Appendix A

Scala Tools

Whatever language you do your software development in, tools are a significant aspect of how you work. The adoption of languages is often driven by the quality of tools in their ecosystem. The choice of tools is completely independent of the learning objectives of this book. There are a number of mentions of Eclipse specific elements in the main text, but for the most part, we ignore what tools you will be using. This appendix is intended to give you a brief rundown of some of the main options that you have available and give you just enough information that you can get up and running with whichever you choose to use.

A.1 Command Line

No matter what tools you choose to use, you should go to `http://scala-lang.org/` and download the latest version of the Scala Software Development Kit (SDK). In order to get this running, you will also need to download a Java SDK (often called the JDK, which is different from the JRE). Once you have those installed and you have added their `bin` directories to your path,[1] you should be able to run `java -version` and `scala -version` from the command line to see information about what you have installed.

There are two main programs in the Scala installation that you can use to complete the tasks in this book. The `scalac` program is the Scala compiler. In the most basic usage, you give it the names of the `.scala` files that you want to compile, and it will create a number of `.class` files that can actually be run. There are two command line options that you are likely to use with `scalac`. The `-d` option tells it where you want to put the `.class` files. Most of the time it is nicer if you do not have all of your compiled files mixed in with your source files. The `-cp` or `-classpath` option lets you specify where the compiler should go to look for things like external libraries. For example, this is a flag you would would need to use to compile GUI programs that use ScalaFX.

The other significant program is `scala`, which actually runs programs. There are three ways to invoke this command. If you provide it with no files to run, it will start the REPL. For this book, that is probably your most common usage scenario. This allows you to play with code, entering one statement at a time. If you provide a file name that ends in `.scala`, it will try to run that file as a Scala script. While we use that approach through

[1]How you add things to your path varies by operating system and even with versions of operating systems. We recommend you look online for how to add elements to the path on your machine.

the majority of *Introduction to Programming and Problem Solving Using Scala*, it is how you run small scripts and is not applicable to object oriented development. Thus, this usage is not generally applicable to the material in this book. The third method is to give it the name of a singleton `object` that you want to run as your application. For this to work, you need to have compiled the code with `scalac` and either be in the directory where it compiled or have that directory in your classpath. Note that when you do this, you provide a fully specified name for the object. For example, to run the version of the drawing program from chapter 17 you might type in `scala regexparser.drawing.DrawingMain`. This command can also accept the `-cp` or `-classpath` option to add elements to the classpath for running. If any were added when you ran `scalac`, they should probably be duplicated in the run of `scala`.

While we strongly recommend that you have Scala installed on your machine, and we believe that there will be times when you go out to the command line to run the REPL or verify compiles from the command line, we also recommend that you not use this approach in general for the material in this book. While all the tools listed below use `scalac` to do their compiling, you do not want to manually use `scalac` to compile all of your programs. Instead, you should use one of the other tools described in this appendix for most of your tasks.

A.2 sbt

There are lots of aspects in building programs that are tedious. To help automate the process of building programs and potentially speed it up by not rebuilding things that have not changed, software developers have created build tools. For Scala, the standard build tool is sbt which you can download and get information on from `http://www.scala-sbt.org/`.

Having sbt installed on your machine is a good idea if you have any thoughts of playing with Scala beyond the contents of this book. There are major projects, like the Play web framework, that make extensive use of sbt. The IDEs that are discussed below also provide methods of interacting with sbt projects, so even if you plan to use an IDE, there is a benefit in having sbt installed and some basic familiarity with how it works.

Given the completeness of their online documentation, and the possibility for future change, we are not going to duplicate their documentation here and instead recommend that you read what they have to say online.

A.3 Eclipse IDE

The closest thing to an official IDE (Integrated Development Environment) for Scala is Eclipse. Eclipse is an open source IDE developed primarily for Java, but which supports many other languages through plug-ins. There are two ways that you can get Eclipse with Scala. One is to go directly to `http://www.eclipse.org/` and download Eclipse, then go to `http://scala-ide.org/` and add the Scala plug-in. The other approach is to just download the bundle from `http://scala-ide.org/`. The second approach is simpler, but often lags behind the more recent releases of Eclipse. If you use the first approach, you should verify that the version of Eclipse that you are using works with the Scala plug-in. If a new version of Eclipse was just released, this might not be the case. You can always try to do an install and if you run into problems you can revert to an older version of Eclipse.

Once you have Eclipse installed with the Scala plug-in, you should make sure that you are in the Scala view, instead of the Java view, and run Eclipse and create a new Scala Project. You can change views using the Window → Show View menu option or with the little icon near the top right corner of the workspace. Note that you really cannot do anything in Eclipse unless you have a project that you are working in. Do not create a separate project for every little program that you write. Use projects to divide larger pieces of work. For example, all of the code for this book was written in a single Eclipse project. Packages were created in this project to separate out the code for different chapters. For your own work, we would recommend that you keep one project for playing around with code from the book and a separate project for each of the different end of chapter "Projects" that you choose to work on.

