and `toByte`.

2.3.2 Booleans

Booleans are exactly the same as Java: `true` or `false`.

```
true
// res: Boolean = true

false
// res: Boolean = false
```

2.3.3 Characters

`Chars` are 16-bit Unicode values written as a single character enclosed in single quotes.

```
'a'
// res: Char = a
```

> **Scala vs Java's Type Hierarchy**
>
> Although they are written with initial capitals, Scala's `Int`, `Double`, `Float`, `Long`, `Short`, `Byte`, `Boolen` and `Char` refer to exactly the same things as `int`, `double`, `float`, `long`, `short`, `byte`, `boolean`, and `char` in Java.
>
> In Scala all of these types act like objects with methods and fields. However, once your code is compiled, a Scala `Int` is exactly the same as a Java `int`. This makes interoperability between the two languages a breeze.

2.3.4 Strings

Strings are exactly Java's strings, and are written the same way.

```
"this is a string"
// res: java.lang.String = this is a string

"the\nusual\tescape characters apply"
// res: java.lang.String =
// the
```

```
//     usual escape characters apply
```

2.3.5 Null

Null is the same as Java, though not used nearly as often. Scala's `null` also has its own type: `Null`.

```
null
// res: Null = null
```

> **Using Nulls in Scala**
>
> Although `null`s are common in Java code, they are considered very bad practice in Scala.
>
> The main use of `null` in Java is to implement *optional* values that have some or no value at different points of a program's execution. However, `null` values cannot be checked by the compiler, leading to possible runtime errors in the form of `NullPointerException`s.
>
> Later we will see that Scala has the means to define optional values that *are* checked by the compiler. This removes the necessity of using `null`, making our programs much safer.

2.3.6 Unit

Unit, written `()`, is the Scala equivalent of Java's `void`. Unit is the result of expressions that evaluate to no interesting value, such as printing to standard output using `println`. The console doesn't print unit but we can ask for the type of an expression to see that unit is in fact the result of some expressions.

```
()

:type ()
// Unit

println("something")
// something

:type println("something")
// Unit
```

Unit is an important concept in Scala. Many of Scala's syntactic constructs are *expressions* that have types and values. We need a placeholder for expressions that don't yield a useful value, and unit provides just that.

2.3.7 Take home points

In this section we have seen *literal* expressions, which evaluate to basic data types. These basics types are mostly identical to Java, except for `Unit` which has no equivalent.

We note that every literal expression has a *type*, and evaluates to a *value*—something which is also true for more complex Scala expressions.

In the next section we will learn how to define our own object literals.

2.3.8 Exercises

2.3.8.1 Literally Just Literals

What are the values and types of the following Scala literals?

```
42

true

123L

42.0
```

See the solution

2.3.8.2 Quotes and Misquotes

What is the difference between the following literals? What is the type and value of each?

```
'a'

"a"
```

See the solution

2.3.8.3 An Aside on Side-Effects

What is the difference between the following expressions? What is the type and value of each?

```
"Hello world!"

println("Hello world!")
```

See the solution

2.3.8.4 Learning By Mistakes

What is the type and value of the following literal? Try writing it on the REPL or in a Scala worksheet and see what happens!

```
'Hello world!'
```

See the solution

2.4 Object Literals

So far we've seen how to create objects of built-in types like `Int` and `String` and combine them into expressions. In this section we will see how to create objects of our own design using *object literals*.

When we write an object literal we use a *declaration*, which is a different kind of program to an expression. A declaration does not evaluate to a value. Instead is gives a name to a value. This name can then be used to refer to the value in other code.

We can declare an empty object as follows:

```
object Test {}
```

This is not an expression—it does not evaluate to a value. Rather, it binds a name (`Test`) to a value (an empty object).

Once we have bound the name `Test` we can use it in expressions, where it evaluates to the object we have declared. The simplest expression is just the name on its own, which evaluates to the value itself:

```
Test
// res: Test.type = Test$@1668bd43
```

This expression is equivalent to writing a literal like `123` or `"abc"`. Note that the type of the object is reported as `Test.type`. This is not like any type we've seen before—it's a new type, created just for our object, called a *singleton type*. We cannot create other values of this type.

Empty objects are not so useful. Within the body (between the braces) of an object declaration we can put expressions. It is more common, however, to put declarations such as declaring methods, fields, or even more objects.

Object Declaration Syntax

The syntax for declaring an object is

```
object name {
  declarationOrExpression ...
}
```

where

- `name` is the name of the object; and
- the optional `declarationOrExpressions` are declarations or expressions.

Let's see how to declare methods and fields.

2.4.1 Methods

We interact with objects via methods so let's create an object with a method.

```
object Test2 {
  def name: String = "Probably the best object ever"
}
```

Here we've create a method called `name`. We can call it in the usual way.

```
Test2.name
// res: String = Probably the best object ever
```

Here's an object with a more complex method:

```
object Test3 {
  def hello(name: String) =
    "Hello " + name
}
defined object Test3

Test3.hello("Noel")
// res: String = Hello Noel
```

Method Declaration Syntax

The syntax for declaring a method is

```
def name(parameter: type, ...): resultType =
  bodyExpression
```

or

```
def name: resultType =
  bodyExpression
```

where

- `name` is the name of the method;
- the optional `parameters` are the names given to parameters to the method;
- the `types` are the types of the method parameters;
- the optional `resultType` is the type of the result of the method;
- the `bodyExpression` is an expression that calling the method evaluates to.

Method parameters are optional, but if a method has parameters their type must be given. Although the result type is optional it is good practice to define it as it serves as (machine checked!) documentation.

The term *argument* may be used interchangably with *parameter*.

Return is Implicit

The return value of the method is determined by evaluating the body—there is no need to write `return` like you would in Java.

2.4.2 Fields

An object can also contain other objects, called *fields*. We introduce these using the keywords `val` or `var`, which look similar to `def`:

```
object Test4 {
  val name = "Noel"
  def hello(other: String): String =
    name + " says hi to " + other
}

Test4.hello("Dave")
// res: String = Noel says hi to Dave
```

Field Declaration Syntax

The syntax for declaring a field is

```
val name: type = valueExpression
```

or

```
var name: type = valueExpression
```

where

- `name` is the name of the field;
- the optional `type` declaration gives the type of the field;
- the `valueExpression` evaluates to the object that is bound to the `name`.

Using `val` defines an *immutable* field, meaning we cannot change the value bound to the name. A `var` field is *mutable*, allowing us to change the bound value.

Always prefer `val` to `var`. Scala programmers prefer to use immutable fields wherever possible, as this maintains substitution. While you will no doubt create the occassional mutable field in your application code, we will stay away from `var` for most of this course and you should do the same in your Scala programming.

2.4.3 Methods versus fields

You might wonder why we need fields when we can have methods of no arguments that seem to work the same. The difference is subtle—a field gives a name to a value, whereas a method gives a name to a computation that produces a value.

Here's an object that shows the difference:

```
object Test7 {
   val simpleField = {
     println("Evaluating simpleField")
     42
   }
   def noParameterMethod = {
     println("Evaluating noParameterMethod")
     42
   }
}
```

Here we have used a `println` expression to print something to the console, and a block expression (expressions surrounded by { and }) to group expressions. We'll see more about block expressions in the next section.

Notice how the console says we've defined a object, but it hasn't run either of our `println` statements? This is due to a quirk of Scala and Java called *lazy loading*.

Objects and classes (which we'll see later) aren't loaded until they are referenced by other code. This is what prevents Scala loading the entire standard library into memory to run a simple `"Hello world!"` app.

Let's force Scala to evaluate our object body by referencing `Test7` in an expression:

```
Test7
// Evaluating simpleField
// res: Test7.type = Test7$@b22e8c9
```

When the object is first loaded, Scala runs through its definitions and calculates the values of each of its fields. This results in the code printing `"Evaluating simpleField"` as a side-effect.

The body expression of a field is run only once after which the final value is stored in the object. The expression is never evaluated again—notice the lack of `println` output below.

```
Test7.simpleField
// res: Int = 42

Test7.simpleField
// res: Int = 42
```

The body of a method, on the other hand, is evaluated every time we call the method—notice the repreated println output below.

```
Test7.noParameterMethod
// Evaluating noParameterMethod
// res: Int = 42

Test7.noParameterMethod
// Evaluating noParameterMethod
// res: Int = 42
```

2.4.4 Take home points

In this section we have created our own objects, given them methods and fields, and referenced them in expressions.

We have seen the syntax for declaring objects

```
object name {
  declarationOrExpression ...
}
```

for declaring methods

```
def name(parameter: type, ...): resultType = bodyExpression
```

and for declaring fields

```
val name = valueExpression
var name = valueExpression
```

All of these are *declarations*, binding names to values. Declarations are different to expressions. They do not evaluate to a value and do not have a type.

We have also seen the difference between methods and fields—fields refer to values stored within an object, whereas methods refer to computations that produce values.

2.4.5 Exercises

2.4.5.1 Cat-o-matique

The table below shows the names, colour, and favourite foods of three cats. Define an object for each cat. (For experienced programmers: we haven't covered classes yet.)

Name	Colour	Food
Oswald	Black	Milk
Henderson	Ginger	Chips
Quentin	Tabby and white	Curry

See the solution

2.4.5.2 Square Dance!

Define an object called `calc` with a method `square` that accepts a `Double` as an argument and... you guessed it... squares its input. Add a method called `cube` that cubes its input *calling* `square` as part of its result calculation.

See the solution

2.4.5.3 Precise Square Dance!

Copy and paste `calc` from the previous exercise to create a `calc2` that is generalized to work with `Int`s as well as `Double`s. If you have Java experience, this should be fairly straightforward. If not, read the solution below.

See the solution

2.4.5.4 Order of evaluation

When entered on the console, what does the following program output, and what is the type and value of the final expression? Think carefully about the types, dependencies, and evaluation behaviour of each field and method.

```
object argh {
  def a = {
    println("a")
    1
  }

  val b = {
    println("b")
    a + 2
  }

  def c = {
    println("c")
    a
    b + "c"
  }
}

argh.c + argh.b + argh.a
```

See the solution

2.4.5.5 Greetings, human

Define an object called `person` that contains fields called `firstName` and `lastName`. Define a second object called `alien` containing a method called `greet` that takes your person as a parameter and returns a greeting using their `firstName`.

What is the type of the `greet` method? Can we use this method to greet other objects?

See the solution

2.4.5.6 The Value of Methods

Are methods values? Are they expressions? Why might this be the case?

See the solution

2.5 Writing Methods

In the previous section we saw the syntax of methods. One of our main goals in this course is to go beyond syntax and give you systematic methods for constructing Scala programs. This is our first section dealing with such matters. In this section we're going to look at a systematic method for constructing methods. As you gain experience with Scala you can drop some of the steps of this method, but we *strongly* suggest you follow this method during the course.

To make the advice concrete we'll use this exercise from the previous section as an example:

Define an object called `calc` with a method `square` that accepts a `Double` as an argument and... you guessed it... squares its input. Add a method called `cube` that cubes its input, calling `square` as part of its result calculation.

2.5.1 Identify the Input and Output

Your first step is to identify the types of the input parameters, if any, and the result of the method.

In many cases the exercises will tell you the types and you can just read them straight from the description. In the example above the input type is given as `Double`. The result type we can infer is also `Double`.

2.5.2 Prepare Test Cases

Types alone don't tell all the story. There are many `Double` to `Double` functions, but few that implement squaring. Thus we should prepare some test cases that illustrate the expected behaviour of the method.

We're not going to use a testing library in this course, as we're trying to avoid external dependencies. We can implement a poor-man's testing library using the `assert` function that Scala provides. For our `square` example we might have test cases like

```
assert(square(2.0) == 4.0)
assert(square(3.0) == 9.0)
assert(square(-2.0) == 4.0)
```

2.5.3 Write the Declaration

With types and test cases ready we can now write the method declaration. We haven't developed the body yet so use `???`, another nifty Scala feature, in its place.

```
def square(in: Double): Double =
  ???
```

This step should be mechanical given the information gathered in the previous steps.

2.5.4 Run the Code

Run the code and check it compiles (and thus we haven't made any typos) and also that our tests fail (and thus are testing something). You may need to place the tests after the method declaration.

2.5.5 Write the Body

We're now ready to write the body of our method. We will develop a number of techniques for this throughout the course. For now, we're going to look at two techniques.

2.5.5.1 Consider the Result Type

The first technique is to look at the result type, in this case `Double`. How can we create `Double` values? We could write a literal, but that obviously won't be correct in this case. The other way we know to create a `Double` is to call a method on some object, which brings us to the next technique.

2.5.5.2 Consider the Input Type

Our next technique is to look at the type of input parameters to the method. In this case we have a `Double`. We have established we need to create a `Double`, so what methods can we call to create a `Double` from our input? There are many such methods, and here we have to use our domain knowledge to select * as the correct method to call.

We can now write our complete method as

```
def square(in: Double): Double =
  in * in
```

2.5.6 Run the Code, Again

Finally we should run the code again and check that the tests all pass in this case.

This is very simple example but practicing the process now will serve you well for the more complicated examples we will encounter later.

Process for Writing Methods

We have a six-step process for writing methods in a systematic way.

1. Identify the type of the inputs and output of the method.
2. Write some test cases for the expected output of the method given example input. We can use the `assert` function to write down these cases.
3. Write the method declaration using `???` for the body like so:

```
def name(parameter: type, ...): resultType =
 ???
```

4. Run the code to check the test cases do in fact fail.
5. Write the body of the method. We currently have two techniques to apply here:

   - consider the result type and how we can create an instance of it; and
   - consider the input type and methods we can call to transform it to the result type.

6. Run the code again and check the test cases pass.

2.6 Compound Expressions

We have almost finished our basic introduction to Scala. In this section we are going to look at two special kinds of expressions, *conditionals* and *blocks*, we will need in more complicated programs.

2.6.1 Conditionals

A conditional allows us to choose an expression to evaluate based on some condition. For example, we can choose a string based on which of two numbers is the smallest.

```
if(1 < 2) "Yes" else "No"
// res: String = Yes
```

Conditionals are Expressions

Scala's `if` statement has the same syntax as Java's. One important difference is that *Scala's conditional is an expression*—it has a type and returns a value.

The expression that is not selected does not get evaluated. This is apparent if we use an expression with a side-effect.

```
if(1 < 2) println("Yes") else println("No")
// Yes
```

We can tell the expression `println("No")` is not evaluated because `No` is not output to the console.

Conditional Expression Syntax

The syntax for a conditional expression is

```
if(condition)
  trueExpression
else
  falseExpression
```

where

- `condition` is an expression with `Boolean` type;
- `trueExpression` is the expression evaluated if `condition` evaluates to `true`; and
- `falseExpression` is the expression evaluated if `condition` evaluates to `false`.

2.6.2 Blocks

Blocks are expressions that allow us to sequence computations together. They are written as a pair of braces containing sub-expressions separated by semicolons or newlines.

```
{ 1; 2; 3 }
// warning: a pure expression does nothing in statement position; you may be omitting
//          necessary parentheses
//  { 1; 2; 3 }
//    ^
// warning: a pure expression does nothing in statement position; you may be omitting
//          necessary parentheses
//  { 1; 2; 3 }
//       ^
// res: Int = 3
```

As you can see, executing this code causes the console to raise a number of warnings and return the `Int` value 3.

A block is a sequence of expressions or declarations surrounded by braces. A block is also an expression: it executes each of its sub-expressions in order and returns the value of the last expression.

Why execute 1 and 2 if we're going to throw their values away? This is a good question, and is the reason the Scala compiler raised those warnings above.

One reason to use a block is to use code that produces side-effects before calculating a final value:

```
{
  println("This is a side-effect")
  println("This is a side-effect as well")
  3
}
// This is a side-effect
// This is a side-effect as well
// res: Int = 3
```

We can also use a block when we want to name intermediate results, such as

```
def name: String = {
  val title = "Professor"
  val name = "Funkenstein"
  title + " " + name
}

name
// res: String = Professor Funkenstein
```

Block Expression Syntax

The syntax of a block expression is

```
{
   declarationOrExpression ...
   expression
}
```

where

- the optional `declarationOrExpressions` are declarations or expression; and
- `expression` is an expression determining the type and value of the block expression.

2.6.3 Take home points

Conditional expressions allow us to choose an expression to evaluate based on a `Boolean` condition. The syntax is

```
if(condition)
  trueExpression
else
  falseExpression
```

A conditional, being an expression, has a type and evaluates to an object.

A block allows us to sequence expressions and declarations. It is commonly used when we want to sequence expressions with side-effects, or name intermediate results in a computation. The syntax is

```
{
   declarationOrExpression ...
   expression
}
```

The type and value of a block is that of the last expression in the block.

2.6.4 Exercises

2.6.4.1 A Classic Rivalry

What is the type and value of the following conditional?

```
if(1 > 2) "alien" else "predator"
```

See the solution

2.6.4.2 A Less Well Known Rivalry

What about this conditional?

```
if(1 > 2) "alien" else 2001
```

See the solution

2.6.4.3 An if Without an else

What about this conditional?

```
if(false) "hello"
```

See the solution

2.7 Conclusion

We have had a very brief introduction to the fundamentals of Scala:

- expressions, which evaluate to values; and
- declarations, which gives names to values.

We've seen how we can write literals for many objects, and use method calls and compound expressions to create new objects from existing ones.

We have also declared our own objects, and constructed methods and fields.

Next we're going to see how a new kind of declaration, a class, provides a template for creating objects. Classes allow us to reuse code and unify similar objects will a common type.

Chapter 3

Objects and Classes

In the previous chapter we saw how to create objects and interact with them via method calls. In this section we're going to see how we can abstract over objects using *classes*. Classes are a template for constructing objects. Given a class we can make many objects that have the same type and share common properties.

3.1 Classes

A class is a template for creating objects that have similar methods and fields. In Scala a class also defines a type, and objects created from a class all share the same type. This allows us to overcome the problem we had in the *Greetings*, *Human* exercise in the last chapter.

3.1.1 Defining a Class

Here is a declaration for a simple `Person` class:

```
class Person {
  val firstName = "Noel"
  val lastName = "Welsh"
  def name = firstName + " " + lastName
}
```

Like an object declaration, a class declaration binds a name (in this case `Person`) and is not an expression. However, unlike an object name, we cannot use a class name in an expression. A class is not a value, and there is a different *namespace* in which classes live.

```
Person
// error: not found: value Person
//  Person
//  ^
```

We can create a new `Person` object using the `new` operator. Objects are values and we access their methods and fields in the usual way:

```
val noel = new Person
// noel: Person = Person@3235186a

noel.firstName
```

```
// res: String = Noel
```

Notice the type of the object is `Person`. Each call to new creates a distinct object of the same type:

```
noel // noel is the object that prints '@3235186a'
// res: Person = Person@3235186a

val newNoel = new Person // each new object prints a new number
// newNoel: Person = Person@2792b987

val anotherNewNoel = new Person
// anotherNewNoel: Person = Person@63ee4826
```

This means we can write a method that takes any `Person` as a parameter:

```
object alien {
  def greet(p: Person) =
    "Greetings, " + p.firstName + " " + p.lastName
}

alien.greet(noel)
// res: String = Greetings, Noel Welsh

alien.greet(newNoel)
// res: String = Greetings, Noel Welsh
```

> **Java Tip**
>
> Scala classes are all subclasses of `java.lang.Object` and are, for the most part, usable from Java as well as Scala. The default printing behaviour of `Person` comes from the `toString` method defined in `java.lang.Object`.

3.1.2 Constructors

As it stands our `Person` class is rather useless: we can create as many new objects as we want but they all have the same `firstName` and `lastName`. What if we want to give each person a different name?

The solution is to introduce a *constructor*, which allows us to pass parameters to new objects as we create them:

```
class Person(first: String, last: String) {
  val firstName = first
  val lastName = last
  def name = firstName + " " + lastName
}

val dave = new Person("Dave", "Gurnell")
// dave: Person = Person@3ed12df7

dave.name
// res: String = Dave Gurnell
```

The constructor parameters `first` and `last` can only be used within the body of the class. We must declare a field or method using `val` or `def` to access data from outside the object.

Constructor arguments and fields are often redundant. Fortunately, Scala provides us a useful short-hand way of declaring both in one go. We can prefix constructor parameters with the `val` keyword to have Scala define fields for them automatically:

```
class Person(val firstName: String, val lastName: String) {
  def name = firstName + " " + lastName
}

new Person("Dave", "Gurnell").firstName
// res: String = Dave
```

`val` fields are *immutable*—they are initialized once after which we cannot change their values. Scala also provides the `var` keyword for defining *mutable* fields.

Scala programmers tend to prefer to write immutability and side-effect-free code so we can reason about it using the substitution model. In this course we will concentrate almost exclusively on immutable `val` fields.

Class Declaration Syntax

The syntax for declaring a class is

```
class Name(parameter: type, ...) {
  declarationOrExpression ...
}
```

or

```
class Name(val parameter: type, ...) {
  declarationOrExpression ...
}
```

where

- `Name` is the name of the class;
- the optional `parameters` are the names given to constructor parameters;
- the `types` are the types of the constructor parameters;
- the optional `declarationOrExpressions` are declarations or expressions.

3.1.3 Default and Keyword Parameters

All Scala methods and constructors support *keyword parameters* and *default parameter values*.

When we call a method or constructor, we can *use parameter names as keywords* to specify the parameters in an arbitrary order:

```
new Person(lastName = "Last", firstName = "First")
// res: Person = Person(First,Last)
```

This comes in doubly useful when used in combination with *default parameter values*, defined like this:

```
def greet(firstName: String = "Some", lastName: String = "Body") =
  "Greetings, " + firstName + " " + lastName + "!"
```

If a parameter has a default value we can omit it in the method call:

```
greet("Awesome")
// res: String = Greetings, Awesome Body!
```

Combining keywords with default parameter values let us skip earlier parameters and just provide values for later ones:

```
greet(lastName = "Dave")
// res: String = Greetings, Some Dave!
```

Keyword Parameters

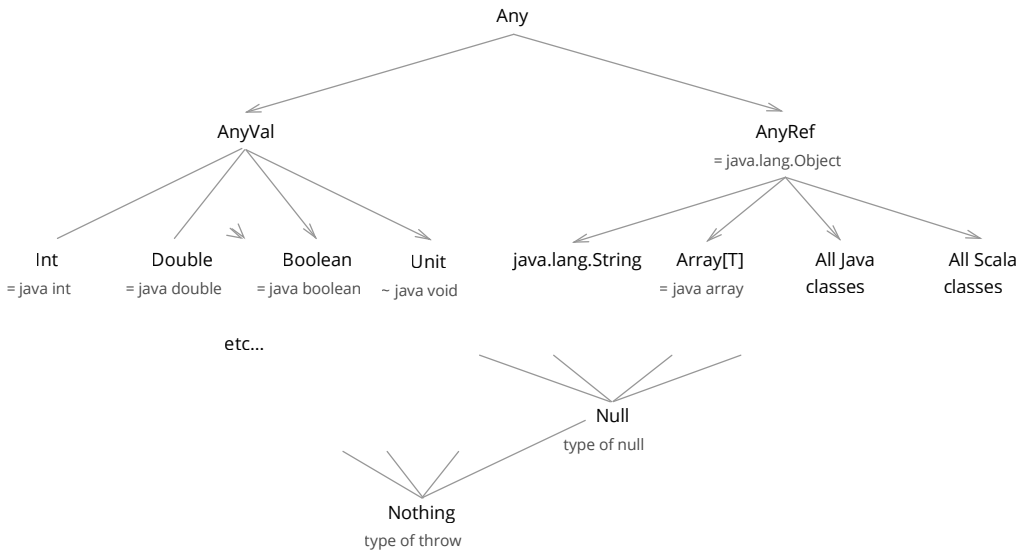
Keyword parameters are robust to changes in the number and order of parameters. For example, if we add a `title` parameter to the `greet` method, the meaning of keywordless method calls changes but keyworded calls remain the same:

```
def greet(title: String = "Citizen", firstName: String = "Some", lastName: String = "Body") =
  "Greetings, " + title + " " + firstName + " " + lastName + "!"

greet("Awesome") // this is now incorrect
// res: String = Greetings, Awesome Some Body

greet(firstName = "Awesome") // this is still correct
// res: String = Greetings, Citizen Awesome Body
```

This is particularly useful when creating methods and constructors with a large number of parameters.

3.1.4 Scala's Type Hierarchy

Unlike Java, which separates primitive and object types, everything in Scala is an object. As a result, "primitive" value types like `Int` and `Boolean` form part of the same type hierarchy as classes and traits.



Scala has a grand supertype called `Any`, under which there are two types, `AnyVal` and `AnyRef`. `AnyVal` is the supertype of all value types, which `AnyRef` is the supertype of all "reference types" or classes. All Scala and Java classes are subtypes of `AnyRef`[1].

[1] We can actually define subtypes of `AnyVal`, which are known as value classes. These are useful in a few specialised circumstances and we're not going to discuss them here.

Some of these types are simply Scala aliases for types that exist in Java: `Int` is `int`, `Boolean` is `boolean`, and `AnyRef` is `java.lang.Object`.

There are two special types at the *bottom* of the hierarchy. `Nothing` is the type of `throw` expressions, and `Null` is the type of the value `null`. These special types are subtypes of everything else, which helps us assign types to `throw` and `null` while keeping other types in our code sane. The following code illustrates this:

```
def badness = throw new Exception("Error")
// badness: Nothing

null
// res: Null = null

val bar = if(true) 123 else badness
// bar: Int = 123

val baz = if(false) "it worked" else null
// baz: String = null
```

Although the types of `badness` and `res` are `Nothing` and `Null` respectively, the types of `bar` and `baz` are still sensible. This is because `Int` is the least common supertype of `Int` and `Nothing`, and `String` is the least common supertype of `String` and `Null`.

3.1.5 Take Home Points

In this section we learned how to define *classes*, which allow us to create many objects with the same *type*. Thus, classes let us *abstract across objects* that have similar properties.

The properties of the objects of a class take the form of *fields* and *methods*. Fields are pre-computed values stored within the object and methods are computations we can call.

The syntax for declaring classes is

```
class Name(parameter: type, ...) {
  declarationOrExpression ...
}
```

We create objects from a class by calling the constructor using the keyword `new`.

We also learned about *keyword parameters* and *default parameters*.

Finally we learned about Scala's type hierarchy, including the overlap with Java's type hierarchy, the special types `Any`, `AnyRef`, `AnyVal`, `Nothing`, `Null`, and `Unit`, and the fact that Java and Scala classes both occupy the same subtree of the type hierarchy.

3.1.6 Exercises

We now have enough machinery to have some fun playing with classes.

3.1.6.1 Cats, Again

Recall the cats from a previous exercise:

Name	Colour	Food
Oswald	Black	Milk
Henderson	Ginger	Chips
Quentin	Tabby and white	Curry

Define a class `Cat` and then create an object for each cat in the table above.

See the solution

3.1.6.2 Cats on the Prowl

Define an object `ChipShop` with a method `willServe`. This method should accept a `Cat` and return `true` if the cat's favourite food is chips, and false otherwise.

See the solution

3.1.6.3 Directorial Debut

Write two classes, `Director` and `Film`, with fields and methods as follows:

- `Director` should contain:
  - a field `firstName` of type `String`
  - a field `lastName` of type `String`
  - a field `yearOfBirth` of type `Int`
  - a method called `name` that accepts no parameters and returns the full name
- `Film` should contain:
  - a field `name` of type `String`
  - a field `yearOfRelease` of type `Int`
  - a field `imdbRating` of type `Double`
  - a field `director` of type `Director`
  - a method `directorsAge` that returns the age of the director at the time of release
  - a method `isDirectedBy` that accepts a `Director` as a parameter and returns a `Boolean`

Copy-and-paste the following demo data into your code and adjust your constructors so that the code works without modification:

```
val eastwood          = new Director("Clint", "Eastwood", 1930)
val mcTiernan         = new Director("John", "McTiernan", 1951)
val nolan             = new Director("Christopher", "Nolan", 1970)
val someBody          = new Director("Just", "Some Body", 1990)

val memento           = new Film("Memento", 2000, 8.5, nolan)
val darkKnight        = new Film("Dark Knight", 2008, 9.0, nolan)
val inception         = new Film("Inception", 2010, 8.8, nolan)

val highPlainsDrifter = new Film("High Plains Drifter", 1973, 7.7, eastwood)
val outlawJoseyWales  = new Film("The Outlaw Josey Wales", 1976, 7.9, eastwood)
val unforgiven        = new Film("Unforgiven", 1992, 8.3, eastwood)
val granTorino        = new Film("Gran Torino", 2008, 8.2, eastwood)
val invictus          = new Film("Invictus", 2009, 7.4, eastwood)

val predator          = new Film("Predator", 1987, 7.9, mcTiernan)
```

```
val dieHard           = new Film("Die Hard", 1988, 8.3, mcTiernan)
val huntForRedOctober = new Film("The Hunt for Red October", 1990, 7.6, mcTiernan)
val thomasCrownAffair = new Film("The Thomas Crown Affair", 1999, 6.8, mcTiernan)

eastwood.yearOfBirth         // should be 1930
dieHard.director.name        // should be "John McTiernan"
invictus.isDirectedBy(nolan) // should be false
```

Implement a method of `Film` called `copy`. This method should accept the same parameters as the constructor and create a new copy of the film. Give each parameter a default value so you can copy a film changing any subset of its values:

```
highPlainsDrifter.copy(name = "L'homme des hautes plaines")
// returns Film("L'homme des hautes plaines", 1973, 7.7, /* etc */)

thomasCrownAffair.copy(yearOfRelease = 1968,
  director = new Director("Norman", "Jewison", 1926))
// returns Film("The Thomas Crown Affair", 1926, /* etc */)

inception.copy().copy().copy()
// returns a new copy of `inception`
```

See the solution

3.1.6.4 A Simple Counter

Implement a `Counter` class. The constructor should take an `Int`. The methods `inc` and `dec` should increment and decrement the counter respectively returning a new `Counter`. Here's an example of the usage:

```
new Counter(10).inc.dec.inc.inc.count
// res: Int = 12
```

See the solution

3.1.6.5 Counting Faster

Augment the `Counter` from the previous exercise to allow the user can optionally pass an `Int` parameter to `inc` and `dec`. If the parameter is omitted it should default to `1`.

See the solution

3.1.6.6 Additional Counting

Here is a simple class called `Adder`.

```
class Adder(amount: Int) {
  def add(in: Int) = in + amount
}
```

Extend `Counter` to add a method called `adjust`. This method should accept an `Adder` and return a new `Counter` with the result of applying the `Adder` to the `count`.

See the solution

3.2 Objects as Functions

In the final exercise of the previous section, we defined a class called `Adder`:

```
class Adder(amount: Int) {
  def add(in: Int): Int = in + amount
}
```

In the discussion we described an `Adder` as an object representing a computation—a bit like having a method that we can pass around as a value.

This is such a powerful concept that Scala has a fully blown set of language features for creating objects that behave like computations. These objects are called *functions*, and are the basis of *functional programming*.

3.2.1 The apply method

For now we are going to look at just one of Scala's features supporting functional programming—*function application syntax*.

In Scala, by convention, an object can be "called" like a function if it has a method called `apply`. Naming a method `apply` affords us a special shortened call syntax: `foo.apply(args)` becomes `foo(args)`.

For example, let's rename the `add` method in `Adder` to `apply`:

```
class Adder(amount: Int) {
  def apply(in: Int): Int = in + amount
}

val add3 = new Adder(3)
// add3: Adder = Adder@1d4f0fb4

add3(2) // shorthand for add3.apply(2)
// res: Int = 5
```

With this one simple trick, objects can "look" syntactically like functions. There are lots of things that we can do with objects that we can't do with methods, including assign them to variables and pass them around as arguments.

> **Function Application Syntax**
>
> The method call `object.apply(parameter, ...)` can also be written as `object(parameter, ...)`

3.2.2 Take home points

In this section we looked at *function application syntax*, which lets us "call" an object as if it is a function.

Function application syntax is available for any object defining an `apply` method.

With function application syntax, we now have first class values that behave like computations. Unlike methods, objects can be passed around as data. This takes us one step closer towards true functional programming in Scala.

3.2.3 Exercises

3.2.3.1 When is a Function not a Function?

We'll get a chance to write some code at the end of the next section. For now we should think about an important theoretical question:

How close does function application syntax get us to creating truly reusable objects to do computations for us? What are we missing?

See the solution

3.3 Companion Objects

Sometimes we want to create a method that logically belongs to a class but is independent of any particular object. In Java we would use a *static method* for this, but Scala has a simpler solution that we've seen already: singleton objects.

One common use case is auxiliary constructors. Although Scala does have syntax that lets us define multiple constructors for a class, Scala programmers almost always prefer to implement additional constructors as `apply` methods on an object with the same name as the class. We refer to the object as the *companion object* of the class. For example:

```
class Timestamp(val seconds: Long)

object Timestamp {
  def apply(hours: Int, minutes: Int, seconds: Int): Timestamp =
    new Timestamp(hours*60*60 + minutes*60 + seconds)
}

Timestamp(1, 1, 1).seconds
// res: Long = 3661
```

> **Using the Console Effectively**
>
> Note our use of the `:paste` command in the transcript above. Companion objects must be defined in the same compilation unit as the classes they support. In a normal codebase this simply means defining the class and object in the same file, but on the REPL we have to enter then in one command using `:paste`.
>
> You can enter `:help` on the REPL to find out more.

As we saw earlier, Scala has two namespaces: a space of *type names* and a space of *value names*. This separation allows us to name our class and companion object the same thing without conflict.

It is important to note that *the companion object is not an instance of the class*—it is a singleton object with its own type:

```
Timestamp // note that the type is `Timestamp.type`, not `Timestamp`
// res: Timestamp.type = Timestamp$@602b24e6
```

Companion Object Syntax

To define a companion object for a class, in the *same file* as the class define an object with the same name.

```
class Name {
  ...
}

object Name {
  ...
}
```

3.3.1 Take home points

Companion objects provide us with a means to associate functionality with a class without associating it with any instance of that class. They are commonly used to provide additional constructors.

Companion objects *replace Java's static methods*. They provide equivalent functionality and are more flexible.

A companion object has the same name as its associated class. This doesn't cause a naming conflict because Scala has two namespaces: the namespace of values and the namespace of types.

A companion object must be defined in the same file as the associated class. When typing on the REPL, the class and companion object must be entered in the same block of code using `:paste` mode.

3.3.2 Exercises

3.3.2.1 Friendly Person Factory

Implement a companion object for `Person` containing an `apply` method that accepts a whole name as a single string rather than individual first and last names.

Tip: you can split a `String` into an `Array` of components as follows:

```
val parts = "John Doe".split(" ")
// parts: Array[String] = Array(John, Doe)

parts(0)
// res: String = John
```

See the solution

3.3.2.2 Extended Body of Work

Write companion objects for `Director` and `Film` as follows:

- the `Director` companion object should contain:
  - an `apply` method that accepts the same parameters as the constructor of the class and returns a new `Director`;
  - a method `older` that accepts two `Directors` and returns the oldest of the two.

- the `Film` companion object should contain:
  - an `apply` method that accepts the same parameters as the constructor of the class and returns a new `Film`;
  - a method `highestRating` that accepts two `Films` and returns the highest `imdbRating` of the two;
  - a method `oldestDirectorAtTheTime` that accepts two `Films` and returns the `Director` who was oldest at the respective time of filming.

See the solution

3.3.2.3 Type or Value?

The similarity in naming of classes and companion objects tends to cause confusion for new Scala developers. When reading a block of code it is important to know which parts refer to a class or *type* and which parts refer to a singleton object or *value*.

This is the inspiration for the new hit quiz, *Type or Value?*, which we will be piloting below. In each case identify whether the word `Film` refers to the type or value:

```
val prestige: Film = bestFilmByChristopherNolan()
```

See the solution

```
new Film("Last Action Hero", 1993, mcTiernan)
```

See the solution

```
Film("Last Action Hero", 1993, mcTiernan)
```

See the solution

```
Film.newer(highPlainsDrifter, thomasCrownAffair)
```

See the solution

Finally a tough one...

```
Film.type
```

See the solution

3.4 Case Classes

Case classes are an exceptionally useful shorthand for defining a class, a companion object, and a lot of sensible defaults in one go. They are ideal for creating lightweight data-holding classes with the minimum of hassle.

Case classes are created simply by prepending a class definition with the keyword `case`:

```
case class Person(firstName: String, lastName: String) {
  def name = firstName + " " + lastName
}
```

Whenever we declare a case class, Scala automatically generates a *class and companion object*:

```
val dave = new Person("Dave", "Gurnell") // we have a class
// dave: Person = Person(Dave,Gurnell)

Person // and a companion object too
// res: Person.type = Person
```

What's more, the class and companion are pre-populated with some very useful features.

3.4.1 Features of a case class

1. *A field for each constructor argument*—we don't even need to write `val` in our constructor definition, although there's no harm in doing so.

```
dave.firstName
// res: String = Dave
```

2. *A default `toString` method* that prints a sensible constructor-like representation of the class (no more `@` signs and cryptic hex numbers):

```
dvae
// res: Person = Person("Dave","Gurnell")
```

3. *Sensible `equals`, and `hashCode` methods* that operate on the field values in the object.

This makes it easy to use case classes with collections like `Lists`, `Sets` and `Maps`. It also means we can compare objects on the basis of their contents rather than their reference identity:

```
new Person("Noel", "Welsh").equals(new Person("Noel", "Welsh"))
// res: Boolean = true

new Person("Noel", "Welsh") == new Person("Noel", "Welsh")
// res: Boolean = true
```

4. *A `copy` method* that creates a new object with the same field values as the current one:

```
dave.copy()
// res: Person = Person(Dave,Gurnell)
```

Note that the `copy` method creates and returns a *new object* of the class rather than returning the current one:

```
dave.copy() eq res0
// res: Boolean = false
```

The `copy` method actually accepts optional parameters matching each of the constructor parameters. If a parameter is specified the new object uses that value instead of the existing value from the current object. This is ideal for use with keyword parameters to let us copy an object while changing the values of one or more fields:

```
dave.copy(firstName = "Dave2")
// res: Person = Person(Dave2,Gurnell)

dave.copy(lastName = "Gurnell2")
// res: Person = Person(Dave,Gurnell2)
```

Value and Reference Equality

Scala's == operator is different from Java's—it delegates to `equals` rather than comparing values on reference identity.

Scala has an operator called `eq` with the same behaviour as Java's ==. However, it is rarely used in application code:

```
new Person("Noel", "Welsh") eq (new Person("Noel", "Welsh"))
// res: Boolean = false

dave eq dave
// res: Boolean = true
```

3.4.2 Features of a case class companion object

The companion object contains an `apply` method with the same arguments as the class constructor. Scala programmers tend to prefer the `apply` method over the constructor for the brevity of omitting `new`, which makes constructors much easier to read inside expressions:

```
Person("Dave", "Gurnell") == Person("Noel", "Welsh")
// res: Boolean = false

Person("Dave", "Gurnell") == Person("Dave", "Gurnell")
// res: Boolean = true
```

Finally, the companion object also contains code to implement an *extractor pattern* for use in *pattern matching*. We'll see this later this chapter.

Case Class Declaration Syntax

The syntax to declare a case class is

```
case class Name(parameter: type, ...) {
  declarationOrExpression ...
}
```

where

- `Name` is the name of the case class;
- the optional `parameters` are the names given to constructor parameters;

- the `types` are the types of the constructor parameters;
- the optional `declarationOrExpressions` are declarations or expressions.

3.4.3 Case objects

A final note. If you find yourself defining a case class with no constructor arguments you can instead a define a *case object*. A case object is defined just like a case class and has the same default methods as a case class.

```
case object Citizen {
  def firstName = "John"
  def lastName  = "Doe"
  def name = firstName + " " + lastName
}
```

The differences between a case object and a regular singleton object are:

- The `case object` keyword defines a class and an object, and makes the object an instance (actually the only instance) of the class:

```
class Citizen { /* ... */ }
object Citizen extends Citizen { /* ... */ }
```

- With a case object we still get all of the functionality defined for case classes above.

3.4.4 Take Home Points

Case classes are the *bread and butter of Scala data types*. Use them, learn them, love them.

The syntax for declaring a case class is the same as for declaring a class, but with `case` appended

```
case class Name(parameter: type, ...) {
  declarationOrExpression ...
}
```

Case classes have numerous auto-generated methods and features that save typing. We can override this behaviour on a piece-by-piece basis by implementing the relevant methods ourselves.

In Scala 2.10 and earlier we can define case classes containing 0 to 22 fields. In Scala 2.11 we gain the ability to define arbitrarily-sized case classes.

3.4.5 Exercises

3.4.5.1 Case Cats

Recall that a `Cat` has a `String` colour and food. Define a case class to represent a `Cat`.

See the solution

3.4.5.2 Roger Ebert Said it Best...

> No good movie is too long and no bad movie is short enough.

The same can't always be said for code, but in this case we can get rid of a lot of boilerplate by converting `Director` and `Film` to case classes. Do this conversion and work out what code we can cut.

See the solution

3.4.5.3 Case Class Counter

Reimplement `Counter` as a case class, using `copy` where appropriate. Additionally initialise `count` to a default value of `0`.

See the solution

3.4.5.4 Application, Application, Application

What happens when we define a companion object for a case class? Let's see.

Take our `Person` class from the previous section and turn it into a case class (hint: the code is above). Make sure you still have the companion object with the alternate `apply` method as well.

See the solution

3.5 Pattern Matching

Until now we have interacted with objects by calling methods or accessing fields. With case classes we can interact in another way, via *pattern matching*.

Pattern matching is like an extended `if` expression that allows us to evaluate an expression depending on the "shape" of the data. Recall the `Person` case class we've seen in previous examples:

```
case class Person(firstName: String, lastName: String)
```

Now imagine we wanted to implement a `Stormtrooper` that is looking for members of the rebellion. We could use pattern matching like this:

```
object Stormtrooper {
  def inspect(person: Person): String =
    person match {
      case Person("Luke", "Skywalker") => "Stop, rebel scum!"
      case Person("Han", "Solo") => "Stop, rebel scum!"
      case Person(first, last) => s"Move along, $first"
    }
}
```

Notice the syntax for a pattern (`Person("Luke", "Skywalker")`) matches the syntax for constructing the object the pattern matches (`Person("Luke", "Skywalker")`).

Here it is in use:

```
Stormtrooper.inspect(Person("Noel", "Welsh"))
// res: String = Move along, Noel

Stormtrooper.inspect(Person("Han", "Solo"))
// res: String = Stop, rebel scum!
```

Pattern Matching Syntax

The syntax of a pattern matching expression is

```
expr0 match {
  case pattern1 => expr1
  case pattern2 => expr2
  ...
}
```

where

- the expression `expr0` evaluates to the value we match;
- the patterns, or *guards*, `pattern1`, `pattern2`, and so on are checked against this value in order; and
- the right-hand side expression (`expr1`, `expr2`, and so on) of the first pattern that matches is evaluated[a].

Pattern matching is itself an expression and thus evaluates to a value—the value of the matched expression.

[a] In reality patterns are compiled to a more efficient form than a sequence of tests, but the semantics are the same.

3.5.1 Pattern Syntax

Scala has an expressive syntax for writing patterns or guards. For case classes the pattern syntax matches the constructor syntax. Take the data

```
Person("Noel", "Welsh")
```

A pattern to match against the `Person` type is written

```
Person(pat0, pat1)
```

where `pat0` and `pat1` are patterns to match against the `firstName` and `lastName` respectively. There are four possible patterns we could use in place of `pat0` or `pat1`:

1. A name, which matches any value at that position and binds it to the given name. For example, the pattern `Person(first, last)` binds the name `first` to the value `"Noel"`, and the name `last` to the value `"Welsh"`.

2. An underscore (_), which matches any value and ignores it. For example, as Stormtroopers only care about the first name of ordinary citizens we could just write `Person(first, _)` to avoid binding a name to the value of the `lastName`.

3. A literal, which successfully matches only the value the literal respresents. So , for example, the pattern `Person("Han", "Solo")` matches the `Person` with first name `"Han"` and last name `"Solo"`.

4. Another case class using the same constructor style syntax.

Note there is a lot more we can do with pattern matching, and pattern matching is actually extensible. We'll look at these features in a later section.

3.5.2 Take Home Points

Case classes allow a new form of interaction, called *pattern matching*. Pattern matching allows us to take apart a case class, and evaluate different expressions depending on what the case class contains.

The syntax for pattern matching is

```
expr0 match {
  case pattern1 => expr1
  case pattern2 => expr2
  ...
}
```

A pattern can be one of

1. a name, binding any value to that name;
2. an underscore, matching any value and ignoring it;
3. a literal, matching the value the literal denotes; or
4. a constructor-style pattern for a case class.

3.5.3 Exercises

3.5.3.1 Feed the Cats

Define an object `ChipShop` with a method `willServe`. This method should accept a `Cat` and return true if the cat's favourite food is chips, and false otherwise. Use pattern matching.

See the solution

3.5.3.2 Get Off My Lawn!

In this exercise we're going to write a simulator of my Dad, the movie critic. It's quite simple: any movie directed by Clint Eastwood gets a rating 10.0, any movie directed by John McTiernan gets a 7.0, while any other movie gets a 3.0. Implement an object called `Dad` with a method `rate` which accepts a `Film` and returns a `Double`. Use pattern matching.

See the solution

3.6 Conclusions

In this section we've explored *classes*. We have seen that classes allow us to abstract over objects. That is, to define objects that share properties in common and have a common type.

We also looked at *companion objects*, which are used in Scala to define auxillary constructors and other utility methods that don't belong on a class.

Finally, we introduced *case classes*, which greatly reduce boilerplate code and allow *pattern-matching*, a new way of interacting with objects, in addition to method calls.

Chapter 4

Modelling Data with Traits

We looked in depth at classes in the previous chapter. Classes provide us with a way to abstract over objects that have similar properties, allowing us to write code that works with any object in a class.

In this chapter we explore *abstraction over classes*, allowing us to write code that works with objects of different classes. We achieve this with a mechanism called *traits*.

This chapter also marks a change in our focus. In previous chapters we have addressed the technical aspects of constructing Scala code. In this chapter we will initially focus on the technical aspects of traits. Our focus will then change to using Scala as a *medium to express our thoughts*.

We will see how we can mechanically transform a description of data, called an *algebraic datatype*, into code. Using *structural recursion* we can mechanically write code that transforms an algebraic datatype.

4.1 Traits

Traits are templates for creating classes, in the same way that classes are templates for creating objects. Traits allow us to express that two or more classes can be considered the same, and thus both implement the same operations. In other words, traits allow us to express that multiple classes share a common super-type (outside of the `Any` super-type that all classes share).

> **Traits vs Java Interfaces**
>
> Traits are very much like Java 8's *interfaces* with *default methods*. If you have not used Java 8, you can think of traits as being like a cross between interfaces and *abstract classes*.

4.1.1 An Example of Traits

Let's start with an example of a trait. Imagine we're modelling visitors to a website. There are two types of visitor: those who have registered on our site and those who are anonymous. We can model this with two classes:

```
import java.util.Date

case class Anonymous(id: String, createdAt: Date = new Date())

case class User(
```

```
  id: String,
  email: String,
  createdAt: Date = new Date()
)
```

With these class definitions we're saying that both anonymous and registered visitors have an id and a creation date. But we only know the email address of registered visitors.

There is obvious duplication here, and it would be nice to not have to write the same definitions twice. More important though, is to create some common type for the two kinds of visitors. If they had some type in common (other than `AnyRef` and `Any`) we could write methods that worked on any kind of visitor. We can do this with a trait like so:

```
import java.util.Date

trait Visitor {
  def id: String      // Unique id assigned to each user
  def createdAt: Date // Date this user first visited the site

  // How long has this visitor been around?
  def age: Long = new Date().getTime - createdAt.getTime
}

case class Anonymous(id: String, createdAt: Date = new Date()) extends Visitor

case class User(
  id: String,
  email: String,
  createdAt: Date = new Date()
) extends Visitor
```

Note the two changes:

- we defined the trait `Visitor`; and
- we declared that `Anonymous` and `User` are subtypes of the `Visitor` trait by using the `extends` keyword.

The `Visitor` trait expresses an interface that any subtype must implement: they must implement a `String` called `id` and a `createdAt` `Date`. Any sub-type of `Visitor` also automatically has a method `age` as defined in `Visitor`.

By defining the `Visitor` trait we can write methods that work with any subtype of visitor, like so:

```
scala> def older(v1: Visitor, v2: Visitor): Boolean =
         v1.createdAt.before(v2.createdAt)

scala> older(Anonymous("1"), User("2", "test@example.com"))
older(Anonymous("1"), User("2", "test@example.com"))
res4: Boolean = true
```

Here the method `older` can be called with either an `Anonymous` or a `User` as they are both subtypes of `Visitor`.

> **Trait Syntax**
>
> To declare a trait we write

```
trait TraitName {
  declarationOrExpression ...
}
```

To declare that a class is a subtype of a trait we write

```
class Name(...) extends TraitName {
  ...
}
```

More commonly we'll use case classes, but the syntax is the same

```
case class Name(...) extends TraitName {
 ...
}
```

4.1.2 Traits Compared to Classes

Like a class, a trait is a named set of field and method definitions. However, it differs from a class in a few important ways:

- *A trait cannot have a constructor*—we can't create objects directly from a trait. Instead we can use a trait to create a class, and then create objects from that class. We can base as many classes as we like on a trait.
- Traits can define *abstract methods* that have names and type signatures but no implementation. We saw this in the `Visitor` trait. We must specify the implementation when we create a class that extends the trait, but until that point we're free to leave definitions abstract.

Let's return to the `Visitor` trait to further explore abstract definitions. Recall the definition of `Visitor` is

```
import java.util.Date

trait Visitor {
  def id: String      // Unique id assigned to each user
  def createdAt: Date // Date this user first visited the site

  // How long has this visitor been around?
  def age: Long = new Date().getTime - createdAt.getTime
}
```

`Visitor` prescribes two abstract methods. That is, methods which do not have an implementation but must be implemented by extending classes. These are `id` and `createdAt`. It also defines a concrete method, `age`, that is defined in terms of one of the abstract methods.

`Visitor` is used as a building block for two classes: `Anonymous` and `User`. Each class `extends Visitor`, meaning it inherits all of its fields and methods:

```
scala> Anonymous("anon1")
res14: Anonymous = Anonymous(anon1)

scala> res14.createdAt
res15: java.util.Date = Mon Mar 24 15:11:45 GMT 2014
```

```
scala> res14.age
res16: Long = 8871
```

`id` and `createdAt` are abstract so they must be defined in extending classes. Our classes implement them as `vals` rather than `defs`. This is legal in Scala, which sees `def` as a more general version of `val`[1]. It is good practice to never define `vals` in a trait, but rather to use `def`. A concrete implementation can then implement it using using a `def` or `val` as appropriate.

4.1.3 Take Home Points

Traits are a way of *abstracting over classes* that have similar properties, just like classes are a way of abstracting over objects.

Using a traits has two parts. Declaring the trait

```
trait TraitName {
  declarationOrExpression ...
}
```

and extending the trait from a class (usually a case class)

```
case class Name(...) extends TraitName {
  ...
}
```

4.1.4 Exercises

4.1.4.1 Cats, and More Cats

Demand for Cat Simulator 1.0 is exploding! For v2 we're going to go beyond the domestic cat to model `Tigers`, `Lions`, and `Panthers` in addition to the `Cat`. Define a trait `Feline` and then define all the different species as subtypes of `Feline`. To make things interesting, define:

- on `Feline` a `colour` as before;
- on `Feline` a `String sound`, which for a cat is `"meow"` and is `"roar"` for all other felines;
- only `Cat` has a favourite food; and
- `Lions` have an `Int maneSize`.

See the solution

4.1.4.2 Shaping Up With Traits

Define a trait called Shape and give it three abstract methods:

- `sides` returns the number of sides;
- `perimeter` returns the total length of the sides;
- `area` returns the area.

[1] This is all part of the uniform access principle we saw in the exercises for Object Literals.

Implement `Shape` with three classes: `Circle`, `Rectangle`, and `Square`. In each case provide implementations of each of the three methods. Ensure that the main constructor parameters of each shape (e.g. the radius of the circle) are accessible as fields.

Tip: The value of π is accessible as `math.Pi`.

See the solution

4.1.4.3 Shaping Up 2 (Da Streets)

The solution from the last exercise delivered three distinct types of shape. However, it doesn't model the relationships between the three correctly. A `Square` isn't just a `Shape`—it's also a type of `Rectangle` where the width and height are the same.

Refactor the solution to the last exercise so that `Square` and `Rectangle` are subtypes of a common type `Rectangular`.

Tip: A trait can extend another trait.

See the solution

4.2 This or That and Nothing Else: Sealed Traits

In many cases we can enumerate all the possible classes that can extend a trait. For example, we previously modelled a website visitor as `Anonymous` or a logged in `User`. These two cases cover all the possibilities as one is the negation of the other. We can model this case with a *sealed trait*, which allows the compiler to provide extra checks for us.

We create a sealed trait by simply writing `sealed` in front of our trait declaration:

```
sealed trait Visitor {
  def id: String
  def createdAt: Date
  def age: Long = new Date().getTime() - createdAt.getTime()
}
```

When we mark a trait as `sealed` we *must* define all of its subtypes in the same file. Once the trait is sealed, the compiler knows the complete set of subtypes and will warn us if a pattern matching expression is missing a case:

```
scala> def missingCase(v: Visitor) =
          v match {
            case User(_, _, _) => "Got a user"
          }
<console>:21: warning: match may not be exhaustive.
It would fail on the following input: Anonymous(_, _)
               v match {
               ^
missingCase: (v: Visitor)String
```

We will *not* get a similar warning from an unsealed trait.

We can still extend the subtypes of a sealed trait outside of the file where they are defined. For example, we could extend `User` or `Anonymous` further elsewhere. If we want to prevent this possibility we should declare them as `sealed` (if we want to allow extensions within the file) or `final` if we want to disallow all extensions. For the visitors example it probably doesn't make sense to allow any extension to `User` or `Anonymous`, so the simplified code should look like this:

```
sealed trait Visitor { /* ... */ }
final case class User(/* ... */) extends Visitor
final case class Anonymous(/* ... */) extends Visitor
```

This is a very powerful pattern and one we will use frequently.

Sealed Trait Pattern

If all the subtypes of a trait are known, seal the trait

```
sealed trait TraitName {
  ...
}
```

Consider making subtypes `final` if there is no case for extending them

```
final case class Name(...) extends TraitName {
  ...
}
```

Remember subtypes must be defined in the same file as a sealed trait.

4.2.1 Take home points

Sealed traits and final (case) classes allow us to control extensibility of types. *The majority of cases* should use the sealed trait / final case class pattern.

```
sealed trait TraitName { ... }
final case class Name(...) extends TraitName
```

The main advantages of this pattern are:

- the compiler will warn if we miss a case in pattern matching; and
- we can control extension points of sealed traits and thus make stronger guarantees about the behaviour of subtypes.

4.2.2 Exercises

4.2.2.1 Printing Shapes

Let's revisit the `Shapes` example from the previous section.

First make `Shape` a sealed trait. Then write a singleton object called `Draw` with an `apply` method that takes a `Shape` as an argument and returns a description of it on the console. For example:

```
Draw(Circle(10))      // returns "A circle of radius 10cm"

Draw(Rectangle(3, 4)) // returns "A rectangle of width 3cm and height 4cm"

// and so on...
```

Finally, verify that the compiler complains when you comment out a `case` clause.

See the solution

4.2.2.2 The Color and the Shape

Write a sealed trait `Color` to make our shapes more interesting.

- give `Color` three properties for its RGB values;
- create three predefined colours: `Red`, `Yellow`, and `Pink`;
- provide a means for people to produce their own custom `Colors` with their own RGB values;
- provide a means for people to tell whether any `Color` is "light" or "dark".

A lot of this exercise is left deliberately open to interpretation. The important thing is to practice working with traits, classes, and objects.

Decisions such as how to model colours and what is considered a light or dark colour can either be left up to you or discussed with other class members.

Edit the code for `Shape` and its subtypes to add a colour to each shape.

Finally, update the code for `Draw.apply` to print the colour of the argument as well as its shape and dimensions:

- if the argument is a predefined colour, print that colour by name:

```
Draw(Circle(10, Yellow)) // returns "A yellow square of size 10cm"
```

- if the argument is a custom colour rather than a predefined one, print the word "light" or "dark" instead.

You may want to deal with the colour in a helper method.

See the solution

4.2.2.3 A Short Division Exercise

Good Scala developers don't just use types to model data. Types are a great way to put artificial limitations in place to ensure we don't make mistakes in our programs. In this exercise we will see a simple (if contrived) example of this—using types to prevent division by zero errors.

Dividing by zero is a tricky problem—it can lead to exceptions. The JVM has us covered as far as floating point division is concerned but integer division is still a problem:

```
scala> 1.0 / 0.0
res0: Double = Infinity

scala> 1 / 0
java.lang.ArithmeticException: / by zero
```

Let's solve this problem once and for all using types!

Create an object called `divide` with an `apply` method that accepts two `Ints` and returns `DivisionResult`. `DivisionResult` should be a sealed trait with two subtypes: a `Finite` type encapsulating the result of a valid division, and an `Infinite` type representing the result of dividing by `0`.

Here's some example usage:

```
scala> divide(1, 2)
res7: DivisionResult = Finite(0)

scala> divide(1, 0)
res8: DivisionResult = Infinite
```

Finally, write a sample function that calls `divide`, matches on the result, and returns a sensible description.

See the solution

4.3 Modelling Data with Traits

In this section we're going to shift our focus from language features to programming patterns. We're going to look at modelling data and learn a process for expressing in Scala any data model defined in terms of *logical ors and ands*. Using the terminology of object-oriented programming, we will express *is-a* and *has-a* relationships. In the terminology of functional programming we are learning about *sum* and *product types*, which are together called *algebraic data types*.

Our goal in this section is to see how to translate a data model into Scala code. In the next section we'll see patterns for code that uses algebraic data types.

4.3.1 The Product Type Pattern

Our first pattern is to model data that contains other data. We might describe this as "`A` *has a* `B` *and* `C`". For example, a `Cat` has a colour and a favourite food; a `Visitor` has an id and a creation date; and so on.

The way we write this is to use a case class. We've already done this many times in exercises; now we're formalising the pattern.

Product Type Pattern

If `A` has a `b` (with type `B`) and a `c` (with type `C`) write

```
case class A(b: B, c: C)
```

or

```
trait A {
  def b: B
  def c: C
}
```

4.4 The Sum Type Pattern

Our next pattern is to model data that is two or more distinct cases. We might describe this as "`A` *is a* `B` *or* `C`". For example, a `Feline` is a `Cat`, `Lion`, or `Tiger`; a `Visitor` is an `Anonymous` or `User`; and so on.

We write this using the sealed trait / final case class pattern.

Sum Type Pattern

If A is a B or C write

```
sealed trait A
final case class B() extends A
final case class C() extends A
```

4.4.1 Algebraic Data Types

An algebraic data type is any data that uses the above two patterns. In the functional programming literature, data using the "has-a and" pattern is known as a *product type*, and the "is-a or" pattern is a *sum type*.

4.4.2 The Missing Patterns

We have looked at relationships along two dimensions: is-a/has-a, and and/or. We can draw up a little table and see we only have patterns for two of the four table cells.

	And	Or
Is-a		Sum type
Has-a	Product type	

What about the missing two patterns?

The "is-a and" pattern means that A is a B and C. This pattern is in some ways the inverse of the sum type pattern, and we can implement it as

```
trait B
trait C
trait A extends B with C
```

In Scala a trait can extend as many traits as we like using the `with` keyword like `A extends B with C with D` and so on. We aren't going to use this pattern in this course. If we want to represent that some data conforms to a number of different interfaces we will often be better off using a *type class*, which we will explore later. There are, however, several legitimate uses of this pattern:

- for modularity, using what's known as the cake pattern; and
- sharing implementation across several classes where it doesn't make sense to make default implementations in the main trait.

The "has-a or" patterns means that A has a B or C. There are two ways we can implement this. We can say that A has a d of type D, where D is a B or C. We can mechanically apply our two patterns to implement this:

```
trait A {
  def d: D
}
sealed trait D
```

```
final case class B() extends D
final case class C() extends D
```

Alternatively we could implement this as `A` is a `D` or `E`, and `D` has a `B` and `E` has a `C`. Again this translates directly into code

```
sealed trait A
final case class D(b: B) extends A
final case class E(c: C) extends A
```

4.4.3 Take Home Points

We have seen that we can mechanically translate data using the "has-a and" and "is-a or" patterns (or, more succintly, the product and sum types) into Scala code. This type of data is known as an algebraic data type. Understanding these patterns is very important for writing idiomatic Scala code.

4.4.4 Exercises

4.4.4.1 Stop on a Dime

A traffic light is red, green, or yellow. Translate this description into Scala code.

See the solution

4.4.4.2 Calculator

A calculation may succeed (with an `Int` result) or fail (with a `String` message). Implement this.

See the solution

4.4.4.3 Water, Water, Everywhere

Bottled water has a size (an `Int`), a source (which is a well, spring, or tap), and a `Boolean` carbonated. Implement this in Scala.

See the solution

4.5 Working With Data

In the previous section we saw how to define algebraic data types using a combination of the sum (or) and product type (and) patterns. In this section we'll see a pattern for using algebraic data types, known as *structural recursion*. We'll actually see two variants of this pattern: one using *polymorphism* and one using *pattern matching*.

Structural recursion is the precise opposite of the process of building an algebraic data type. If `A` has a `B` and `C` (the product-type pattern), to construct an `A` we must have a `B` and a `C`. The sum and product type patterns tell us how to combine data to make bigger data. Structural recursion says that if we have an `A` as defined before, we must break it into its constituent `B` and `C` that we then combine in some way to get closer to our desired answer. Structural recursion is essentially the process of breaking down data into smaller pieces.

Just as we have two patterns for building algebraic data types, we will have two patterns for decomposing them using structural recursion. We will actually have two variants of each pattern, one using polymorphism, which

is the typical object-oriented style, and one using pattern matching, which is typical functional style. We'll end this section with some rules for choosing which pattern to use.

4.5.1 Structural Recursion using Polymorphism

Polymorphic dispatch, or just polymorphism for short, is a fundamental object-oriented technique. If we define a method in a trait, and have different implementations in classes extending that trait, when we call that method the implementation on the actual concrete instance will be used. Here's a very simple example. We start with a simple definition using the familiar product type (or) pattern.

```
sealed trait A {
  def foo: String
}
final case class B() extends A {
  def foo: String =
    "It's B!"
}
final case class C() extends A {
  def foo: String =
    "It's C!"
}
```

We declare a value with type `A` but we see the concrete implementation on `B` or `C` is used.

```
scala> val anA: A = B()
anA: A = B()

scala> anA.foo
res1: String = It's B!

scala> val anA: A = C()
anA: A = C()

scala> anA.foo
res2: String = It's C!
```

We can define an implementation in a trait, and change the implementation in an extending class using the `override` keyword.

```
sealed trait A {
  def foo: String =
    "It's A!"
}
final case class B() extends A {
  override def foo: String =
    "It's B!"
}
final case class C() extends A {
  override def foo: String =
    "It's C!"
}
```

The behaviour is as before; the implementation on the concrete class is selected.

```
scala> val anA: A = B()
anA: A = B()

scala> anA.foo
res3: String = It's B!
```

Remember that if you provide a default implementation in a trait, you should ensure that implementation is valid for all subtypes.

Now we understand how polymorphism works, how do we use it with an algebraic data types? We've actually seen everything we need, but let's make it explicit and see the patterns.

The Product Type Polymorphism Pattern

If `A` has a `b` (with type `B`) and a `c` (with type `C`), and we want to write a method `f` returning an `F`, simply write the method in the usual way.

```
case class A(b: B, c: C) {
  def f: F = ???
}
```

In the body of the method we must use `b`, `c`, and any method parameters to construct the result of type `F`.

The Sum Type Polymorphism Pattern

If `A` is a `B` or `C`, and we want to write a method `f` returning an `F`, define `f` as an abstract method on `A` and provide concrete implementations in `B` and `C`.

```
sealed trait A {
  def f: F
}
final case class B() extends A {
  def f: F =
    ???
}
final case class C() extends A {
  def f: F =
    ???
}
```

4.5.2 Structural Recursion using Pattern Matching

Structural recursion with pattern matching proceeds along the same lines as polymorphism. We simply have a case for every subtype, and each pattern matching case must extract the fields we're interested in.

The Product Type Pattern Matching Pattern

If `A` has a `b` (with type `B`) and a `c` (with type `C`), and we want to write a method `f` that accepts an `A` and returns an `F`, write

```
def f(a: A): F =
  a match {
    case A(b, c) => ???
  }
```

In the body of the method we use `b` and `c` to construct the result of type `F`.

The Sum Type Pattern Matching Pattern

If `A` is a `B` or `C`, and we want to write a method `f` accepting an `A` and returning an `F`, define a pattern matching case for `B` and `C`.

```
def f(a: A): F =
  a match {
    case B() => ???
    case C() => ???
  }
```

4.5.3 A Complete Example

Let's look at a complete example of the algebraic data type and structural recursion patterns, using our familiar `Feline` data type.

We start with a description of the data. A `Feline` is a `Lion`, `Tiger`, `Panther`, or `Cat`. We're going to simplify the data description, and just say that a `Cat` has a `String favouriteFood`. From this description we can immediately apply our pattern to define the data.

```
sealed trait Feline
final case class Lion() extends Feline
final case class Tiger() extends Feline
final case class Panther() extends Feline
final case class Cat(favouriteFood: String) extends Feline
```

Now let's implement a method using both polymorphism and pattern matching. Our method, `dinner`, will return the appropriate food for the feline in question. For a `Cat` their dinner is their `favouriteFood`. For `Lions` it is antelope, for `Tigers` it is tiger food, and for `Panthers` it is licorice.

We could represent food as a `String`, but we can do better and represent it with a type. This avoids, for example, spelling mistakes in our code. So let's define our `Food` type using the now familiar patterns.

```
sealed trait Food
final case object Antelope extends Food
final case object TigerFood extends Food
final case object Licorice extends Food
final case class CatFood(food: String) extends Food
```

Now we can implement `dinner` as a method returning `Food`. First using polymorphism:

```
sealed trait Feline {
  def dinner: Food
}
final case class Lion() extends Feline {
  def dinner: Food =
    Antelope
}
final case class Tiger() extends Feline {
  def dinner: Food =
    TigerFood
}
final case class Panther() extends Feline {
  def dinner: Food =
    Licorice
}
final case class Cat(favouriteFood: String) extends Feline {
  def dinner: Food =
    CatFood(favouriteFood)
}
```

Now using pattern matching. We actually have two choices when using pattern matching. We can implement our code in a single method on `Feline` or we can implement it in a method on another object. Let's see both.

```
sealed trait Feline {
  def dinner: Food =
    this match {
      case Lion() => Antelope
      case Tiger() => TigerFood
      case Panther() => Licorice
      case Cat(favouriteFood) => CatFood(favouriteFood)
    }
}

object Diner {
  def dinner(feline: Feline): Food =
    feline match {
      case Lion() => Antelope
      case Tiger() => TigerFood
      case Panther() => Licorice
      case Cat(food) => CatFood(food)
    }
}
```

Note how we can directly apply the patterns, and the code falls out. This is the main point we want to make with structural recursion: the code follows the shape of the data, and can be produced in an almost mechanical way.

4.5.4 Choosing Which Pattern to Use

We have three way of implementing structural recursion:

1. polymorphism;
2. pattern matching in the base trait; and
3. pattern matching in an external object (as in the `Diner` example above).

Which should we use? The first two methods give the same result: a method defined on the classes of interest. We should use whichever is more convenient. This normally ends up being pattern matching on the base trait as it requires less code duplication.

When we implement a method in the classes of interest we can have only one implementation of the method, and everything that method requires to work must be contained within the class and parameters we pass to the method. When we implement methods using pattern matching in an external object we can provide multiple implementations, one per object (multiple `Diners` in the example above).

The general rule is: if a method only depends on other fields and methods in a class it is a good candidate to be implemented inside the class. If the method depends on other data (for example, if we needed a `Cook` to make dinner) consider implementing is using pattern matching outside of the classes in question. If we want to have more than one implementation we should use pattern matching and implement it outside the classes.

4.5.5 Object-Oriented vs Functional Extensibility

In classic functional programming style we have no objects, only data without methods and functions. This style of programming makes extensive use of pattern matching. We can mimic it in Scala using the algebraic data type pattern and pattern matching in methods defined on external objects.

Classic object oriented style uses polymorphism and allow open extension of classes. In Scala terms this means no sealed traits.

What are the tradeoffs we make in the two different styles?

One advantage of functional style is it allows the compiler to help us more. By sealing traits we are telling the compiler it knows all the possible subtypes of that trait. It can then tell us if we miss out a case in our pattern matching. This is especially useful if we add or remove subtypes later in development. We could argue we get the same benefit from object-oriented style, as we must implement all methods defined on the base trait in any subtypes. This is true, but in practice classes with a large number of methods are very difficult to maintain and we'll inevitably end up factoring some of the code into different classes – essentially duplicating the functional style.

This doesn't mean functional style is to be preferred in all cases. There is a fundamental difference between the kind of extensibility that object-oriented style and functional style gives us. With OO style we can easily add new data, by extending a trait, but adding a new method requires us to change existing code. With functional style we can easily add a new method but adding new data requires us to modify existing code. In tabular form:

	Add new method	Add new data
OO	Change existing code	Existing code unchanged
FP	Existing code unchanged	Change existing code

In Scala we have the flexibility to use both polymorphism and pattern matching, and we should use whichever is appropriate. However we generally prefer sealed traits as it gives us greater guarantees about our code's semantics, and we can use typeclasses, which we'll explore later, to get us OO-style extensibility.

4.5.6 Exercises

4.5.6.1 Traffic Lights

In the previous section we implemented a `TrafficLight` data type like so:

```
sealed trait TrafficLight
final case object Red extends TrafficLight
final case object Green extends TrafficLight
final case object Yellow extends TrafficLight
```

Using polymorphism and then using pattern matching implement a method called `next` which returns the next `TrafficLight` in the standard `Red -> Green -> Yellow -> Red` cycle. Do you think it is better to implement this method inside or outside the class? If inside, would you use pattern matching or polymorphism? Why?

See the solution

4.5.6.2 Calculation

In the last section we created a `Calculation` data type like so:

```
sealed trait Calculation
final case class Success(result: Int) extends Calculation
final case class Failure(reason: String) extends Calculation
```

We're now going to write some methods that use a `Calculation` to perform a larger calculation. These methods will have a somewhat unusual shape—this is a precursor to things we'll be exploring soon—but if you follow the patterns you will be fine.

Create a `Calculator` object. On `Calculator` define methods `+` and `-` that accept a `Calculation` and an `Int`, and return a new `Calculation`. Here are some examples

```
assert(Calculator.+(Success(1), 1) == Success(2))
assert(Calculator.-(Success(1), 1) == Success(0))
assert(Calculator.+(Failure("Badness"), 1) == Failure("Badness"))
```

See the solution

Now write a division method that fails if the divisor is 0. The following tests should pass. Note the behavior for the last test. This indicates "fail fast" behavior. If a calculation has already failed we keep that failure and don't process any more data even if, as is the case in the test, doing so would lead to another failure.

```
assert(Calculator./(Success(4), 2) == Success(2))
assert(Calculator./(Success(4), 0) == Failure("Division by zero"))
assert(Calculator./(Failure("Badness"), 0) == Failure("Badness"))
```

See the solution

4.5.6.3 Email

Recall the `Visitor` trait we looked at earlier: a website `Visitor` is either `Anonymous` or a signed-in `User`. Now imagine we wanted to add the ability to send emails to visitors. We can only email signed-in users, and sending an email requires a lot of knowledge about SMTP settings, MIME headers, and so on. Would an `email` method be better implemented using polymorphism on the `Visitor` trait or using pattern matching in an `EmailService` object? Why?

See the solution

4.6 Recursive Data

A particular use of algebraic data types that comes up very often is defining *recursive data*. This is data that is defined in terms of itself, and allows us to create data of potentially unbounded size (though any concrete instance will be finite).

We can't define recursive data like[2]

```
final case class Broken(broken: Broken)
```

as we could never actually create an instance of such a type—the recursion never ends. To define valid recursive data we must define a *base case*, which is the case that ends the recursion.

Here is a more useful recursive definition: an `IntList` is either the empty list `End`, or a `Pair`[3] containing an `Int` and an `IntList`. We can directly translate this to code using our familiar patterns:

```
sealed trait IntList
final case object End extends IntList
final case class Pair(head: Int, tail: IntList) extends IntList
```

Here `End` is the base case. We construct the list containing 1, 2, and 3 as follows:

```
Pair(1, Pair(2, Pair(3, End)))
```

This data structure is known as a singly-linked list. In this example we have four links in our chain. We can write this out in a longer form to better understand the structure of the list. Below, `d` represents an empty list, and `a`, `b`, and `c` are pairs built on top of it.

```
val d = End()
val c = Pair(3, d)
val b = Pair(2, c)
val a = Pair(1, b)
```

In addition to being links in a chain, these data structures all represent complete sequences of integers:

- `a` represents the sequence `1, 2, 3`
- `b` represents the sequence `2, 3`
- `c` represents the sequence 3 (only one element)
- `d` represents an empty sequence

Using this implementation, we can build lists of arbitrary length by repeatedly taking an existing list and prepending a new element[4].

[2] We actually can define data in this manner if we delay the construction of the recursive case, like `final case class LazyList(head: Int, tail: () => LazyList)`. This uses a feature of Scala, functions, that we haven't seen yet. We can do some fairly mind-bending things with this construction, such as defining an infinite stream of ones with the declaration `val ones: LazyList = LazyList(1, () => ones)`. Since we only ever realise a finite amount of this list we can use it to implement certain types of data that would be difficult to implement in other ways. If you're interested in exploring this area further, what we have implemented in called a lazy list, and an "odd lazy list" in particular. The "even list", described in How to add laziness to a strict language wihtout even being odd, is a better implementation. If you wish to explore further, there is a rich literature on lazy datastructures and more mind melting theory under the name of "coinductive data".

[3] The traditional name this element is a `Cons` cell. We don't use this name as it's a bit confusing if you don't know the story behind it.

[4] This is how Scala's built-in `List` data structure works. We will be introduced to `List` in the chapter on *Collections*.

We can apply the same structural recursion patterns to process a recursive algebraic data type. The only wrinkle is that we must make a recursive call when the data definition is recursion.

Let's add together all the elements of an `IntList`. We'll use pattern matching, but as we know the same process applies to using polymorphism.

Start with the tests and method declaration.

```
val example = Pair(1, Pair(2, Pair(3, End)))
assert(sum(example) == 6)
assert(sum(example.tail) == 5)
assert(sum(End) == 0)

def sum(list: IntList): Int = ???
```

Note how the tests define `0` to be the sum of the elements of an `End` list. It is important that we define an appropriate base case for our method as we will build our final result of this base case.

Now we apply our structural recursion pattern to fill out the body of the method.

```
def sum(list: IntList): Int =
  list match {
    case End => ???
    case Pair(hd, tl) => ???
  }
```

Finally we have to decide on the bodies of our cases. We have already decided that `0` is answer for `End`. For `Pair` we have two bits of information to guide us. We know we need to return an `Int` and we know that we need to make a recursive call on `tl`. Let's fill in what we have.

```
def sum(list: IntList): Int =
  list match {
    case End => 0
    case Pair(hd, tl) => ??? sum(tl)
  }
```

The recursive call will return the sum of the tail of the list, by definition. Thus the correct thing to do is to add hd to this result. This gives us our final result:

```
def sum(list: IntList): Int =
  list match {
    case End => 0
    case Pair(hd, tl) => hd + sum(tl)
  }
```

4.6.1 Understanding the Base Case and Recursive Case

Our patterns will carry us most of the way to a correct answer, but we still need to supply the method bodies for the base and recursive cases. There is some general guidance we can use:

- For the base case we should generally return the *identity* for the function we're trying to compute. The identity is an element that doesn't change the result. E.g. 0 is the identity for addition, because `a + 0 == a` for any `a`. If we were calculating the product of elements the identity would be 1 as `a * 1 == a` for all `a`.

- For the recursive case, assume the recursion will return the correct result and work out what you need to add to get the correct answer. We saw this for `sum`, where we assume the recursive call will give us the correct result for the tail of the list and we then just add on the head.

Recursive Algebraic Data Types Pattern

When defining recursive algebraic data types, there must be at least two cases: one that is recursive, and one that is not. Cases that are not recursive are known as base cases. In code, the general skeleton is:

```
sealed trait RecursiveExample
final case class RecursiveCase(recursion: RecursiveExample) extends RecursiveExample
final case object BaseCase extends RecursiveExample
```

Recursive Structural Recursion Pattern

When writing structurally recursive code on a recursive algebraic data type:

- whenever we encounter a recursive element in the data we make a recursive call to our method; and
- whenever we encounter a base case in the data we return the identity for the operation we are performing.

4.6.2 Tail Recursion

You may be concerned that recursive calls will consume excessive stack space. Scala can apply an optimisation, called *tail recursion*, to many recursive functions to stop them consuming stack space.

A tail call is a method call where the caller immediately returns the value. So this is a tail call

```
def method1: Int =
  1

def tailCall: Int =
  method1
```

because `tailCall` immediately returns the result of calling `method1` while

```
def notATailCall: Int =
  method1 + 2
```

because `notATailCall` does not immediatley return—it adds an number to the result of the call.

A tail call can be optimised to not use stack space. Due to limitations in the JVM, Scala only optimises tail calls where the caller calls itself. Since tail recursion is an important property to maintain, we can use the `@tailrec` annotation to ask the compiler to check that methods we believe are tail recursion really are. Here we have two versions of `sum` annotated. One is tail recursive and one is not. You can see the compiler complains about the method that is not tail recursive.

```
scala> import scala.annotation.tailrec
import scala.annotation.tailrec

scala> @tailrec
       def sum(list: IntList): Int =
         list match {
           case End => 0
           case Pair(hd, tl) => hd + sum(tl)
         }
<console>:15: error: could not optimize @tailrec annotated method sum: it contains a recursive call
      □
                    not in tail position
         list match {
         ^

scala> @tailrec
       def sum(list: IntList, total: Int = 0): Int =
         list match {
           case End => total
           case Pair(hd, tl) => sum(tl, total + hd)
         }
sum: (list: IntList, total: Int)Int
```

Any non-tail recursion function can be transformed into a tail recursive version by adding an accumulator as we have done with `sum` above. This transforms stack allocation into heap allocation, which sometimes is a win, and other times is not.

In Scala we tend not to work directly with tail recursive functions as there is a rich collections library that covers the most common cases where tail recursion is used. Should you need to go beyond this, because you're implementing your own datatypes or are optimising code, it is useful to know about tail recursion.

4.6.3 Exercises

4.6.3.1 A List of Methods

Using our definition of `IntList`

```
sealed trait IntList
final case object End extends IntList
final case class Pair(head: Int, tail: IntList) extends IntList
```

define a method `length` that returns the length of the list. There is test data below you can use to check your solution. For this exercise it is best to use pattern matching in the base trait.

```
val example = Pair(1, Pair(2, Pair(3, End)))

assert(example.length == 3)
assert(example.tail.length == 2)
assert(End.length == 0)
```

See the solution

Define a method to compute the product of the elements in an `IntList`. Test cases are below.

```
assert(example.product == 6)
assert(example.tail.product == 6)
assert(End.product == 1)
```

See the solution

Define a method to double the value of each element in an `IntList`, returning a new `IntList`. The following test cases should hold:

```
assert(example.double == Pair(2, Pair(4, Pair(6, End))))
assert(example.tail.double == Pair(4, Pair(6, End)))
assert(End.double == End)
```

See the solution

4.6.3.2 The Forest of Trees

A binary tree of integers can be defined as follows:

A `Tree` is a `Node` with a left and right `Tree` or a `Leaf` with an element of type `Int`.

Implement this algebraic data type.

See the solution

Implement `sum` and `double` on `Tree` using polymorphism and pattern matching.

See the solution

4.7 Extended Examples

To test your skills with algebraic data types and structural recursion here are some larger projects to attempt.

4.7.0.1 A Calculator

In this exercise we'll implement a simple interpreter for programs containing only numeric operations.

We start by defining some types to represent the expressions we'll be operating on. In the compiler literature this is known as an *abstract syntax tree*.

Our representation is:

- An `Expression` is an `Addition`, `Subtraction`, or a `Number`;
- An Addition has a `left` and `right` Expression;
- A Subtraction has a `left` and `right` Expression; or
- A Number has a `value` of type `Double`.

Implement this in Scala.

See the solution

Now implement a method `eval` that converts an `Expression` to a `Double`. Use polymorphism or pattern matching as you see fit. Explain your choice of implementation method.

See the solution

We're now going to add some expressions that call fail: division and square root. Start by extending the abstract syntax tree to include representations for `Division` and `SquareRoot`.

See the solution

Now we're going to change `eval` to represent that a computation can fail. (`Double` uses `NaN` to indicate a computation failed, but we want to be helpful to the user and tell them why the computation failed.) Implement an appropriate algebraic data type.

See the solution

Now change `eval` to return your result type, which I have called `Calculation` in my implementation. Here are some examples:

```
assert(Addition(SquareRoot(Number(-1.0)), Number(2.0)).eval ==
       Failure("Square root of negative number"))
assert(Addition(SquareRoot(Number(4.0)), Number(2.0)).eval == Success(4.0))
assert(Division(Number(4), Number(0)).eval == Failure("Division by zero"))
```

See the solution

4.7.0.2 JSON

In the calculator exercise we gave you the algebraic data type representation. In this exercise we want you to design the algebraic data type yourself. We're going to work in what is hopefully a familiar domain: [JSON][link-json].

Design an algebraic data type to represent JSON. Don't go directly to code. Start by sketching out the design in terms of logical ands and ors—the building blocks of algebraic data types. You might find it useful to use a notation similar to [BNF][link-bnf]. For example, we could represent the `Expression` data type from the previous exercise as follows:

```
Expression ::= Addition left:Expression right:Expression
             | Subtraction left:Expression right:Expression
             | Division left:Expression right:Expression
             | SquareRoot value:Expression
             | Number value:Int
```

This simplified notation allows us to concentrate on the structure of the algebraic data type without worrying about the intricacies of Scala syntax.

Note you'll need a sequence type to model JSON, and we haven't looked at Scala's collection library yet. However we have seen how to implement a list as an algebraic data type.

Here are some examples of JSON you'll need to be able to represent

```
["a string", 1.0, true]
{
  "a": [1,2,3],
  "b": ["a","b","c"]
  "c": { "doh":true, "ray":false, "me":1 }
}
```

See the solution

Translate your representation to Scala code.

See the solution

Now add a method to convert your JSON representation to a `String`. Make sure you enclose strings in quotes, and handle arrays and objects properly.

See the solution

Test your method works. Here are some examples using the representation I chose.

```
SeqCell(JsString("a string"), SeqCell(JsNumber(1.0), SeqCell(JsBoolean(true), SeqEnd))).print
// res: String = ["a string", 1.0, true]

ObjectCell(
  "a", SeqCell(JsNumber(1.0), SeqCell(JsNumber(2.0), SeqCell(JsNumber(3.0), SeqEnd))),
  ObjectCell(
    "b", SeqCell(JsString("a"), SeqCell(JsString("b"), SeqCell(JsString("c"), SeqEnd))),
    ObjectCell(
      "c", ObjectCell("doh", JsBoolean(true),
             ObjectCell("ray", JsBoolean(false),
               ObjectCell("me", JsNumber(1.0), ObjectEnd))),
      ObjectEnd
    )
  )
).print
// res: String = {"a": [1.0, 2.0, 3.0], "b": ["a", "b", "c"], "c": {"doh": true, "ray": false, "me":
     1.0}}
```

4.7.0.3 Music

In the JSON exercise there was a well defined specification to model. In this exercise we want to work on modelling skills given a rather fuzzy specification. The goal is to model music. You can choose to interpret this how you want, making your model as simple or complex as you like. The critical thing is to be able to justify the decisions you made, and to understand the limits of your model.

You might find it easiest to use the BNF notation, introduced in the JSON exercise, to write down your model.

See the solution

4.8 Conclusions

In this chapter we have made an extremely important change in our focus, away from language features and towards the programming patterns they support. This continues for the rest of the book.

We have explored two extremely important patterns: *algebraic data types* and *structural recursion*. These patterns allow us to go from a mental model of data, to the representation and processing of that data in Scala in an almost entirely mechanical way. Not only in the structure of our code formulaic, and thus easy to comprehend, but the compiler can catch common errors for us which makes development and maintenance easier. These two tools are among the most commonly used in idiomatic functional code, and it is hard to over-emphasize their importance.

In the exercises we developed a few common data structures, but we were limited to storing a fixed type of data, and our code contained a lot of repetition. In the next section we will look at how we can abstract over types and methods, and introduce some important concepts of sequencing operations.

Chapter 5

Sequencing Computations

In this section we're going to look at two more language features, *generics* and *functions*, and see some abstractions we can build using these features: *functors*, and *monads*.

Our starting point is code that we developed in the previous section. We developed `IntList`, a list of integers, and wrote code like the following:

```
sealed trait IntList {
  def length: Int =
    this match {
      case End => 0
      case Pair(hd, tl) => 1 + tl.length
    }
  def double: IntList =
    this match {
      case End => End
      case Pair(hd, tl) => Pair(hd * 2, tl.double)
    }
  def product: Int =
    this match {
      case End => 1
      case Pair(hd, tl) => hd * tl.product
    }
  def sum: Int =
    this match {
      case End => 0
      case Pair(hd, tl) => hd + tl.sum
    }
}
final case object End extends IntList
final case class Pair(head: Int, tail: IntList) extends IntList
```

There are two problems with this code. The first is that our list is restricted to storing `Int`s. The second problem is that here is a lot of repetition. The code has the same general structure, which is unsurprising given we're using our structural recursion pattern, and it would be nice to reduce the amount of duplication.

We will address both problems in this section. For the former we will use generics to *abstract over types*, so we can create data that works with user specified types. For the latter we will use functions to *abstract over methods*, so we can reduce duplication in our code.