Once you have a project, you should expand it and create packages and source files in the `src` directory. When you have an application that you want to run, you can right click on the editor when the main is up or on the file that contains the main and select the `Run` option. There is also a `Run` menu that will allow you to run your programs.

A.3.1 Useful Keyboard Shortcuts

One of the main benefits of IDEs is that they can automate a number of common tasks that can prevent you from having to do tedious things while programming. Many of these cannot only be reached from the menus, they also have keyboard shortcuts, which most developers find remarkably handy. Here are a few using the PC keyboard layout. If you use a Mac, you can find them in the menus to see the keystrokes that are used on that platform.

- Ctrl-Alt-O - Automatically adds and organizes imports. If you are happy with having import statements added to the top of your file with one import per line, this keystroke will do that. If there are multiple types that share a name, it will bring up a dialog asking you which one you want to use. If there is nothing to import, this will "organize" the existing imports. When Eclipse does this, it puts them in alphabetical order and breaks them into groups. Unfortunately, this can do things for some Scala imports that you likely do not want. So Hitting this once when you have written code that needs one or more imports is a great time saver. Hitting it a second time you might want to avoid though.

- Ctrl-Alt-F - Format the code. Hopefully when you program you do a good job of indenting your code and putting meaningful whitespace in various places. Even if you are careful about this, it is still possible for your code to get messed up in certain places. Hitting this keyboard shortcut will reformat the code according to certain rules that you can change in the settings of Eclipse. Most all of the code in this book is presented in a format that was produced using this automatic formatter.

- Ctrl-Space - Brings up the context sensitive menu. One of the most beneficial aspects of IDEs in statically typed languages is that they can show you menus of the members of a `class` or other things that can be used to complete your current typing. Eclipse brings this up automatically after you enter a period if you pause for a second. This keystroke will bring it up for you at any time, even if you have not gotten to the period yet. If there is only one option, it will just fill in that option for you. Also, if you are typing the name of a `class`, `trait`, or singleton `object` that has not yet been imported, it will do any import of that one type in the same manner as Ctrl-Shift-O.

- Alt-Shift-R - Renames something. This is available under the “Refactoring” menu in Eclipse, but it is something you do commonly enough that it is probably worth learning the keystroke for.
- F3 or Ctrl-click - this takes you to the declaration of the element that the cursor is on. If it is in a different file, it will open that file and go to the proper point in the file.

There are many other keystrokes that you can use in Eclipse, this is only a small sample of the ones we have found to be most useful. A quick web search will pull up many others.

A.3.2 External Libraries

One of the things that you have to do a fair bit for some of the material in this book, including ScalaFX, Akka, XML, and parsers, is add external JAR files to the build path. In Eclipse you can right click on the project and select “Build Path” → “Add External Archives”. This will bring up a file selector that you can use to select the appropriate JAR files. If you want to see what JAR files are already there or possibly remove them, select “Build Path” → “Configure Build Path...” and go to the “Libraries” tab.

One of the great benefits of sbt is its ability to manage dependencies for you. That includes downloading them and adding them into the classpath. You can use sbt in conjunction with Eclipse by setting up an sbt project and running `sbteclipse`. This will create an Eclipse project for the sbt project that you can bring into Eclipse using “File” → “Import...” → “General” → “Existing Projects into Workspace”.

A.4 IntelliJ IDEA

While Eclipse has official support for the Scala plug-in, the developer community seems very evenly split between Eclipse and IntelliJ IDEA, which is another IDE that was primarily created for Java, but which includes a Scala plug-in. You can find information on IntelliJ and download it from `https://www.jetbrains.com/idea/`. The biggest difference between Eclipse and IntelliJ is that IntelliJ has a version that you can pay for with additional features. There is a “Community Edition” that you can download and use for free, but they also have a version they call the “Ultimate” version that includes extra features and that you have to pay for beyond a 30 day trial period. The material covered in this book can all be done with the “Community Edition”, so there is no need to spend money on the other version for this book. When you install IntelliJ, you will be given the option to add the Scala plug-in. You should select this option.

In many ways, you use IntelliJ much like you would Eclipse. You have to create a project to put your code in. You can choose if you want a regular IntelliJ project or an sbt based project. Then you can select the option to add a “Scala Class” to your project. That brings up a dialog in which tou can change from a class to a trait or an object.

A.4.1 External Libraries

If you use the sbt based project in IntelliJ, you should use sbt to handle your dependencies. However, if you use a standard IntelliJ project, “File” → “Project Structure” →

"Modules" $\rightarrow$ "Dependencies" $\rightarrow$ "Add". That will bring up a dialog that you can use to add a JAR file to your classpath.

Appendix B

Recursion Refresher

Recursion is a standard approach to problem solving. It is a valuable tool to have in your "bag of tricks" for programming as there are many problems with solutions that can be expressed more simply using recursion than they can be without it. Recursion also happens to be the standard tool for producing most repetition, including iteration, in functional languages.