As we work with these techniques we'll see some general patterns emerge. We'll name and investigate these patterns in more detail at the end of this section.

5.1 Generics

Generic types allow us to *abstract over types*. There are useful for all sorts of data structures, but commonly encountered in collections so that's where we'll start.

5.1.1 Pandora's Box

Let's start with a collection that is even simpler than our list—a box that stores a single value. We don't care what type is stored in the box, but we want to make sure we preserve that type when we get the value out of the box. To do this we use a generic type.

```
final case class Box[A](value: A)

Box(2)
// res0: Box[Int] = Box(2)

res0.value
// res1: Int = 2

Box("hi") // if we omit the type parameter, scala will infer its value
// res2: Box[String] = Box(hi)

res2.value
// res3: String = hi
```

The syntax `[A]` is called a *type parameter*. We can also add type parameters to methods, which limits the scope of the parameter to the method declaration and body:

```
def generic[A](in: A): A = in

generic[String]("foo")
// res: String = foo

generic(1) // again, if we omit the type parameter, scala will infer it
/ res: Int = 1
```

Type parameters work in a way analogous to method parameters. When we call a method we bind the method's parameter names to the values given in the method call. For example, when we call `generic(1)` the name `in` is bound to the value `1` within the body of `generic`.

When we call a method or construct a class with a type parameter, the type parameter is bound to the concrete type within the method or class body. So when we call `generic(1)` the type parameter `A` is bound to `Int` in the body of `generic`.

Type Parameter Syntax

We declare generic types with a list of type names within square brackets like `[A, B, C]`. By convention we use single uppercase letters for generic types.

Generic types can be declared in a class or trait declaration in which case they are visible throughout the rest of the declaration.

```
case class Name[A](...){ ... }
trait Name[A]{ ... }
```

Alternatively they may be declared in a method declaration, in which case they are only visible within the method.

```
def name[A](...){ ... }
```

5.1.2 Generic Algebraic Data Types

We described type parameters as analogous to method parameters, and this analogy continues when extending a trait that has type parameters. Extending a trait, as we do in a sum type, is the type level equivalent of calling a method and we must supply values for any type parameters of the trait we're extending.

In previous sections we've seen sum types like the following:

```
sealed trait Calculation
final case class Success(result: Double) extends Calculation
final case class Failure(reason: String) extends Calculation
```

Let's generalise this so that our result is not restricted to a `Double` but can be some generic type. In doing so let's change the name from `Calculation` to `Result` as we're not restricted to numeric calculations anymore. Now our data definition becomes:

A `Result` of type `A` is either a `Success` of type `A` or a `Failure` with a `String` reason. This translates to the following code

```
sealed trait Result[A]
case class Success[A](result: A) extends Result[A]
case class Failure[A](reason: String) extends Result[A]
```

Notice that both `Success` and `Failure` introduce a type parameter `A` which is passed to `Result` when it is extended. `Success` also has a value of type `A`, but `Failure` only introduces `A` so it can pass it onward to `Result`. In a later section we'll introduce *variance*, giving us a cleaner way to implement this, but for now this is the pattern we'll use.

Invariant Generic Sum Type Pattern

If A of type T is a B or C write

```
sealed trait A[T]
final case class B[T]() extends A[T]
final case class C[T]() extends A[T]
```

5.1.3 Exercises

5.1.3.1 Generic List

Our `IntList` type was defined as

```
sealed trait IntList
final case object End extends IntList
final case class Pair(head: Int, tail: IntList) extends IntList
```

Change the name to `LinkedList` and make it generic in the type of data stored in the list.

See the solution

5.1.3.2 Working With Generic Types

There isn't much we can do with our `LinkedList` type. Remember that types define the available operations, and with a generic type like `A` there isn't a concrete type to define any available operations. (Generic types are made concrete when a class is instantiated, which is too late to make use of the information in the definition of the class.)

However, we can still do some useful things with our `LinkedList`! Implement `length`, returning the length of the `LinkedList`. Some test cases are below.

```
val example = Pair(1, Pair(2, Pair(3, End())))
assert(example.length == 3)
assert(example.tail.length == 2)
assert(End().length == 0)
```

See the solution

On the JVM we can compare all values for equality. Implement a method `contains` that determines whether or not a given item is in the list. Ensure your code works with the following test cases:

```
val example = Pair(1, Pair(2, Pair(3, Empty())))
assert(example.contains(3) == true)
assert(example.contains(4) == false)
assert(Empty().contains(0) == false)
// This should not compile
// example.contains("not an Int")
```

See the solution

Implement a method `apply` that returns the nth item in the list

Hint: If you need to signal an error in your code (there's one situation in which you will need to do this), consider throwing an exception. Here is an example:

```
throw new Exception("Bad things happened")
```

Ensure your solution works with the following test cases:

```
val example = Pair(1, Pair(2, Pair(3, Empty())))
assert(example(0) == 1)
assert(example(1) == 2)
assert(example(2) == 3)
assert(try {
  example(3)
  false
} catch {
  case e: Exception => true
})
```

See the solution

Throwing an exception isn't cool. Whenever we throw an exception we lose type safety as there is nothing in the type system that will remind us to deal with the error. It would be much better to return some kind of result that encodes we can succeed or failure. We introduced such a type in this very section.

```
sealed trait Result[A]
case class Success[A](result: A) extends Result[A]
case class Failure[A](reason: String) extends Result[A]
```

Change `apply` so it returns a `Result`, with a failure case indicating what went wrong. Here are some test cases to help you:

```
assert(example(0) == Success(1))
assert(example(1) == Success(2))
assert(example(2) == Success(3))
assert(example(3) == Failure("Index out of bounds"))
```

See the solution

5.2 Functions

Functions allow us to *abstract over methods*, turning methods into values that we can pass around and manipulate within our programs.

Let's look at three methods we wrote that manipulate `IntList`.

```
sealed trait IntList {
  def length: Int =
    this match {
      case End => 0
      case Pair(hd, tl) => 1 + tl.length
    }
  def double: IntList =
    this match {
      case End => End
      case Pair(hd, tl) => Pair(hd * 2, tl.double)
    }
  def product: Int =
    this match {
      case End => 1
      case Pair(hd, tl) => hd * tl.product
    }
  def sum: Int =
    this match {
      case End => 0
      case Pair(hd, tl) => hd + tl.sum
    }
}
```

All of these methods have the same general pattern, which is not surprising as they all use structural recursion. It would be nice to be able to remove the duplication.

Let's start by focusing on the methods that return an `Int`: `length`, `product`, and `sum`. We want to write a method like

```
def abstraction(end: Int, f: ???): Int =
  this match {
    case End => end
    case Pair(hd, tl) => f(hd, tl.abstraction(f, end))
  }
```

I've used `f` to denote some kind of object that does the combination of the head and recursive call for the `Pair` case. At the moment we don't know how to write down the type of this value, or how to construct one. However, we can guess from the title of this section that what we want is a function!

A function is like a method: we can call it with parameters and it evaluates to a result. Unlike a method a function is value. We can pass a function to a method or to another function. We can return a function from a method, and so on.

Much earlier in this course we introduced the `apply` method, which lets us treat objects as functions in a syntactic sense:

```
object add1 {
  def apply(in: Int) = in + 1
}

add1(2)
// res: Int = 3
```

This is a big step towards doing real functional programming in Scala but we're missing one important component: *types*.

As we have seen, types allow us to abstract across values. We've seen all sorts of special case functions like `Adders` and `ActionListeners`, but what we really want is a generalised set of types that allow us to represent computations of any kind.

Enter Scala's `Function` types.

5.2.1 Function Types

We write a function type like `(A, B) => C` where `A` and `B` are the types of the parameters and `C` is the result type. The same pattern generalises from functions of no arguments to an arbitrary number of arguments.

In our example above we want `f` to be a function that accepts two `Int`s as parameters and returns an `Int`. Thus we can write it as `(Int, Int) => Int`.

Function Type Declaration Syntax

To declare a function type, write

```
(A, B, ...) => C
```

where

- `A, B, ...` are the types of the input parameters; and
- `C` is the type of the result.

If a function only has one parameter the parentheses may be dropped:

```
A => B
```

5.2.2 Function literals

Scala also gives us a *function literal syntax* specifically for creating new functions. Here are some example function literals:

```
val sayHi = () => "Hi!"
// sayHi: () => String = <function0>

sayHi()
// res: String = Hi!

val add1 = (x: Int) => x + 1
// add1: Int => Int = <function1>

add1(10)
// res: Int = 11

val sum = (x: Int, y:Int) => x + y
// sum: (Int, Int) => Int = <function2>

sum(10, 20)
// res: Int = 30
```

In code where we know the argument types, we can sometimes *drop the type annotations* and allow Scala to infer them[1]. There is no syntax for declaring the result type of a function and it is normally inferred, but if we find ourselves needing to do this we can put a type on the function's body expression:

```
(x: Int) => (x + 1): Int
```

Function Literal Syntax

The syntax for declaring a function literal is

```
(parameter: type, ...) => expression
```

where - the optional `parameters` are the names given to the function parameters; - the `types` are the types of the function parameters; and - the `expression` determines the result of the function.

5.2.3 Exercises

5.2.3.1 A Better Abstraction

We started developing an abstraction over `sum`, `length`, and `product` which we sketched out as

```
def abstraction(end: Int, f: ???): Int =
  this match {
    case End => end
    case Pair(hd, tl) => f(hd, tl.abstraction(f, end))
  }
```

[1] Note that we only can drop the parentheses around the argument list on single-argument functions—we still have to write `() => foo` and `(a, b) => foo` on functions of other arities.

Rename this function to `fold`, which is the name it is usually known as, and finish the implementation.

See the solution

Now reimplement `sum`, `length`, and `product` in terms of `fold`.

See the solution

Is it more convenient to rewrite methods in terms of `fold` if they were implemented using pattern matching or polymorphic? What does this tell us about the best use of `fold`?

See the solution

Why can't we write our `double` method in terms of `fold`? Is it feasible we could if we made some change to `fold`?

See the solution

Implement a generalised version of `fold` and rewrite `double` in terms of it.

See the solution

5.3 Generic Folds for Generic Data

We've seen that when we define a class with generic data, we cannot implement very many methods on that class. The user supplies the generic type, and thus we must ask the user to supply functions that work with that type. Nonetheless, there are some common patterns for using generic data, which is what we explore in this section. We have already seen *fold* in the context of our `IntList`. Here we will explore fold in more detail, and learn the pattern for implementing fold for any algebraic data type.

5.3.1 Fold

Last time we saw fold we were working with a list of integers. Let's generalise to a list of a generic type. We're already see all the tools we need. First our data definition, in this instance slightly modified to use the invariant sum type pattern.

```
sealed trait LinkedList[A]
final case class Pair[A](head: A, tail: LinkedList[A]) extends LinkedList[A]
final case class End[A]() extends LinkedList[A]
```

The last version of `fold` that we saw on `IntList` was

```
def fold[A](end: A, f: (Int, A) => A): A =
  this match {
    case End => end
    case Pair(hd, tl) => f(hd, tl.fold(end, f))
  }
```

It's reasonably straightforward to extend this to `LinkedList[A]`. We merely have to account for the head element of a `Pair` being of type `A` not `Int`.

```
sealed trait LinkedList[A] {
  def fold[B](end: B, f: (A, B) => B): B =
    this match {
      case End() => end
      case Pair(hd, tl) => f(hd, tl.fold(end, f))
    }
```

```
}
final case class Pair[A](head: A, tail: LinkedList[A]) extends LinkedList[A]
final case class End[A]() extends LinkedList[A]
```

Fold is just an adaptation of structural recursion where we allow the user to pass in the functions we apply at each case. As structural recursion is the generic pattern for writing any function that transforms an algebraic datatype, fold is the concrete realisation of this generic pattern. That is, fold is the generic transformation or iteration method. *Any function* you care to write on an algebraic datatype can be written in terms of fold.

> **Fold Pattern**
>
> For an algebraic datatype A, fold converts it to a generic type B. Fold is a structural recursion with:
>
> - one function parameter for each case in A;
> - each function takes as parameters the fields for its associated class;
> - if A is recursive, any function parameters that refer to a recursive field take a parameter of type B.
>
> The right-hand side of pattern matching cases, or the polymorphic methods as appropriate, consists of calls to the appropriate function.

Let's apply the pattern to derive the `fold` method above. We start with our basic template:

```
sealed trait LinkedList[A] {
  def fold[B](???): B =
    this match {
      case End() => ???
      case Pair(hd, tl) => ???
    }
}
final case class Pair[A](head: A, tail: LinkedList[A]) extends LinkedList[A]
final case class End[A]() extends LinkedList[A]
```

This is just the structural recursion template with the addition of a generic type parameter for the return type.

Now we add one function for each of the two classes in `LinkedList`.

```
def fold[B](end: ???, pair: ???): B =
  this match {
    case End() => ???
    case Pair(hd, tl) => ???
  }
```

From the rules for the function types:

- end has no parameters (as End stores no values) and returns B. Thus its type is `() => B`, which we can optimise to just a value of type B; and
- `pair` has two parameters, one for the list head and one for the tail. The argument for the head has type A, and the tail is recursive and thus has type B. The final type is therefore `(A, B) => B`.

Substituting in we get

```
def fold[B](end: B, pair: (A, B) => B): B =
  this match {
    case End() => end
    case Pair(hd, tl) => pair(hd, tl.fold(end, pair))
  }
```

5.3.2 Working With Functions

There are a few tricks in Scala for working with functions and methods that accept functions (known as higher-order methods). Here we are going to look at:

1. a compact syntax for writing functions;
2. converting methods to functions; and
3. a way to write higher-order methods that assists type inference.

5.3.2.1 Placeholder syntax

In very simple situations we can write inline functions using an extreme shorthand called *placeholder syntax*. It looks like this:

```
((_: Int) * 2)
// res: Int => Int = <function1>
```

`(_: Int) * 2` is expanded by the compiler to `(a: Int) => a * 2`. It is more idiomatic to use the placeholder syntax only in the cases where the compiler can infer the types. Here are a few more examples:

```
_ + _     // expands to `(a, b) => a + b`
foo(_)    // expands to `(a) => foo(a)`
foo(_, b) // expands to `(a) => foo(a, b)`
_(foo)    // expands to `(a) => a(foo)`
// and so on...
```

Placeholder syntax, while wonderfully terse, can be confusing for large expressions and should only be used for very small functions.

5.3.3 Converting methods to functions

Scala contains another feature that is directly relevant to this section—the ability to convert method calls to functions. This is closely related to placeholder syntax—simply follow a method with an underscore:

```
object Sum {
  def sum(x: Int, y: Int) = x + y
}

Sum.sum
// <console>:9: error: missing arguments for method sum in object Sum;
// follow this method with `_' if you want to treat it as a partially applied function
//               Sum.sum
//                   ^

(Sum.sum _)
// res: (Int, Int) => Int = <function2>
```

In situations where Scala can infer that we need a function, we can even drop the underscore and simply write the method name—the compiler will promote the method to a function automatically:

```
object MathStuff {
  def add1(num: Int) = num + 1
}

Counter(2).adjust(MathStuff.add1)
// res: Counter = Counter(3)
```

5.3.3.1 Multiple Parameter Lists

Methods in Scala can actually have multiple parameter lists. Such methods work just like normal methods, except we must bracket each parameter list separately.

```
def example(x: Int)(y: Int) = x + y
// example: (x: Int)(y: Int)Int

example(1)(2)
// res: Int = 3
```

Multiple parameter lists have two relevant uses: they look nicer when defining functions inline and they assist with type inference.

The former is the ability to write functions that look like code blocks. For example, if we define `fold` as

```
def fold[B](end: B)(pair: (A, B) => B): B =
  this match {
    case End() => end
    case Pair(hd, tl) => pair(hd, tl.fold(end, pair))
  }
```

then we can call it as

```
fold(0){ (total, elt) => total + elt }
```

which is a bit easier to read than

```
fold(0, (total, elt) => total + elt)
```

More important is the use of multiple parameter lists to ease type inference. Scala's type inference algorithm cannot use a type inferred for one parameter for another parameter in the same list. For example, given `fold` with a signature like

```
def fold[B](end: B, pair: (A, B) => B): B
```

if Scala infers `B` for `end` it cannot then use this inferred type for the `B` in`pair`, so we must often write a type declaration on `pair`. However, Scala can use types inferred for one parameter list in another parameter *list*. So if we write `fold` as

```
def fold[B](end: B)(pair: (A, B) => B): B
```

then inferring `B` from `end` (which is usually easy) allows `B` to be used when inferring the type `pair`. This means fewer type declarations and a smoother development process.

5.3.4 Exercises

5.3.4.1 Tree

A binary tree can be defined as follows:

A `Tree` of type `A` is a `Node` with a left and right `Tree` or a `Leaf` with an element of type `A`.

Implement this algebraic data type along with a fold method.

See the solution

Using `fold` convert the following `Tree` to a `String`

```
val tree: Tree[String] =
  Node(Node(Leaf("To"), Leaf("iterate")),
       Node(Node(Leaf("is"), Leaf("human,")),
            Node(Leaf("to"), Node(Leaf("recurse"), Leaf("divine")))))
```

Remember you can append `Strings` using the + method.

See the solution

5.4 Modelling Data with Generic Types

In this section we'll see the additional power the generic types give us when modelling data. We see that with generic types we can implement *generic sum and product types*, and also model some other useful abstractions such as *optional values*.

5.4.1 Generic Product Types

Let's look at using generics to model a *product type*. Consider a method that returns two values—for example, an `Int` and a `String`, or a `Boolean` and a `Double`:

```
def intAndString: ??? = // ...

def booleanAndDouble: ??? = // ...
```

The question is what do we use as the return types? We could use a regular class without any type parameters, with our usual algebraic data type patterns, but then we would have to implement one version of the class for each combination of return types:

```
case class IntAndString(intValue: Int, stringValue: String)

def intAndString: IntAndString = // ...

case class BooleanAndDouble(booleanValue: Boolean, doubleValue: Double)

def booleanAndDouble: BooleanAndDouble = // ...
```

The answer is to use generics to create a *product type*—for example a `Pair`—that contains the relevant data for *both* return types:

```
def intAndString: Pair[Int, String] = // ...

def booleanAndDouble: Pair[Boolean, Double] = // ...
```

Generics provide a different approach to defining product types— one that relies on aggregation as opposed to inheritance.

5.4.1.1 Exercise: Pairs

Implement the `Pair` class from above. It should store two values—`one` and `two`—and be generic in both arguments. Example usage:

```
val pair = Pair[String, Int]("hi", 2)
// pair: Pair[String,Int] = Pair(hi,2)

pair.one
// res: String = hi

pair.two
// res: Int = 2
```

See the solution

5.4.2 Tuples

A *tuple* is the generalisation of a pair to more terms. Scala includes built-in generic tuple types with up to 22 elements, along with special syntax for creating them. With these classes we can represent any kind of *this and that* relationship between almost any number of terms.

The classes are called `Tuple1[A]` through to `Tuple22[A, B, C, ...]` but they can also be written in the sugared[2] form `(A, B, C, ...)`. For example:

```
Tuple2("hi", 1) // unsugared syntax
// res: (String, Int) = (hi,1)

("hi", 1) // sugared syntax
// res: (String, Int) = (hi,1)

("hi", 1, true)
// res: (String, Int, Boolean) = (hi,1,true)
```

We can define methods that accept tuples as parameters using the same syntax:

```
def tuplized[A, B](in: (A, B)) = in._1
// tuplized: [A, B](in: (A, B))A

tuplized(("a", 1))
// res: String = a
```

We can also pattern match on tuples as follows:

[2]The term "syntactic sugar" is used to refer to convenience syntax that is not needed but makes programming sweeter. Operator syntax is another example of syntactic sugar that Scala provides.

```
(1, "a") match {
  case (a, b) => a + b
}
// res: String = 1a
```

Although pattern matching is the natural way to deconstruct a tuple, each class also has a complement of fields named _1, _2 and so on:

```
val x = (1, "b", true)
// x: (Int, String, Boolean) = (1,b,true)

x._1
// res: Int = 1

x._3
// res: Boolean = true
```

5.4.3 Generic Sum Types

Now let's look at using generics to model a *sum type*. Again, we have previously implemented this using our algebraic data type pattern, factoring out the common aspects into a supertype. Generics allow us to abstract over this pattern, providing a ... well ... generic implementation.

Consider a method that, depending on the value of its parameters, returns one of two types:

```
def intOrString(input: Boolean) =
  if(input == true) 123 else "abc"
// intOrString: (input: Boolean)Any
```

We can't simply write this method as shown above because the compiler infers the result type as `Any`. Instead we have to introduce a new type to explicitly represent the disjunction:

```
def intOrString(input: Boolean): Sum[Int, String] =
  if(input == true) {
    Left[Int, String](123)
  } else {
    Right[Int, String]("abc")
  }
```

How do we implement `Sum`? We just have to use the patterns we've already seen, with the addition of generic types.

5.4.3.1 Exercise: Generic Sum Type

Implement a trait `Sum[A, B]` with two subtypes `Left` and `Right`. Create type parameters so that `Left` and `Right` can wrap up values of two different types.

Hint: you will need to put both type parameters on all three types. Example usage:

```
Left[Int, String](1).value
// res: Int = 1

Right[Int, String]("foo").value
// res: String = foo

val sum: Sum[Int, String] = Right("foo")
```

```
// sum: Sum[Int,String] = Right(foo)

sum match {
  case Left(x) => x.toString
  case Right(x) => x
}
// res: String = foo
```

See the solution

5.4.4 Generic Optional Values

Many expressions may sometimes produce a value and sometimes not. For example, when we look up an element in a hash table (associative array) by a key, there may not be a value there. If we're talking to a web service, that service may be down and not reply to us. If we're looking for a file, that file may have been deleted. There are a number of ways to model this situation of an optional value. We could throw an exception, or we could return `null` when a value is not available. The disadvantage of both these methods is they don't encode any information in the type system.

We generally want to write robust programs, and in Scala we try to utilise the type system to encode properties we want our programs to maintain. One common property is "correctly handle errors". If we can encode an *optional value* in the type system, the compiler will force us to consider the case where a value is not available, thus increasing the robustness of our code.

5.4.4.1 Exercise: Maybe that Was a Mistake

Create a generic trait called `Maybe` of a generic type `A` with two subtypes, `Full` containing an `A`, and `Empty` containing no value. Example usage:

```
val perhaps: Maybe[Int] = Empty[Int]
// perhaps: Maybe[Int] = Empty()

val perhaps: Maybe[Int] = Full(1)
// perhaps: Maybe[Int] = Full(1)
```

See the solution

5.4.5 Take Home Points

In this section we have used generics to model sum types, product types, and optional values using generics.

These abstractions are commonly used in Scala code and have implementations in the Scala standard library. The sum type is called `Either`, products are tuples, and optional values are modelled with `Option`.

5.4.6 Exercises

5.4.6.1 Generics versus Traits

Sum types and product types are general concepts that allow us to model almost any kind of data structure. We have seen two methods of writing these types—traits and generics. When should we consider using each?

See the solution

5.4.6.2 Folding Maybe

In this section we implemented a sum type for modelling optional data:

```
sealed trait Maybe[A]
final case class Full[A](value: A) extends Maybe[A]
final case class Empty[A]() extends Maybe[A]
```

Implement fold for this type.

See the solution

5.4.6.3 Folding Sum

In this section we implemented a generic sum type:

```
sealed trait Sum[A, B]
final case class Left[A, B](value: A) extends Sum[A, B]
final case class Right[A, B](value: B) extends Sum[A, B]
```

Implement `fold` for `Sum`.

See the solution

5.5 Sequencing Computation

We have now mastered generic data and folding over algebraic data types. Now we will look as some other common patterns of computation that are 1) often more convenient to use than fold for algebraic data types and 2) can be implemented for certain types of data that do not support a fold. These methods are known as *map* and *flatMap*.

5.5.1 Map

Imagine we have a list of `Int` user IDs, and a function which, given a user ID, returns a `User` record. We want to get a list of user records for all the IDs in the list. Written as types we have `List[Int]` and a function `Int => User`, and we want to get a `List[User]`.

Imagine we have an optional value representing a user record loaded from the database and a function that will load their most recent order. If we have a record we want to then lookup the user's most recent order. That is, we have a `Maybe[User]` and a function `User => Order`, and we want a `Maybe[Order]`.

Imagine we have a sum type representing an error message or a completed order. If we have a completed order we want to get the total value of the order. That is, we have a `Sum[String, Order]` and a function `Order => Double`, and we want `Sum[String, Double]`.

What these all have in common is we have a type `F[A]` and a function `A => B`, and we want a result `F[B]`. The method that performs this operation is called `map`.

Let's implement `map` for `LinkedList`. We start by outlining the types and adding the general structural recursion skeleton:

```
sealed trait LinkedList[A] {
  def map[B](fn: A => B): LinkedList[B] =
    this match {
      case Pair(hd, tl) => ???
      case End() => ???
    }
}
final case class Pair[A](head: A, tail: LinkedList[A]) extends LinkedList[A]
final case class End[A]() extends LinkedList[A]
```

We know we can use the structural recursion pattern as we know that `fold` (which is just the structural recursion pattern abstracted) is the universal iterator for an algebraic data type. Thus:

- For `Pair` we have to combine `head` and `tail` to return a `LinkedList[B]` (as the types tell us) and we also know we need to recurse on `tail`. We can write

```
case Pair(hd, tl) => {
  val newTail: LinkedList[B] = tail.map(fn)
  // Combine newTail and head to create LinkedList[B]
}
```

We can convert `head` to a `B` using `fn`, and then build a larger list from `newTail` and our `B` giving us the final solution

```
case Pair(hd, tl) => Pair(fn(hd), tl.map(fn))
```

- For `End` we don't have any value of `A` to apply to the function. The only thing we can return is an `End`.

Therefore the complete solution is

```
sealed trait LinkedList[A] {
  def map[B](fn: A => B): LinkedList[B] =
    this match {
      case Pair(hd, tl) => Pair(fn(hd), tl.map(fn))
      case End() => End[B]()
    }
}
```

Notice how using the types and patterns guided us to a solution.

5.5.2 FlatMap

Now imagine the following examples:

- We have a list of users and we want to get a list of all their orders. That is, we have `LinkedList[User]` and a function `User => LinkedList[Order]`, and we want `LinkedList[Order]`.
- We have an optional value representing a user loaded from the database, and we want to lookup their most recent order—another optional value. That is, we have `Maybe[User]` and `User => Maybe[Order]`, and we want `Maybe[Order]`.

- We have a sum type holding an error message or an `Order`, and we want to email an invoice to the user. Emailing returns either an error message or a message ID. That is, we have `Sum[String, Order]` and a function `Order => Sum[String, Id]`, and we want `Sum[String, Id]`.

What these all have in common is we have a type `F[A]` and a function `A => F[B]`, and we want a result `F[B]`. The method that performs this operation is called `flatMap`.

Let's implement `flatMap` for `Maybe` (we need an append method to implement `flatMap` for `LinkedList`). We start by outlining the types:

```
sealed trait Maybe[A] {
  def flatMap[B](fn: A => Maybe[B]): Maybe[B] = ???
}
final case class Full[A](value: A) extends Maybe[A]
final case object Empty[A]() extends Maybe[A]
```

We use the same pattern as before: it's a structural recursion and our types guide us in filling in the method bodies.

```
sealed trait Maybe[A] {
  def flatMap[B](fn: A => Maybe[B]): Maybe[B] =
    this match {
      case Full(v) => fn(v)
      case Empty() => Empty[B]()
    }
}
final case class Full[A](value: A) extends Maybe[A]
final case class Empty[A]() extends Maybe[A]
```

5.5.3 Functors and Monads

A type like `F[A]` with a `map` method is called a *functor*. If a functor also has a `flatMap` method it is called a *monad*[^monads].

[^monads:] There is a little bit more to being a functor or monad. For a monad we require a constructor, typically called `point`, and there are some algebraic laws that our `map` and `flatMap` operations must obey. A quick search online will find more information on monads, or they are covered in more detail in our "Advanced Scala" book.

Although the most immediate applications of `map` and `flatMap` are in collection classes like lists, the bigger picture is sequencing computations. Imagine we have a number of computations that can fail. For instance

```
def mightFail1: Maybe[Int] =
  Full(1)

def mightFail2: Maybe[Int] =
  Full(2)

def mightFail3: Maybe[Int] =
  Empty // This one failed
```

We want to run these computations one after another. If any one of them fails the whole computation fails. Otherwise we'll add up all the numbers we get. We can do this with `flatMap` as follows.

```
mightFail1 flatMap { x =>
  mightFail2 flatMap { y =>
    mightFail3 flatMap { z =>
      Full(x + y + z)
    }
  }
}
```

The result of this is `Empty`. If we drop `mightFail3`, leaving just

```
mightFail1 flatMap { x =>
  mightFail2 flatMap { y =>
    Full(x + y)
  }
}
```

the computation succeeds and we get `Full(3)`.

The general idea is a monad represents a value in some context. The context depends on the monad we're using. We've seen examples where the context is:

- an optional value, such as we might get when retrieving a value from a database;
- an sum of values, which might represent a error message and a value we're computing with; and
- a list of values.

We use `map` when we want to transform the value within the context to a new value, while keeping the context the same. We use `flatMap` when we want to transform the value *and* provide a new context.

5.5.4 Exercises

5.5.4.1 Mapping Lists

Given the following list

```
val list: LinkedList[Int] = Pair(1, Pair(2, Pair(3, Empty)))
```

- double all the elements in the list;
- add one to all the elements in the list; and
- divide by three all the elements in the list.

See the solution

5.5.4.2 Mapping Maybe

Implement `map` for `Maybe`.

See the solution

For bonus points, implement `map` in terms of `flatMap`.

See the solution

5.5.4.3 Sequencing Computations

We're going to use Scala's builtin `List` class for this exercise as it has a `flatMap` method.

Given this list

```
val list = List(1, 2, 3)
```

return a `List[Int]` containing both all the elements and their negation. Order is not important. Hint: Given an element create a list containing it and its negation.

See the solution

Given this list

```
val list = List(Full(3), Full(2), Full(1))
```

return a `List[Maybe[Int]]` containing `None` for the odd elements. Hint: If `x % 2 == 0` then `x` is even.

See the solution

5.5.4.4 Sum

Recall our `Sum` type.

```
sealed trait Sum[A, B] {
  def fold[C](left: A => C, right: B => C): C =
    this match {
      case Left(a) => left(a)
      case Right(b) => right(b)
    }
}
final case class Left[A, B](value: A) extends Sum[A, B]
final case class Right[A, B](value: B) extends Sum[A, B]
```

To prevent a name collision between the built-in `Either`, rename the `Left` and `Right` cases to `Failure` and `Success` respectively.

See the solution

Now things are going to get a bit trickier. We are going to implement `map` and `flatMap`, again using pattern matching in the `Sum` trait. Start with `map`. The general recipe for `map` is to start with a type like `F[A]` and apply a function `A => B` to get `F[B]`. `Sum` however has two generic type parameters. To make it fit the `F[A]` pattern we're going to fix one of these parameters and allow `map` to alter the other one. The natural choice is to fix the type parameter associated with `Failure` and allow `map` to alter a `Success`. This corresponds to "fail-fast" behaviour. If our `Sum` has failed, any sequenced computations don't get run.

In summary `map` should have type

```
def map[C](f: B => C): Sum[A, C] =
```

See the solution

Now implement `flatMap` using the same logic as `map`.

See the solution

5.6 Variance

In this section we cover *variance annotations*, which allow us to control subclass relationships between types with type parameters. To motivate this, let's look again at our invariant generic sum type pattern.

Recall our `Maybe` type, which we defined as

```
sealed trait Maybe[A]
final case class Full[A](value: A) extends Maybe[A]
final case class Empty[A]() extends Maybe[A]
```

Ideally we would like to drop the unused type parameter on `Empty` and write something like

```
sealed trait Maybe[A]
final case class Full[A](value: A) extends Maybe[A]
final case object Empty extends Maybe[???]
```

Objects can't have type parameters. In order to make `Empty` an object we need to provide a concrete type in the `extends Maybe` part of the definition. But what type parameter should we use? In the absence of a preference for a particular data type, we could use something like `Unit` or `Nothing`. However this leads to type errors:

```
scala> :paste
sealed trait Maybe[A]
final case class Full[A](value: A) extends Maybe[A]
final case object Empty extends Maybe[Nothing]
^D

defined trait Maybe
defined class Full
defined module Empty

scala> val possible: Maybe[Int] = Empty
<console>:9: error: type mismatch;
 found   : Empty.type
 required: Maybe[Int]
       val possible: Maybe[Int] = Empty
```

The problem here is that `Empty` is a `Maybe[Nothing]` and a `Maybe[Nothing]` is not a subtype of `Maybe[Int]`. To overcome this issue we need to introduce variance annotations.

5.6.1 Invariance, Covariance, and Contravariance

> **Variance is Hard**
>
> Variance is one of the trickier aspects of Scala's type system. Although it is useful to be aware of its existence, we rarely have to use it in application code.

If we have some type `Foo[A]`, and `A` is a subtype of `B`, is `Foo[A]` a subtype of `Foo[B]`? The answer depends on the *variance* of the type `Foo`. The variance of a generic type determines how its supertype/subtype relationships change with respect with its type parameters:

A type `Foo[T]` is *invariant* in terms of `T`, meaning that the types `Foo[A]` and `Foo[B]` are unrelated regardless of the relationship between `A` and `B`. This is the default variance of any generic type in Scala.

A type `Foo[+T]` is *covariant* in terms of `T`, meaning that `Foo[A]` is a supertype of `Foo[B]` if `A` is a supertype of `B`. Most Scala collection classes are covariant in terms of their contents. We'll see these next chapter.

A type `Foo[-T]` is *contravariant* in terms of `T`, meaning that `Foo[A]` is a *subtype* of `Foo[B]` if `A` is a *supertype* of `B`. The only example of contravariance that I am aware of is function arguments.

5.6.2 Function Types

When we discussed function types we glossed over how exactly they are implemented. Scala has 23 built-in generic classes for functions of 0 to 22 arguments. Here's what they look like:

```
trait Function0[+R] {
  def apply: R
}

trait Function1[-A, +B] {
  def apply(a: A): B
}

trait Function2[-A, -B, +C] {
  def apply(a: A, b: B): C
}

// and so on...
```

Functions are contravariant in terms of their arguments and covariant in terms of their return type. This seems counterintuitive but it makes sense if we look at it from the point of view of function arguments. Consider some code that expects a `Function1[A, B]`:

```
class Box[A](value: A) {
  /** Apply `func` to `value`, returning a `Box` of the result. */
  def map[B](func: Function1[A, B]): Box[B] =
    Box(func(a))
}
```

To understand variance, consider what functions can we safely pass to this `map` method:

- A function from `A` to `B` is clearly ok.
- A function from `A` to a subtype of `B` is ok because it's result type will have all the properties of `B` that we might depend on. This indicates that functions are covariant in their result type.
- A function expecting a supertype of `A` is also ok, because the `A` we have in the Box will have all the properties that the function expects.
- A function expecting a subtype of `A` is not ok, because our value may in reality be a different subtype of `A`.

5.6.3 Covariant Sum Types

Now we know about variance annotations we can solve our problem with `Maybe` by making it covariant.

```
sealed trait Maybe[+A]
final case class Full[A](value: A) extends Maybe[A]
final case object Empty extends Maybe[Nothing]
```

In use we get the behaviour we expect. `Empty` is a subtype of all `Full` values.

```
val perhaps: Maybe[Int] = Empty
// perhaps: Maybe[Int] = Empty
```

This pattern is the most commonly used one with generic sum types. We should only use covariant types where the container type is immutable. If the container allows mutation we should only use invariant types.

Covariant Generic Sum Type Pattern

If `A` of type `T` is a `B` or `C`, and `C` is not generic, write

```
sealed trait A[+T]
final case class B[T](t: T) extends A[T]
final case object C extends A[Nothing]
```

This pattern extends to more than one type parameter. If a type parameter is not needed for a specific case of a sum type, we can substitute `Nothing` for that parameter.

5.6.4 Contravariant Position

There is another pattern we need to learn for covariant sum types, which involves the interaction of covariant type parameters and contravariant method and function parameters. To illustrate this issue let's develop a covariant `Sum`.

5.6.4.1 Exercise: Covariant Sum

Implement a covariant `Sum` using the covariant generic sum type pattern.

See the solution

Now let's see what happens when we implement `flatMap` on `Sum`.

5.6.4.2 Exercise: Some sort of flatMap

Implement `flatMap` and verify you receive an error like

```
error: covariant type A occurs in contravariant position in type B => Sum[A,C] of value f
  def flatMap[C](f: B => Sum[A, C]): Sum[A, C] =
                 ^
```

See the solution

What is going on here? Let's momentarily switch to a simpler example that illustrates the problem.

```
case class Box[+A](value: A) {
  def set(a: A): Box[A] = Box(a)
}
```

which causes the error

```
error: covariant type A occurs in contravariant position in type A of value a
  def set(a: A): Box[A] = Box(a)
          ^
```

Remember that functions, and hence methods, which are just like functions, are contravariant in their input parameters. In this case we have specified that `A` is covariant but in `set` we have a parameter of type `A` and the type rules requires `A` to be contravariant here. This is what the compiler means by a "contravariant position".

The solution is introduce a new type that is a supertype of `A`. We can do this with the notation `[AA >: A]` like so:

```
case class Box[+A](value: A) {
  def set[AA >: A](a: AA): Box[AA] = Box(a)
}
```

This successfully compiles.

Back to `flatMap`, the function `f` is a parameter, and thus in a contravariant position. This means we accept *supertypes* of `f`. It is declared with type `B => Sum[A, C]` and thus a supertype is *covariant* in `B` and *contravariant* in `A` and `C`. `B` is declared as covariant, so that is fine. `C` is invariant, so that is fine as well. `A` on the other hand is covariant but in a contravariant position. Thus we have to apply the same solution we did for `Box` above.

```
sealed trait Sum[+A, +B] {
  def flatMap[AA >: A, C](f: B => Sum[AA, C]): Sum[AA, C] =
    this match {
      case Failure(v) => Failure(v)
      case Success(v) => f(v)
    }
}
final case class Failure[A](value: A) extends Sum[A, Nothing]
final case class Success[B](value: B) extends Sum[Nothing, B]
```

Contravariant Position Pattern

If `A` of a covariant type `T` and a method `f` of `A` complains that `T` is used in a contravariant position, introduce a type `TT >: T` in `f`.

```
case class A[+T] {
  def f[TT >: T](t: TT): A[TT]
}
```

5.6.5 Type Bounds

We have see some type bounds above, in the contravariant position pattern. Type bounds extend to specify subtypes as well as supertypes. The syntax is `A <: Type` to declare `A` must be a subtype of `Type` and `A >: Type` to declare a supertype.

For example, the following type allows us to store a `Visitor` or any subtype:

```
case class WebAnalytics[A <: Visitor](
  visitor: A,
  pageViews: Int,
  searchTerms: List[String],
  isOrganic: Boolean
)
```

5.6.6 Exercises

5.6.6.1 Covariance and Contravariance

Using the notation `A <: B` to indicate `A` is a subtype of `B` and assuming:

- `Siamese <: Cat <: Animal`; and
- `Purr <: CatSound <: Sound`

if I have a method

```
def groom(groomer: Cat => CatSound): CatSound =
  val oswald = Cat("Black", "Cat food")
  groomer(oswald)
}
```

which of the following can I pass to `groom`?

- A function of type `Animal => Purr`
- A function of type `Siamese => Purr`
- A function of type `Animal => Sound`

See the solution

5.6.6.2 Calculator Again

We're going to return to the interpreter example we saw at the end of the last chapter. This time we're going to use the general abstractions we've created in this chapter, and our new knowledge of `map`, `flatMap`, and `fold`.

We're going to represent calculations as `Sum[String, Double]`, where the `String` is an error message. Extend `Sum` to have `map` and `fold` method.

See the solution

Now we're going to reimplement the calculator from last time. We have an abstract syntax tree defined via the following algebraic data type:

```
sealed trait Expression
final case class Addition(left: Expression, right: Expression) extends Expression
final case class Subtraction(left: Expression, right: Expression) extends Expression
final case class Division(left: Expression, right: Expression) extends Expression
final case class SquareRoot(value: Expression) extends Expression
final case class Number(value: Double) extends Expression
```

Now implement a method `eval: Sum[String, Double]` on `Expression`. Use `flatMap` and `map` on `Sum` and introduce any utility methods you see fit to make the code more compact. Here are some test cases:

```
assert(Addition(Number(1), Number(2)).eval == Success(3))
assert(SquareRoot(Number(-1)).eval == Failure("Square root of negative number"))
assert(Division(Number(4), Number(0)).eval == Failure("Division by zero"))
assert(Division(Addition(Subtraction(Number(8), Number(6)), Number(2)), Number(2)).eval == Success
    (2.0))
```

See the solution

5.7 Conclusions

In this section we have explored generic types and functions, which allow us to abstract over types and methods respectively.

We have seen new patterns for generic algebraic types, and generic structural recursion. Using these building blocks we have seen some common patterns for working with generic types, namely *fold*, *map*, and *flatMap*.

In the next section we will explore these topics further by working with the collections classes in Scala.

Chapter 6

Collections

We hardly need to state how important collection classes are. The Collections API was one of the most significant additions to Java, and Scala's collections framework, completely revised and updated in 2.8, is an equally important addition to Scala.

In this section we're going to look at three key datastructures in Scala's collection library: *sequences*, *options*, *and maps*.

We will start with *sequences*. We begin with basic operations on sequences, and then briefly examine the distinction Scala makes between interface and implementation, and mutable and immutable sequences. We then explore in depth the methods Scala provides to transform sequences.

After covering the main collection types we turn to *for comprehensions*, a syntax that allows convenient specification of operations on collections.

With for comprehensions under our belt we will move onto *options*, which are used frequently in the APIs for sequences and maps. Options provide a means to sequence computations and are an essential companion to for comprehensions.

We'll then look at *monads*, which we have introduced before, and see how they work with for comprehensions.

Next we will cover the other main collection classes: *maps and sets*. We will discover that they share a great deal in common with sequences, so most of our knowledge transfers directly.

We finish with discussion of *ranges*, which can represent large sequences of integers without storing every intermediate value in memory.

In the previous two chapters we have been focused on Scala concepts. The focus in this chapter is not on fundamental concepts, but on gaining practice with an important API and reinforcing concepts we have previously seen.

6.1 Sequences

A *sequence* is a collection of items with a defined and stable order. Sequences are one of the most common data structures. In this section we're going to look at the basics of sequences: creating them, key methods on sequences, and the distinction between mutable and immutable sequences.

Here's how you create a sequence in Scala:

```
val sequence = Seq(1, 2, 3)
// sequence: Seq[Int] = List(1, 2, 3)
```

This immediately shows off a key feature of Scala's collections, the *separation between interface and implementation*. In the above, the value has type `Seq[Int]` but is implemented by a `List`.

6.1.1 Basic operations

Sequences implement many methods. Let's look at some of the most common.

6.1.1.1 Accessing elements

We can access the elements of a sequence using its `apply` method, which accepts an `Int` index as a parameter. Indices start from `0`.

```
sequence.apply(0)
// res: Int = 1

sequence(0) // sugared syntax
// res: Int = 1
```

An exception is raised if we use an index that is out of bounds:

```
sequence(3)
// java.lang.IndexOutOfBoundsException: 3
//        at ...
```

We can also access the head and tail of the sequence:

```
sequence.head
// res: Int = 1

sequence.tail
// res: Seq[Int] = List(2, 3)

sequence.tail.head
// res: Int = 2
```

Again, trying to access an element that doesn't exist throws an exception:

```
Seq().head
// java.util.NoSuchElementException: head of empty list
//   at scala.collection.immutable.Nil$.head(List.scala:337)
//   ...

Seq().tail
// java.lang.UnsupportedOperationException: tail of empty list
//   at scala.collection.immutable.Nil$.tail(List.scala:339)
//   ...
```

If we want to safely get the `head` without risking an exception, we can use `headOption`:

```
sequence.headOption
// res: Option[Int] = Some(1)

Seq().headOption
// res: Option[Nothing] = None
```

The `Option` class here is Scala's built-in equivalent of our `Maybe` class from earlier. It has two subtypes—`Some` and `None`—representing the presence and absence of a value respectively.

6.1.2 Sequence length

Finding the length of a sequence is straightforward:

```
sequence.length
// res: Int = 3
```

6.1.3 Searching for elements

There are a few ways of searching for elements. The `contains` method tells us whether a sequence contains an element (using == for comparison):

```
sequence.contains(2)
// res: Boolean = true
```

The `find` method is like a generalised version of `contains` - we provide a test function and the sequence returns the first item for which the test returns `true`:

```
sequence.find(_ == 3)
// res: Option[Int] = Some(3)

sequence.find(_ > 4)
// res: Option[Int] = None
```

The `filter` method is a variant of `find` that returns *all* the matching elements in the sequence:

```
sequence.filter(_ > 1)
// res: Seq[Int] = List(2, 3)
```

6.1.4 Sorting elements

We can use the `sortWith` method to sort a list using a binary function. The function takes two list items as parameters and returns `true` if they are in the correct order and `false` if they are the wrong way around:

```
sequence.sortWith(_ < _)
// res: Seq[Int] = List(3, 2, 1)
```

6.1.5 Appending/prepending elements

There are many ways to add elements to a sequence. We can append an element with the `:+` method:

```
sequence.:+(4)
// res: Seq[Int] = List(1, 2, 3, 4)
```

It is more idiomatic to call `:+` as an infix operator:

```
sequence :+ 4
// res: Seq[Int] = List(1, 2, 3, 4)
```

We can similarly *prepend* an element using the `+:` method:

```
sequence.+:(0)
// res: Seq[Int] = List(0, 1, 2, 3)
```

Again, it is more idiomatic to call `+:` as an infix operator. Here *the trailing colon makes it right associative*, so we write the operator-style expression the other way around:

```
0 +: sequence
// res: Seq[Int] = List(0, 1, 2, 3)
```

This is another of Scala's general syntax rules—any method ending with a `:` character becomes *right associative* when written as an infix operator. This rule is designed to replicate Haskell-style operators for things like list prepend (`::`) and list concatenation (`:::`). We'll look at this in more detail in a moment.

Finally we can concatenate entire sequences using the ++ method.

```
sequence ++ Seq(4, 5, 6)
// res: Seq[Int] = List(1, 2, 3, 4, 5, 6)
```

6.1.6 Lists

The default implementation of `Seq` is a `List`, which is a classic linked list data structure similar to the one we developed in an earlier exercise. Some Scala libraries work specifically with `Lists` rather than using more generic types like `Seq`. For this reason we should familiarize ourselves with a couple of list-specific methods.

We can write an empty list using the singleton object `Nil`:

```
Nil
// res: scala.collection.immutable.Nil.type = List()
```

Longer lists can be created by prepending elements in classic linked-list style using the `::` method, which is equivalent to `+:`:

```
val list = 1 :: 2 :: 3 :: Nil
// list: List[Int] = List(1, 2, 3)

4 :: 5 :: list
// res: List[Int] = List(4, 5, 1, 2, 3)
```

We can also use the `List.apply` method for a more conventional constructor notation:

```
List(1, 2, 3)
// res: List[Int] = List(1, 2, 3)
```

Finally, the `:::` method is a right-associative `List`-specific version of ++:

```
List(1, 2, 3) ::: List(4, 5, 6)
// res: List[Int] = List(1, 2, 3, 4, 5, 6)
```

`::` and `:::` are specific to lists whereas `+:`, `:+` and ++ work on any type of sequence.

Lists have well known performance characteristics—constant-time prepend and head/tail operations and linear-time append and search operations. Other immutable sequences are available in Scala with different performance characteristics to match all situations. It is up to us as developers to decide whether we want to tie our code to a specific sequence type like `List` or refer to our sequences as `Seqs` to simplify swapping implementations.

6.1.7 Importing Collections and Other Libraries

The Seq and `List` types are so ubiquitous in Scala that they are made automatically available at all times. Other collections like `Vector` and `Queue` have to be brought into scope manually.

The main collections package is called `scala.collection.immutable`. We can import specific collections from this package as follows:

```
import scala.collection.immutable.Vector

Vector(1, 2, 3)
// res: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)
```

We can also use *wildcard imports* to import everything in a package:

```
import scala.collection.immutable._

Queue(1, 2, 3)
// res: scala.collection.immutable.Queue[Int] = Queue(1, 2, 3)
```

We can also use `import` to bring methods and fields into scope from a singleton:

```
import scala.collection.immutable.Vector.apply

apply(1, 2, 3)
// res: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)
```

We can write import statements anywhere in our code—imported identifiers are lexically scoped to the block where we use them:

```
// `empty` is unbound here

def someMethod = {
  import scala.collection.immutable.Vector.empty

  // `empty` is bound to `Vector.empty` here
  empty[Int]
}

// `empty` is unbound again here
```

> **Import Statements**
>
> Import statements in Scala are very flexible. The main points are nicely described in the Scala Wikibook.

6.1.8 Take Home Points

Seq is Scala's general sequence datatype. It has a number of general subtypes such as `List`, `Stack`, `Vector`, `Queue`, and `Array`, and specific subtypes such as `String`.

The default sequences in Scala are immutable. We also have access to mutable sequences, which are covered separately in the Collections Redux chapter.

We have covered a variety of methods that operate on sequences. Here is a type table of everything we have seen so far:

Method	We have	We provide	We get
`Seq(...)`		`[A], ...`	`Seq[A]`
`apply`	`Seq[A]`	`Int`	`A`
`head`	`Seq[A]`		`A`
`tail`	`Seq[A]`		`Seq[A]`
`length`	`Seq[A]`		`Int`
`contains`	`Seq[A]`	`A`	`Boolean`
`find`	`Seq[A]`	`A => Boolean`	`Option[A]`
`filter`	`Seq[A]`	`A => Boolean`	`Seq[A]`
`sortWith`	`Seq[A]`	`(A, A) => Boolean`	`Seq[A]`
`:+, +:`	`Seq[A]`	`A`	`Seq[A]`
`++`	`Seq[A]`	`Seq[A]`	`Seq[A]`
`::`	`List[A]`	`A`	`List[A]`
`:::`	`List[A]`	`List[A]`	`List[A]`

We can always use `Seq` and `List` in our code. Other collections can be brought into scope using the `import` statement as we have seen.

6.1.9 Exercises

6.1.9.1 Documentation

Discovering Scala's collection classes is all about knowing how to read the API documentation. Look up the `Seq` and `List` types now and answer the following:

- There is a synonym of `length` defined on `Seq`—what is it called?
- There are two methods for retrieving the first item in a `List` – what are they called and how do they differ?
- What method can be used used to display the elements of the sequence as a string?
- What method of `Option` can be used to determine whether the option contains a value?

Tip: There is a link to the Scala API documentation at http://scala-lang.org.

See the solution

6.1.9.2 Animals

Create a `Seq` containing the `Strings` `"cat"`, `"dog"`, and `"penguin"`. Bind it to the name `animals`.

See the solution

Append the element `"tyrannosaurus"` to `animals` and prepend the element `"mouse"`.

See the solution

What happens if you prepend the `Int` `2` to `animals`? Why? Try it out... were you correct?

See the solution

6.1.9.3 Intranet Movie Database

Let's revisit our films and directors example from the Classes chapter.

The code below is a partial rewrite of the previous sample code in which `Films` are stored as a field of `Director` instead of the other way around. Copy and paste this into a new Scala worksheet and continue with the exercises below:

```
case class Film(
  name: String,
  yearOfRelease: Int,
  imdbRating: Double)

case class Director(
  firstName: String,
  lastName: String,
  yearOfBirth: Int,
  films: Seq[Film])

val memento           = new Film("Memento", 2000, 8.5)
val darkKnight        = new Film("Dark Knight", 2008, 9.0)
val inception         = new Film("Inception", 2010, 8.8)

val highPlainsDrifter = new Film("High Plains Drifter", 1973, 7.7)
val outlawJoseyWales  = new Film("The Outlaw Josey Wales", 1976, 7.9)
val unforgiven        = new Film("Unforgiven", 1992, 8.3)
val granTorino        = new Film("Gran Torino", 2008, 8.2)
val invictus          = new Film("Invictus", 2009, 7.4)

val predator          = new Film("Predator", 1987, 7.9)
val dieHard           = new Film("Die Hard", 1988, 8.3)
val huntForRedOctober = new Film("The Hunt for Red October", 1990, 7.6)
val thomasCrownAffair = new Film("The Thomas Crown Affair", 1999, 6.8)

val eastwood = new Director("Clint", "Eastwood", 1930,
  Seq(highPlainsDrifter, outlawJoseyWales, unforgiven, granTorino, invictus))

val mcTiernan = new Director("John", "McTiernan", 1951,
  Seq(predator, dieHard, huntForRedOctober, thomasCrownAffair))

val nolan = new Director("Christopher", "Nolan", 1970,
  Seq(memento, darkKnight, inception))

val someGuy = new Director("Just", "Some Guy", 1990,
  Seq())

val directors = Seq(eastwood, mcTiernan, nolan, someGuy)

// TODO: Write your code here!
```

Using this sample code, write implementations of the following methods:

- Accept a parameter `numberOfFilms` of type `Int`—find all directors who have directed more than `numberOfFilms`:

See the solution

- Accept a parameter `year` of type `Int`—find a director who was born before that year:

See the solution

- Accept two parameters, `year` and `numberOfFilms`, and return a list of directors who were born before `year` who have also directed more than than `numberOfFilms`:

See the solution

- Accept a parameter `ascending` of type `Boolean` that defaults to `true`. Sort the directors by age in the specified order:

See the solution

6.2 Working with Sequences

In the previous section we looked at the basic operations on sequences. Now we're going to look at practical aspects of working with sequences—how functional programming allows us to process sequences in a terse and declarative style.

6.2.1 Bulk Processing of Elements

When working with sequences we often want to deal with the collection as a whole, rather than accessing and manipulating individual elements. Scala gives us a number of powerful options that allow us to solve many problems more directly.

6.2.2 Map

Let's start with something simple—suppose we want to double every element of a sequence. You might wish to express this as a loop. However, this requires writing several lines of looping machinery for only one line of actual doubling functionality.

In Scala we can use the `map` method defined on any sequence. `Map` takes a function and applies it to every element, creating a sequence of the results. To double every element we can write:

```
val sequence = Seq(1, 2, 3)
// sequence: Seq[Int] = List(1, 2, 3)

sequence.map(elt => elt * 2)
// res: Seq[Int] = List(2, 4, 6)
```

If we use *placeholder syntax* we can write this even more compactly:

```
sequence.map(_ * 2)
// res: Seq[Int] = List(2, 4, 6)
```

Given a sequence with type `Seq[A]`, the function we pass to `map` must have type `A => B` and we get a `Seq[B]` as a result. This isn't right for every situation. For example, suppose we have a sequence of strings, and we want to generate a sequence of all the permutations of those strings. We can call the `permutations` method on a string to get all permutations of it:

```
"dog".permutations
// res: Iterator[String] = non-empty iterator
```

This returns an `Iterable`, which is a bit like a Java `Iterator`. We're going to look at iterables in more detail later. For now all we need to know is that we can call the `toList` method to convert an `Iterable` to a `List`.

```
"dog".permutations.toList
// res: List[String] = List(dog, dgo, odg, ogd, gdo, god)
```

Thus we could write

```
Seq("a", "wet", "dog").map(_.permutations.toList)
// res: Seq[List[String]] = List(List(a), List(wet, wte, ewt, etw, twe, tew), List(dog, dgo, odg,
    ogd, □
                                                                                        gdo,
    god))
```

but we end up with a sequence of sequences. Let's look at the types in more detail to see what's gone wrong:

Method	We have	We provide	We get
`map`	`Seq[A]`	`A => B`	`Seq[B]`
`map`	`Seq[String]`	`String => List[String]`	`Seq[List[String]]`
`???`	`Seq[A]`	`A => Seq[B]`	`Seq[B]`

What is the method `???` that we can use to collect a single flat sequence?

6.2.3 FlatMap

Our mystery method above is called `flatMap`. If we simply replace `map` with `flatMap` we get the answer we want:

```
Seq("a", "wet", "dog").flatMap(_.permutations.toList)
// res: Seq[String] = List(a, wet, wte, ewt, etw, twe, tew, dog, dgo, odg, ogd, gdo, god)
```

`flatMap` is similar to `map` except that it expects our function to return a sequence. The sequences for each input element are appended together. For example:

```
Seq(1, 2, 3).flatMap(num => Seq(num, num * 10))
// res: List[Int] = List(1, 10, 2, 20, 3, 30)
```

The end result is (nearly) always the same type as the original sequence: `aList.flatMap(...)` returns another `List`, `aVector.flatMap(...)` returns another `Vector`, and so on:

```
import scala.collection.immutable.Vector

Vector(1, 2, 3).flatMap(num => Seq(num, num * 10))
// res: scala.collection.immutable.Vector[Int] = Vector(1, 10, 2, 20, 3, 30)
```

6.2.4 Folds

Now let's look at another kind of operation. Say we have a `Seq[Int]` and we want to add all the numbers together. `map` and `flatMap` don't apply here for two reasons:

1. they expect a *unary* function, whereas + is a *binary* operation;
2. they both return sequences of items, whereas we want to return a single `Int`.

There are also two further wrinkles to consider.

1. What result do we expect if the sequence is empty? If we're adding items together then `0` seems like a natural result, but what is the answer in general?
2. Although + is commutative (i.e. `a+b == b+a`), in general we may need to specify an order in which to pass arguments to our binary function.

Let's make another type table to see what we're looking for:

Method	We have	We provide	We get
`???`	`Seq[Int]`	`0` and `(Int, Int) => Int`	`Int`

The methods that fit the bill are called folds, with two common cases `foldLeft` and `foldRight` corresponding to the order the fold is applied. The job of these methods is to traverse a sequence and accumulate a result. The types are as follows:

Method	We have	We provide	We get
`foldLeft`	`Seq[A]`	`B` and `(B, A) => B`	`B`
`foldRight`	`Seq[A]`	`B` and `(A, B) => B`	`B`

Given the sequence `Seq(1, 2, 3)`, `0`, and + the methods calculate the following:

Method	Operations	Notes
`Seq(1, 2, 3).foldLeft(0)(_ + _)`	`(((0 + 1) + 2) + 3)`	Evaluation starts on the left
`Seq(1, 2, 3).foldRight(0)(_ + _)`	`(1 + (2 + (3 + 0)))`	Evaluation starts on the right

As we know from studying algebraic data types, the fold methods are very flexible. We can write *any* transformation on a sequence in terms of fold.

6.2.5 Foreach

There is one more traversal method that is commonly used: `foreach`. Unlike `map`, `flatMap` and the `folds`, `foreach` does not return a useful result—we use it purely for its side-effects. The type table is:

Method	We have	We provide	We get
`foreach`	`Seq[A]`	`A => Unit`	`Unit`

A common example of using `foreach` is printing the elements of a sequence:

```
List(1, 2, 3).foreach(num => println("And a " + num + "..."))
And a 1...
And a 2...
And a 3...
```

6.2.6 Algebra of Transformations

We've seen the four major traversal functions, `map`, `flatMap`, `fold`, and `foreach`. It can be difficult to know which to use, but it turns out there is a simple way to decide: look at the types! The type table below gives the types for all the operations we've seen so far. To use it, start with the data you have (always a `Seq[A]` in the table below) and then look at the functions you have available and the result you want to obtain. The final column will tell you which method to use.

We have	We provide	We want	Method
`Seq[A]`	`A => Unit`	`Unit`	`foreach`
`Seq[A]`	`A => B`	`Seq[B]`	`map`
`Seq[A]`	`A => Seq[B]`	`Seq[B]`	`flatMap`
`Seq[A]`	`B` and `(B, A) => B`	`B`	`foldLeft`
`Seq[A]`	`B` and `(A, B) => B`	`B`	`foldRight`

This type of analysis may see foreign at first, but you will quickly get used to it. Your two steps in solving any problem with sequences should be: think about the types, and experiment on the REPL!

6.2.7 Exercises

The goals of this exercise are for you to learn your way around the collections API, but more importantly to learn to use types to drive implementation. When approaching each exercise you should answer:

1. What is the type of the data we have available?
2. What is the type of the result we want?
3. What is the type of the operations we will use?

When you have answered these questions look at the type table above to find the correct method to use. Done in this way the actual programming should be straightforward.

6.2.7.1 Heroes of the Silver Screen

These exercises re-use the example code from the *Intranet Movie Database* exercise from the previous section:

Nolan Films

Starting with the definition of `nolan`, create a list containing the names of the films directed by Christopher Nolan.

See the solution

Cinephile

Starting with the definition of `directors`, create a list containing the names of all films by all directors.

See the solution

Vintage McTiernan

Starting with `mcTiernan`, find the date of the earliest McTiernan film.

Tip: you can concisely find the minimum of two numbers `a` and `b` using `math.min(a, b)`.

See the solution

High Score Table

Starting with `directors`, find all films sorted by descending IMDB rating:

See the solution

Starting with `directors` again, find the *average score* across all films:

See the solution

Tonight's Listings

Starting with `directors`, print the following for every film: `"Tonight only! FILM NAME by DIRECTOR!"`

See the solution

From the Archives

Finally, starting with `directors` again, find the *earliest film* by any director:

See the solution

6.2.7.2 Do-It-Yourself

Now we know the essential methods of `Seq`, we can write our own versions of some other library methods.

Minimum

Write a method to find the smallest element of a `Seq[Int]`.

See the solution

Unique

Given `Seq(1, 1, 2, 4, 3, 4)` create the sequence containing each number only once. Order is not important, so `Seq(1, 2, 4, 3)` or `Seq(4, 3, 2, 1)` are equally valid answers. Hint: Use `contains` to check if a sequence contains a value.

See the solution

Reverse

Write a function that reverses the elements of a sequence. Your output does not have to use the same concrete implementation as the input. Hint: use `foldLeft`.

See the solution

Map

Write `map` in terms of `foldRight`.

See the solution

Fold Left

Write your own implementation of `foldLeft` that uses `foreach` and mutable state. Remember you can create a mutable variable using the `var` keyword, and assign a new value using `=`. For example

```
var mutable = 1
var mutable = 1
mutable: Int = 1

mutable = 2
mutable = 2
mutable: Int = 2
```

See the solution

There are many other methods on sequences. Consult the API documentation for the Seq trait for more information.

6.3 For Comprehensions

We've discussed the main collection transformation functions—`map`, `flatMap`, `foldLeft`, `foldRight`, and `foreach`—and seen that they provide a powerful way of working with collections. They can become unwieldy to work with when dealing with many collections or many nested transformations. Fortunately Scala has special syntax for working with collections (in fact any class that implements `map` and `flatMap`) that makes complicated operations simpler to write. This syntax is known as a *for comprehension*.

> **Not Your Father's For Loops**
>
> *for comprehensions* in Scala are very different to the C-style *for loops* in Java. There is no direct equivalent of either language's syntax in the other.

Let's start with a simple example. Say we have the sequence `Seq(1, 2, 3)` and we wish to create a sequence with every element doubled. We know we can write

```
Seq(1, 2, 3).map(_ * 2)
// res: Seq[Int] = List(2, 4, 6)
```

The equivalent program written with a for comprehension is:

```
for {
  x <- Seq(1, 2, 3)
} yield x * 2
// res: Seq[Int] = List(2, 4, 6)
```

We call the expression containing the `<-` a *generator*, with a *pattern* on the left hand side and a *generator expression* on the right. A for comprehension iterates over the elements in the generator, binding each element to the pattern and calling the `yield` expression. It combines the yielded results into a sequence of the same type as the original generator.

In simple examples like this one we don't really see the power of for comprehensions—direct use of `map` and `flatMap` are often more compact in the simplest case. Let's try a more complicated example. Say we want to double all the numbers in `Seq(Seq(1), Seq(2, 3), Seq(4, 5, 6))` and return a flattened sequence of the results. To do this with `map` and `flatMap` we must nest calls:

```
val data = Seq(Seq(1), Seq(2, 3), Seq(4, 5, 6))
// data: Seq[Seq[Int]] = List(List(1), List(2, 3), List(4, 5, 6))

data.flatMap(_.map(_ * 2))
// res: Seq[Int] = List(2, 4, 6, 8, 10, 12)
```

This is getting complicated. The equivalent for comprehension is much more ... comprehensible.

```
for {
  subseq  <- data
  element <- subseq
} yield element * 2
// res: Seq[Int] = List(2, 4, 6, 8, 10, 12)
```

This gives us an idea of what the for comprehensions does. A general for comprehension:

```
for {
  x <- a
  y <- b
  z <- c
} yield e
```

translates to:

```
a.flatMap(x => b.flatMap(y => c.map(z => e)))
```

The intuitive understanding of the code is to iterate through all of the sequences in the generators, mapping the `yield` expression over every element therein, and accumulating a result of the same type as sequence fed into the first generator.

Note that if we omit the `yield` keyword before the final expression, the overall type of the `for` comprehension becomes `Unit`. This version of the `for` comprehension is executed purely for its side-effects, and any result is ignored. Revisiting the doubling example from earlier, we can print the results instead of returning them:

```
for {
  seq <- Seq(Seq(1), Seq(2, 3))
  elt <- seq
} println(elt * 2) // Note: no 'yield' keyword
// 2
// 4
// 6
```

The equivalent method calls use `flatMap` as usual and `foreach` in place of the final `map`:

```
a.flatMap(x => b.flatMap(y => c.foreach(z => e)))
```

We can use parentheses instead of braces to delimit the generators in a for loop. However, we must use semicolons to separate the generators if we do. Thus:

```
for (
  x <- a;
  y <- b;
  z <- c;
) yield e
```

is equivalent to:

```
for {
  x <- a
  y <- b
  z <- c
} yield e
```

Some developers prefer to use parentheses when there is only one generator and braces otherwise:

```
for(x <- Seq(1, 2, 3)) yield {
  x * 2
}
```

We can also use braces to wrap the yield expression and convert it to a *block* as usual:

```
for {
  // ...
} yield {
  // ...
}
```

6.3.1 Exercises

(More) Heroes of the Silver Screen

Repeat the following exercises from the previous section *without using `map` or `flatMap`*:

Nolan Films

List the names of the films directed by Christopher Nolan.

See the solution

Cinephile

List the names of all films by all directors.

See the solution

High Score Table

Find all films sorted by descending IMDB rating:

See the solution

Tonight's Listings

Print the following for every film: `"Tonight only! FILM NAME by DIRECTOR!"`

See the solution

6.4 Options

We have seen `Options` in passing a number of times already—they represent values that may or may not be present in our code. Options are an alternative to using `null` that provide us with a means of chaining computations together without risking `NullPointerExceptions`. We have previously produced code in the spirit of `Option` with our `DivisionResult` and `Maybe` types in previous chapters.

Let's look into Scala's built-in `Option` type in more detail.

6.4.1 Option, Some, and None

`Option` is a generic sealed trait with two subtypes—`Some` and `None`. Here is an abbreviated version of the code—we will fill in more methods as we go on:

```
sealed trait Option[+A] {
  def getOrElse(default: A): A

  def isEmpty: Boolean
  def isDefined: Boolean = !isEmpty

  // other methods...
}

final case class Some[A](x: A) extends Option[A] {
  def getOrElse(default: A) = x

  def isEmpty: Boolean = false

  // other methods...
}

final case object None extends Option[Nothing] {
  def getOrElse(default: A) = default

  def isEmpty: Boolean = true

  // other methods...
}
```

Here is a typical example of code for generating an option—reading an integer from the user:

```
def readInt(str: String): Option[Int] =
  if(str matches "\\d+") Some(str.toInt) else None
```

The `toInt` method of `String` throws a `NumberFormatException` if the string isn't a valid series of digits, so we guard its use with a regular expression. If the number is correctly formatted we return `Some` of the `Int` result. Otherwise we return `None`. Example usage:

```
readInt("123")
// res: Option[Int] = Some(123)

readInt("abc")
// res: Option[Int] = None
```

6.4.2 Extracting Values from Options

There are several ways to safely extract the value in an option without the risk of throwing any exceptions.

Alternative 1: the `getOrElse` method—useful if we want to fall back to a default value:

```
readInt("abc").getOrElse(0)
// res: Int = 0
```

Alternative 2: pattern matching—`Some` and `None` both have associated patterns that we can use in a `match` expression:

```
readInt("123") match {
  case Some(number) => number + 1
  case None         => 0
}
// res: Int = 124
```

Alternative 3: `map` and `flatMap`—`Option` supports both of these methods, enabling us to chain off of the value within producing a new `Option`. This bears a more explanation—let's look at it in a little more detail.

6.4.3 Options as Sequences

One way of thinking about an `Option` is as a sequence of 0 or 1 elements. In fact, `Option` supports many of the sequence operations we have seen so far:

```
sealed trait Option[+A] {
  def getOrElse(default: A): A

  def isEmpty: Boolean
  def isDefined: Boolean = !isEmpty

  def filter(func: A => Boolean): Option[A]
  def find(func: A => Boolean): Option[A]

  def map[B](func: A => B): Option[B]
  def flatMap(func: A => Option[B]): Option[B]
  def foreach(func: A => Unit): Unit

  def foldLeft[B](initial: B)(func: (B, A) => B): B
  def foldRight[B](initial: B)(func: (A, B) => B): B
}
```

Because of the limited size of `0` or `1`, there is a bit of redundancy here: `filter` and `find` effectively do the same thing, and `foldLeft` and `foldRight` only differ in the order of their arguments. However, these methods give us a lot flexibility for manipulating optional values. For example, we can use `map` and `flatMap` to define optional versions of common operations:

```
def sum(optionA: Option[Int], optionB: Option[Int]): Option[Int] =
  optionA.flatMap(a => optionB.map(b => a + b))

sum(readInt("1"), readInt("2"))
// res: Option[Int] = Some(3)

sum(readInt("1"), readInt("b"))
// res: Option[Int] = None

sum(readInt("a"), readInt("2"))
// res: Option[Int] = None
```

The implementation of `sum` looks complicated at first, so let's break it down:

- If `optionA` is `None`, the result of `optionA.flatMap(foo)` is also `None`. The return value of `sum` is therefore `None`.
- If `optionA` is `Some`, the result of `optionA.flatMap(foo)` is whatever value `foo` returns. This value is determined by the outcome of `optionB.map`:

- If `optionB` is `None`, the result of `optionB.map(bar)` is also `None`. The return value of `sum` is therefore `None`.
- If `optionB` is `Some`, the result of `optionB.map(bar)` is `Some` of the result of `bar`. In our case, the return value of `sum` is `a + b`.

Although `map` and `flatMap` don't allow us to *extract* values from our `Options`, they allow us to *compose computations together* in a safe manner. If all arguments to the computation are `Some`, the result is a `Some`. If any of the arguments are `None`, the result is `None`.

We can use `map` and `flatMap` in combination with pattern matching or `getOrElse` to combine several `Options` and yield a single non-optional result:

```
sum(readInt("1"), readInt("b")).getOrElse(0)
// res: Int = 0
```

It's worth noting that `Option` and `Seq` are also compatible in some sense. We can turn a `Seq[Option[A]]` into a `Seq[A]` using `flatMap`:

```
Seq(readInt("1"), readInt("b"), readInt("3")).flatMap(x => x)
// res: Seq[Int] = List(1, 3)
```

6.5 Options as Flow Control

Because `Option` supports `map` and `flatMap`, it also works with for comprehensions. This gives us a nice syntax for combining values without resorting to building custom methods like `sum` to keep our code clean:

```
val optionA = readInt("123")
val optionB = readInt("234")

for {
  a <- optionA
  b <- optionB
} yield a + b
```

In this code snippet `a` and `b` are both `Ints`—we can add them together directly using `+` in the `yield` block.

Let's stop to think about this block of code for a moment. There are three ways of looking at it:

1. We can expand the block into calls to `map` and `flatMap`. You will be unsurprised to see that the resulting code is identical to our implementation of `sum` above:

   ```
   optionA.flatMap(a => optionB.map(b => a + b))
   ```

2. We can think of `optionA` and `optionB` as sequences of zero or one elements, in which case the result is going to be a flattened sequence of length `optionA.size * optionB.size`. If either `optionA` or `optionB` is `None` then the result is of length `0`.

3. We can think of each clause in the for comprehension as an expression that says: *if this clause results in a Some, extract the value and continue... if it results in a None, exit the for comprehension and return None*.

Once we get past the initial foreignness of using for comprehensions to "iterate through" options, we find a useful control structure that frees us from excessive use of `map` and `flatMap`.

6.5.1 Exercises

6.5.1.1 Adding Things

Write a method `addOptions` that accepts two parameters of type `Option[Int]` and adds them together. Use a for comprehension to structure your code.

See the solution

Write a second version of your code using `map` and `flatMap` instead of a for comprehension.

See the solution

6.5.1.2 Adding All of the Things

Overload `addOptions` with another implementation that accepts three `Option[Int]` parameters and adds them all together.

See the solution

Write a second version of your code using `map` and `flatMap` instead of a for comprehension.

See the solution

6.5.1.3 A(nother) Short Division Exercise

Write a method `divide` that accepts two `Int` parameters and divides one by the other. Use `Option` to avoid exceptions when the denominator is `0`.

See the solution

Using your `divide` method and a for comprehension, write a method called `divideOptions` that accepts two parameters of type `Option[Int]` and divides one by the other:

See the solution

6.5.1.4 A Simple Calculator

A final, longer exercise. Write a method called `calculator` that accepts three string parameters:

```
def calculator(operand1: String, operator: String, operand2: String): Unit
```

and behaves as follows:

1. Convert the operands to `Int`s;
2. Perform the desired mathematical operator on the two operands:
    - provide support for at least four operations: `+`, `-`, `*` and `/`;
    - use `Option` to guard against errors (invalid inputs or division by zero).
3. Finally print the result or a generic error message.

Tip: Start by supporting just one operator before extending your method to other cases.

See the solution

For the enthusiastic only, write a second version of your code using `flatMap` and `map`.

See the solution

6.6 Monads

We've seen that by implementing a few methods (`map`, `flatMap`, and optionally `filter` and `foreach`), we can use any class with a *for comprehension*. In the previous chapter we learned this such a class is called a *monad*. Here we are going to look in a bit more depth at monads.

6.6.1 What's in a Monad?

The concept of a monad is notoriously difficult to explain because it is so general. We can get a good intuitive understanding by comparing some of the types of monad that we will deal with on a regular basis.

Broadly speaking, a monad is a generic type that allows us to sequence computations while abstracting away some technicality. We do the sequencing using *for comprehensions*, worrying only about the programming logic we care about. The code hidden in the monad's `map` and `flatMap` methods does all of the plumbing for us. For example:

- `Option` is a monad that allows us to sequence computations on optional values without worrying about the fact that they may or may not be present;
- `Seq` is a monad that allows us to sequence computations that return multiple possible answers without worrying about the fact that there are lots of possible combinations involved;
- `Future` is another popular monad that allows us to sequence asynchronous computations without worrying about the fact that they are asynchronous.

To demonstrate the generality of this principle, here are some examples. This first example calculates the sum of two numbers that may or may not be there:

```
for {
  a <- getFirstNumber  // getFirstNumber  returns Option[Int]
  b <- getSecondNumber // getSecondNumber returns Option[Int]
} yield a + b

// The final result is an Option[Int]---the result of
// applying `+` to `a` and `b` if both values are present
```

This second example calculate the sums of all possible pairs of numbers from two sequences:

```
for {
  a <- getFirstNumbers  // getFirstNumbers  returns Seq[Int]
  b <- getSecondNumbers // getSecondNumbers returns Seq[Int]
} yield a + b

// The final result is a Seq[Int]---the results of
// applying `+` to all combinations of `a` and `b`
```

This third example asynchronously calculates the sum of two numbers that can only be obtained asynchronously (all without blocking):

```
for {
  a <- getFirstNumber   // getFirstNumber  returns Future[Int]
  b <- getSecondNumber  // getSecondNumber returns Future[Int]
} yield a + b

// The final result is a Future[Int]---a data structure
// that will eventually allow us to access the result of
```

```
// applying `+` to `a` and `b`
```

The important point here is that, if we ignore the comments, *these three examples look identical*. Monads allow us to forget about one part of the problem at hand—optional values, multiple values, or asynchronously available values—and focus on just the part we care about—adding two numbers together.

There are many other monads that can be used to simplify problems in different circumstances. You may come across some of them in your future use of Scala. In this course we will concentrate entirely on `Seq` and `Option`.

6.6.2 Exercises

6.6.2.1 Adding All the Things ++

We've already seen how we can use a for comprehension to neatly add together three optional values. Let's extend this to other monads. Use the following definitions:

```
import scala.util.Try

val opt1 = Some(1)
val opt2 = Some(2)
val opt3 = Some(3)

val seq1 = Seq(1)
val seq2 = Seq(2)
val seq3 = Seq(3)

val try1 = Try(1)
val try2 = Try(2)
val try3 = Try(3)
```

Add together all the options to create a new option. Add together all the sequences to create a new sequence. Add together all the trys to create a new try. Use a for comprehension for each. It shouldn't take you long!

See the solution

6.7 For Comprehensions Redux

Earlier we looked at the fundamentals of for comprehensions. In this section we're going to looking at some handy additional features they offer, and at idiomatic solutions to common problems.

6.7.1 Filtering

It's quite common to only process selected elements. We can do this with comprehensions by adding an `if` clause after the generator expression. So to process only the positive elements of sequence we could write

```
for(x <- Seq(-2, -1, 0, 1, 2) if x > 0) yield x
// res: Seq[Int] = List(1, 2)
```

The code is converted to a `withFilter` call, or if that doesn't exist to `filter`.

Note that, unlike the normal `if` expression, an `if` clause in a generator does not have parentheses around the condition. So we write `if x > 0` not `if(x > 0)` in a for comprehension.

6.7.2 Parallel Iteration

Another common problem is to iterate over two or more collections in parallel. For example, say we have the sequences `Seq(1, 2, 3)` and `Seq(4, 5, 6)` and we want to add together elements with the same index yielding `Seq(5, 7 , 9)`. If we write

```
for {
  x <- Seq(1, 2, 3)
  y <- Seq(4, 5, 6)
} yield x + y
// res: Seq[Int] = List(5, 6, 7, 6, 7, 8, 7, 8, 9)
```

we see that iterations are nested. We traverse the first element from the first sequence and then all the elements of the second sequence, then the second element from the first sequence and so on.

The solution is to `zip` together the two sequences, giving a sequence containing pairs of corresponding elements

```
Seq(1, 2, 3).zip(Seq(4, 5, 6))
// res: Seq[(Int, Int)] = List((1,4), (2,5), (3,6))
```

With this we can easily compute the result we wanted

```
for(x <- Seq(1, 2, 3).zip(Seq(4, 5, 6))) yield { val (a, b) = x; a + b }
// res: Seq[Int] = List(5, 7, 9)
```

Sometimes you want to iterate over the values in a sequence and their indices. For this case the `zipWithIndex` method is provided.

```
for(x <- Seq(1, 2, 3).zipWithIndex) yield x
// res: Seq[(Int, Int)] = List((1,0), (2,1), (3,2))
```

Finally note that `zip` and `zipWithIndex` are available on all collection classes, including `Map` and `Set`.

6.7.3 Pattern Matching

The pattern on the left hand side of a generator is not named accidentally. We can include any pattern there and only process results matching the pattern. This provides another way of filtering results. So instead of:

```
for(x <- Seq(1, 2, 3).zip(Seq(4, 5, 6))) yield { val (a, b) = x; a + b }
// res: Seq[Int] = List(5, 7, 9)
```

we can write:

```
for((a, b) <- Seq(1, 2, 3).zip(Seq(4, 5, 6))) yield a + b
// res: Seq[Int] = List(5, 7, 9)
```

6.7.4 Intermediate Results

It is often useful to create an intermediate result within a sequence of generators. We can do this by inserting an assignment expression like so:

```
for {
  x     <- Seq(1, 2, 3)
  square = x * x
  y     <- Seq(4, 5, 6)
} yield square * y
// res: Seq[Int] = List(4, 5, 6, 16, 20, 24, 36, 45, 54)
```

6.8 Maps and Sets

Up to now we've spent all of our time working with sequences. In this section we'll go through the two other most common collection types: `Maps` and `Sets`.

6.8.1 Maps

A `Map` is very much like its counterpart in Java - it is a collection that maps *keys* to *values*. The keys must form a set and in most cases are unordered. Here is how to create a basic map:

```
val example = Map("a" -> 1, "b" -> 2, "c" -> 3)
// res: scala.collection.immutable.Map[java.lang.String,Int] =
        Map(a -> 1, b -> 2, c -> 3)
```

The type of the resulting map is `Map[String,Int]`, meaning all the keys are type `String` and all the values are of type `Int`.

A quick aside on `->`. The constructor function for `Map` actually accepts an arbitrary number of `Tuple2` arguments. `->` is actually a function that generates a Tuple2.

```
"a" -> 1
// res: (java.lang.String, Int) = (a,1)
```

Let's look at the most common operations on a map.

6.8.1.1 Accessing values using keys

The raison d'etre of a map is to convert keys to values. There are two main methods for doing this: `apply` and `get`.

```
example("a") // The same as example.apply("a")
// res: Int = 1

example.get("a")
// res: Option[Int] = Some(1)
```

`apply` attempts to look up a key and throws an exception if it is not found. By contrast, `get` returns an `Option`, forcing you to handle the not found case in your code.

```
example("d")
java.util.NoSuchElementException: key not found: d

example.get("d")
// res: Option[Int] = None
```

Finally, the `getOrElse` method accepts a default value to return if the key is not found.

```
example.getOrElse("d", -1)
// res: Int = -1
```

6.8.1.2 Determining membership

The `contains` method determines whether a map contains a key.

```
example.contains("a")
// res: Boolean = true
```

6.8.1.3 Determining size

Finding the size of a map is just as easy as finding the size of a sequence.

```
example.size
// res: Int = 3
```

6.8.1.4 Adding and removing elements

As with `Seq`, the default implementation of `Map` is immutable. We add and remove elements by creating new maps as opposed to mutating existing ones.

We can add new elements using the `+` method. Note that, as with Java's `HashMap`, keys are overwritten and order is non-deterministic.

```
example.+("c" -> 10, "d" -> 11, "e" -> 12)
// res: scala.collection.immutable.Map[java.lang.String,Int] =
        Map(e -> 12, a -> 1, b -> 2, c -> 10, d -> 11)
```

We can remove keys using the `-` method:

```
example.-("b", "c")
// res: scala.collection.immutable.Map[java.lang.String,Int] =
        Map(a -> 1)
```

If we are only specifying a single argument, we can write `+` and `-` as infix operators.

```
example + ("d" -> 4) - "c"
// res: scala.collection.immutable.Map[java.lang.String,Int] =
         Map(a -> 1, b -> 2, d -> 4)
```

Note that we still have to write the pair `"d" -> 4` in parentheses because `+` and `->` have the same precedence.

There are many other methods for manipulating immutable maps. For example, the `++` and `--` methods return the union and intersection of their arguments. See the Scaladoc for `Map` for more information.

6.8.1.5 Mutable maps

The `scala.collection.mutable` package contains several mutable implementations of `Map`:

```
val example2 = scala.collection.mutable.Map("x" -> 10, "y" -> 11, "z" -> 12)
// example2: scala.collection.mutable.Map[java.lang.String,Int] =
              Map(x -> 10, z -> 12, y -> 11)
```

The in-place mutation equivalents of + and - are += and -= respectively.

```
example2 += ("x" -> 20)
// res: example2.type = Map(x -> 20, z -> 12, y -> 11)

example2 -= ("y", "z")
// res: example2.type = Map(x -> 20)
```

Note that, like their immutable cousins, += and -= both return a result of type Map. In this case, however, the return value is *the same object* that we called the method on. The return value is useful for chaining method calls together, but we can discard it if we see fit.

We can also use the update method, or its assignment-style syntactic-sugar, to update elements in the map:

```
example2("w") = 30

example2
// res: scala.collection.mutable.Map[java.lang.String,Int] = Map(x -> 20, w -> 30)
```

Note that, as with mutable sequences, a(b) = c is shorthand for a.update(b, c). The update method does not return a value, but the map is mutated as a side-effect.

There are many other methods for manipulating mutable maps. See the Scaladoc for scala.collection.mutable.Map for more information.

6.8.1.6 Sorted maps

The maps we have seen so far do not guarantee an ordering over their keys. For example, note that in this example, the order of keys in the resulting map is different from the order of addition operations.

```
Map("a" -> 1) + ("b" -> 2) + ("c" -> 3) +
         ("d" -> 4) + ("e" -> 5)
// res: scala.collection.immutable.Map[java.lang.String,Int] =
         Map(e -> 5, a -> 1, b -> 2, c -> 3, d -> 4)
```

Scala also provides ordered immutable and mutable versions of a ListMap class that preserves the order in which keys are added:

```
scala.collection.immutable.ListMap("a" -> 1) + ("b" -> 2) + ("c" -> 3) +
         ("d" -> 4) + ("e" -> 5)
// res: scala.collection.immutable.ListMap[java.lang.String,Int] =
          Map(a -> 1, b -> 2, c -> 3, d -> 4, e -> 5)
```

Scala's separation of interface and implementation means that the methods on ordered and unordered maps are almost identical, although their performance may vary. See this useful page for more information on the performance characteristics of the various types of collection.

6.8.1.7 map and flatMap

Maps, like sequences, extend the `Traversable` trait, which means they inherit the standard `map` and `flatMap` methods. In fact, a `Map[A,B]` is a `Traversable[Tuple2[A,B]]`, which means that `map` and `flatMap` operate on instances of `Tuple2`.

Here is an example of `map`:

```
example.map(pair => pair._1 -> pair._2 * 2)
// res: scala.collection.immutable.Map[java.lang.String,Int] =
          Map(a -> 2, b -> 4, c -> 6)
```

Note that the resulting object is also a `Map` as you might expect. However, what happens when the function we supply doesn't return a pair? What does `map` return then? Is it a compile error? Let's try it.

```
example.map(pair => pair._1 + " = " + pair._2)
// res: scala.collection.immutable.Iterable[java.lang.String] =
          List(a = 1, b = 2, c = 3)
```

It turns out the code does work, but we get back an `Iterable` result (look at the type, not the value)—a far more general data type.

Scala's collections framework is built in a clever (and complicated) way that always ensures you get something sensible back out of one of the standard operations like `map` and `flatMap`. We won't go into the details here (it's practically a training course in its own right). Suffice to say that you can normally guess using common sense (and judicious use of the REPL) the type of collection you will get back from any operation.

Here is a more complicated example using `flatMap`:

```
example.flatMap {
         case (str, num) =>
           (1 to 3).map(x => (str + x) -> (num * x))
       }
// res: scala.collection.immutable.Map[String,Int] =
         Map(c3 -> 9, b2 -> 4, b3 -> 6, c2 -> 6, b1 -> 2,
             c1 -> 3, a3 -> 3, a1 -> 1, a2 -> 2)
```

and the same example written using `for` syntax:

```
for{
         (str, num) <- example
          x         <- 1 to 3
       } yield (str + x) -> (num * x)
// res: scala.collection.immutable.Map[String,Int] =
         Map(c3 -> 9, b2 -> 4, b3 -> 6, c2 -> 6, b1 -> 2,
             c1 -> 3, a3 -> 3, a1 -> 1, a2 -> 2)
```

Note that the result is a `Map` again. The argument to `flatMap` returns a sequence of pairs, so in the end we are able to make a new `Map` from them. If our function returns a sequence of non-pairs, we get back a more generic data type.

```
for{
         (str, num) <- example
          x         <- 1 to 3
       } yield (x + str) + "=" + (x * num)
// res: scala.collection.immutable.Iterable[java.lang.String] =
         List(1a=1, 2a=2, 3a=3, 1b=2, 2b=4, 3b=6, 1c=3, 2c=6, 3c=9)
```

6.8.1.8 In summary

Here is a type table of all the methods we have seen so far:

Method	We have	We provide	We get
`Map(...)`		`Tuple2[A,B], ...`	`Map[A,B]`
`apply`	`Map[A,B]`	`A`	`B`
`get`	`Map[A,B]`	`A`	`Option[B]`
`+`	`Map[A,B]`	`Tuple2[A,B], ...`	`Map[A,B]`
`-`	`Map[A,B]`	`Tuple2[A,B], ...`	`Map[A,B]`
`++`	`Map[A,B]`	`Map[A,B]`	`Map[A,B]`
`--`	`Map[A,B]`	`Map[A,B]`	`Map[A,B]`
`contains`	`Map[A,B]`	`A`	`Boolean`
`size`	`Map[A,B]`		`Int`
`map`	`Map[A,B]`	`Tuple2[A,B] => Tuple2[C,D]`	`Map[C,D]`
`map`	`Map[A,B]`	`Tuple2[A,B] => E`	`Iterable[E]`
`flatMap`	`Map[A,B]`	`Tuple2[A,B] => Traversable[Tuple2[C,D]]`	`Map[C,D]`
`flatMap`	`Map[A,B]`	`Tuple2[A,B] => Traversable[E]`	`Iterable[E]`

Here are the extras for mutable Sets:

Method	We have	We provide	We get
`+=`	`Map[A,B]`	`A`	`Map[A,B]`
`-=`	`Map[A,B]`	`A`	`Map[A,B]`
`update`	`Map[A,B]`	`A, B`	`Unit`

6.8.2 Sets

Sets are unordered collections that contain no duplicate elements. You can think of them as sequences without an order, or maps with keys and no values. Here is a type table of the most important methods:

Method	We have	We provide	We get
`+`	`Set[A]`	`A`	`Set[A]`
`-`	`Set[A]`	`A`	`Set[A]`
`++`	`Set[A]`	`Set[A]`	`Set[A]`
`--`	`Set[A]`	`Set[A]`	`Set[A]`
`contains`	`Set[A]`	`A`	`Boolean`
`apply`	`Set[A]`	`A`	`Boolean`
`size`	`Set[A]`		`Int`
`map`	`Set[A]`	`A => B`	`Set[B]`
`flatMap`	`Set[A]`	`A => Traversable[B]`	`Set[B]`

and the extras for mutable Sets:

Method	We have	We provide	We get
`+=`	`Set[A]`	`A`	`Set[A]`
`-=`	`Set[A]`	`A`	`Set[A]`

6.8.3 Exercises

6.8.3.1 Favorites

Copy and paste the following code into an editor:

```
val people = Set(
  "Alice",
  "Bob",
  "Charlie",
  "Derek",
  "Edith",
  "Fred")

val ages = Map(
  "Alice"   -> 20,
  "Bob"     -> 30,
  "Charlie" -> 50,
  "Derek"   -> 40,
  "Edith"   -> 10,
  "Fred"    -> 60)

val favoriteColors = Map(
  "Bob"     -> "green",
  "Derek"   -> "magenta",
  "Fred"    -> "yellow")

val favoriteLolcats = Map(
  "Alice"   -> "Long Cat",
  "Charlie" -> "Ceiling Cat",
  "Edith"   -> "Cloud Cat")
```

Use the code as test data for the following exercises:

Write a method `favoriteColor` that accepts a person's name as a parameter and returns their favorite colour.

See the solution

Update `favoriteColor` to return a person's favorite color *or* beige as a default.

See the solution

Write a method `printColors` that prints everyone's favorite color!

See the solution

Write a method `lookup` that accepts a name and one of the maps and returns the relevant value from the map. Ensure that the return type of the method matches the value type of the map.

See the solution

Calculate the color of the oldest person:

See the solution

6.8.4 Do-It-Yourself Part 2

Now we have some practice with maps and sets let's see if we can implement some useful library functions for ourselves.

6.8.4.1 Union of Sets

Write a method that takes two sets and returns a set containing the union of the elements. Use iteration, like `map` or `foldLeft`, not the built-in `union` method to do so!

See the solution

6.8.4.2 Union of Maps

Now let's write union for maps. Assume we have two `Map[A, Int]` and add corresponding elements in the two maps. So the union of `Map('a' -> 1, 'b' -> 2)` and `Map('a' -> 2, 'b' -> 4)` should be `Map('a' -> 3, 'b' -> 6)`.

See the solution

6.8.4.3 Generic Union

There are many things that can be added, such as strings (string concatenation), sets (union), and of course numbers. It would be nice if we could generalise our `union` method on maps to handle anything for which a sensible `add` operation can be defined. How can we go about doing this?

See the solution

6.9 Ranges

So far we've seen lots of ways to iterate over sequences but not much in the way of iterating over numbers. In Java and other languages it is common to write code like

```
for(i = 0; i < array.length; i++) {
  doSomething(array[i])
}
```

We've seen that for comprehensions provide a succinct way of implementing these programs. But what about classics like this?

```
for(i = 99; i > 0; i--) {
  System.out.println(i + "bottles of beer on the wall!")
  // Full text omitted for the sake of brevity
}
```

Scala provides the `Range` class for these occasions. A `Range` represents a sequence of integers from some starting value to less than the end value with a non-zero step. We can construct a `Range` using the `until` method on `Int`.

```
1 until 10
// res: scala.collection.immutable.Range = Range(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

By default the step size is 1, so trying to go from high to low gives us an empty Range.

```
10 until 1
// res: scala.collection.immutable.Range = Range()
```

We can rectify this by specifying a different step, using the by method on Range.

```
10 until 1 by -1
// res: scala.collection.immutable.Range = Range(10, 9, 8, 7, 6, 5, 4, 3, 2)
```

Now we can write the Scala equivalent of our Java program.

```
for(i <- 99 until 0 by -1) println(i + " bottles of beer on the wall!")
// 99 bottles of beer on the wall!
// 98 bottles of beer on the wall!
// 97 bottles of beer on the wall!
// etc ...
```

This gives us a hint of the power of ranges. Since they are sequences we can combine them with other sequences in interesting ways. For example, to create a range with a gap in the middle we can concatenate two ranges:

```
(1 until 10) ++ (20 until 30)
// res: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 20, 21, 22,
    23, □
                                                                                    24, 25, 26, 27, 28,

    29)
```

Note that the result is a `Vector` not a Range but this doesn't matter. As they are both sequences we can use both them in a for comprehension without any code change!

6.10 Generating Random Data

In this section we have an extended case study generating random data. The ideas here have many applications. For example, in generating data for testing, as used in *property based testing*, in *probabilistic programming*, a new area of machine learning, and, if you're going through the extended case study, in *generative art*.

6.10.1 Random Words

We'll start by generating text. Imagine we wanted to generate (somewhat) realistic text, perhaps to use as a placeholder to fill in parts of a website design. If we took a large amount of real text we could analyse to work out for each word what the most common words following it are. Such a model is known as a *Markov chain*.

To keep this example to a reasonable size we're going to deal with a really simplified version of the problem, where all sentences have the form *subject-verb-object*. For example, "Noel wrote code".

Write a program to generate all possible sentences given the following model:

- subjects are `List("Noel", "The cat", "The dog");`

- verbs are `List("wrote", "chased", "slept on")`; and
- objects are `List("the book", "the ball", "the bed")`.

See the solution

This model creates some clearly nonsensical sentences. We can do better by making the choice of verb depen-dend on the subject, and the object depend on the verb.

Let's use the following model:

- The subjects are as before.
- For the verbs:
- If the subject is "Noel" the possible verbs are "wrote", "chased", and "slept on".
- If the subject is "The cat" the possible verbs are "meowed at", "chased", and "slept on".
- If the subject is "The dog" the possible verbs are "barked at", "chased", and "slept on".
- For the objects:
- If the verb is "wrote" the possible objects are "the book", "the letter", and "the code".
- If the verb is "chased" the possible objects are "the ball", "the dog", and "the cat".
- If the verb is "slept on" the possible objects are "the bed", "the mat", and "the train".
- If the verb is "meowed at" the possible objects are "Noel", "the door", "the food cupboard".
- If the verb is "barked at" the possible objects are "the postman", "the car", and "the cat".

Implement this.

See the solution

This model has all the features we need for our full random generation model. In particular we have *conditional distributions*, meaning the choice of, say, verb is dependent or conditional on what has come before.

6.10.2 Probabilities

We now have a model that we can imagine making arbitrarily complex to generate more and more realistic data, but we're missing the element of probability that would allow us to weight the data generation towards more common outcomes.

Let's extend our model to work on `List[(A, Double)]`, where `A` is the type of data we are generating and the `Double` is a probability. We're still enumerating all possibilities but we're now associating a probability with each possible outcome.

Start by defining a class `Distribution` that will wrap a `List[(A, Double)]`. (Why?)

See the solution

We should create some convenience constructors for `Distribution`. A useful one is `uniform` which will accept a `List[A]` and create a `Distribution[A]` where each element has equal probability. Make it so.

See the solution

What are the other methods we must add to implement the models we've seen so far? What are their signatures?

See the solution

Now implement these methods. Start with `map`, which is simpler. We might end up with elements appearing multiple times in the list of events after calling `map`. That's absolutely ok.

See the solution

Now implement `flatMap`. To do so you'll need to combine the probability of an event with the probability of the event it depends on. The correct way to do so is to multiply the probabilities together. The may lead to *unnormalised* probabilities—probabilities that do not sum up to 1. You might find the following two utilities useful, though you don't need to normalise probabilities or ensure that elements are unique for the model to work.

```
def normalize: Distribution[A] = {
  val totalWeight = (events map { case (a, p) => p }).sum
  Distribution(events map { case (a,p) => a -> (p / totalWeight) })
}

def compact: Distribution[A] = {
  val distinct = (events map { case (a, p) => a }).distinct
  def prob(a: A): Double =
    (events filter { case (x, p) => x == a } map { case (a, p) => p }).sum

  Distribution(distinct map { a => a -> prob(a) })
}
```

See the solution

6.10.3 Examples

With `Distribution` we can now define some interesting model. We could do some classic problems, such as working out the probability that a coin flip gives three heads in a row.

```
sealed trait Coin
final case object Heads extends Coin
final case object Tails extends Coin

val fairCoin: Distribution[Coin] = Distribution.uniform(List(Heads, Tails))
val threeFlips =
  for {
    c1 <- fairCoin
    c2 <- fairCoin
    c3 <- fairCoin
  } yield (c1, c2, c3)
// threeFlips: Distribution[(Coin, Coin, Coin)] =
// Distribution(List(
//    ((Heads,Heads,Heads),0.125),
//    ((Heads,Heads,Tails),0.125),
//    ((Heads,Tails,Heads),0.125),
//    ((Heads,Tails,Tails),0.125),
//    ((Tails,Heads,Heads),0.125),
//    ((Tails,Heads,Tails),0.125),
//    ((Tails,Tails,Heads),0.125),
//    ((Tails,Tails,Tails),0.125)
// ))
```

From this we can read of the probability of three heads being 0.125, as we'd expect.

Let's create a more complex model. Imagine the following situation:

I put my food into the oven and after some time it ready to eat and produces delicious smell with probability 0.3 and otherwise it is still raw and produces no smell with probability 0.7. If there are delicious smells the cat comes to harass me with probability 0.8, and otherwise it stays asleep. If there is no smell the cat harasses me for the hell of it with probability 0.4 and otherwise stays asleep.

Implement this model and answer the question: if the cat comes to harass me what is the probability my food is producing delicious smells (and therefore is ready to eat.)

I found it useful to add this constructor to the companion object of `Distribution`:

```
def discrete[A](events: List[(A,Double)]): Distribution[A] =
  Distribution(events).compact.normalize
```

See the solution

6.10.4 Next Steps

The current library is limited to working with discrete events. If we wanted to work with continuous domains, such as coordinates in the plane, we need a different representation as we clearly can't represent all possible outcomes. We can even run into issues with complex discrete models, as the number of events increases exponentially with each `flatMap`.

Instead of representing all events we can sample from the distributions of interest and maintain a set of samples. Varying the size of the set allows us to tradeoff accuracy with computational resources.

We could use the same style of implementation with a sampling representation, but this requires we fix the number of samples in advance. It's more useful to be able to repeatedly sample from the same model, so the user can ask for more samples if they decide they need higher accuracy. To do so requires we separate defining the structure of the model from the process of sampling from it, and reify the model. We're not going to go further into this implementation here, but if you're going through the case study you'll pick up the techniques need to implement it.

Chapter 7

Type Classes

Type classes are a powerful feature of Scala that allow us to extend existing libraries with new functionality, without using inheritance and without having access to the original library source code. In this chapter we will learn how to use and implement type classes, using a Scala feature called *implicits*.

In the section on traits we compared object oriented and functional style in terms of extensibility, using this table.

	Add new method	Add new data
OO	Change existing code	Existing code unchanged
FP	Existing code unchanged	Change existing code

Type classes give us a third implementation technique which is more flexible than either. A type class is like a trait, defining an interface. However, with type classes we can:

- plug in different implementations of an interface for a given class; and
- implement an interface without modifying existing code.

This means we can add new methods *or* new data without changing any existing code.

It's difficult to understand these concepts without an example. We'll start this section by exploring how we can use type classes. We'll then turn to implementing them ourselves. We'll finish with a discussion of best practices.

7.1 Type Class Instances

Type classes in Scala involve the interaction of a number of components. To simplify the presentation we are going to start by looking at *using* type classes before we look at how to *build them ourselves*.

7.1.1 Ordering

A simple example of a type class is the `Ordering` trait. For a type `A`, an `Ordering[A]` defines a comparison method `compare` that compares two instances of `A` by some ordering. To construct an `Ordering` we can use the convenience method `fromLessThan` defined the companion object.

Imagine we want to sort a `List` of `Int`s. There are many different ways to sort such a list. For example, we could sort from highest to lowest, or we could sort from lowest to highest. There is a method `sorted` on `List` that will sort a list, but to use it we must pass in an `Ordering` to give the particular ordering we want.

Let's define some `Ordering`s and see them in action.

```
import scala.math.Ordering

val minOrdering = Ordering.fromLessThan[Int](_ < _)
// minOrdering: scala.math.Ordering[Int] = scala.math.Ordering$$anon$9@787f32b7

val maxOrdering = Ordering.fromLessThan[Int](_ > _)
// maxOrdering: scala.math.Ordering[Int] = scala.math.Ordering$$anon$9@4bf324f9

List(3, 4, 2).sorted(minOrdering)
// res: List[Int] = List(2, 3, 4)

List(3, 4, 2).sorted(maxOrdering)
// res: List[Int] = List(4, 3, 2)
```

Here we define two orderings: `minOrdering`, which sorts from lowest to highest, and `maxOrdering`, which sorts from highest to lowest. When we call `sorted` we pass the `Ordering` we want to use. These implementations of a type class are called *type class instances*.

The type class pattern separates the implementation of functionality (the type class instance, an `Ordering[A]` in our example) from the type the functionality is provided for (the `A` in an `Ordering[A]`). *This is the basic pattern for type classes.* Everything else we will see just provides extra convenience.

7.1.2 Implicit Values

It can be inconvenient to continually pass the type class instance to a method when we want to repeatedly use the same instance. Scala provides a convenience, called an *implicit value*, that allows us to get the compiler to pass the type class instance for us. Here's an example of use:

```
implicit val ordering = Ordering.fromLessThan[Int](_ < _)

scala> List(2, 4, 3).sorted
// res: List[Int] = List(2, 3, 4)

List(1, 7 ,5).sorted
// res: List[Int] = List(1, 5, 7)
```

Note we didn't supply an ordering to `sorted`. Instead, the compiler provides it for us.

We have to tell the compiler which values it is allowed pass to methods for us. We do this by annotating a value with `implicit`, as in the declaration `implicit val ordering = ...`. The method must also indicate that it accepts implicit values. If you look at the documentation for the `sorted` method on `List` you see that the single parameter is declared `implicit`. We'll talk more about implicit parameter lists in a bit. For now we just need to know that we can get the compiler to supply implicit values to parameters that are themselves marked implicit.

7.1.3 Declaring Implicit Values

We can tag any `val`, `var`, `object` or zero-argument `def` with the `implicit` keyword, making it a potential candidate for an implicit parameter.

```
implicit val exampleOne = ...
implicit var exampleTwo = ...
implicit object exampleThree = ...
implicit def exampleFour = ...
```

An implicit value must be declared within a surrounding object, class, or trait.

7.1.4 Implicit Value Ambiguity

What happens when multiple implicit values are in scope? Let's ask the console.

```
implicit val minOrdering = Ordering.fromLessThan[Int](_ < _)

implicit val maxOrdering = Ordering.fromLessThan[Int](_ > _)

List(3,4,5).sorted
//  <console>:12: error: ambiguous implicit values:
//  both value ordering of type => scala.math.Ordering[Int]
//  and value minOrdering of type => scala.math.Ordering[Int]
//  match expected type scala.math.Ordering[Int]
//                List(3,4,5).sorted
//                            ^
```

The rule is simple: the compiler will signal an error if there is any ambiguity in which implicit value should be used.

7.1.5 Take Home Points

In this section we've seen the basics for using type classes. In Scala, a type class is just a trait. To use a type class we:

- create implementations of that trait, called type class instances; and
- typically we mark the type class instances as implicit values.

Marking values as implicit tells the compiler it can supply them as a parameter to a method call if none is explicitly given. For the compiler to supply a value:

1. the parameter must be marked implicit in the method declaration;
2. there must be an implicit value available of the same type as the parameter; and
3. there must be only one such implicit value available.

7.1.6 Exercises

7.1.6.1 More Orderings

Define an `Ordering` that orders `Int`s from lowest to highest by absolute value. The following test cases should pass.

```
assert(List(-4, -1, 0, 2, 3).sorted(absOrdering) == List(0, -1, 2, 3, -4))
assert(List(-4, -3, -2, -1).sorted(absOrdering) == List(-1, -2, -3, -4))
```

See the solution

Now make your ordering an implicit value, so the following test cases work.

```
assert(List(-4, -1, 0, 2, 3).sorted == List(0, -1, 2, 3, -4))
assert(List(-4, -3, -2, -1).sorted == List(-1, -2, -3, -4))
```

See the solution

7.1.6.2 Rational Orderings

Scala doesn't have a class to represent rational numbers, but we can easily implement one ourselves.

```
final case class Rational(numerator: Int, denominator: Int)
```

Implement an `Ordering` for `Rational` to order rationals from smallest to largest. The following test case should pass.

```
assert(List(Rational(1, 2), Rational(3, 4), Rational(1, 3)).sorted ==
       List(Rational(1, 3), Rational(1, 2), Rational(3, 4)))
```

See the solution

7.2 Organising Type Class Instances

In section we'll learn about the places the compiler searches for type class instances (implicit values), known as the *implicit scope*, and we'll discuss how to organise type class instances to make their use more convenient.

7.2.1 Implicit Scope

The compiler searches the implicit scope when it tries to find an implicit value to supply as an implicit parameter. The implicit scope is composed of several parts, and there are rules that prioritise some parts over others.

The first part of the implicit scope is the normal scope where other identifiers are found. This includes identifiers declared in the local scope, within any enclosing class, object, or trait, or `imported` from elsewhere. An eligible implicit value must be a single identifier (i.e. `a`, not `a.b`). This is referred to as the *local scope*.

The implicit scope also includes the companion objects of types involved in the method call with the implicit parameter. Let's look at `sorted` for example. The signature for `sorted`, defined on `List[A]`, is

```
sorted[B >: A](implicit ord: math.Ordering[B]): List[A]
```

The compiler will look in the following places for `Ordering` instances:

- the companion object of `List`;
- the companion object of `Ordering`; and
- the companion object of the type `B`, which is the type of elements in the list or any superclass.

The practical upshot is we can define type class instances in the companion object of our types (the type `A` in this example) and they will be found by the compiler without the user having to import them explicitly.

In the previous section we defined an `Ordering` for a `Rational` type we created. Let's see how we can use the companion object to make this `Ordering` easier to use.

First let's define the ordering in the local scope.

```
final case class Rational(numerator: Int, denominator: Int)

object Example {
  def example = {
    implicit val ordering = Ordering.fromLessThan[Rational]((x, y) =>
      (x.numerator.toDouble / x.denominator.toDouble) <
      (y.numerator.toDouble / y.denominator.toDouble)
    )
    assert(List(Rational(1, 2), Rational(3, 4), Rational(1, 3)).sorted ==
           List(Rational(1, 3), Rational(1, 2), Rational(3, 4)))
  }
}
```

This works as we expect.

Now let's shift the type class instance out of the local scope and see that it doesn't compile.

```
final case class Rational(numerator: Int, denominator: Int)

object Instance {
  implicit val ordering = Ordering.fromLessThan[Rational]((x, y) =>
    (x.numerator.toDouble / x.denominator.toDouble) <
    (y.numerator.toDouble / y.denominator.toDouble)
  )
}

object Example {
  def example =
    assert(List(Rational(1, 2), Rational(3, 4), Rational(1, 3)).sorted ==
           List(Rational(1, 3), Rational(1, 2), Rational(3, 4)))
}
```

Here I get an error at compilation time

```
No implicit Ordering defined for Rational.
assert(List(Rational(1, 2), Rational(3, 4), Rational(1, 3)).sorted ==
                                                            ^
```

Finally let's move the type class instance into the companion object of `Rational` and see that the code compiles again.

```
final case class Rational(numerator: Int, denominator: Int)

object Rational {
  implicit val ordering = Ordering.fromLessThan[Rational]((x, y) =>
    (x.numerator.toDouble / x.denominator.toDouble) <
    (y.numerator.toDouble / y.denominator.toDouble)
  )
}

object Example {
  def example =
```

```
    assert(List(Rational(1, 2), Rational(3, 4), Rational(1, 3)).sorted ==
           List(Rational(1, 3), Rational(1, 2), Rational(3, 4)))
}
```

This leads us to our first pattern for packaging type class instances.

> **Type Class Instance Packaging: Companion Objects**
>
> When defining a type class instance, if
>
> 1. there is a single instance for the type; and
> 2. you can edit the code for the type that you are defining the instance for
>
> then *define the type class instance in the companion object of the type.*

7.2.2 Implicit Priority

If we look in the companion object for `Ordering` we see some type class instances are already defined. In particular there is an instance for `Int`, yet we could define our own instances for `Ordering[Int]` (which we did in the previous section) and not have an issue with ambiguity.

To understand this we need to learn about the priority rules for selecting implicits. An ambiguity error is only raised if there are multiple type class instances with the same priority. Otherwise the highest priority implicit is selected.

The full priority rules are rather complex, but that complexity has little impact in most cases. The practical implication is that the local scope takes precedence over instances found in companion objects. This means that implicits that the programmer explicitly pulls into scope, by importing or defining them in the local scope, will be used in preference.

Let's see this in practice, by defining an `Ordering` for `Rational` within the local scope.

```
final case class Rational(numerator: Int, denominator: Int)

object Rational {
  implicit val ordering = Ordering.fromLessThan[Rational]((x, y) =>
    (x.numerator.toDouble / x.denominator.toDouble) <
    (y.numerator.toDouble / y.denominator.toDouble)
  )
}

object Example {
  implicit val higherPriorityImplicit = Ordering.fromLessThan[Rational]((x, y) =>
      (x.numerator.toDouble / x.denominator.toDouble) >
      (y.numerator.toDouble / y.denominator.toDouble)
  )

  def example =
    assert(List(Rational(1, 2), Rational(3, 4), Rational(1, 3)).sorted ==
           List(Rational(3, 4), Rational(1, 2), Rational(1, 3)))
}
```

Notice that `higherPriorityImplicit` defines a different ordering to the one defined in the companion object for `Rational`. We've also changed the expected ordering in `example` to match this new ordering. This code both compiles and runs correctly, illustrating the effect of the priority rules.

Type Class Instance Packaging: Companion Objects Part 2

When defining a type class instance, if

1. there is a single good default instance for the type; and
2. you can edit the code for the type that you are defining the instance for

then *define the type class instance in the companion object of the type*. This allows users to override the instance by defining one in the local scope whilst still providing sensible default behaviour.

7.2.3 Packaging Implicit Values Without Companion Objects

If there is no good default instance for a type class instance, or if there are several good defaults, we should not place an type class instances in the companion object but instead require the user to explicitly import an instance into the local scope.

In this case, one simple way to package instances is to place each in its own object that the user can import into the local scope. For instance, we might define orderings for `Rational` as follows:

```
final case class Rational(numerator: Int, denominator: Int)

object RationalLessThanOrdering {
  implicit val ordering = Ordering.fromLessThan[Rational]((x, y) =>
    (x.numerator.toDouble / x.denominator.toDouble) <
    (y.numerator.toDouble / y.denominator.toDouble)
  )
}

object RationalGreaterThanOrdering {
  implicit val ordering = Ordering.fromLessThan[Rational]((x, y) =>
    (x.numerator.toDouble / x.denominator.toDouble) >
    (y.numerator.toDouble / y.denominator.toDouble)
  )
}
```

In use the user would `import RationalLessThanOrdering._` or `import RationalGreaterThanOrdering._` as appropriate.

7.2.4 Take Home Points

The compiler looks for type class instances (implicit values) in two places:

1. the local scope; and
2. the companion objects of types involved in the method call.

Implicits found in the local scope take precedence over those found in companion objects.

When packaging type class instances, if there is a single instance or a single good default we should put it in the companion object if possible. Otherwise, one way to package implicits is to place each one in an object and require the user to explicitly import them.

7.2.5 Exercises

7.2.5.1 Ordering Orders

Here is a case class to store orders of some arbitrary item.

```
final case class Order(units: Int, unitPrice: Double) {
  val totalPrice: Double = units * unitPrice
}
```

We have a requirement to order `Orders` in three different ways:

1. by `totalPrice`;
2. by number of `units`; and
3. by `unitPrice`.

Implement and package implicits to provide these orderings, and justify your packaging.

See the solution

7.3 Creating Type Classes

In the previous sections we saw how to create and use type class instances. Now we're going to explore creating our own type classes.

7.3.1 Elements of Type Classes

There are four components of the type class pattern:

- the actual type class itself;
- the type class instances;
- interfaces using implicit parameters; and
- interfaces using enrichment and implicit parameters.

We have already seen type class instances and talked briefly about implicit parameters. Here we will look at defining our own type class, and in the following section we will look at the two styles of interface.

7.3.2 Creating a Type Class

Let's start with an example—converting data to HTML. This is a fundamental operation in any web application, and it would be great to be able to provide a `toHtml` method across the board in our application.

One implementation strategy is to create a trait we `extend` wherever we want this functionality:

```
trait HtmlWriteable {
  def toHtml: String
}

final case class Person(name: String, email: String) extends HtmlWriteable {
  def toHtml = s"<span>$name &lt;$email&gt;</span>"
}
```

```
Person("John", "john@example.com").toHtml
// res: String = <span>John &lt;john@example.com&gt;</span>
```

This solution has a number of drawbacks. First, we are restricted to having just one way of rendering a `Person`. If we want to list people on our company homepage, for example, it is unlikely we will want to list everybody's email addresses without obfuscation. For logged in users, however, we probably want the convenience of direct email links. Second, this pattern can only be applied to classes that we have written ourselves. If we want to render a `java.util.Date` to HTML, for example, we will have to write some other form of library function.

Polymorphism has failed us, so perhaps we should try pattern matching instead? We could write something like

```
object HtmlWriter {
  def write(in: Any): String =
    in match {
      case Person(name, email) => ...
      case Date => ...
      case _ => throw new Exception(s"Can't render ${in} to HTML")
    }
}
```

This implementation has its own issues. We have lost type safety because there is no useful supertype that covers just the elements we want to render and no more. We can't have more than one implemnetation of rendering for a given type. We also have to modify this code whenever we want to render a new type.

We can overcome all of these problems by moving our HTML rendering to an adapter class:

```
trait HtmlWriter[A] {
  def write(in: A): String
}

object PersonWriter extends HtmlWriter[Person] {
  def write(person: Person) = s"<span>${person.name} &lt;${person.email}&gt;</span>"
}

PersonWriter.write(Person("John", "john@example.com"))
// res: String = <span>John &lt;john@example.com&gt;</span>
```

This is better. We can now define `HtmlWriter` functionality for other types, including types we have not written ourselves:

```
import java.util.Date

object DateWriter extends HtmlWriter[Date] {
  def write(in: Date) = s"<span>${in.toString}</span>"
}

DateWriter.write(new Date)
// res: String = <span>Sat Apr 05 16:01:58 BST 2014</span>
```

We can also write another `HtmlWriter` for writing `People` on our homepage:

```
object ObfuscatedPersonWriter extends HtmlWriter[Person] {
  def write(person: Person) =
    s"<span>${person.name} (${person.email.replaceAll("@", " at ")})</span>"
}
```

```
ObfuscatedPersonWriter.write(Person("John", "john@example.com"))
// res: String = John (john at example.com)
```

Much safer—it'll take a spam bot more than a few microseconds to decypher that!

You might recognise `PersonWriter`, `DateWriter`, and `ObfuscatedPersonWriter` as following the type class instance pattern (though we haven't made them implicit values at this point). The `HtmlWriter` trait, which the instances implement, is the type class itself.

> **Type Class Pattern**
>
> A type class is a trait with at least one type variable. The type variables specify the concrete types the type class instances are defined for. Methods in the trait usually use the type variables.
>
> ```
> trait ExampleTypeClass[A] {
>   def doSomething(in: A): Foo
> }
> ```

The next step is to introduce implicit parameters, so we can use type classes with less boilerplate.

7.3.3 Take Home Points

We have seen the basic pattern for implementing type classes.

- We declare some interface for the functionality we want

```
trait HtmlWriter[A] {
  def toHtml(in: A): String
}
```

- We write type class instances for each concrete class we want to use and for each different situation we want to use it in

```
object PersonWriter extends HtmlWriter[Person] {
  def write(person: Person) =
    s"${person.name} (${person.email})"
}

object ObfuscatedPersonWriter extends HtmlWriter[Person] {
  def write(person: Person) =
    s"${person.name} (${person.email.replaceAll("@", " at ")})"
}
```

- This allows us to implement the functionality for any type, and to provide different implementations for the same type.

7.3.4 Exercises

7.3.4.1 Equality

Scala provides two equality predicates: by value (==) and by reference (eq). Nonetheless, we sometimes need additional predicates. For instance, we could compare people by just email address if we were validating new user accounts in some web application.

Implement a trait `Equal` of some type `A`, with a method `equal` that compares two values of type `A` and returns a `Boolean`. `Equal` is a type class.

See the solution

Our `Person` class is

```
case class Person(name: String, email: String)
```

Implement instances of `Equal` that compare for equality by email address only, and by name and email.

See the solution

7.4 Implicit Parameter and Interfaces

We've seen the basics of the type class pattern. Now let's look at how we can make it easier to use. Recall our starting point is a trait `HtmlWriter` which allows us to implement HTML rendering for classes without requiring access to their source code, and allows us to render the same class in different ways.

```
trait HtmlWriter[A] {
  def write(in: A): String
}

object PersonWriter extends HtmlWriter[Person] {
  def write(person: Person) = s"<span>${person.name} &lt;${person.email}&gt;</span>"
}
```

This issue with this code is that we need manage a lot of `HtmlWriter` instances when we render any complex data. We have already seen that we can manage this complexity using implicit values and have mentioned *implicit parameters* in passing. In this section we go in depth on implicit parameters.

7.4.1 Implicit Parameter Lists

Here is an example of an implicit parameter list:

```
object HtmlUtil {
  def htmlify[A](data: A)(implicit writer: HtmlWriter[A]): String = {
    writer.write(data)
  }
}
```

The `htmlify` method accepts two arguments: some `data` to convert to HTML and a `writer` to do the conversion. The `writer` is an implicit parameter.

The `implicit` keyword applies to the *whole parameter list*, not just an individual parameter. This makes the parameter list optional—when we call `HtmlUtil.htmlify` we can either specify the list as normal

```
HtmlUtil.htmlify(Person("John", "john@example.com"))(PersonWriter)
// res: String = <span>John &lt;john@example.com&gt;</span>
```

or we can omit the implicit parameters. If we omit the implicit parameters, the compiler searches for implicit values of the correct type it can use to fill in the missing arguments. We have already learned about implicit values, but let's see a quick example to refresh our memory. First we define an implicit value.

```
implicit object ApproximationWriter extends HtmlWriter[Int] {
  def write(in: Int): String =
    s"It's definitely less than ${((in / 10) + 1) * 10}"
}
```

When we use `HtmlUtil` we don't have to specify the implicit parameter if an implicit value can be found.

```
HtmlUtil.htmlify(2)
// res: String = It's definitely less than 10
```

7.4.2 Interfaces Using Implicit Parameters

A complete use of the type class pattern requires an interface using implicit parameters, along with implicit type class instances. We've seen two examples already: the `sorted` method using `Ordering`, and the `htmlify` method above. The best interface depends on the problem being solved, but there is a pattern that occurs frequently enough that it is worth explaining here.

In many case the interface defined by the type class is the same interface we want to use. This is the case for `HtmlWriter` – the only method of interest is `write`. We could write something like

```
object HtmlWriter {
  def write[A](in: A)(implicit writer: HtmlWriter[A]): String =
    writer.write(in)
}
```

We can avoid this indirection (which becomes more painful to write as our interfaces become larger) with the following construction:

```
object HtmlWriter {
  def apply[A](implicit writer: HtmlWriter[A]): HtmlWriter[A] =
    writer
}
```

In use it looks like

```
HtmlWriter[Person].write(Person("Noel", "noel@example.org"))
```

The idea is to simply select a type class instance by type (done by the no-argument `apply` method) and then directly call the methods defined on that instance.

> **Type Class Interface Pattern**
>
> If the desired interface to a type class `TypeClass` is exactly the methods defined on the type class trait, define an interface on the companion object using a no-argument `apply` method like

```
object TypeClass {
  def apply[A](implicit instance: TypeClass[A]): TypeClass[A] =
    instance
}
```

7.4.3 Take Home Points

Implicit parameters make type classes more convenient to use. We can make an entire parameter list with the `implicit` keyword to make it an implicit parameter list.

```
def method[A](normalParam1: NormalType, ...)(implicit implicitParam1: ImplicitType[A], ...)
```

If we call a method and do not explicitly supply an explicit parameter, the compiler will search for an implicit value of the correct type and insert it as the parameter.

Using implicit parameters we can make more convenient interfaces using type class instances. If the desired interface to a type class is exactly the methods defined on the type class we can create a convenient interface using the pattern

```
object TypeClass {
  def apply[A](implicit instance: TypeClass[A]): TypeClass[A] =
    instance
}
```

7.4.4 Exercises

7.4.4.1 Equality Again

In the previous section we defined a trait `Equal` along with some implementations for `Person`.

```
case class Person(name: String, email: String)

trait Equal[A] {
  def equal(v1: A, v2: A): Boolean
}

object EmailEqual extends Equal[Person] {
  def equal(v1: Person, v2: Person): Boolean =
    v1.email == v2.email
}

object NameEmailEqual extends Equal[Person] {
  def equal(v1: Person, v2: Person): Boolean =
    v1.email == v2.email && v1.name == v2.name
}
```

Implement an object called `Eq` with an `apply` method. This method should accept two explicit parameters of type `A` and an implicit `Equal[A]`. It should perform the equality checking using the provided `Equal`. With appropriate implicits in scope, the following code should work

```
Eq(Person("Noel", "noel@example.com"), Person("Noel", "noel@example.com"))
```

See the solution

Package up the different `Equal` implementations as implicit values in their own objects, and show you can control the implicit selection by changing which object is imported.

See the solution

Now implement an interface on the companion object for `Equal` using the no-argument apply method pattern. The following code should work.

```
import NameAndEmailImplicit._
Equal[Person].equal(Person("Noel", "noel@example.com"), Person("Noel", "noel@example.com"))
```

Which interface style do you prefer?

See the solution

7.5 Enriched Interfaces

A second type of type class interface, called *type enrichment*[1] allow us to create interfaces that act as if they were methods defined on the classes of interest. For example, suppose we have a method called `numberOfVowels`:

```
def numberOfVowels(str: String) =
  str.filter(Seq('a', 'e', 'i', 'o', 'u').contains(_)).length

numberOfVowels("the quick brown fox")
// res: Int = 5
```

This is a method that we use all the time. It would be great if `numberOfVowels` was a built-in method of `String` so we could write `"a string".numberOfVowels`, but of course we can't change the source code for `String`. Scala has a feature called called *implicit classes* that allow us to add new functionality to an existing class without editing its source code. This is a similar concept to *categories* in Objective C or *extension methods* in C#, but the implementation is different in each case.

7.5.1 Implicit Classes

Let's build up implicit classes piece by piece. We can wrap `String` in a class that adds our `numberOfVowels`:

```
class ExtraStringMethods(str: String) {
  val vowels = Seq('a', 'e', 'i', 'o', 'u')

  def numberOfVowels =
    str.toList.filter(vowels contains _).length
}
```

We can use this to wrap up our `String` and gain access to our new method:

```
new ExtraStringMethods("the quick brown fox").numberOfVowels
```

Writing new `ExtraStringMethods` every time we want to use `numberOfVowels` is unwieldy. However, if we tag our class with the `implicit` keyword, we give Scala the ability to insert the constructor call automatically into our code:

[1] Type enrichment is sometimes referred to as pimping in older literature. We will not use that term.

```
implicit class ExtraStringMethods(str: String) { /* ... */ }

"the quick brown fox".numberOfVowels
// res: Int = 5
```

When the compiler processes our call to `numberOfVowels`, it interprets it as a type error because there is no such method in `String`. Rather than give up, the compiler attempts to fix the error by searching for an implicit class that provides the method and can be constructed from a `String`. It finds `ExtraStringMethods`. The compiler then inserts an invisible constructor call, and our code type checks correctly.

Implicit classes follow the same scoping rules as implicit values. Like implicit values, they must be defined within an enclosing object, class, or trait (except when writing Scala at the console).

There is one additional restriction for implicit classes: only a single implicit class will be used to resolve a type error. The compiler will not look to construct a chain of implicit classes to access the desired method.

7.6 Combining Type Classes and Type Enrichment

Implicit classes can be used on their own but we most often combine them with type classes to create a more natural style of interface. We keep the type class (`HtmlWriter`) and adapters (`PersonWriter`, `DateWriter` and so on) from our type class example, and add an implicit class with methods that themselves take implicit parameters. For example:

```
implicit class HtmlOps[T](data: T) {
  def toHtml(implicit writer: HtmlWriter[T]) =
    writer.write(data)
}
```

This allows us to invoke our type-class pattern on any type for which we have an adapter *as if it were a built-in feature of the class*:

```
Person("John", "john@example.com").toHtml
// res: String = <span>John &lt;john@example.com&gt;</span>
```

This gives us many benefits. We can extend existing types to give them new functionality, use simple syntax to invoke the functionality, *and* choose our preferred implementation by controlling which implicits we have in scope.

7.6.1 Take Home Points

Implicit classes are a Scala language feature that allows us to define extra functionality on existing data types without using conventional inheritance. This is a programming pattern called *type enrichment*.

The Scala compiler uses implicit classes to *fix type errors in our code*. When it encounters us accessing a method or field that doesn't exist, it looks through the available implicits to find some code it can insert to fix the error.

The rules for implicit classes are the same as for implicit values, with the additional restriction that only a single implicit class will be used to fix a type error.

7.6.2 Exercises

7.6.2.1 Drinking the Kool Aid

Use your newfound powers to add a method `yeah` to `Int`, which prints `Oh yeah!` as many times as the `Int` on which it is called if the `Int` is positive, and is silent otherwise. Here's an example of usage:

```
2.yeah
// Oh yeah!
// Oh yeah!

3.yeah
// Oh yeah!
// Oh yeah!
// Oh yeah!

-1.yeah
```

When you have written your implicit class, package it in an `IntImplicits` object.

See the solution

7.6.2.2 Times

Extend your previous example to give `Int` an extra method called `times` that accepts a function of type `Int => Unit` as an argument and executes it `n` times. Example usage:

```
3.times(i => println(s"Look - it's the number $i!"))
// Look - it's the number 0!
// Look - it's the number 1!
// Look - it's the number 2!
```

For bonus points, re-implement `yeah` in terms of `times`.

See the solution

7.6.3 Easy Equality

Recall our `Equal` type class from a previous section.

```
trait Equal[A] {
  def equal(v1: A, v2: A): Boolean
}
```

Implement an enrichment so we can use this type class via a triple equal (===) method. For example, if the correct implicits are in scope the following should work.

```
"abcd".===("ABCD") // Assumes case-insensitive equality implicit
```

See the solution

7.7 Using Type Classes

We have seen how to define type classes. In this section we'll see some conveniences for using them: *context bounds* and the *implicitly* method.

7.7.1 Context Bounds

When we use type classes we often end up requiring implicit parameters that we pass onward to a type class interface. For example, using our `HtmlWriter` example we might want to define some kind of page template that accepts content rendered by a writer.

```
def pageTemplate[A](body: A)(implicit writer: HtmlWriter[A]): String = {
  val renderedBody = body.toHtml

  s"<html><head>...</head><body>${renderedBody}</body></html>"
}
```

We don't explicitly use the implicit `writer` in our code, but we need it in scope so the compiler can insert it for the `toHtml` enrichment.

Context bounds allow us to write this more compactly, with a notation that is reminiscent of a type bound.

```
def pageTemplate[A : HtmlWriter](body: A): String = {
  val renderedBody = body.toHtml

  s"<html><head>...</head><body>${renderedBody}</body></html>"
}
```

The context bound is the notation `[A : HtmlWriter]` and it expands into the equivalent implicit parameter list in the prior example.

> **Context Bound Syntax**
>
> A context bound is an annotation on a generic type variable like so:
>
> ```
> [A : Context]
> ```
>
> It expands into a generic type parameter `[A]` along with an implicit parameter for a `Context[A]`.

7.7.2 Implicitly

Context bounds give us a short-hand syntax for declaring implicit parameters, but since we don't have an explicit name for the parameter we cannot use it in our methods. Normally we use context bounds when we don't need explicit access to the implicit parameter, but rather just implicitly pass it on to some other method. However if we do need access for some reason we can use the `implicitly` method.

```
case class Example(name: String)
implicit val implicitExample = Example("implicit")

implicitly[Example]
// res: Example = Example(implicit)
```

```
implicitly[Example] == implicitExample
// res: Boolean = true
```

The `implicitly` method takes no parameters but has a generic type parameters. It returns the implicit matching the given type, assuming there is no ambiguity.

7.8 Implicit Conversions

So far we have seen two programming patterns using implicits: *type enrichment*, which we implement using *implicit classes*, and *type classes*, which we implement using *implicit values and parameter lists*.

Scala has a third implicit mechanism called *implicit conversions* that we will cover here for completeness. Implicit conversions can be seen as a more general form of implicit classes, and can be used in a wider variety of contexts.

> **The Dangers of Implicit Conversions**
>
> As we shall see later in this section, undisciplined use of implicit conversions can cause as many problems as it fixes for the beginning programmer. Scala even requires us to write a special import statement to silence compiler warnings resulting from the use of implicit conversions:
>
> ```
> import scala.language.implicitConversions
> ```
>
> We recommend using implicit classes and implicit values/parameters over implicit conversions wherever possible. By sticking to the type enrichment and type class design patterns you should find very little cause to use implicit conversions in your code.
>
> You have been warned!

7.8.1 Implicit conversions

Implicit conversions are a more general form of implicit classes. We can tag any single-argument method with the `implicit` keyword to allow the compiler to implicitly use the method to perform automated conversions from one type to another:

```
class B {
  def bar = "This is the best method ever!"
}

class A

implicit def aToB(in: A): B = new B()

new A().bar
// res: String = This is the best method ever!
```

Implicit classes are actually just syntactic sugar for the combination of a regular class and an implicit conversion. With an implicit class we have to define a new type as a target for the conversion; with an implicit method we can convert from any type to any other type as long as an implicit is available in scope.

7.8.2 Designing with Implicit Conversions

The power of implicit conversions tends to cause problems for newer Scala developers. We can easily define very general type conversions that play strange games with the semantics of our programs:

```
implicit def intToBoolean(int: Int) = int == 0

if(1) "yes" else "no"
// res: String = no

if(0) "yes" else "no"
// res: String = yes
```

This example is ridiculous, but it demonstrates the potential problems implicits can cause. `intToBoolean` could be defined in a library in a completely different part of our codebase, so how would we debug the bizarre behaviour of the `if` expressions above?

Here are some tips for designing using implicits that will prevent situations like the one above:

- Wherever possible, stick to the type enrichment and type class programming patterns.
- Wherever possible, use implicit classes, values, and parameter lists over implicit conversions.
- Package implicits clearly, and bring them into scope only where you need them. We recommend using the packaging guidelines introduced earlier this chapter.
- Avoid creating implicit conversions that convert from one general type to another general type—the more specific your types are, the less likely the implicit is to be applied incorrectly.

7.8.3 Exercises

7.8.3.1 Implicit Class Conversion

Any implicit class can be reimplemented as a class paired with an implicit method. Re-implement the `IntOps` class from the *type enrichment* section in this way. Verify that the class still works the same way as it did before.

See the solution

7.9 JSON Serialisation

In this section we have an extended example involving serializing Scala data to JSON, which is one of the classic use cases for type classes. The typical process for converting data to JSON in Scala involves two steps. First we convert our data types to an intermediate case class representation, then we serialize the intermediate representation to a string.

Here is a suitable case class representation of a subset of the JSON language. We have a `sealed trait` `JsValue` that defines a `stringify` method, and a set of subtypes for two of the main JSON data types—objects and strings:

```
sealed trait JsValue {
  def stringify: String
}

final case class JsObject(values: Map[String, JsValue]) extends JsValue {
  def stringify = values
```

```
    .map { case (name, value) => "\"" + name + "\":" + value.stringify }
    .mkString("{", ",", "}")
}

final case class JsString(value: String) extends JsValue {
  def stringify = "\"" + value.replaceAll("\\|\"", "\\\\$1") + "\""
}
```

You should recognise this as the algebraic data type pattern.

We can construct JSON objects and serialize them as follows:

```
JsObject(Map("foo" -> JsString("a"), "bar" -> JsString("b"), "baz" -> JsString("c")))
// res: JsObject = JsObject(Map(foo -> JsString(a), bar -> JsString(b), baz -> JsString(c)))

res4.stringify
// res: String = {"foo":"a","bar":"b","baz":"c"}
```

7.9.1 Convert X to JSON

Let's create a type class for converting Scala data to JSON. Implement a `JsWriter` trait containing a single abstract method `write` that converts a value to a `JsValue`.

See the solution

Now let's create the dispatch part of our type class. Write a `JsUtil` object containing a single method `toJson`. The method should accept a value of an arbitrary type `A` and convert it to JSON.

Tip: your method will have to accept an implicit `JsWriter` to do the actual conversion.

See the solution

Now, let's revisit our data types from the web site visitors example in the Sealed traits section:

```
import java.util.Date

sealed trait Visitor {
  def id: String
  def createdAt: Date
  def age: Long = new Date().getTime() - createdAt.getTime()
}

final case class Anonymous(
  val id: String,
  val createdAt: Date = new Date()
) extends Visitor

final case class User(
  val id: String,
  val email: String,
  val createdAt: Date = new Date()
) extends Visitor
```

Write `JsWriter` instances for `Anonymous` and `User`.

See the solution

Given these two definitions we can implement a `JsWriter` for `Visitor` as follows. This uses a new type of pattern – `a: B` – which matches any value of type `B` and binds it to a variable `a`:

```
implicit object VisitorWriter extends JsWriter[Visitor] {
  def write(value: Visitor) = value match {
    case anon: Anonymous => JsUtil.toJson(anon)
    case user: User      => JsUtil.toJson(user)
  }
}
```

Finally, verify that your code works by converting the following list of users to JSON:

```
val visitors: Seq[Visitor] = Seq(Anonymous("001", new Date), User("003", "dave@xample.com", new Date
    ))
```

See the solution

7.9.2 Prettier Conversion Syntax

Let's improve our JSON syntax by combining type classes and type enrichment. Convert `JsUtil` to an `implicit class` with a `toJson` method. Sample usage:

```
Anonymous("001", new Date).toJson
```

See the solution

Chapter 8

Conclusions

This completes Essential Scala. To recap our journey, we have learned Scala via the major patterns of usage:

- algebraic data types and structural recursion;
- sequencing computations using `map`, `flatMap`, and `fold`; and
- type classes.

These are the patterns we use daily in our Scala coding, which we have found work well across many Scala projects, and they make up by the far the majority of our Scala code. They will serve you well.

We have tried to emphasise that if you can model the problem correctly the code follows in an almost mechanical way. Learning how to think in the Scala way (or, more broadly, in a functional way) is by far the most important lesson of this book.

We have introduced language features as they support the patterns. In the appendices you will find additional material covering some inessential functionality we have skipped over in the main text. Scala has a few other features, such as self types, that we have found so little use for in our years of programming Scala that we have omitted them entirely in this introductory text.

8.1 What Now?

The journey to mastering Scala has not finished with this book. You will benefit greatly from active participation in the Scala community. We have setup an online chat room for discussion of all Scala related matters. Any and all Scala related questions are welcome there. There are many other forums, conferences, and user groups where you can find an enthusiastic and welcoming community of fellow programmers.

If you have enjoyed Essential Scala we hope you'll consider our followup book Advanced Scala. As the name suggests, it covers more advanced concepts with an emphasis on patterns for larger programs.

Finally, we would love hear your thoughts on Essential Scala. Any feedback—good or bad—helps to improve the book. We can be reached at hello@underscore.io. Any improvements we make to Essential Scala will of course be made available to every reader as part of our policy of free liftime updates.

Thank you for reading Essential Scala, and we hope you future coding in Scala is productive and fun.

Appendix A

Pattern Matching

We have seen the duality between algebraic data types and pattern matching. Armed with this information, we are in a good position to return to pattern matching and see some of its more powerful features.

As we discussed earlier, patterns are written in their own DSL that only superficially resembles regular Scala code. *Patterns serve as tests that match a specific set of Scala values.* The `match` expression compares a value to each pattern in turn, finds the first pattern that matches, and executes the corresponding block of Scala code.

Some patterns bind values to variables that can be used on the right hand side of the corresponding => symbol, and *some patterns contain other patterns*, allowing us to build complex tests that simultaneously examine many parts of a value. Finally, *we can create our own custom patterns*, implemented in Scala code, to match any cross-section of values we see fit.

We have already seen case class patterns and certain types of sequence patterns. Each of the remaining types of pattern is described below together with an example of its use.

A.1 Standard patterns

A.1.1 Literal patterns

Literal patterns match a particular value. Any Scala literals work except function literals: primitive values, `Strings`, `nulls`, and `()`:

```
(1 + 1) match {
  case 1 => "It's one!"
  case 2 => "It's two!"
  case 3 => "It's three!"
}
// res: String = It's two!

Person("Dave", "Gurnell") match {
  case Person("Noel", "Welsh") => "It's Noel!"
  case Person("Dave", "Gurnell") => "It's Dave!"
}
// res: String = It's Dave!

println("Hi!") match {
  case () => "It's unit!"
}
```

```
// Hi!
// res: String = It's unit!
```

A.1.2 Constant patterns

Identifiers starting with an uppercase letter are *constants* that match a single predefined constant value:

```
val X = "Foo"
// X: String = Foo

val Y = "Bar"
// Y: String = Bar

val Z = "Baz"
// Z: String = Baz

"Bar" match {
  case X => "It's foo!"
  case Y => "It's bar!"
  case Z => "It's baz!"
}
// res: String = It's bar!
```

A.1.3 Alternative patterns

Vertical bars can be used to specify alternatives:

```
"Bar" match {
  case X | Y => "It's foo or bar!"
  case Z     => "It's baz!"
}
// res: String = It's baz!
```

A.1.4 Variable capture

Identifiers starting with lowercase letters bind values to variables. The variables can be used in the code to the right of the =>:

```
Person("Dave", "Gurnell") match {
  case Person(f, n) => f + " " + n
}
// res: String = "Dave Gurnell"
```

The @ operator, written `x @ y`, allows us to capture a value in a variable x while also matching it against a pattern y. x must be a variable pattern and y can be any type of pattern. For example:

```
Person("Dave", "Gurnell") match {
  case p @ Person(_, s) => s"The person $p has the surname $s"
}
// res: String = "The person Person(Dave,Gurnell) is called Dave Gurnell"
```

A.1.5 Wildcard patterns

The _ symbol is a pattern that matches any value and simply ignores it. This is useful in two situations: when nested inside other patterns, and when used on its own to provide an "else" clause at the end of a match expression:

```
Person("Dave", "Gurnell") match {
  case Person("Noel", _) => "It's Noel!"
  case Person("Dave", _) => "It's Dave!"
}
// res: String = It's Dave!

Person("Dave", "Gurnell") match {
  case Person(name, _) => s"It's $name!"
}
// res: String = It's Dave!

Person("John", "Doe") match {
  case Person("Noel", _) => "It's Noel!"
  case Person("Dave", _) => "It's Dave!"
  case _ => "It's someone else!"
}
// res: String = It's someone else!
```

A.1.6 Type patterns

A type pattern takes the form `x: Y` where `Y` is a type and `x` is a wildcard pattern or a variable pattern. The pattern matches any value of type `Y` and binds it to `x`:

```
val shape: Shape = Rectangle(1, 2)
// shape: Shape = Rectangle(1.0,2.0)

shape match {
  case c : Circle    => s"It's a circle: $c!"
  case r : Rectangle => s"It's a rectangle: $r!"
  case s : Square    => s"It's a square: $s!"
}
// res: String = It's a rectangle: Rectangle(1.0,2.0)!
```

A.1.7 Tuple patterns

Tuples of any arity can be matched with parenthesised expressions as follows:

```
(1, 2) match {
  case (a, b) => a + b
}
// res: Int = 3
```

A.1.8 Guard expressions

This isn't so much a pattern as a feature of the overall `match` syntax. We can add an extra condition to any `case` clause by suffixing the pattern with the keyword `if` and a regular Scala expression. For example:

```
123 match {
  case a if a % 2 == 0 => "even"
  case _ => "odd"
}
// res: String = odd
```

To reiterate, the code between the `if` and => keywords is a regular Scala expression, not a pattern.

A.2 Custom Patterns

In the last section we took an in-depth look at all of the types of pattern that are embedded into the pattern matching language. However, in that list we didn't see some of the patterns that we've been using in the course so far—case class and sequence patterns were nowhere to be seen!

There is a final aspect of pattern matching that we haven't covered that truly makes it a universal tool—we can define our own custom *extractor* patterns using regular Scala code and use them along-side the built-in patterns in our `match` expressions.

A.2.1 Extractors

An extractor pattern looks like a function call of zero or more arguments: `foo(a, b, c)`, where each argument is itself an arbitrary pattern.

Extractor patterns are defined by creating objects with a method called `unapply` or `unapplySeq`. We'll dive into the guts of these methods in a minute. For now let's look at some of the predefined extractor patterns from the Scala library.

A.2.1.1 Case class extractors

The companion object of every `case class` is equipped with an extractor that creates a pattern of the same arity as the constructor. This makes it easy to capture fields in variables:

```
Person("Dave", "Gurnell") match {
  case Person(f, l) => List(f, l)
}
// res: List[String] = List(Dave, Gurnell)
```

A.2.1.2 Regular expressions

Scala's regular expression objects are outfitted with a pattern that binds each of the captured groups:

```
import scala.util.matching.Regex

val r = new Regex("""(\d+)\.(\d+)\.(\d+)\.(\d+)""")
// r: scala.util.matching.Regex = (\d+)\.(\d+)\.(\d+)\.(\d+)

"192.168.0.1" match {
  case r(a, b, c, d) => List(a, b, c, d)
}
// res: List[String] = List(192, 168, 0, 1)
```

A.2.1.3 Lists and Sequences

Lists and sequences can be captured in several ways:

The `List` and `Seq` companion objects act as patterns that match fixed-length sequences.

```
List(1, 2, 3) match {
   case List(a, b, c) => a + b + c
}
// res: Int = 6
```

- `Nil` matches the empty list:

```
Nil match {
  case List(a) => "length 1"
  case Nil => "length 0"
}
// res: String = length 0
```

There is also a singleton object `::` that matches the head and tail of a list.

```
List(1, 2, 3) match {
  case ::(head, tail) => s"head $head tail $tail"
  case Nil => "empty"
}
// res: String = head 1 tail List(2, 3)
```

This perhaps makes more sense when you realise that binary extractor patterns can also be written infix.

```
List(1, 2, 3) match {
  case head :: tail => s"head $head tail $tail"
  case Nil => "empty"
}
// res: String = head 1 tail List(2, 3)
```

Combined use of `::`, `Nil`, and `_` allow us to match the first elements of any length of list.

```
List(1, 2, 3) match {
  case Nil => "length 0"
  case a :: Nil => s"length 1 starting $a"
  case a :: b :: Nil => s"length 2 starting $a $b"
  case a :: b :: c :: _ => s"length 3+ starting $a $b $c"
}
// res: String = length 3+ starting 1 2 3
```

A.2.1.4 Creating custom fixed-length extractors

You can use any object as a fixed-length extractor pattern by giving it a method called `unapply` with a particular type signature:

```
def unapply(value: A): Boolean           // pattern with 0 parameters
def unapply(value: A): Option[B]                    // 1 parameter
def unapply(value: A): Option[(B1, B2)]             // 2 parameters
                                                    // etc...
```

Each pattern matches values of type `A` and captures arguments of type `B`, `B1`, and so on. Case class patterns and `::` are examples of fixed-length extractors.

For example, the extractor below matches email addresses and splits them into their user and domain parts:

```
object Email {
  def unapply(str: String): Option[(String, String)] = {
    val parts = str.split("@")
    if (parts.length == 2) Some((parts(0), parts(1))) else None
  }
}

"dave@underscore.io" match {
  case Email(user, domain) => List(user, domain)
}
// res: List[String] = List(dave, underscore.io)

"dave" match {
  case Email(user, domain) => List(user, domain)
  case _ => Nil
}
// res: List[String] = List()
```

This simpler pattern matches any string and uppercases it:

```
object Uppercase {
  def unapply(str: String): Option[String] =
    Some(str.toUpperCase)
}

Person("Dave", "Gurnell") match {
  case Person(f, Uppercase(l)) => s"$f $l"
}
// res: String = Dave GURNELL
```

A.2.1.5 Creating custom variable-length extractors

We can also create extractors that match arbitrary numbers of arguments by defining an `unapplySeq` method of the following form:

```
def unapplySeq(value: A): Option[Seq[B]]
```

Variable-length extractors match a value only if the pattern in the `case` clause is the same length as the `Seq` returned by `unapplySeq`. `Regex` and `List` are examples of variable-length extractors.

The extractor below splits a string into its component words:

```
object Words {
  def unapplySeq(str: String) = Some(str.split(" ").toSeq)
}

"the quick brown fox" match {
  case Words(a, b, c)    => s"3 words: $a $b $c"
  case Words(a, b, c, d) => s"4 words: $a $b $c $d"
}
// res: String = 4 words: the quick brown fox
```

A.2.1.6 Wildcard sequence patterns

There is one final type of pattern that can only be used with variable-length extractors. The *wildcard sequence* pattern, written _*, matches zero or more arguments from a variable-length pattern and discards their values. For example:

```
List(1, 2, 3, 4, 5) match {
  case List(a, b, _*) => a + b
}
// res: Int = 3

"the quick brown fox" match {
  case Words(a, b, _*) => a + b
}
// res: String = "thequick"
```

We can combine wildcard patterns with the @ operator to capture the remaining elements in the sequence.

```
"the quick brown fox" match {
  case Words(a, b, rest @ _*) => rest
}
// res: Seq[String] = WrappedArray("brown", "fox")
```

A.2.2 Exercises

A.2.2.1 Positive Matches

Custom extractors allow us to abstract away complicated conditionals. In this example we will build a very simple extractor, which we probably wouldn't use in real code, but which is representative of this idea.

Create an extractor `Positive` that matches any positive integer. Some test cases:

```
assert(
  "No" ==
    (0 match {
       case Positive(_) => "Yes"
       case _ => "No"
     })
)

assert(
  "Yes" ==
    (42 match {
       case Positive(_) => "Yes"
       case _ => "No"
     })
)
```

See the solution

A.2.2.2 Titlecase extractor

Extractors can also transform their input. In this exercise we'll write an extractor that converts any string to titlecase by uppercasing the first letter of every word. A test case:

```
assert(
  "Sir Lord Doctor David Gurnell" ==
    ("sir lord doctor david gurnell" match {
       case Titlecase(str) => str
     })
)
```

Tips:

- Java `Strings` have the methods `split(String)`, `toUpperCase` and `substring(Int, Int)`.
- The method `split(String)` returns a Java `Array[String]`. You can convert this to a `List[String]` using `array.toList` so you can `map` over it and manipulate each word.
- A `List[String]` can be converted back to a `String` with the code `list.mkString(" ")`.

This extractor isn't particularly useful, and in general defining your own extractors is not common in Scala. However it can be a useful tool in certain circumstances.

See the solution

Appendix B

Collections Redux

This optional section covers some more details of the collections framework that typically aren't used in day-to-day programming. This includes the different sequence implementations available, details of collections operations on arrays and strings, some of the core traits in the framework, and details of Java interoperation.

B.1 Sequence Implementations

We've seen that the Scala collections seperate interface from implementation. This means we can work with all collections in a generic manner. However different concrete implementations have different performance characteristics, so we must be aware of the available implementations so we can choose appropriately. Here we look at the mostly frequently used implementations of `Seq`. For full details on all the available implementation see the docs.

B.1.1 Peformance Characteristics

The collections framework distinguishes at the type level two general classes of sequences. Sequences implementing `IndexedSeq` have efficient `apply`, `length`, and (if mutable) `update` operations, while `LinearSeq`s have efficient `head` and `tail` operations. Neither have any additional operations over `Seq`.

B.1.2 Immutable Implementations

The main immutable `Seq` implementations are `List`, and `Stream`, and `Vector`.

B.1.2.1 List

A `List` is a singly linked list. It has constant time access to the first element and remainder of the list (`head`, and `tail`) and is thus a `LinearSeq`. It also has constant time prepending to the front of the list, but linear time appending to the end. `List` is the default `Seq` implementation.

B.1.2.2 Stream

A `Stream` is like a list except its elements are computed on demand, and thus it can have infinite size. Like other collections we can create streams by calling the `apply` method on the companion object.

```
Stream(1, 2, 3)
// res: scala.collection.immutable.Stream[Int] = Stream(1, ?)
```

Note that only the first element is printed. The others will be computed when we try to access them.

We can also use the `#::` method to construct a stream from individual elements, starting from `Stream.empty`.

```
Stream.empty.#::(3).#::(2).#::(1)
// res: scala.collection.immutable.Stream[Int] = Stream(1, ?)
```

We can also use the more natural operator syntax.

```
1 #:: 2 #:: 3 #:: Stream.empty
// res: scala.collection.immutable.Stream[Int] = Stream(1, ?)
```

This method allows us to create a infinite stream. Here's an infinite stream of 1s:

```
def streamOnes: Stream[Int] = 1 #:: streamOnes

streamOnes
// res: Stream[Int] = Stream(1, ?)
```

Because elements are only evaluated as requested, calling `streamOnes` doesn't lead to infinte recursion. When we take the first five elements (and convert them to a `List`, so they'll all print out) we see we have what we want.

```
streamOnes.take(5).toList
// res: List[Int] = List(1, 1, 1, 1, 1)
```

B.1.2.3 Vector

`Vector` is the final immutable sequence we'll consider. Unlike `Stream` and `List` it is an `IndexedSeq`, and thus offers fast random access and updates. It is the default immutable `IndexedSeq`, which we can see if we create one.

```
scala.collection.immutable.IndexedSeq(1, 2, 3)
// res: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 2, 3)
```

Vectors are a good choice if you want both random access and immutability.

B.1.3 Mutable Implementations

The mutable collections are probably more familiar. In addition to linked lists and arrays (which we discuss in more detail later) there are buffers, which allow for efficient construction of certain data structures.

B.1.3.1 Buffers

`Buffers` are used when you want to efficiently create a data structure an item at a time. An `ArrayBuffer` is an `IndexedSeq` which also has constant time appends. A `ListBuffer` is like a `List` with constant time prepend *and* append (though note it is mutable, unlike `List`).

Buffers' add methods to support destructive prepends and appends. For example, the += is destructive append.

```
val buffer = new scala.collection.mutable.ArrayBuffer[Int]()
// buffer: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer()

buffer += 1
// res: buffer.type = ArrayBuffer(1)

buffer
// res: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1)
```

B.1.3.2 StringBuilder

A `StringBuilder` is essentially a buffer for building strings. It is mostly the same as Java's `StringBuilder` except that it implements standard Scala collections method where there is a conflict. So, for example, the `reverse` method creates a new `StringBuilder` unlike in Java.

B.1.3.3 LinkedLists

Mutable singly `LinkedLists` and `DoubleLinkedLists` work for the most part just like `List`. A `DoubleLikeList` maintains both a `prev` and `next` pointer and so allows for efficient removal of an element.

B.2 Arrays and Strings

Arrays and strings in Scala correspond to Java's arrays and strings.

```
"this is not a string"
// res: java.lang.String = this is not a string
```

Yet all the familiar collection methods are available on them.

```
"is it true?".map(elt => true)
// res: scala.collection.immutable.IndexedSeq[Boolean] = Vector(true, true, true, true, true, true,
        □
                                                                    true, true, true, true,
    true)

Array(1, 2, 3).map(_ * 2)
// res: Array[Int] = Array(2, 4, 6)
```

This conversion is done automatically using implicit conversions. There are two conversions. The Wrapped conversions (`WrappedArray` and `WrappedString`) wrap the original array or string in an object supporting the Seq methods. Operations on such a wrapped object return another wrapped object.

```
val sequence = new scala.collection.immutable.WrappedString("foo")
// sequence: scala.collection.immutable.WrappedString = foo

sequence.reverse
// res: scala.collection.immutable.WrappedString = oof
```

The Ops conversions (`ArrayOps` and `StringOps`) add methods that return an object of the original type. Thus these objects are short-lived.

```
val sequence = new scala.collection.immutable.StringOps("foo")
// sequence: scala.collection.immutable.StringOps = foo

sequence.reverse
// res: String = oof
```

The choice of conversion is based on the required type. If we use a string, say, where a Seq is expected the string will be wrapped. If we just want to use a Seq method on a string then an Op conversion will be used.

```
val sequence: Seq[Char] = "foo"
// sequence: Seq[Char] = foo

sequence.getClass
// res: java.lang.Class[_ <: Seq[Char]] = class scala.collection.immutable.WrappedString
```

B.2.1 Performance

You might be worried about the performance of implicit conversions. The Ops conversions are normally optimised away. The Wrapped conversions can give a small performance hit which may be an issue in particularly performance sensitive code.

B.3 Iterators and Views

Iterators and views are two parts of the collection library that don't find much use outside of a few special cases.

B.3.1 Iterators

Scala's iterators are like Java's iterators. You can use them to walk through the elements of a collection, but only once. Iterators have `hasNext` and `next` methods, with the obvious semantics. Otherwise they behave like sequences, though they don't inherit from `Seq`.

Iterators don't find a great deal of use in Scala. Two primary use cases are operating on collections that are too large to fit in memory or in particularly high performance code.

B.3.2 Views

When performing a sequence of transformations on a collection, a number of intermediate collections will be constructed. For example, in the below example two intermediate collections will be created by the first and second call to `map`.

```
Seq(1, 2, 3).map(_ * 2).map(_ + 4).map(_.toString)
// res: Seq[java.lang.String] = List(6, 8, 10)
```

It is as if we'd written

```
val intermediate1 = Seq(1, 2, 3).map(_ * 2)
val intermediate2 = intermediate1.map(_ + 4)
val result = intermediate2.map(_.toString)
```

These intermediate collections are not strictly necessary. We could instead do the full sequence of transformations on an element-by-element basis. Views allows this. We create a view by calling the `view` method on any collection. Any traversals of a view are only applied when the `force` method is called.

```
val view = Seq(1, 2, 3).view.map(_ * 2).map(_ + 4).map(_.toString)
// view: scala.collection.SeqView[java.lang.String,Seq[_]] = SeqViewMMM(...)

view.force
// res: Seq[java.lang.String] = List(6, 8, 10)
```

Note that when a view is forced the original type is retained.

For very large collections of items with many stages of transformations a view can be worthwhile. For modest sizes views are usually slower than creating the intermediate data structures.

B.4 Traversable and Iterable

So far we've avoided discussing the finer details of the collection class hierarchy. As we near the end of this section it is time to quickly go over some of the intricacies.

B.4.1 Traversable

The trait `Traversable` sits at the top of the collection hierarchy and represents a collection that allows traversal of its contents. The only abstract operation is `foreach`. Most of the collection methods are implemented in `Traversable`, though classes extending it may reimplement methods for performance.

B.4.1.1 TraversableOnce

`TraversableOnce` represents a collection that can be traversed one or more times. It is primarily used to reduce code duplication between `Iterators` and `Traversable`.

B.4.2 Iterable

`Iterable` is the next trait below `Traversable`. It has a single abstract method `iterator` that should return an `Iterator` over the collection's contents. The `foreach` method is implemented in terms of this. It adds a few methods to `Traversable` that can only be efficiently implemented when an iterator is available.

B.5 Java Interoperation

The prefered way to convert between Scala and Java collections is use the `JavaConverters` implicit conversions. We use it by importing `scala.collection.JavaConverters._` and then methods `asJava` and `asScala` become available on many of the collections.

```
import scala.collection.JavaConverters._

Seq(1, 2, 3).asJava
// res: java.util.List[Int] = [1, 2, 3]
```

Java does not distinguish mutable and immutable collections at the type level but the conversions do preserve this property by throwing `UnsupportOperationException` as appropriate.

```
val java = Seq(1, 2, 3).asJava
// java: java.util.List[Int] = [1, 2, 3]

java.set(0, 5)
// java.lang.UnsupportedOperationException
//  at java.util.AbstractList.set(AbstractList.java:115)
//     ...
```

The conversions go the other way as well.

```
val list: java.util.List[Int] = new java.util.ArrayList[Int]()
// list: java.util.List[Int] = []

list.asScala
// res: scala.collection.mutable.Buffer[Int] = Buffer()
```

Note that the Scala equivalent is a mutable collection. If we mutate an element we see that the underlying Java collection is also changed. This holds for all conversions; they always share data and are not copied.

```
list.asScala += 5
// res: scala.collection.mutable.Buffer[Int] = Buffer(5)

list
// res: java.util.List[Int] = [5]
```

B.5.1 JavaConversions

There is another set of conversions in `scala.collection.JavaConversions`, which perform conversions without needing the calls to `asJava` or `asScala`. Many people find this confusing in large systems and thus it is not recommended.

B.6 Mutable Sequences

Most of the interfaces we've have covered so far do not have any side-effects—like the `copy` method on a case class, they return a new copy of the sequence. Sometimes, however, we need mutable collections. Fortunately, Scala provides two parallel collections hierarchies, one in the `scala.collection.mutable` package and one in the `scala.collection.immutable` package.

The default `Seq` is defined to be `scala.collection.immutable.Seq`. If we want a mutable sequence we can use `scala.collection.mutable.Seq`.

```
val mutable = scala.collection.mutable.Seq(1, 2, 3)
// mutable: scala.collection.mutable.Seq[Int] = ArrayBuffer(1, 2, 3)
```

Note that the concrete implementation class is now an `ArrayBuffer` and not a `List`.

B.6.1 Destructive update

In addition to all the methods of an immutable sequence, a mutable sequence can be updated using the `update` method. Note that `update` returns `Unit`, so no value is printed in the REPL after this call. When we print the original sequence we see it is changed:

```
mutable.update(0, 5)

mutable
// res: scala.collection.mutable.Seq[Int] = ArrayBuffer(5, 2, 3)
```

A more idiomatic way of calling `update` is to use *assignment operator syntax*, which is another special syntax built in to Scala, similar to infix operator syntax and function application syntax:

```
mutable(1) = 7

mutable
// res: scala.collection.mutable.Seq[Int] = ArrayBuffer(5, 7, 3)
```

B.6.2 Immutable methods on mutable sequences

Methods defined on both mutable and immutable sequences will never perform destructive updates. For example, `:+` always returns a new copy of the sequence without updating the original:

```
val mutable = scala.collection.mutable.Seq[Int](1, 2, 3)
// mutable: scala.collection.mutable.Seq[Int] = ArrayBuffer(1, 2, 3)

mutable :+ 4
// res: scala.collection.mutable.Seq[Int] = ArrayBuffer(1, 2, 3, 4)

mutable
// res: scala.collection.mutable.Seq[Int] = ArrayBuffer(1, 2, 3)
```

B.6.2.1 Using Mutable Collections Safely

Scala programmers tend to favour immutable collections and only bring in mutable ones in specific circumastances. Using `import scala.collection.mutable._` at the top of a file tends to create a whole series of naming collisions that we have to work around.

To work around this, I suggest importing the `mutable` package iteself rather than its contents. We can then explicitly refer to any mutable collection using the package name as a prefix, leaving the unprefixed names referring to the immutable versions:

```
import scala.collection.mutable
import scala.collection.mutable

mutable.Seq(1, 2, 3)
// res: scala.collection.mutable.Seq[Int] = ArrayBuffer(1, 2, 3)

Seq(1, 2, 3)
// res: Seq[Int] = List(1, 2, 3)
```

B.6.3 In summary

Scala's collections library includes mutable sequences in the `scala.collection.mutable` package. The main extra operation is `update`:

Method	We have	We provide	We get
`update`	`Seq[A]`	`Int, A`	`Unit`

B.6.4 Exercises

B.6.4.1 Animals

Create a `Seq` containing the `Strings` `"cat"`, `"dog"`, and `"penguin"`. Bind it to the name `animals`.

See the solution

Append the element `"tyrannosaurus"` to `animals` and prepend the element `"mouse"`.

See the solution

What happens if you prepend the `Int` `2` to `animals`? Why? Try it out... were you correct?

See the solution

Now create a mutable sequence containing `"cat"`, `"dog"`, and `"penguin"` and `update` an element to be an `Int`. What happens?

See the solution

Appendix C

Solutions to Exercises

C.1 Expressions, Types, and Values

C.1.1 Solution to: Type and Value

Type is `Int` and value is 3.

Return to the exercise

C.1.2 Solution to: Type and Value Part 2

Type is `Int` and value is 3.

Return to the exercise

C.1.3 Solution to: Type and Value Part 3

Type is `Int`, but this one doesn't evaluate to a value—it raises an exception instead, and a raised exception is not a value. How can we tell this? We can't continuing computing with the result of the expression. For example, we can't print it. Compare

```
println("foo")
foo
```

and

```
println("foo".toInt)
java.lang.NumberFormatException: For input string: "foo"
```

In the latter no printing occurs indicating the `println` is never evaluated.

Return to the exercise

C.1.4 Solution to: Operator Style

```
"foo" take 1
```

Return to the exercise

C.1.5 Solution to: Operator Style Part 2

```
1.+(2).+(3)
```

Return to the exercise

C.1.6 Solution to: Substitution

The expressions have the same result type and return value. However, they arrive at their results in different ways. The first computes its result through a series of additions, while the later is simply a literal.

As neither expression has any side-effects, they are interchangeable from a user's point of view. Anywhere you can write `1 + 2 + 3` you can also write `6`, and vice versa, without changing the meaning of any program. This is known as *substitution* and you may remember the principle from simplifying algebraic formulae at school.

As programmers we must develop a mental model of how our code operates. The *substitution model of evaluation* is a particularly simple model that says anywhere we see an expression we may substitute its result. In the absence of side-effects, the substitution model always works[1]. If we know the types and values of each component of an expression, we know the type and value of the expression as a whole. In functional programming we aim to avoid side-effects for this reason: it makes our programs easier to understand.

Return to the exercise

C.1.7 Solution to: Literally Just Literals

`42` is an `Int`. `true` is a `Boolean`. `123L` is a `Long`. `42.0` is a `Double`.

This exercise just gives you some experience using the Scala console or Worksheet.

Return to the exercise

C.1.8 Solution to: Quotes and Misquotes

The first is a literal `Char` and the second is a literal `String`.

Return to the exercise

C.1.9 Solution to: An Aside on Side-Effects

The literal expression `"Hello world!"` evaluates to a `String` value. The expression `println("Hello world!")` evaluates to `Unit` and, as a side-effect, prints `"Hello world!"` on the console.

This an important distinction between a program that evaluates to a value and a program that prints a value as a side-effect. The former can be used in a larger expression but the latter cannot.

Return to the exercise

[1] What exactly is a side-effect? One workable definition is anything that causes substitution to yield an incorrect result. Does substitution *always* work, in the absence of side-effects? To truly have a correct model of Scala we must define the order in which we apply substitutions. There are a number of possible orders. (For example, we perform substitution left-to-right, or right-to-left? Do we substitute as soon as possible or delay until we need a value?) Most of the time order of substitution doesn't matter, but there are cases where it does. Scala always applies substitution from left-to-right and at the earliest possible time.

C.1.10 Solution to: Learning By Mistakes

You should see an error message. Take the time to read and get used to the error messages in your development environment—you'll see plenty more of them soon!

Return to the exercise

C.1.11 Solution to: Cat-o-matique

This is just a finger exercise to get you used to the syntax of defining objects. You should have a solution similar to the code below.

```
object Oswald {
  val colour: String = "Black"
  val food: String = "Milk"
}

object Henderson {
  val colour: String = "Ginger"
  val food: String = "Chips"
}

object Quentin {
  val colour: String = "Tabby and white"
  val food: String = "Curry"
}
```

Return to the exercise

C.1.12 Solution to: Square Dance!

Here is the solution. `cube(x)` calls `square(x)` and multiplies its value by `x` one more time. The return type of each method is inferred by the compiler as `Double`.

```
object calc {
  def square(x: Double) = x * x
  def cube(x: Double) = x * square(x)
}
```

Return to the exercise

C.1.13 Solution to: Precise Square Dance!

Like Java, Scala can't generalize particularly well across `Int`s and `Double`s. However, it will allow us to *"overload"* the `square` and `cube` methods by defining them for each type of parameter.

```
object calc2 {
  def square(value: Double) = value * value
  def cube(value: Double) = value * square(value)

  def square(value: Int) = value * value
  def cube(value: Int) = value * square(value)
}
```

"Overloaded" methods are ones we have defined several times for different argument types. Whenever we call an overloaded method type, Scala automatically determines which variant we need by looking at the type of the argument.

```
calc2.square(1.0) // calls the `Double` version of `square`
calc2.square(1)   // calls the `Int` version `square`
```

The Scala compiler is able to insert automatic conversions between numeric types wherever you have a lower precision and require a higher precision. For example, if you write `calc.square(2)`, the compiler determines that the only version of `calc.square` takes a `Double` and automatically infers that you really mean `calc.square(2.toDouble)`.

Conversions in the opposite direction, from high precision to low precision, are not handled automatically because they can lead to rounding errors. For example, the code below will not compile because `x` is an `Int` and its body expression is a `Double` (try it and see)!

```
val x: Int = calc.square(2) // compile error
```

You can manually use the `toInt` method of `Double` to work around this:

```
val x: Int = calc.square(2).toInt // toInt rounds down
```

> **The Dangers of String Concatenation**
>
> To maintain similar behaviour to Java, Scala also automatically converts any object to a `String` where required. This is to make it easy to write things like `println("a" + 1)`, which Scala automatically rewrites as `println("a" + 1.toString)`.
>
> The fact that string concatenation and numeric addition share the same + method can sometimes cause unexpected bugs, so watch out!

Return to the exercise

C.1.14 Solution to: Order of evaluation

Here is the solution:

```
b
a
c
a
a
// res: String = 3c31
```

The full sequence of evaluation is as follows:

```
- We calculate the main sum at the end of the program, which...
  - Loads `argh`, which...
    - Calculates all the fields in `argh`, which...
```

```
      - Calculates `b`, which...
        - Prints `"b"`
        - Evaluates `a + 2`, which...
          - Calls `a`, which...
            - Prints `"a"`
            - Returns `1`
          - Returns `1 + 2`
        - Stores the value `3` in `b`
  - Calls `argh.c`, which...
    - Prints `"c"`
    - Evaluates `a`
      - Prints `"a"`
      - Returns `1` - Which we discard
    - Evaluates `b + "c"`, which...
      - Retrieves the value `3` from `b`
        - Retrieves the value `"c"`
        - Evaluates the `+`, determining that it actually refers to string
          concatenation and converting `3` to `"3"`
        - Returns the `String` `"3c"`
  - Calls `argh.b`, which...
    - Retrieves the value `3` from `b`
  - Evaluates the first `+`, determining that it actually refers to string
    concatentation, and yielding `"3c3"`
  - Calls `argh.a`, which...
    - Prints `"a"`
    - Returns `1`
  - Evaluates the first `+`, determining that it actually refers to string
    concatentation, and yielding `"3c31"`
```

Whew! That's a lot for such a simple piece of code.

Return to the exercise

C.1.15 Solution to: Greetings, human

```
object person {
  val firstName = "Dave"
  val lastName = "Gurnell"
}

object alien {
  def greet(p: person.type) =
    "Greetings, " + p.firstName + " " + p.lastName
}

alien.greet(person)
```

Notice the type on the p parameter of `greet`: `person.type`. This is one of the *singleton types* we were referring to earlier. In this case it is specific to the object `person`, which prevents us using `greet` on any other object. This is very different from a type such as `Int` that is shared by all Scala integers.

This imposes a significant limitation on our ability to write programs in Scala. We can only write methods that work with built-in types or single objects of our own creation. In order to build useful programs we need the

ability to *define our own types* and create multiple values of each. We can do this using `classes`, which we will cover in the next section.

Return to the exercise

C.1.16 Solution to: The Value of Methods

First let's deal with the equivalence between methods and expressions. As we know, expressions are program fragments that produce values. A simple test of whether something is an expression is to see if we can assign it to a field.

```
object calculator {
  def square(x: Int) = x * x
}

val someField = calculator.square
// error: missing arguments for method square in object calculator;
// follow this method with `_' if you want to treat it as a partially applied function
//        val someField = calculator.square
//                                   ^
```

Although we don't understand this error message fully yet (we shall learn about "partially applied functions" later), it does show us that `square` *is not an expression*. However, a *call* to `square` *does* yield a value:

```
val someField = calculator.square(2)
// someField: Int = 4
```

A method with no arguments looks like it behaves differently. However, this is a trick of the syntax.

```
object clock {
  def time = System.currentTimeMillis
}

val now = clock.time
// now: Long = 1395402828639
```

Although it looks like now is being assigned `clock.time` as a value, it is actually being assigned the *value returned by calling* `clock.time`. We can demonstrate this by calling the method again:

```
val aBitLaterThanNow = clock.time
// aBitLaterThanNow: Long = 1395403220551
```

As we saw above, references to fields and calls to argumentless methods look identical in Scala. This is by design, to allow us to swap the implementation of a field for a method (and vice versa) without affecting other code. It is a programming language feature called the *uniform access principle*.

So, in summary, *calls to methods are expressions* but *methods themselves are not expressions*. In addition to methods, Scala also has a concept called *functions*, which are objects that can be invoked like methods. As we know objects are values, so functions are also values and can be treated as data. As you may have guessed, functions are a critical part of *functional programming*, which is one of Scala's major strengths. We will learn about functions and functional programming in a bit.

Return to the exercise

C.1.17 Solution to: A Classic Rivalry

It's a `String` with value `"predator"`. Predators are clearly best.

The type is determined by the upper bound of the types in the *then* and *else* expressions. In this case both expressions are `Strings` so the result is also a `String`.

The value is determined at runtime. `2` is greater than `1` so the conditional evaluates to the value of the *else* expression.

Return to the exercise

C.1.18 Solution to: A Less Well Known Rivalry

It's a value of type `Any` with value `2001`.

This is similar to the previous exercise—the difference is the type of the result. We saw earlier that the type is the *upper bound* of the positive and negative arms of the expression. `"alien"` and `2001` are completely different types - their closest common ancestor is `Any`, which is the grand supertype of all Scala types.

This is an important observation: types are determined at compile time, before the program is run. The compiler doesn't know which of `1` and `2` is greater before running the program, so it can only make a best guess at the type of the result of the conditional. `Any` is as close as it can get in this program, whereas in the previous exercise it can get all the way down to `String`.

We'll learn more about `Any` in the following sections. Java programmers shouldn't confuse it with `Object` because it subsumes value types like `Int` and `Boolean` as well.

Return to the exercise

C.1.19 Solution to: An if Without an else

The result type and value are `Any` and `()` respectively.

All code being equal, conditionals without `else` expressions only evaluate to a value half of the time. Scala works around this by returning the `Unit` value if the `else` branch should be evaluated. We would usually only use these expressions for their side-effects.

Return to the exercise

C.2 Objects and Classes

C.2.1 Solution to: Cats, Again

This is a finger exercise to get you used to the syntax of defining classes.

```
class Cat(val colour: String, val food: String)

val oswald = new Cat("Black", "Milk")
val henderson = new Cat("Ginger", "Chips")
val quentin = new Cat("Tabby and white", "Curry")
```

Return to the exercise

C.2.2 Solution to: Cats on the Prowl

```
object ChipShop {
  def willServe(cat: Cat): Boolean =
    if(cat.food == "Chips")
      true
    else
      false
}
```

Return to the exercise

C.2.3 Solution to: Directorial Debut

This exercise provides some hands on experience writing Scala classes, fields and methods. The model solution is as follows:

```
class Director(
  val firstName: String,
  val lastName: String,
  val yearOfBirth: Int) {

  def name: String =
    s"$firstName $lastName"

  def copy(
    firstName: String = this.firstName,
    lastName: String = this.lastName,
    yearOfBirth: Int = this.yearOfBirth): Director =
    new Director(firstName, lastName, yearOfBirth)
}

class Film(
  val name: String,
  val yearOfRelease: Int,
  val imdbRating: Double,
  val director: Director) {

  def directorsAge =
    yearOfRelease - director.yearOfBirth

  def isDirectedBy(director: Director) =
    this.director == director

  def copy(
    name: String = this.name,
    yearOfRelease: Int = this.yearOfRelease,
    imdbRating: Double = this.imdbRating,
    director: Director = this.director): Film =
    new Film(name, yearOfRelease, imdbRating, director)
}
```

Return to the exercise

C.2.4 Solution to: A Simple Counter

```
class Counter(val count: Int) {
  def dec = new Counter(count - 1)
  def inc = new Counter(count + 1)
}
```

Aside from practicing with classes and objects, this exercise has a second goal—to think about why `inc` and `dec` return a new `Counter`, rather than updating the same counter directly.

Because `val` fields are immutable, we need to come up with some other way of propagating the new value of `count`. Methods that return new `Counter` objects give us a way of returning new state without the side-effects of assignment. They also permit *method chaining*, allowing us to write whole sequences of updates in a single expression

The use-case `new Counter(10).inc.dec.inc.inc.count` actually creates 5 instances of `Counter` before returning its final `Int` value. You may be concerned about the extra memory and CPU overhead for such a simple calculation, but don't be. Modern execution environments like the JVM render the extra overhead of this style of programming negligable in all but the most performance critical code.

Return to the exercise

C.2.5 Solution to: Counting Faster

The simplest solution is this:

```
class Counter(val count: Int) {
  def dec(amount: Int = 1) = new Counter(count - amount)
  def inc(amount: Int = 1) = new Counter(count + amount)
}
```

However, this adds parentheses to `inc` and `dec`. If we omit the parameter we now have to provide an empty pair of parentheses:

```
new Counter(10).inc
// error: missing arguments for method inc in class Counter;
// follow this method with `_' if you want to treat it as a partially applied function
//               new Counter(10).inc
//                                ^
```

We can work around this using *method overloading* to recreate our original parenthesis-free methods. Note that overloading methods requires us to specify the return types:

```
class Counter(val count: Int) {
  def dec: Counter = dec()
  def inc: Counter = inc()
  def dec(amount: Int = 1): Counter = new Counter(count - amount)
  def inc(amount: Int = 1): Counter = new Counter(count + amount)
}

new Counter(10).inc.inc(10).count
// res: Int = 21
```

Return to the exercise

C.2.6 Solution to: Additional Counting

```
class Counter(val count: Int) {
  def dec = new Counter(count - 1)
  def inc = new Counter(count + 1)
  def adjust(adder: Adder) =
    new Counter(adder.add(count))
}
```

This is an interesting pattern that will become more powerful as we learn more features of Scala. *We are using `Adders` to capture computations* and pass them to `Counter`. Remember from our earlier discussion that *methods are not expressions*—they cannot be stored in fields or passed around as data. However, *`Adders` are both objects and computations*.

Using objects as computations is a common paradigm in object oriented programming languages. Consider, for example, the classic `ActionListener` from Java's Swing:

```
public class MyActionListener implements ActionListener {
  public void actionPerformed(ActionEvent evt) {
    // Do some computation
  }
}
```

The disadvantage of objects like `Adders` and `ActionListeners` is that they are limited to use in one particular circumstance. Scala includes a much more general concept called *functions* that allow us to represent any kind of computation as an object. We will be introduced to some of the concepts behind functions in this chapter.

Return to the exercise

C.2.7 Solution to: When is a Function not a Function?

The main thing we're missing is *types*, which are the way we properly abstract across values.

At the moment we can define a class called `Adder` to capture the idea of adding to a number, but that code isn't properly portable across codebases—other developers need to know about our specific class to use it.

We could define a library of common function types with names like `Handler`, `Callback`, `Adder`, `BinaryAdder`, and so on, but this quickly becomes impractical.

Later on we will see how Scala copes with this problem by defining a generic set of function types that we can use in a wide variety of situations.

Return to the exercise

C.2.8 Solution to: Friendly Person Factory

Here is the code:

```
object Person {
  def apply(name: String): Person = {
    val parts = name.split(" ")
    new Person(parts(0), parts(1))
  }
}
```

And here it is in use:

```
Person.apply("John Doe").firstName // full method call
// res: String = John

Person("John Doe").firstName // sugared apply syntax
// res: String = John
```

Return to the exercise

C.2.9 Solution to: Extended Body of Work

This exercise is inteded to provide more practice writing code. The model solution, including the class definitions from the previous section, is now:

```
class Director(
  val firstName: String,
  val lastName: String,
  val yearOfBirth: Int) {

  def name: String =
    s"$firstName $lastName"

  def copy(
    firstName: String = this.firstName,
    lastName: String = this.lastName,
    yearOfBirth: Int = this.yearOfBirth) =
    new Director(firstName, lastName, yearOfBirth)
}

object Director {
  def apply(firstName: String, lastName: String, yearOfBirth: Int): Director =
    new Director(firstName, lastName, yearOfBirth)

  def older(director1: Director, director2: Director): Director =
    if (director1.yearOfBirth < director2.yearOfBirth) director1 else director2
}

class Film(
  val name: String,
  val yearOfRelease: Int,
  val imdbRating: Double,
  val director: Director) {

  def directorsAge =
    director.yearOfBirth - yearOfRelease

  def isDirectedBy(director: Director) =
    this.director == director

  def copy(
    name: String = this.name,
    yearOfRelease: Int = this.yearOfRelease,
    imdbRating: Double = this.imdbRating,
    director: Director = this.director) =
    new Film(name, yearOfRelease, imdbRating, director)
}

object Film {
  def apply(
```

```
    name: String,
    yearOfRelease: Int,
    imdbRating: Double,
    director: Director): Film =
    new Film(name, yearOfRelease, imdbRating, director)

  def newer(film1: Film, film2: Film): Film =
    if (film1.yearOfRelease < film2.yearOfRelease) film1 else film2

  def highestRating(film1: Film, film2: Film): Double = {
    val rating1 = film1.imdbRating
    val rating2 = film2.imdbRating
    if (rating1 > rating2) rating1 else rating2
  }

  def oldestDirectorAtTheTime(film1: Film, film2: Film): Director =
    if (film1.directorsAge > film2.directorsAge) film1.director else film2.director
}
```

Return to the exercise

C.2.10 Solution to: Type or Value?

Type!—this code is defining a value `prestige` of type `Film`.

Return to the exercise

C.2.11 Solution to: Type or Value? Part 2

Type!—this is a reference to the *constructor* of `Film`. The constructor is part of the *class* `Film`, which is a *type*.

Return to the exercise

C.2.12 Solution to: Type or Value? Part 3

Value!—this is shorthand for:

```
Film.apply("Last Action Hero", 1993, mcTiernan)
```

`apply` is a method defined on the *singleton object* (or value) `Film`.

Return to the exercise

C.2.13 Solution to: Type or Value? Part 4

Value!—`newer` is another method defined on the *singleton object* `Film`.

Return to the exercise

C.2.14 Solution to: Type or Value? Part 5

Value!—This is tricky! You'd be forgiven for getting this one wrong.

`Film.type` refers to the type of the singleton object `Film`, so in this case `Film` is a reference to a value. However, the whole fragment of code is a type.

Return to the exercise

C.2.15 Solution to: Case Cats

Another simple finger exercise.

```
case class Cat(colour: String, food: String)
```

Return to the exercise

C.2.16 Solution to: Roger Ebert Said it Best...

Case classes provide our `copy` methods and our `apply` methods and remove the need to write val' before each constructor argument. The final codebase looks like this:

```
case class Director(firstName: String, lastName: String, yearOfBirth: Int) {
  def name: String =
    s"$firstName $lastName"
}

object Director {
  def older(director1: Director, director2: Director): Director =
    if (director1.yearOfBirth < director2.yearOfBirth) director1 else director2
}

case class Film(
  name: String,
  yearOfRelease: Int,
  imdbRating: Double,
  director: Director) {

  def directorsAge =
    yearOfRelease - director.yearOfBirth

  def isDirectedBy(director: Director) =
    this.director == director
}

object Film {
  def newer(film1: Film, film2: Film): Film =
    if (film1.yearOfRelease < film2.yearOfRelease) film1 else film2

  def highestRating(film1: Film, film2: Film): Double = {
    val rating1 = film1.imdbRating
    val rating2 = film2.imdbRating
    if (rating1 > rating2) rating1 else rating2
  }

  def oldestDirectorAtTheTime(film1: Film, film2: Film): Director =
    if (film1.directorsAge > film2.directorsAge) film1.director else film2.director
```

```
}
```

Not only is this code significantly shorter, it also provides us with `equals` methods, `toString` methods, and pattern matching functionality that will set us up for later exercises.

Return to the exercise

C.2.17 Solution to: Case Class Counter

```
case class Counter(count: Int = 0) {
  def dec = copy(count = count - 1)
  def inc = copy(count = count + 1)
  def adjust(adder: Adder) = copy(count = adder(count))
}
```

This is almost a trick exercise—there are very few differences with the previous implementation However, notice the extra functionality we got for free:

```
Counter(0) // construct objects without `new`
// res: Counter = Counter(0)

Counter().inc // printout shows the value of `count`
// res: Counter = Counter(1)

Counter().inc.dec == Counter().dec.inc // semantic equality check
// res: Boolean = true
```

Return to the exercise

C.2.18 Solution to: Application, Application, Application

Here's the code:

```
case class Person(firstName: String, lastName: String) {
  def name = firstName + " " + lastName
}

object Person {
  def apply(name: String): Person = {
    val parts = name.split(" ")
    apply(parts(0), parts(1))
  }
}
```

Even though we are defining a companion object for `Person`, Scala's case class code generator is still working as expected—it adds the auto-generated companion methods to the object we have defined, which is why we need to place the class and companion in a single compilation unit.

This means we end up with a companion object with an overloaded `apply` method with two possible type signatures:

```
def apply(name: String): Person // and ...
def apply(firstName: String, lastName: String): Person
```

Return to the exercise

C.2.19 Solution to: Feed the Cats

We can start by writing the skeleton suggested by the problem text.

```
object ChipShop {
  def willServe(cat: Cat): Boolean =
    cat match {
      case Cat(???, ???, ???) => ???
    }
}
```

As the return type is `Boolean` we know we need at least two cases, one for true and one for false. The text of the exercise tells us what they should be: cats that prefer chips, and all other cats. We can implement this with a literal pattern and an _ pattern.

```
object ChipShop {
  def willServe(cat: Cat): Boolean =
    cat match {
      case Cat(_, "Chips") => true
      case Cat(_, _) => false
    }
}
```

Return to the exercise

C.2.20 Solution to: Get Off My Lawn!

```
object Dad {
  def rate(film: Film): Double =
    film match {
      case Film(_, _, _, Director("Clint", "Eastwood", _)) => 10.0
      case Film(_, _, _, Director("John", "McTiernan", _)) => 7.0
      case _ => 3.0
    }
}
```

Pattern matching is a bit annoying in this case. Later on we'll learn how we can use pattern matching to match a particular value, called a *constant pattern*.

Return to the exercise

C.3 Modelling Data with Traits

C.3.1 Solution to: Cats, and More Cats

This is mostly a finger exercise to get you used to trait syntax but there are a few interesting things in the solution.

```
trait Feline {
  def colour: String
  def sound: String
}
case class Lion(colour: String, maneSize: Int) extends Feline {
  val sound = "roar"
}
```

```
case class Tiger(colour: String) extends Feline {
  val sound = "roar"
}
case class Panther(colour: String) extends Feline {
  val sound = "roar"
}
case class Cat(colour: String, food: String) extends Feline {
  val sound = "meow"
}
```

Notice that `sound` is not defined as a constructor argument. Since it is a constant, it doesn't make sense to give users a chance to modify it. There is a lot of duplication in the definition of `sound`. We could define a default value in `Feline` like so

```
trait Feline {
  def colour: String
  def sound: String = "roar"
}
```

This is generally a bad practice. If we define a default implementation it should be an implementation that is suitable for all subtypes.

Another alternative to define an intermediate type, perhaps called `BigCat` that defines sound as `"roar"`. This is a better solution.

```
trait BigCat extends Feline {
  val sound = "roar"
}
case class Tiger(...) extends BigCat
case class Lion(...) extends BigCat
case class Panther(...) extends BigCat
```

Return to the exercise

C.3.2 Solution to: Shaping Up With Traits

```
trait Shape {
  def sides: Int
  def perimeter: Double
  def area: Double
}

case class Circle(radius: Double) extends Shape {
  val sides = 1
  val perimeter = 2 * math.Pi * radius
  val area = math.Pi * radius * radius
}

case class Rectangle(width: Double, height: Double) extends Shape {
  val sides = 4
  val perimeter = 2 * width + 2 * height
  val area = width * height
}

case class Square(size: Double) extends Shape {
  val sides = 4
  val perimeter = 4 * size
  val area = size * size
```

```
}
```

Return to the exercise

C.3.3 Solution to: Shaping Up 2 (Da Streets)

The new code looks like this:

```
// trait Shape ...

// case class Circle ...

trait Rectangular extends Shape {
  def width: Int
  def height: Int
  val sides = 4
  val perimeter = 2*width + 2*height
  val area = width*height
}

case class Square(val size: Int) extends Rectangular {
  val width = size
  val height = size
}

case class Rectangle(val width: Int, val height: Int) extends Rectangular
```

Return to the exercise

C.3.4 Solution to: Printing Shapes

```
object Draw {
  def apply(shape: Shape): String = shape match {
    case Rectangle(width, height) =>
      s"A rectangle of width ${width}cm and height ${height}cm"

    case Square(size) =>
      s"A square of size ${size}cm"

    case Circle(radius) =>
      s"A circle of radius ${radius}cm"
  }
}
```

Return to the exercise

C.3.5 Solution to: The Color and the Shape

One solution to this exercise is presented below. Remember that a lot of the implementation details are unimportant—the crucial aspects of a correct solution are:

- There must be a `sealed trait Color`:
  - The trait should contain three `def` methods for the RGB values.

  - The trait should contains the `isLight` method, defined in terms of the RGB values.

- There must be three objects representing the predefined colours:
  - Each object must `extend Color`.
  - Each object should override the RGB values as `vals`.
  - Marking the objects as `final` is optional.
  - Making the objects `case objects` is also optional.
- There must be a class representing custom colours:
  - The class must `extend Color`.
  - Marking the class `final` is optional.
  - Making the class a `case class` is optional (although highly recommended).
- There should ideally be two methods in `Draw`:
  - One method should accept a `Color` as a parameter and one a `Shape`.
  - The method names are unimportant.
  - Each method should perform a `match` on the supplied value and provide enough `cases` to cover all possible subtypes.
- The whole codebase should compile and produce sensible values when tested!

```
// Shape uses Color so we define Color first:
sealed trait Color {
  // We decided to store RGB values as doubles between 0.0 and 1.0.
  //
  // It is always good practice to define abstract members as `defs`
  // so we can implement them with `defs`, `vals` or `vars`.
  def red: Double
  def green: Double
  def blue: Double

  // We decided to define a "light" colour as one with
  // an average RGB of more than 0.5:
  def isLight = (red + green + blue) / 3.0 > 0.5
  def isDark = !isLight
}

final case object Red extends Color {
  // Here we have implemented the RGB values as `vals`
  // because the values cannot change:
  val red = 1.0
  val green = 0.0
  val blue = 0.0
 }

final case object Yellow extends Color {
  // Here we have implemented the RGB values as `vals`
  // because the values cannot change:
  val red = 1.0
  val green = 1.0
  val blue = 0.0
}

final case object Pink extends Color {
  // Here we have implemented the RGB values as `vals`
  // because the values cannot change:
  val red = 1.0
```

```
  val green = 0.0
  val blue = 1.0
}

// The arguments to the case class here generate `val` declarations
// that implement the RGB methods from `Color`:
final case class CustomColor(
  red: Double,
  green: Double,
  blue: Double) extends Color

// The code from the previous exercise comes across almost verbatim,
// except that we add a `color` field to `Shape` and its subtypes:
sealed trait Shape {
  def sides: Int
  def perimeter: Double
  def area: Double
  def color: Color
}

final case class Circle(radius: Double, color: Color) extends Shape {
  val sides = 1
  val perimeter = 2 * math.Pi * radius
  val area = math.Pi * radius * radius
}

sealed trait Rectangular extends Shape {
  def width: Double
  def height: Double
  val sides = 4
  val perimeter = 2 * width + 2 * height
  val area = width * height
}

final case class Square(size: Double, color: Color) extends Rectangular {
  val width = size
  val height = size
}

final case class Rectangle(
  width: Double,
  height: Double,
  color: Color) extends Rectangular

// We decided to overload the `Draw.apply` method for `Shape` and
// `Color` on the basis that we may want to reuse the `Color` code
// directly elsewhere:
object Draw {
  def apply(shape: Shape): String = shape match {
    case Circle(radius, color) =>
      s"A ${Draw(color)} circle of radius ${radius}cm"

    case Square(size, color) =>
      s"A ${Draw(color)} square of size ${size}cm"

    case Rectangle(width, height, color) =>
      s"A ${Draw(color)} rectangle of width ${width}cm and height ${height}cm"
  }

  def apply(color: Color): String = color match {
```

```
    // We deal with each of the predefined Colors with special cases:
    case Red    => "red"
    case Yellow => "yellow"
    case Pink   => "pink"
    case color  => if(color.isLight) "light" else "dark"
  }
}

// Test code:

Draw(Circle(10, Pink))
// returns "A pink circle of radius 10.0cm"

Draw(Rectangle(3, 4, CustomColor(0.4, 0.4, 0.6)))
// returns "A dark rectangle of width 3.0cm and height 4.0cm"
```

Return to the exercise

C.3.6 Solution to: A Short Division Exercise

Here's the code:

```
sealed trait DivisionResult
final case class Finite(value: Int) extends DivisionResult
final case object Infinite extends DivisionResult

object divide {
  def apply(num: Int, den: Int): DivisionResult =
    if(den == 0) Infinite else Finite(num / den)
}

divide(1, 0) match {
  case Finite(value) => s"It's finite: ${value}"
  case Infinite      => s"It's infinite"
}
```

The result of `divide.apply` is a `DivisionResult`, which is a `sealed trait` with two subtypes. The subtype `Finite` is a `case class` encapsulting the result, but the subtype `Infinite` can simply be an object. We've used a `case object` for parity with `Finite`.

The implementation of `divide.apply` is simple - we perform a test and return a result. Note that we haven't annotated the method with a result type—Scala is capable of inferring the type `DivisionResult` as the least upper bound of `Infinite` and `Finite`.

Finally, the match illustrates a case class pattern with the parentheses, and a case object pattern without.

Return to the exercise

C.3.7 Solution to: Stop on a Dime

This is a direct application of the sum type pattern.

```
sealed trait TrafficLight
final case object Red extends TrafficLight
final case object Green extends TrafficLight
```

```
final case object Yellow extends TrafficLight
```

As there are fields or methods on the three cases, and thus there is no need to create than one instance of them, I used case objects instead of case classes.

Return to the exercise

C.3.8 Solution to: Calculator

```
sealed trait Calculation
final case class Success(result: Int) extends Calculation
final case class Failure(reason: String) extends Calculation
```

Return to the exercise

C.3.9 Solution to: Water, Water, Everywhere

Crank the handle on the product and sum type patterns.

```
final case class BottledWater(size: Int, source: Source, carbonated: Boolean)
sealed trait Source
final case object Well extends Source
final case object Spring extends Source
final case object Tap extends Source
```

Return to the exercise

C.3.10 Solution to: Traffic Lights

First with polymorphism:

```
sealed trait TrafficLight {
  def next: TrafficLight
}
final case object Red extends TrafficLight {
  def next: TrafficLight =
    Green
}
final case object Green extends TrafficLight {
  def next: TrafficLight =
    Yellow
}
final case object Yellow extends TrafficLight {
  def next: TrafficLight =
    Red
}
```

Now with pattern matching:

```
sealed trait TrafficLight {
  def next: TrafficLight =
    this match {
      case Red => Green
      case Green => Yellow
      case Yellow => Red
    }
```

```
}
final case object Red extends TrafficLight
final case object Green extends TrafficLight
final case object Yellow extends TrafficLight
```

In this case I think implementing inside the class using pattern matching is best. `Next` doesn't depend on any external data and we probably only want one implementation of it. Pattern matching makes the structure of the state machine clearer than polymorphism.

Ultimately there are no hard-and-fast rules, and we must consider our design decisions in the context of the larger program we are writing.

Return to the exercise

C.3.11 Solution to: Calculation

Start by implementing the framework the exercise calls for:

```
object Calculator {
  def +(calc: Calculation, operand: Int): Calculation = ???
  def -(calc: Calculation, operand: Int): Calculation = ???
}
```

Now apply the structural recursion pattern:

```
object Calculator {
  def +(calc: Calculation, operand: Int): Calculation =
    calc match {
        case Success(result) => ???
        case Failure(reason) => ???
    }
  def -(calc: Calculation, operand: Int): Calculation =
    calc match {
      case Success(result) => ???
      case Failure(reason) => ???
    }
}
```

To write the remaining bodies of the methods we can no longer rely on the patterns. However, a bit of thought quickly leads us to the correct answer. We know that + and - are binary operations; we need two integers to use them. We also know we need to return a `Calculation`. Looking at the `Failure` cases, we don't have two `Int`s available. The only result that makes sense to return is `Failure`. On the `Success` side, we *do* have two `Int`s and thus we should return `Success`. This gives us:

```
object Calculator {
  def +(calc: Calculation, operand: Int): Calculation =
    calc match {
        case Success(result) => Success(result + operand)
        case Failure(reason) => Failure(reason)
    }
  def -(calc: Calculation, operand: Int): Calculation =
    calc match {
      case Success(result) => Success(result - operand)
      case Failure(reason) => Failure(reason)
    }
}
```

Return to the exercise

C.3.12 Solution to: Calculation Part 2

The important points here are:

1. We have the same general pattern as before, matching on the `Calculation` *first* to implement our fail fast behavior.
2. After matching on our `Calculation` we then check for division by zero.

```
def /(calc: Calculation, operand: Int): Calculation =
  calc match {
    case Success(result) =>
      operand match {
        case 0 => Failure("Division by zero")
        case _ => Success(result / operand)
      }
    case Failure(reason) => Failure(reason)
  }
```

Return to the exercise

C.3.13 Solution to: Email

I would implement the method in an `EmailService` object. There are a lot of details to do with sending an email that have nothing to do with our `Visitor` class. I would rather keep these details in a separate abstraction.

Return to the exercise

C.3.14 Solution to: A List of Methods

```
sealed trait IntList {
  def length: Int =
    this match {
      case End => 0
      case Pair(hd, tl) => 1 + tl.length
    }
}
final case object End extends IntList
final case class Pair(head: Int, tail: IntLIst) extends IntList
```

Return to the exercise

C.3.15 Solution to: A List of Methods Part 2

```
sealed trait IntList {
  def product: Int =
    this match {
      case End => 1
      case Pair(hd, tl) => hd * tl.product
    }
}
```

```
final case object End extends IntList
final case class Pair(head: Int, tail: IntList) extends IntList
```

Return to the exercise

C.3.16 Solution to: A List of Methods Part 3

```
sealed trait IntList {
  def double: IntList =
    this match {
      case End => End
      case Pair(hd, tl) => Pair(hd * 2, tl.double)
    }
}
final case object End extends IntList
final case class Pair(head: Int, tail: IntList) extends IntList
```

Return to the exercise

C.3.17 Solution to: The Forest of Trees

```
sealed trait Tree
final case class Node(val l: Tree, val r: Tree) extends Tree
final case class Leaf(val elt: Int) extends Tree
```

Return to the exercise

C.3.18 Solution to: The Forest of Trees Part 2

```
object TreeOps {
  def sum(tree: Tree): Int =
    tree match {
      case Leaf(elt) => elt
      case Node(l, r) => sum(l) + sum(r)
    }

  def double(tree: Tree): Tree =
    tree match {
      case Leaf(elt) => Leaf(elt * 2)
      case Node(l, r) => Node(double(l), double(r))
    }
}

sealed trait Tree {
  def sum: Int
  def double: Tree
}
final case class Node(val l: Tree, val r: Tree) extends Tree {
  def sum: Int =
    l.sum + r.sum

  def double: Tree =
    Node(l.double, r.double)
}
final case class Leaf(val elt: Int) extends Tree {
  def sum: Int =
```

```
    elt

  def double: Tree =
    Leaf(elt * 2)
}
```

Return to the exercise

C.3.19 Solution to: A Calculator

This is a straightforward algebraic data type.

```
sealed trait Expression
final case class Addition(left: Expression, right: Expression) extends Expression
final case class Subtraction(left: Expression, right: Expression) extends Expression
final case class Number(value: Double) extends Expression
```

Return to the exercise

C.3.20 Solution to: A Calculator Part 2

I used pattern matching as it's more compact and I feel this makes the code easier to read.

```
sealed trait Expression {
  def eval: Double =
    this match {
      case Addition(l, r) => l.eval + r.eval
      case Subtraction(l, r) => l.eval - r.eval
      case Number(v) => v
    }
}
final case class Addition(left: Expression, right: Expression) extends Expression
final case class Subtraction(left: Expression, right: Expression) extends Expression
final case class Number(value: Int) extends Expression
```

Return to the exercise

C.3.21 Solution to: A Calculator Part 3

```
sealed trait Expression
final case class Addition(left: Expression, right: Expression) extends Expression
final case class Subtraction(left: Expression, right: Expression) extends Expression
final case class Division(left: Expression, right: Expression) extends Expression
final case class SquareRoot(value: Expression) extends Expression
final case class Number(value: Double) extends Expression
```

Return to the exercise

C.3.22 Solution to: A Calculator Part 4

We did this in the previous section.

```
sealed trait Calculation
final case class Success(result: Double) extends Calculation
final case class Failure(reason: String) extends Calculation
```

Return to the exercise

C.3.23 Solution to: A Calculator Part 5

All this repeated pattern matching gets very tedious, doesn't it! We're going to see how we can abstract this in the next section.

```
sealed trait Expression {
  def eval: Calculation =
    this match {
      case Addition(l, r) =>
          l.eval match {
            case Failure(reason) => Failure(reason)
            case Success(r1) =>
              r.eval match {
                case Failure(reason) => Failure(reason)
                case Success(r2) => Success(r1 + r2)
              }
          }
      case Subtraction(l, r) =>
          l.eval match {
            case Failure(reason) => Failure(reason)
            case Success(r1) =>
              r.eval match {
                case Failure(reason) => Failure(reason)
                case Success(r2) => Success(r1 - r2)
              }
          }
      case Division(l, r) =>
        l.eval match {
          case Failure(reason) => Failure(reason)
          case Success(r1) =>
            r.eval match {
              case Failure(reason) => Failure(reason)
              case Success(r2) =>
                if(r2 == 0)
                  Failure("Division by zero")
                else
                  Success(r1 / r2)
            }
        }
      case SquareRoot(v) =>
        v.eval match {
          case Success(r) =>
            if(r < 0)
              Failure("Square root of negative number")
            else
              Success(Math.sqrt(r))
          case Failure(reason) => Failure(reason)
        }
      case Number(v) => Success(v)
    }
}
final case class Addition(left: Expression, right: Expression) extends Expression
```

```
final case class Subtraction(left: Expression, right: Expression) extends Expression
final case class Division(left: Expression, right: Expression) extends Expression
final case class SquareRoot(value: Expression) extends Expression
final case class Number(value: Int) extends Expression
```

Return to the exercise

C.3.24 Solution to: JSON

There are many possible ways to model JSON. Here's one, which is a fairly direct translation of the railroad diagrams in the JSON spec.

```
Json ::= JsNumber value:Double
       | JsString value:String
       | JsBoolean value:Boolean
       | JsNull
       | JsSequence
       | JsObject
JsSequence ::= SeqCell head:Json tail:JsSequence
             | SeqEnd
JsObject ::= ObjectCell key:String value:Json tail:JsObject
           | ObjectEnd
```

Return to the exercise

C.3.25 Solution to: JSON Part 2

This should be a mechanical process. This is the point of algebraic data types—we do the work in modelling the data, and the code follows directly from that model.

```
sealed trait Json
final case class JsNumber(value: Double) extends Json
final case class JsString(value: String) extends Json
final case class JsBoolean(value: Boolean) extends Json
final case object JsNull extends Json
sealed trait JsSequence extends Json
final case class SeqCell(head: Json, tail: JsSequence) extends JsSequence
final case object SeqEnd extends JsSequence
sealed trait JsObject extends Json
final case class ObjectCell(key: String, value: Json, tail: JsObject) extends JsObject
final case object ObjectEnd extends JsObject
```

Return to the exercise

C.3.26 Solution to: JSON Part 3

This is an application of structural recursion, as all transformations on algebraic data types are, with the wrinkle that we have to treat the sequence types specially. Here is my solution.

```
sealed trait Json {
  def print: String = {
    def quote(s: String): String =
      '"'.toString ++ s ++ '"'.toString
    def seqToJson(seq: SeqCell): String =
      seq match {
        case SeqCell(h, t @ SeqCell(_, _)) =>
          s"${h.print}, ${seqToJson(t)}"
        case SeqCell(h, SeqEnd) => h.print
      }

    def objectToJson(obj: ObjectCell): String =
      obj match {
        case ObjectCell(k, v, t @ ObjectCell(_, _, _)) =>
          s"${quote(k)}: ${v.print}, ${objectToJson(t)}"
        case ObjectCell(k, v, ObjectEnd) =>
          s"${quote(k)}: ${v.print}"
      }

    this match {
      case JsNumber(v) => v.toString
      case JsString(v) => quote(v)
      case JsBoolean(v) => v.toString
      case JsNull => "null"
      case s @ SeqCell(_, _) => "[" ++ seqToJson(s) ++ "]"
      case SeqEnd => "[]"
      case o @ ObjectCell(_, _, _) => "{" ++ objectToJson(o) ++ "}"
      case ObjectEnd => "{}"
    }
  }
}
final case class JsNumber(value: Double) extends Json
final case class JsString(value: String) extends Json
final case class JsBoolean(value: Boolean) extends Json
final case object JsNull extends Json
sealed trait JsSequence extends Json
final case class SeqCell(head: Json, tail: JsSequence) extends JsSequence
final case object SeqEnd extends JsSequence
sealed trait JsObject extends Json
final case class ObjectCell(key: String, value: Json, tail: JsObject) extends JsObject
final case object ObjectEnd extends JsObject
```

Return to the exercise

C.3.27 Solution to: Music

My solution models a very simplified version of Western music. My fundamental "atom" is the note, which consists of a pitch and a duration.

```
Note ::= pitch:Pitch duration:Duration
```

I'm assuming I have a data for `Pitch` representing tones on the standard musical scale from `C0` (about 16Hz) to `C8`. Something like

```
Pitch ::= C0 | CSharp0 | D0 | DSharp0 | F0 | FSharp0 | ... | C8 | Rest
```

Note that I included `Rest` as a pitch, so I can model silence.

We already seem some limitations. I'm not modelling notes that fall outside the scale (microtones) or music systems that use other scales. Furthermore, in most tuning systems flats and their enharmonic sharps (e.g. C-sharp and D-flat) are not the same note, but I'm ignoring that distinction here.

We could break this representation down further into a tone

```
Tone ::= C | CSharp | D | DSharp | F | FSharp | ... | B
```

and an octave

```
Octave ::= 0 | 1 | 2 | ... | 8
```

and then

```
Pitch ::= tone:Tone octave:Octave
```

Durations are a mess in standard musical notation. There are a bunch of named durations (semitone, quaver, etc.) along with dots and tied notes to represent other durations. We can do better by simply saying our music has an atomic unit of time, which we'll call a beat, and each duration is zero or more beats.

```
Duration ::= 0 | 1 | 2 | ...
```

In other words, `Duration` is a natural number. In Scala we might model this with an `Int`, or create a type to represent the additional constraint we put over `Int`.

Again, this representation comes with limitations. Namely we can't represent music that doesn't fit cleanly into some division of time—so called free time music.

Finally we should get to means of composition of notes. There are two main ways: we can play notes in sequence or at the same time.

```
Phrase ::= Sequence | Parallek
Sequence ::= SeqCell phrase:Phrase tail:Sequence
           | SeqEnd

Parallel ::= ParCell phrase:Phrase tail:Parallel
           | ParEnd
```

This representation allows us to arbitrarily nest parallel and sequential units of notes. We might prefer a normalised representation, such as

```
Sequence ::= SeqCell note:Note tail:Sequence
           | SeqEnd

Parallel ::= ParCell sequence:Sequence tail:Parallel
           | ParEnd
```

There are many things missing from this model. Some of them include:

- We don't model musical dynamics in any way. Notes can be louder or softer, and volume can change while a note is being played. Notes do not always have constant pitch, either. Pitch bends or slurs are examples of changing pitches in a single note
- We haven't modelled different instruments at all.
- We haven't modelled effects, like echo and distortion, that make up an important part of modern music.

Return to the exercise

C.4 Sequencing Computations

C.4.1 Solution to: Generic List

This is an application of the generic sum type pattern.

```
sealed trait LinkedList[A]
final case class Pair[A](head: A, tail: LinkedList[A]) extends LinkedList[A]
final case class End[A]() extends LinkedList[A]
```

Return to the exercise

C.4.2 Solution to: Working With Generic Types

This code is largely unchanged from the implementation of `length` on `IntList`.

```
sealed trait LinkedList[A] {
  def length: Int =
    this match {
      case Pair(hd, tl) => 1 + tl.length
      case End() => 0
    }
}
final case class Pair[A](head: A, tail: LinkedList[A]) extends LinkedList[A]
final case class End[A]() extends LinkedList[A]
```

Return to the exercise

C.4.3 Solution to: Working With Generic Types Part 2

This is another example of the standard structural recursion pattern. The important point is `contains` takes a parameter of type `A`.

```
sealed trait LinkedList[A] {
  def contains(item: A): Boolean =
    this match {
      case Pair(hd, tl) =>
        if(hd == item)
          true
        else
          tl.contains(item)
      case End() => false
    }
```

```
}

final case class Pair[A](head: A, tail: LinkedList[A]) extends LinkedList[A]
final case class End[A]() extends LinkedList[A]
```

Return to the exercise

C.4.4 Solution to: Working With Generic Types Part 3

There are a few interesting things in this exercise. Possibly the easiest part is the use of the generic type as the return type of the `apply` method.

Next up is the End case, which the hint suggested you through an `Exception` for. Strictly speaking we should throw Java's `IndexOutOfBoundsException` in this instance, but we will shortly see a way to remove exception handling from our code altogether.

Finally we get to the actual structural recursion, which is perhaps the trickiest part. The key insight is that if the index is zero, we're selecting the current element, otherwise we subtract one from the index and recurse. We can recursively define the integers in terms of addition by one. For example, 3 = 2 + 1 = 1 + 1 + 1. Here we are performing structural recursion on the list *and* on the integers.

```
sealed trait LinkedList[A] {
  def apply(index: Int): A =
    this match {
      case Pair(hd, tl) =>
        if(index == 0)
          hd
        else
          tl(index - 1)
      case End() =>
        throw new Exception("Attempted to get element from an Empty list")
    }
}
final case class Pair[A](head: A, tail: LinkedList[A]) extends LinkedList[A]
final case class End[A]() extends LinkedList[A]
```

Return to the exercise

C.4.5 Solution to: Working With Generic Types Part 4

```
sealed trait Result[A]
case class Success[A](result: A) extends Result[A]
case class Failure[A](reason: String) extends Result[A]

sealed trait LinkedList[A] {
  def apply(index: Int): Result[A] =
    this match {
      case Pair(hd, tl) =>
        if(index == 0)
          Success(hd)
        else
          tl(index - 1)
      case End() =>
        Failure("Index out of bounds")
    }
}
```

```
final case class Pair[A](head: A, tail: LinkedList[A]) extends LinkedList[A]
final case class End[A]() extends LinkedList[A]
```

Return to the exercise

C.4.6 Solution to: A Better Abstraction

```
def fold(end: Int, f: (Int, Int) => Int): Int =
  this match {
    case End => end
    case Pair(hd, tl) => f(hd, tl.fold(f, end))
  }
```

Return to the exercise

C.4.7 Solution to: A Better Abstraction Part 2

```
sealed trait IntList {
  def fold(end: Int, f: (Int, Int) => Int): Int =
    this match {
      case End => end
      case Pair(hd, tl) => f(hd, tl.fold(end, f))
    }
  def length: Int =
    fold(0, (_, tl) => 1 + tl)
  def product: Int =
    fold(1, (hd, tl) => hd * tl)
  def sum: Int =
    fold(0, (hd, tl) => hd + tl)
}
final case object End extends IntList
final case class Pair(head: Int, tail: IntList) extends IntList
```

Return to the exercise

C.4.8 Solution to: A Better Abstraction Part 3

When using `fold` in polymorphic implementations we have a lot of duplication; the polymorphic implementations without `fold` were simpler to write. The pattern matching implementations benefited from `fold` as we remove the duplication in the pattern matching.

In general `fold` makes a good interface for users *outside* the class, but not necessarily for use *inside* the class.

Return to the exercise

C.4.9 Solution to: A Better Abstraction Part 4

The types tell us it won't work. `fold` returns an `Int` and `double` returns an `IntList`. However the general structure of `double` is captured by `fold`. This is apparent if we look at them side-by-side:

```
def double: IntList =
  this match {
    case End => End
    case Pair(hd, tl) => Pair(hd * 2, tl.double)
```

```
  }

def fold(end: Int, f: (Int, Int) => Int): Int =
  this match {
    case End => end
    case Pair(hd, tl) => f(hd, tl.fold(end, f))
  }
```

If we could generalise the types of `fold` from `Int` to some general type then we could write `double`. And that, dear reader, is what we turn to next.

Return to the exercise

C.4.10 Solution to: A Better Abstraction Part 5

We want to generalise the return type of `fold`. Our starting point is

```
def fold(end: Int, f: (Int, Int) => Int): Int
```

Replacing the return type and tracing it back we arrive at

```
def fold[A](list: IntList, f: (Int, A) => A, end: A): A
```

where we've used a generic type on the method to capture the changing return type. With this we can implement `double`. When we try to do so we'll see that type inference fails, so we have to give it a bit of help.

```
sealed trait IntList {
  def fold[A](end: A, f: (Int, A) => A): A =
    this match {
      case End => end
      case Pair(hd, tl) => f(hd, tl.fold(end, f))
    }
  def length: Int =
    fold[Int](0, (_, tl) => 1 + tl)
  def product: Int =
    fold[Int](1, (hd, tl) => hd * tl)
  def sum: Int =
    fold[Int](0, (hd, tl) => hd + tl)
  def double: IntList =
    fold[IntList](End, (hd, tl) => Pair(hd * 2, tl))
}
final case object End extends IntList
final case class Pair(head: Int, tail: IntList) extends IntList
```

Return to the exercise

C.4.11 Solution to: Tree

This is another recursive data type just like list. Follow the patterns and you should be ok.

```
sealed trait Tree[A] {
  def fold[B](node: (B, B) => B, leaf: A => B): B
}
final case class Node[A](left: Tree[A], right: Tree[A]) extends Tree[A] {
  def fold[B](node: (B, B) => B, leaf: A => B): B =
    node(left.fold(node, leaf), right.fold(node, leaf))
}
```

```
final case class Leaf[A](value: A) extends Tree[A] {
  def fold[B](node: (B, B) => B, leaf: A => B): B =
    leaf(value)
}
```

Return to the exercise

C.4.12 Solution to: Tree Part 2

Note it is necessary to instantiate the generic type variable for `fold`. Type inference fails in this case.

```
tree.fold[String]((a, b) => a + " " + b, str => str)
```

Return to the exercise

C.4.13 Solution to: Pairs

If one type parameter is good, two type parameters are better:

```
case class Pair[A, B](one: A, two: B)
```

This is just the product type pattern we have seen before, but we introduce generic types.

Note that we don't always need to specify the type parameters when we construct `Pairs`. The compiler will attempt to infer the types as usual wherever it can:

```
val pair = Pair("hi", 2)
// pair: Pair[String,Int] = Pair(hi,2)
```

Return to the exercise

C.4.14 Solution to: Generic Sum Type

The code is an adaptation of our invariant generic sum type pattern, with another type parameter:

```
sealed trait Sum[A, B]
final case class Left[A, B](value: A) extends Sum[A, B]
final case class Right[A, B](value: B) extends Sum[A, B]
```

Scala's standard library has the generic sum type `Either` for two cases, but it does not have types for more cases.

Return to the exercise

C.4.15 Solution to: Maybe that Was a Mistake

We can apply our invariant generic sum type pattern and get

```
sealed trait Maybe[A]
final case class Full[A](value: A) extends Maybe[A]
final case class Empty[A]() extends Maybe[A]
```

Return to the exercise

C.4.16 Solution to: Generics versus Traits

Ultimately the decision is up to us. Different teams will adopt different programming styles. However, we look at the properties of each approach to inform our choices:

Inheritance-based approaches—traits and classes—allow us to create permanent data structures with specific types and names. We can name every field and method and implement use-case-specific code in each class. Inheritance is therefore better suited to modelling significant aspects of our programs that are re-used in many areas of our codebase.

Generic data structures—`Tuples`, `Options`, `Eithers`, and so on—are extremely broad and general purpose. There are a wide range of predefined classes in the Scala standard library that we can use to quickly model relationships between data in our code. These classes are therefore better suited to quick, one-off pieces of data manipulation where defining our own types would introduce unnecessary verbosity to our codebase.

Return to the exercise

C.4.17 Solution to: Folding Maybe

The code is very similar to the implementation for `LinkedList`. I choose pattern matching in the base trait for my solution.

```
sealed trait Maybe[A] {
  def fold[B](full: A => B, empty: B): B =
    this match {
      case Full(v) => full(v)
      case Empty() => empty
    }
}
final case class Full[A](value: A) extends Maybe[A]
final case class Empty[A]() extends Maybe[A]
```

Return to the exercise

C.4.18 Solution to: Folding Sum

```
sealed trait Sum[A, B] {
  def fold[C](left: A => C, right: B => C): C =
    this match {
      case Left(a) => left(a)
      case Right(b) => right(b)
    }
}
final case class Left[A, B](value: A) extends Sum[A, B]
final case class Right[A, B](value: B) extends Sum[A, B]
```

Return to the exercise

C.4.19 Solution to: Mapping Lists

These exercises just get you used to using `map`.

```
list.map(_ * 2)
list.map(_ + 1)
list.map(_ / 3)
```

Return to the exercise

C.4.20 Solution to: Mapping Maybe

```
sealed trait Maybe[A] {
  def flatMap[B](fn: A => Maybe[B]): Maybe[B] =
    this match {
      case Full(v) => fn(v)
      case Empty() => Empty[B]()
    }
  def map[B](fn: A => B): Maybe[B] =
    this match {
      case Full(v) => Full(fn(v))
      case Empty() => Empty[B]()
    }
}
final case class Full[A](value: A) extends Maybe[A]
final case class Empty[A]() extends Maybe[A]
```

Return to the exercise

C.4.21 Solution to: Mapping Maybe Part 2

```
sealed trait Maybe[A] {
  def flatMap[B](fn: A => Maybe[B]): Maybe[B] =
    this match {
      case Full(v) => fn(v)
      case Empty() => Empty[B]()
    }
  def map[B](fn: A => B): Maybe[B] =
    flatMap[B](v => Full(fn(v)))
}
final case class Full[A](value: A) extends Maybe[A]
final case class Empty[A]() extends Maybe[A]
```

Return to the exercise

C.4.22 Solution to: Sequencing Computations

```
list.flatMap(x => List(x, -x))
```

Return to the exercise

C.4.23 Solution to: Sequencing Computations Part 2

```
list.map(maybe => maybe flatMap { x => if(x % 2 == 0) Full(x) else Empty })
```

Return to the exercise

C.4.24 Solution to: Sum

```
sealed trait Sum[A, B] {
  def fold[C](error: A => C, success: B => C): C =
    this match {
      case Failure(v) => error(v)
      case Success(v) => success(v)
    }
}
final case class Failure[A, B](value: A) extends Sum[A, B]
final case class Success[A, B](value: B) extends Sum[A, B]
```

Return to the exercise

C.4.25 Solution to: Sum Part 2

```
sealed trait Sum[+A, +B] {
  def fold[C](error: A => C, success: B => C): C =
    this match {
      case Failure(v) => error(v)
      case Success(v) => success(v)
    }
  def map[C](f: B => C): Sum[A, C] =
    this match {
      case Failure(v) => Failure(v)
      case Success(v) => Success(f(v))
    }
}
```

Return to the exercise

C.4.26 Solution to: Sum Part 3

```
sealed trait Sum[A, B] {
  def fold[C](error: A => C, success: B => C): C =
    this match {
      case Failure(v) => error(v)
      case Success(v) => success(v)
    }
  def map[C](f: B => C): Sum[A, C] =
    this match {
      case Failure(v) => Failure(v)
      case Success(v) => Success(f(v))
    }
  def flatMap[C](f: B => Sum[A, C]) =
    this match {
      case Failure(v) => Failure(v)
      case Success(v) => f(v)
    }
}
```

```
final case class Failure[A, B](value: A) extends Sum[A, B]
final case class Success[A, B](value: B) extends Sum[A, B]
```

Return to the exercise

C.4.27 Solution to: Covariant Sum

```
sealed trait Sum[+A, +B]
final case class Failure[A](value: A) extends Sum[A, Nothing]
final case class Success[B](value: B) extends Sum[Nothing, B]
```

Return to the exercise

C.4.28 Solution to: Some sort of flatMap

```
sealed trait Sum[+A, +B] {
  def flatMap[C](f: B => Sum[A, C]): Sum[A, C] =
    this match {
      case Failure(v) => Failure(v)
      case Success(v) => f(v)
    }
}
final case class Failure[A](value: A) extends Sum[A, Nothing]
final case class Success[B](value: B) extends Sum[Nothing, B]
```

Return to the exercise

C.4.29 Solution to: Covariance and Contravariance

The only function that will work is the the function of type `Animal => Purr`. The `Siamese => Purr` function will not work because the Oswald is a not a Siamese cat. The `Animal => Sound` function will not work because we require the return type is a `CatSound`.

Return to the exercise

C.4.30 Solution to: Calculator Again

```
sealed trait Sum[+A, +B] {
  def fold[C](error: A => C, success: B => C): C =
    this match {
      case Failure(v) => error(v)
      case Success(v) => success(v)
    }
  def map[C](f: B => C): Sum[A, C] =
    this match {
      case Failure(v) => Failure(v)
      case Success(v) => Success(f(v))
    }
  def flatMap[AA >: A, C](f: B => Sum[AA, C]): Sum[AA, C] =
    this match {
      case Failure(v) => Failure(v)
      case Success(v) => f(v)
    }
}
```

```
final case class Failure[A](value: A) extends Sum[A, Nothing]
final case class Success[B](value: B) extends Sum[Nothing, B]
```

Return to the exercise

C.4.31 Solution to: Calculator Again Part 2

Here's my solution. I used a helper method `lift2` to "lift" a function into the result of two expressions. I hope you'll agree the code is both more compact and easier to read than our previous solution!

```
sealed trait Expression {
  def eval: Sum[String, Double] =
    this match {
      case Addition(l, r) => lift2(l, r, (left, right) => Success(left + right))
      case Subtraction(l, r) => lift2(l, r, (left, right) => Success(left - right))
      case Division(l, r) => lift2(l, r, (left, right) =>
        if(right == 0)
          Failure("Division by zero")
        else
          Success(left / right)
      )
      case SquareRoot(v) =>
        v.eval flatMap { value =>
          if(value < 0)
            Failure("Square root of negative number")
          else
            Success(Math.sqrt(value))
        }
      case Number(v) => Success(v)
    }

  def lift2(l: Expression, r: Expression, f: (Double, Double) => Sum[String, Double]) =
    l.eval flatMap { left =>
      r.eval flatMap { right =>
        f(left, right)
      }
    }
}
final case class Addition(left: Expression, right: Expression) extends Expression
final case class Subtraction(left: Expression, right: Expression) extends Expression
final case class Division(left: Expression, right: Expression) extends Expression
final case class SquareRoot(value: Expression) extends Expression
final case class Number(value: Int) extends Expression
```

Return to the exercise

C.5 Collections

C.5.1 Solution to: Documentation

The synonym for `length` is `size`.

The methods for retrieving the first element in a list are: - `head` —returns `A`, throwing an exception if the list is empty - `headOption`—returns `Option[A]`, returning `None` if the list is empty

The `mkString` method allows us to quickly display a `Seq` as a `String`:

```
Seq(1, 2, 3).mkString(",")              // returns "1,2,3"
Seq(1, 2, 3).mkString("[ ", ", ", " ]"") // returns "[ 1, 2, 3 ]"
```

`Options` contain two methods, `isDefined` and `isEmpty`, that we can use as a quick test:

```
Some(123).isDefined // returns true
Some(123).isEMpty   // returns false
None.isDefined      // returns false
None.isEMpty        // returns true
```

Return to the exercise

C.5.2 Solution to: Animals

```
val animals = Seq("cat", "dog", "penguin")
animals: Seq[String] = List(cat, dog, penguin)
```

Return to the exercise

C.5.3 Solution to: Animals Part 2

```
"mouse" +: animals :+ "tyrannosaurus"
// res: Seq[String] = List(mouse, cat, dog, penguin, tyrannosaurus)
```

Return to the exercise

C.5.4 Solution to: Animals Part 3

The returned sequence has type `Seq[Any]`. It is perfectly valid to return a supertype (in this case `Seq[Any]`) from a non-destructive operation.

```
2 +: animals
// res: Seq[Any] = List(2, cat, dog, penguin)
```

You might expect a type error here, but Scala is capable of determining the least upper bound of `String` and `Int` and setting the type of the returned sequence accordingly.

In most real code appending an `Int` to a `Seq[String]` would be an error. In practice, the type annotations we place on methods and fields protect against this kind of type error, but be aware of this behaviour just in case.

Return to the exercise

C.5.5 Solution to: Intranet Movie Database

We use `filter` because we are expecting more than one result:

```
def directorsWithBackCatalogOfSize(numberOfFilms: Int): Seq[Director] =
 directors.filter(_.films.length > numberOfFilms)
```

Return to the exercise

C.5.6 Solution to: Intranet Movie Database Part 2

We use `find` because we are expecting at most one result. This solution will return the first director found who matches the criteria of the search:

```
def directorBornBefore(year: Int): Option[Director] =
 directors.find(_.yearOfBirth < year)
```

The `Option` type is discussed in more detail later this chapter.

Return to the exercise

C.5.7 Solution to: Intranet Movie Database Part 3

This solution performs each part of the query separately and uses `filter` and `contains` to calculate the intersection of the results:

```
def directorBornBeforeWithBackCatalogOfSize(year: Int, numberOfFilms: Int): Seq[Director] = {
 val byAge   = directors.filter(_.yearOfBirth < year)
 val byFilms = directors.filter(_.films.length > numberOfFilms)
 byAge.filter(byFilms.contains)
}
```

Return to the exercise

C.5.8 Solution to: Intranet Movie Database Part 4

Here is one solution. Note that sorting by ascending age is the same as sorting by descending year of birth:

```
def directorsSortedByAge(ascending: Boolean = true) =
 if(ascending) {
   directors.sortWith((a, b) => a.yearOfBirth < b.yearOfBirth)
 } else {
   directors.sortWith((a, b) => a.yearOfBirth > b.yearOfBirth)
 }
```

Because Scala is a functional language, we can also factor our code as follows:

```
def directorsSortedByAge(ascending: Boolean = true) = {
 val comparator =
   if(ascending) {
     (a, b) => a.yearOfBirth < b.yearOfBirth
   } else {
     (a, b) => a.yearOfBirth > b.yearOfBirth
   }

 directors.sortWith(comparator)
}
```

Here is a final refactoring that is slightly less efficient because it rechecks the value of `ascending` multiple times.

```
def directorsSortedByAge(ascending: Boolean = true) =
 directors.sortWith { (a, b) =>
   if(ascending) {
     a.yearOfBirth < b.yearOfBirth
   } else {
     a.yearOfBirth > b.yearOfBirth
   }
 }
```

Note the use of braces instead of parentheses on the call to `sortWith` in the last example. We can use this syntax on any method call of one argument to give it a control-structure-like look and feel.

Return to the exercise

C.5.9 Solution to: Heroes of the Silver Screen

```
nolan.films.map(_.name)
```

Return to the exercise

C.5.10 Solution to: Heroes of the Silver Screen Part 2

```
directors.flatMap(director => director.films.map(film => film.name))
```

Return to the exercise

C.5.11 Solution to: Heroes of the Silver Screen Part 3

There are a number of ways to do this. We can sort the list of films and then retrieve the smallest element.

```
mcTiernan.films.sortWith { (a, b) =>
  a.yearOfRelease < b.yearOfRelease
}.headOption
```

We can also do this by using a `fold`.

```
mcTiernan.films.foldLeft(Int.MaxValue) { (current, film) =>
  math.min(current, film.yearOfRelease)
}
```

Return to the exercise

C.5.12 Solution to: Heroes of the Silver Screen Part 4

```
directors.
  flatMap(director => director.films).
  sortWith((a, b) => a.imdbRating > b.imdbRating)
```

Return to the exercise

C.5.13 Solution to: Heroes of the Silver Screen Part 5

We cache the list of films in a variable because we use it twice—once to calculate the sum of the ratings and once to fetch the number of films:

```
val films = directors.flatMap(director => director.films)

films.foldLeft(0)((a, b) => a.imdbRating + b.imdbRating) / films.length
```

Return to the exercise

C.5.14 Solution to: Heroes of the Silver Screen Part 6

Println is used for its side-effects so we don't need to accumulate a result—we use `println` as a simple iterator:

```
directors.foreach { director =>
  director.films.foreach { film =>
    println(s"Tonight! ${film.name} by ${director.name}!")
  }
}
```

Return to the exercise

C.5.15 Solution to: Heroes of the Silver Screen Part 7

Here's the solution:

```
directors
  .flatMap(director => director.films)
  .sortWith((a, b) => a.yearOfRelease < b.yearOfRelease)
  .headOption
```

Return to the exercise

C.5.16 Solution to: Do-It-Yourself

This is another fold. We have a `Seq[Int]`, the minimum operation is `(Int, Int) => Int`, and we want an `Int`. The challenge is to find the zero value.

What is the identity for `min` so that `min(x, identity) = x`. It is positive infinity, which in Scala we can write as `Int.MaxValue` (see, fixed width numbers do have benefits).

Thus the solution is:

```
def smallest(seq: Seq[Int]): Int =
  seq.foldLeft(Int.MaxValue)(math.min)
```

Return to the exercise

C.5.17 Solution to: Do-It-Yourself Part 2

Once again we follow the same pattern. The types are:

1. We have a `Seq[Int]`
2. We want a `Seq[Int]`
3. Constructing the operation we want to use requires a bit more thought. The hint is to use `contains`. We can keep a sequence of the unique elements we've seen so far, and use `contains` to test if the sequence contains the current element. If we have seen the element we don't add it, otherwise we do. In code

```
def insert(seq: Seq[Int], elt: Int): Seq[Int] = {
 if(seq.contains(elt))
   seq
 else
   elt +: seq
}
```

We these three pieces we can solve the problem. Looking at the type table we see we want a `fold`. Once again we must find the identity element. In this case the empty sequence is what we want. Why so? Think about what the answer should be if we try to find the unique elements of the empty sequence.

Thus the solution is

```
def insert(seq: Seq[Int], elt: Int): Seq[Int] = {
  if(seq.contains(elt))
    seq
  else
    elt +: seq
}

def unique(seq: Seq[Int]): Seq[Int] = {
  seq.foldLeft(Seq.empty[Int]){ insert _ }
}

unique(Seq(1, 1, 2, 4, 3, 4))
```

Note how I created the empty sequence. I could have written `Seq[Int]()` but in both cases I need to supply a type (`Int`) to help the type inference along.

Return to the exercise

C.5.18 Solution to: Do-It-Yourself Part 3

In this exercise, and the ones that follow, using the types are particularly important. Start by writing down the type of `reverse`.

```
def reverse[A, B](seq: Seq[A], f: A => B): Seq[B] = {
  ???
}
```

The hint says to use `foldLeft`, so let's go ahead and fill in the body as far as we can.

```
def reverse[A](seq: Seq[A]): Seq[A] = {
  seq.foldLeft(???){ ??? }
}
```

We need to work out the function to provide to `foldLeft` and the zero or identity element. For the function, the type of `foldLeft` requires it is `(Seq[A], A) => Seq[A]`. If we flip the types around the `+:` method on `Seq` has the right types.

For the zero element we know that it must have the same type as the return type of `reverse` (because the result of the fold is the result of `reverse`). Thus it's a `Seq[A]`. Which sequence? There are a few ways to answer this:

- The only `Seq[A]` we can create in this method, before we know what `A` is, is the empty sequence `Seq.empty[A]`.
- The identity element is one such that `x +: zero = Seq(x)`. Again this must be the empty sequence.

So we now we can fill in the answer.

```
def reverse[A](seq: Seq[A]): Seq[A] = {
  seq.foldLeft(Seq.empty[A]){ (seq, elt) => elt +: seq }
}
```

Return to the exercise

C.5.19 Solution to: Do-It-Yourself Part 4

Follow the same process as before: write out the type of the method we need to create, and fill in what we know. We start with `map` and `foldRight`.

```
def map[A, B](seq: Seq[A], f: A => B): Seq[B] = {
  seq.foldRight(???){ ??? }
}
```

As usual we need to fill in the zero element and the function. The zero element must have type `Seq[B]`, and the function has type `(A, Seq[B]) => Seq[B])`. The zero element is straightforward: `Seq.empty[B]` is the only sequence we can construct of type `Seq[B]`. For the function, we clearly have to convert that `A` to a `B` somehow. There is only one way to do that, which is with the function supplied to `map`. We then need to add that `B` to our `Seq[B]`, for which we can use the `+:` method. This gives us our final result.

```
def map[A, B](seq: Seq[A], f: A => B): Seq[B] = {
  seq.foldRight(Seq.empty[B]){ (elt, seq) => f(elt) +: seq }
}
```

Return to the exercise

C.5.20 Solution to: Do-It-Yourself Part 5

Once again, write out the skeleton and then fill in the details using the types. We start with

```
def foldLeft[A, B](seq: Seq[A], zero: B, f: (B, A) => B): B = {
  seq.foreach { ??? }
}
```

Let's look at what we have need to fill in. `foreach` returns `Unit` but we need to return a `B`. `foreach` takes a function of type `A => Unit` but we only have a `(B, A) => B` available. The `A` can come from `foreach` and by now we know that the `B` is the intermediate result. We have the hint to use mutable state and we know that we need to keep a `B` around and return it, so let's fill that in.

```
def foldLeft[A, B](seq: Seq[A], zero: B, f: (B, A) => B): B = {
  var result: B = ???
  seq.foreach { (elt: A) => ??? }
  result
}
```

At this point we can just follow the types. `result` must be initially assigned to the value of `zero` as that is the only `B` we have. The body of the function we pass to `foreach` must call `f` with `result` and `elt`. This returns a `B` which we must store somewhere—the only place we have to store it is in `result`. So the final answer becomes

```
def foldLeft[A, B](seq: Seq[A], zero: B, f: (B, A) => B): B = {
  var result = zero
  seq.foreach { elt => result = f(result, elt) }
  result
}
```

Return to the exercise

C.5.21 Solution to: Exercises

```
for {
  film <- nolan.films
} yield film.name
```

Return to the exercise

C.5.22 Solution to: Exercises Part 2

```
for {
  director <- directors
  film     <- director.films
} yield film.name
```

Return to the exercise

C.5.23 Solution to: Exercises Part 3

This one's a little trickier. We have to calculate the complete list of films first before sorting them with `sortWith`. Precedence rules require us to wrap the whole `for / yield` expression in parentheses to achieve this in one expression:

```
(for {
  director <- directors
  film     <- director.films
} yield film).sortWith((a, b) => a.imdbRating > b.imdbRating)
```

Many developers prefer to use a temporary variable to make this code tidier:

```
val films = for {
  director <- directors
  film     <- director.films
} yield film

films sortWith { (a, b) =>
  a.imdbRating > b.imdbRating
}
```

Return to the exercise

C.5.24 Solution to: Exercises Part 4

We can drop the `yield` keyword from the `for` expression to achieve `foreach`-like semantics:

```
for {
  director <- directors
  film     <- director.films
} println(s"Tonight! ${film.name} by ${director.name}!")
```

Return to the exercise

C.5.25 Solution to: Adding Things

We can reuse code from the text above for this:

```
def addOptions(opt1: Option[Int], opt2: Option[Int]) =
  for {
    a <- opt1
    b <- opt2
  } yield a + b
```

Return to the exercise

C.5.26 Solution to: Adding Things Part 2

The pattern is to use `flatMap` for all clauses except the innermost, which becomes a `map`:

```
def addOptions2(opt1: Option[Int], opt2: Option[Int]) =
  opt1 flatMap { a =>
    opt2 map { b =>
      a + b
    }
  }
```

Return to the exercise

C.5.27 Solution to: Adding All of the Things

For comprehensions can have as many clauses as we want so all we need to do is add an extra line to the previous solution:

```
def addOptions(opt1: Option[Int], opt2: Option[Int], opt3: Option[Int]) =
  for {
    a <- opt1
    b <- opt2
    c <- opt3
  } yield a + b + c
```

Return to the exercise

C.5.28 Solution to: Adding All of the Things Part 2

Here we can start to see the simplicity of for comprehensions:

```
def addOptions2(opt1: Option[Int], opt2: Option[Int], opt3: Option[Int]) =
  opt1 flatMap { a =>
    opt2 flatMap { b =>
      opt3 map { c =>
        a + b + c
      }
    }
  }
```

Return to the exercise

C.5.29 Solution to: A(nother) Short Division Exercise

We saw this code in the Traits chapter when we wrote the `DivisionResult` class. The implementation is much simpler now we can use `Option` to do the heavy lifting:

```
def divide(numerator: Int, denominator: Int) =
  if(denominator < 1) None else Some(numerator / denominator)
```

Return to the exercise

C.5.30 Solution to: A(nother) Short Division Exercise Part 2

In this example the `divide` operation returns an `Option[Int]` instead of an `Int`. In order to process the result we need to move the calculation from the `yield` block to a for-clause:

```
def divideOptions(numerator: Option[Int], denominator: Option[Int]) =
  for {
    a <- numerator
    b <- denominator
    c <- divide(a, b)
  } yield c
```

Return to the exercise

C.5.31 Solution to: A Simple Calculator

The trick to this one is realising that each clause in the *for* comprehension can contain an entire block of Scala code:

```
def calculator(operand1: String, operator: String, operand2: String): Unit = {
  val result = for {
    a   <- readInt(operand1)
    b   <- readInt(operand2)
    ans <- operator match {
             case "+" => Some(a + b)
             case "-" => Some(a - b)
             case "*" => Some(a * b)
             case "/" => divide(a, b)
             case _   => None
           }
  } yield ans

  ans match {
    case Some(number) => println(s"The answer is $number!")
    case None         => println(s"Error calculating $operand1 $operator $operand2")
  }
}
```

Another approach involves factoring the calculation part out into its own private function:

```
def calculator(operand1: String, operator: String, operand2: String): Unit = {
  def calcInternal(a: Int, b: Int) =
    operator match {
      case "+" => Some(a + b)
      case "-" => Some(a - b)
      case "*" => Some(a * b)
      case "/" => divide(a, b)
      case _   => None
    }

  val result = for {
    a   <- readInt(operand1)
    b   <- readInt(operand2)
    ans <- calcInternal(a, b)
  } yield ans

  ans match {
    case Some(number) => println(s"The answer is $number!")
    case None         => println(s"Error calculating $operand1 $operator $operand2")
  }
}
```

Return to the exercise

C.5.32 Solution to: A Simple Calculator Part 2

This version of the code is much clearer if we factor out the calculation part into its own function. Without this it would be very hard to read:

```
def calculator(operand1: String, operator: String, operand2: String): Unit = {
  def calcInternal(a: Int, b: Int) =
    operator match {
      case "+" => Some(a + b)
      case "-" => Some(a - b)
      case "*" => Some(a * b)
      case "/" => divide(a, b)
      case _   => None
    }

  val result =
    readInt(operand1) flatMap { a =>
      readInt(operand2) flatMap { b =>
        calcInternal(a, b) map { result =>
          result
        }
      }
    }

  result match {
    case Some(number) => println(s"The answer is $number!")
    case None         => println(s"Error calculating $operand1 $operator $operand2")
  }
}
```

Return to the exercise

C.5.33 Solution to: Adding All the Things ++

```
for {
 x <- opt1
 y <- opt2
 z <- opt3
} yield x + y + z

for {
 x <- seq1
 y <- seq2
 z <- seq3
} yield x + y + z

for {
 x <- try1
 y <- try2
 z <- try3
} yield x + y + z
```

How's that for a cut-and-paste job?

Return to the exercise

C.5.34 Solution to: Favorites

The person may or may not be a key in the `favoriteColors` map so the function should return an `Option` result:

```
def favoriteColor(person: String): Option[String] =
  favoriteColours.get(person)
```

Return to the exercise

C.5.35 Solution to: Favorites Part 2

Now we have a default value we can return a `String` instead of an `Option[String]`:

```
def favoriteColor(person: String): String =
  favoriteColours.get(person).getOrElse("beige")
```

Return to the exercise

C.5.36 Solution to: Favorites Part 3

We can write this one using `foreach` or a for comprehension:

```
def printColors = for {
  person <- people
} println(s"${person}'s favorite color is ${favoriteColor(person)}!")
```

or:

```
def printColors = people foreach { person =>
  println(s"${person}'s favorite color is ${favoriteColor(person)}!")
}
```

Return to the exercise

C.5.37 Solution to: Favorites Part 4

Here we write a generic method using a type parameter:

```
def lookup[A](name: String, values: Map[String, A]) =
  values get name
```

Return to the exercise

C.5.38 Solution to: Favorites Part 5

First we find the oldest person, then we look up the answer:

```
val oldest: Option[String] =
  people.foldLeft(Option.empty[String]) { (older, person) =>
    if(ages.getOrElse(person, 0) > older.flatMap(ages.get).getOrElse(0)) {
      Some(person)
    } else {
      older
    }
  }

val favorite: Option[String] =
  for {
```

```
    oldest <- oldest
    color  <- favoriteColors.get(oldest)
  } yield color
```

Return to the exercise

C.5.39 Solution to: Union of Sets

As always, start by writing out the types and then follow the types to fill-in the details.

```
def union[A](set1: Set[A], set2: Set[A]): Set[A] = {
  ???
}
```

We need to think of an algorithm for computing the union. We can start with one of the sets and add the elements from the other set to it. The result will be the union. What types does this result in? Our result has type `Set[A]` and we need to add every `A` from the two sets to our result, which is an operation with type `(Set[A], A) => Set[A]`. This means we need a fold. Since order is not important any fold will do.

```
def union[A](set1: Set[A], set2: Set[A]): Set[A] = {
  set1.foldLeft(set2){ (set, elt) => (set + elt) }
}
```

Return to the exercise

C.5.40 Solution to: Union of Maps

The solution follows the same pattern as the union for sets, but here we have to handle adding the values as well.

```
def union[A](map1: Map[A, Int], map2: Map[A, Int]): Map[A, Int] = {
  map1.foldLeft(map2){ (map, elt) =>
    val (k, v) = elt
    val newV = map.getOrElse(k, v)
    map + (k -> newV)
  }
}
```

Return to the exercise

C.5.41 Solution to: Generic Union

With the tools we've seen far, we could add another function parameter like so:

```
def union[A, B](map1: Map[A, B], map2: Map[A, B], add: (B, B) => B): Map[A, B] = {
  map1.foldLeft(map2){ (map, elt) =>
    val (k, v) = elt
    val newV = map.get(k).map(v2 => add(v, v2)).getOrElse(v)
    map + (k -> newV)
  }
}
```

Later we'll see a nicer way to do this using type classes.

Return to the exercise

C.5.42 Solution to: Random Words

The following code will compute all possible sentences. The equivalent with explicit `flatMap` and `map` would also work.

Note that `flatMap` has more power than we need for this example. We could use the `subject` to alter how we choose the `verb`, for example. We'll use this ability in the next exercise.

```
val subjects = List("Noel", "The cat", "The dog")
val verbs = List("wrote", "chased", "slept on")
val objects = List("the book", "the ball", "the bed")

def allSentences: List[(String, String, String)] =
  for {
    subject <- subjects
    verb <- verbs
    obj <- objects
  } yield (subject, verb, obj)
```

Return to the exercise

C.5.43 Solution to: Random Words Part 2

We're now using the full power of `flatMap` to make decisions in our code that are dependent on what has happened before.

```
def verbsFor(subject: String): List[String] =
  subject match {
    case "Noel" => List("wrote", "chased", "slept on")
    case "The cat" => List("meowed at", "chased", "slept on")
    case "The dog" => List("barked at", "chased", "slept on")
  }

def objectsFor(verb: String): List[String] =
  verb match {
    case "wrote" => List("the book", "the letter", "the code")
    case "chased" => List("the ball", "the dog", "the cat")
    case "slept on" => List("the bed", "the mat", "the train")
    case "meowed at" => List("Noel", "the door", "the food cupboard")
    case "barked at" => List("the postman", "the car", "the cat")
  }

def allSentencesConditional: List[(String, String, String)] =
  for {
    subject <- subjects
    verb <- verbsFor(subject)
    obj <- objectsFor(verb)
  } yield (subject, verb, obj)
```

Return to the exercise

C.5.44 Solution to: Probabilities

There are no subtypes involved here, so a simple `final case class` will do. We wrap the `List[(A, Double)]` within a class so we can encapsulate manipulating the probabilities—external code can view the probabilities but probably shouldn't be directly working with them.

```
final case class Distribution[A](events: List[(A, Double)])
```

Return to the exercise

C.5.45 Solution to: Probabilities Part 2

As per Scala convention, convenience constructors should live on the companion object.

```
object Distribution {
  def uniform[A](atoms: List[A]): Distribution[A] = {
    val p = 1.0 / atoms.length
    Distribution(atoms.map(a => a -> p))
  }
}
```

Return to the exercise

C.5.46 Solution to: Probabilities Part 3

We need `flatMap` and `map`. The signatures follow the patterns that `flatMap` and `map` always have:

`def flatMap[B](f: A => Distribution[B]): Distribution[B]`

and

`def map[B](f: A => B): Distribution[B]`

Return to the exercise

C.5.47 Solution to: Probabilities Part 4

Implementing `map` merely requires we follow the types.

```
final case class Distribution[A](events: List[(A, Double)]) {
  def map[B](f: A => B): Distribution[B] =
    Distribution(events map { case (a, p) => f(a) -> p })
}
```

Return to the exercise

C.5.48 Solution to: Probabilities Part 5

Once we know how to combine probabilities we just have to follow the types. I've decided to normalise the probabilities after `flatMap` as it helps avoid numeric underflow, which can occur in complex models. An alternative is to use log-probabilities, replacing multiplication with addition.

```
final case class Distribution[A](events: List[(A, Double)]) {
  def map[B](f: A => B): Distribution[B] =
    Distribution(events map { case (a, p) => f(a) -> p })

  def flatMap[B](f: A => Distribution[B]): Distribution[B] =
    Distribution(events flatMap { case (a, p1) =>
                   f(a).events map { case (b, p2) => b -> (p1 * p2) }
                 }).compact.normalize
```

```
  def normalize: Distribution[A] = {
    val totalWeight = (events map { case (a, p) => p }).sum
    Distribution(events map { case (a,p) => a -> (p / totalWeight) })
  }

  def compact: Distribution[A] = {
    val distinct = (events map { case (a, p) => a }).distinct
    def prob(a: A): Double =
      (events filter { case (x, p) => x == a } map { case (a, p) => p }).sum

    Distribution(distinct map { a => a -> prob(a) })
  }
}
object Distribution {
  def uniform[A](atoms: List[A]): Distribution[A] = {
    val p = 1.0 / atoms.length
    Distribution(atoms.map(a => a -> p))
  }
}
```

Return to the exercise

C.5.49 Solution to: Examples

First I constructed the model

```
// We assume cooked food makes delicious smells with probability 1.0, and raw
// food makes no smell with probability 0.0.
sealed trait Food
final case object Raw extends Food
final case object Cooked extends Food

val food: Distribution[Food] =
  Distribution.discrete(List(Cooked -> 0.3, Raw -> 0.7))

sealed trait Cat
final case object Asleep extends Cat
final case object Harassing extends Cat

def cat(food: Food): Distribution[Cat] =
  food match {
    case Cooked => Distribution.discrete(List(Harassing -> 0.8, Asleep -> 0.2))
    case Raw => Distribution.discrete(List(Harassing -> 0.4, Asleep -> 0.6))
  }

val foodModel: Distribution[(Food, Cat)] =
  for {
    f <- food
    c <- cat(f)
  } yield (f, c)
```

From `foodModel` we could read off the probabilities of interest, but it's more fun to write some code to do this for us. Here's what I did.

```
// Probability the cat is harassing me
val pHarassing: Double =
  foodModel.events filter {
    case ((_, Harassing), _) => true
    case ((_, Asleep), _) => false
```

```
  } map { case (a, p) => p } sum

// Probability the food is cooked given the cat is harassing me
val pCookedGivenHarassing: Option[Double] =
  foodModel.events collectFirst[Double] {
    case ((Cooked, Harassing), p) => p
  } map (_ / pHarassing)
```

From this we can see the probability my food is cooked given the cat is harassing me is probably 0.46. I should probably check the oven even though it's more likely the food isn't cooked because leaving my food in and it getting burned is a far worse outcome than checking my food while it is still raw.

This example also shows us that to use this library for real we'd probably want to define a lot of utility functions, such as `filter`, directly on distribution. We also need to keep probabilities unnormalised after certain operations, such as filtering, so we can compute conditional probabilities correctly.

Return to the exercise

C.6 Type Classes

C.6.1 Solution to: More Orderings

```
val absOrdering = Ordering.fromLessThan[Int]{ (x, y) =>
  Math.abs(x) < Math.abs(y)
}
```

Return to the exercise

C.6.2 Solution to: More Orderings Part 2

Simply mark the value as implicit (and make sure it is in scope)

```
implicit val absOrdering = Ordering.fromLessThan[Int]{ (x, y) =>
  Math.abs(x) < Math.abs(y)
}
```

Return to the exercise

C.6.3 Solution to: Rational Orderings

```
implicit val ordering = Ordering.fromLessThan[Rational]((x, y) =>
  (x.numerator.toDouble / x.denominator.toDouble) <
  (y.numerator.toDouble / y.denominator.toDouble)
)
```

Return to the exercise

C.6.4 Solution to: Ordering Orders

My implementation is below. I decided that ordering by `totalPrice` is likely to be the most common choice, and therefore should be the default. Thus I placed it in the companion object for `Order`. The other two orderings I placed in objects so the user could explicitly import them.

```
final case class Order(units: Int, unitPrice: Double) {
  val totalPrice: Double = units * unitPrice
}

object Order {
  implicit val lessThanOrdering = Ordering.fromLessThan[Order]{ (x, y) =>
    x.totalPrice < y.totalPrice
  }
}

object OrderUnitPriceOrdering {
  implicit val unitPriceOrdering = Ordering.fromLessThan[Order]{ (x, y) =>
    x.unitPrice < y.unitPrice
  }
}

object OrderUnitsOrdering {
  implicit val unitsOrdering = Ordering.fromLessThan[Order]{ (x, y) =>
    x.units < y.units
  }
}
```

Return to the exercise

C.6.5 Solution to: Equality

```
trait Equal[A] {
  def equal(v1: A, v2: A): Boolean
}
```

Return to the exercise

C.6.6 Solution to: Equality Part 2

```
object EmailEqual extends Equal[Person] {
  def equal(v1: Person, v2: Person): Boolean =
    v1.email == v2.email
}

object NameEmailEqual extends Equal[Person] {
  def equal(v1: Person, v2: Person): Boolean =
    v1.email == v2.email && v1.name == v2.name
}
```

Return to the exercise

C.6.7 Solution to: Equality Again

```
object Eq {
  def apply[A](v1: A, v2: A)(implicit equal: Equal[A]): Boolean =
    equal.equal(v1, v2)
}
```

Return to the exercise

C.6.8 Solution to: Equality Again Part 2

```
object NameAndEmailImplicit {
  implicit object NameEmailEqual extends Equal[Person] {
    def equal(v1: Person, v2: Person): Boolean =
      v1.email == v2.email && v1.name == v2.name
  }
}

object EmailImplicit {
  implicit object EmailEqual extends Equal[Person] {
    def equal(v1: Person, v2: Person): Boolean =
      v1.email == v2.email
  }
}

object Examples {
  def byNameAndEmail = {
    import NameAndEmailImplicit._
    Eq(Person("Noel", "noel@example.com"), Person("Noel", "noel@example.com"))
  }

  def byEmail = {
    import EmailImplicit._
    Eq(Person("Noel", "noel@example.com"), Person("Dave", "noel@example.com"))
  }
}
```

Return to the exercise

C.6.9 Solution to: Equality Again Part 3

The following code is what we're looking for:

```
object Equal {
  def apply[A](implicit instance: Equal[A]): Equal[A] =
    instance
}
```

In this case the Eq interface is slightly easier to use, as it requires less typing. For most complicated interfaces, with more than a single method, the companion object pattern would be preferred. In the next section we'll see how we can make interfaces that appear to be methods defined on the objects of interest.

Return to the exercise

C.6.10 Solution to: Drinking the Kool Aid

```
object IntImplicits {
  implicit class IntOps(n: Int) {
    def yeah = for{ _ <- 0 until n } println("Oh yeah!")
  }
}

import IntImplicits._

2.yeah
// Oh yeah!
// Oh yeah!
```

The solution uses a `for` comprehension and a range to iterate through the correct number of iterations. Remember that the range `0 until n` is the same as `0 to n-1`—it contains all numbers from `0` inclusive to `n` exclusive.

The names `IntImplicits` and `IntOps` are quite vague—we would probably name them something more specific in a production codebase. However, for this exercise they will suffice.

Return to the exercise

C.6.11 Solution to: Times

```
object IntImplicits {
  implicit class IntOps(n: Int) {
    def yeah =
      times(_ => println("Oh yeah!"))

    def times(func: Int => Unit) =
      for(i <- 0 until n) func(i)
  }
}
```

Return to the exercise

C.6.12 Solution to: Easy Equality

We just need to define an implicit class, which I have here placed in the companion object of `Equal`.

```
trait Equal[A] {
  def equal(v1: A, v2: A): Boolean
}
object Equal {
  def apply[A](implicit instance: Equal[A]): Equal[A] =
    instance

  implicit class ToEqual[A](in: A) {
    def ===(other: A)(implicit equal: Equal[A]): Boolean =
      equal.equal(in, other)
  }
}
```

Here is an example of use.

```
implicit val caseInsensitiveEquals = new Equal[String] {
  def equal(s1: String, s2: String) =
    s1.toLowerCase == s2.toLowerCase
}

import Equal._

"foo".===("FOO")
```

Return to the exercise

C.6.13 Solution to: Implicit Class Conversion

Here is the solution. The methods `yeah` and `times` are exactly as we implemented them previously. The only differences are the removal of the `implicit` keyword on the `class` and the addition of the `implicit def` to do the job of the implicit constructor:

```
object IntImplicits {
  class IntOps(n: Int) {
    def yeah =
      times(_ => println("Oh yeah!"))

    def times(func: Int => Unit) =
      for(i <- 0 until n) func(i)
  }

  implicit def intToIntOps(value: Int) =
    new IntOps(value)
}
```

The code still works the same way it did previously. The implicit conversion is not available until we bring it into scope:

```
5.yeah
// <console>:8: error: value yeah is not a member of Int
//               5.yeah
//                 ^
```

Once the conversion has been brought into scope, we can use `yeah` and `times` as usual:

```
import IntImplicits._

5.yeah
// Oh yeah!
// Oh yeah!
// Oh yeah!
// Oh yeah!
// Oh yeah!
```

Return to the exercise

C.6.14 Solution to: Convert X to JSON

The *type class* is generic in a type `A`. The `write` method converts a value of type `A` to some kind of `JsValue`.

```
trait JsWriter[A] {
  def write(value: A): JsValue
}
```

Return to the exercise

C.6.15 Solution to: Convert X to JSON Part 2

```
object JsUtil {
  def toJson[A](value: A)(implicit writer: JsWriter[A]) =
    writer write value
}
```

Return to the exercise

C.6.16 Solution to: Convert X to JSON Part 3

```
implicit object AnonymousWriter extends JsWriter[Anonymous] {
  def write(value: Anonymous) = JsObject(Map(
    "id"           -> JsString(value.id),
    "createdAt"    -> JsString(value.createdAt.toString)
  ))
}

implicit object UserWriter extends JsWriter[User] {
  def write(value: User) = JsObject(Map(
    "id"           -> JsString(value.id),
    "email"        -> JsString(value.email),
    "createdAt"    -> JsString(value.createdAt.toString)
  ))
}
```

Return to the exercise

C.6.17 Solution to: Convert X to JSON Part 4

```
visitors.map(visitor => JsUtil.toJson(visitor))
```

Return to the exercise

C.6.18 Solution to: Prettier Conversion Syntax

```
implicit class JsUtil[A](value: A) {
  def toJson(implicit writer: JsWriter[A]) =
    writer write value
}
```

In the previous exercise we only defined `JsWriters` for our main case classes. With this convenient syntax, it makes sense for us to have an complete set of `JsWriters` for all the serializable types in our codebase, including `Strings` and `Dates`:

```
implicit object StringWriter extends JsWriter[String] {
  def write(value: String) = JsString(value)
}

implicit object DateWriter extends JsWriter[Date] {
  def write(value: Date) = JsString(value.toString)
}
```

With these definitions we can simplify our existing `JsWriters` for `Anonymous`, `User`, and `Visitor`:

```
implicit object AnonymousWriter extends JsWriter[Anonymous] {
  def write(value: Anonymous) = JsObject(Map(
    "id"        -> value.id.toJson,
    "createdAt" -> value.createdAt.toJson
  ))
}

implicit object UserWriter extends JsWriter[User] {
  def write(value: User) = JsObject(Map(
    "id"        -> value.id.toJson,
    "email"     -> value.email.toJson,
    "createdAt" -> value.createdAt.toJson
  ))
}

implicit object VisitorWriter extends JsWriter[Visitor] {
  def write(value: Visitor) = value match {
    case anon: Anonymous => anon.toJson
    case user: User      => user.toJson
  }
}
```

Return to the exercise

C.7 Pattern Matching

C.7.1 Solution to: Positive Matches

To implement this extractor we define an `unapply` method on an object `Postiive`. ~ scala object Positive { def unapply(in: Int): Option[Int] = if(in > 0) Some(in) else None } ~

Return to the exercise

C.7.2 Solution to: Titlecase extractor

The model solution splits the string into a list of words and maps over the list, manipulating each word before re-combining the words into a string.

```
object Titlecase {
  def unapply(str: String) = {
    str.split(" ").toList.map {
      case "" => ""
      case word => word.substring(0, 1).toUpperCase + word.substring(1)
    }.mkString(" ")
  }
```

```
}
```

Return to the exercise

C.8 Collections Redux

C.8.1 Solution to: Animals

```
val animals = Seq("cat", "dog", "penguin")
// animals: Seq[String] = List(cat, dog, penguin)
```

Return to the exercise

C.8.2 Solution to: Animals Part 2

```
"mouse" +: animals :+ "tyrannosaurus"
// res: Seq[String] = List(mouse, cat, dog, penguin, tyrannosaurus)
```

Return to the exercise

C.8.3 Solution to: Animals Part 3

The returned sequence has type `Seq[Any]`. It is perfectly valid to return a supertype (in this case `Seq[Any]`) from a non-destructive operation.

```
2 +: animals
// res: Seq[Any] = List(2, cat, dog, penguin)
```

You might expect a type error here, but Scala is capable of determining the least upper bound of `String` and `Int` and setting the type of the returned sequence accordingly.

In most real code appending an `Int` to a `Seq[String]` would be an error. In practice, the type annotations we place on methods and fields protect against this kind of type error, but be aware of this behaviour just in case.

Return to the exercise

C.8.4 Solution to: Animals Part 4

If we try to mutate a sequence we *do* get a type error:

```
val mutable = scala.collection.mutable.Seq("cat", "dog", "penguin")
// mutable: scala.collection.mutable.Seq[String] = ArrayBuffer(cat, dog, penguin)

mutable(0) = 2
// <console>:9: error: type mismatch;
//  found   : Int(2)
//  required: String
//              mutable(0) = 2
//                           ^
```

Return to the exercise