B.1 Basics of Recursion

Recursion is a concept that comes from mathematics. A mathematical function is recursive if it is defined in terms of itself. For example, consider the factorial of an integer. You might recall from math classes that $n!$ is the product of all the integers from 1 up to n. We might write this as $n! = 1 * 2 * ... * n$. A factorial can be more formally defined like this.

$$n! = \begin{cases} 1 & n < 2 \\ n * (n-1)! & otherwise \end{cases}$$

In this definition, the factorial function is defined in terms of itself, thus it is a RECURSIVE DEFINITION.

To see how this works, let us take the factorial of 3. By our definition we get that $3! = 3 * 2!$. This is because 3 is not less than 2. Subsequently, we can see that $2! = 2 * 1!$ so $3! = 3 * 2 * 1!$. Finally, $1! = 1$ so $3! = 3 * 2 * 1$.

This definition and its application illustrate two of the key aspects of recursion. There are two possibilities for the value of this function. In the case where n is less than 2, the value of the factorial is 1. This is called a BASE CASE. All recursive functions need some kind of base case. The critical thing about a base case is that it is not recursive and should have a value that can be calculated directly without reference back to the function. The base case stops the recursion. Without a base case, every recursive call would result in another recursive call. This algorithm would result in having infinite recursion. There must be at least one base case, but there can be multiple different base cases. In the case where n is greater than or equal to 2, the function refers back to itself, but it does so with a different value and that value is moving toward the base case. This is the recursive case and is often called the GENERAL CASE. From this example you can see that every recursive definition must have one (or more) base cases, the base case(s) must stop the recursion, and the recursive case must eventually be reduced to a base case.

There are many things that could be defined recursively. We could define multiplication, recursively. After all, at least for the positive integers, multiplication is nothing more than repeated addition? We can write a recursive definition for multiplication like this.

$$m * n = \begin{cases} 0 & n = 0 \\ m + (m * (n-1)) & \textit{otherwise} \end{cases}$$

This function has two cases again, with a base case and a recursive case. Note that the recursive case is defined in terms of the value we are recursing on using a smaller value of that argument.

A function that directly calls itself is called DIRECTLY RECURSIVE. A function that calls another function and eventually results in the original function being called is INDIRECTLY RECURSIVE. For example, if function1 called function2 which eventually called function1, then function1 is indirectly recursive. If there is only one recursive call, it occurs in the last statement in a recursive function, and there is no work to do in that statement after the recursive call, then this function is called a TAIL RECURSIVE FUNCTION.

B.2 Writing Recursive Functions

The translation from math functions to programming functions is not hard. In fact, little will change from the math notation to the Scala notation.

As before, we will begin with the factorial function. Here is the factorial function written in Scala in the REPL.

```
scala> def fact(n:Int):Int = if (n<2) 1 else n*fact(n-1)
fact: (n: Int)Int
```

We have called the function `fact`, short for factorial. The body of the function is a single `if` expression. First it checks the value of `n` to see if it is less than 2. If it is, the expression has a value of 1. Otherwise, it is `n*fact(n-1)`. We can see the results of using this function here:

```
scala> fact(5)
res1: Int = 120
```

We see here that it correctly calculates the factorial of 5.

One significant difference between recursive and non-recursive functions in Scala is that we have to specify the return type of recursive functions. If you do not, Scala will quickly let you know it is needed.

```
scala> def fact(n:Int) = if (n<2) 1 else n*fact(n-1)
<console>:6: error: recursive method fact needs result type
      def fact(n:Int) = if (n<2) 1 else n*fact(n-1)
                                        ^
```

To test this, let us take the factorial of a few different values.

```
scala> fact(10)
res2: Int = 3628800

scala> fact(15)
res3: Int = 2004310016

scala> fact(20)
res4: Int = -2102132736
```

The first two show you that the factorial function grows very quickly. Indeed, programs that do factorial work are referred to as intractable because you cannot use them for even modest size problems. The third example though shows something else interesting. The value of 20! should be quite large. It certainly should not be negative. The problem we are seeing is a result of integers on computers being represented by a finite number of bits. As a result, they can only get so large. The built in number representations in Scala use 32 bits for `Int`. If we changed our function just a bit to use `Long` we could get 64 bits. Let's see that in action.

```
scala> def fact(n:Long):Long = if (n<2) 1L else n*fact(n-1)
fact: (n: Long)Long

scala> fact(20)
res5: Long = 2432902008176640000
```

If you wish to calculate even larger values there is a `BigInt` type as well.

```
scala> def fact(n: BigInt): BigInt = if (n < 2) 1 else n * fact(n - 1)
fact: (n: BigInt)BigInt
```

This can be fun to play with, but operations on `BigInt` are much slower than on either `Int` or `Long`, so you should not use `BigInt` unless you really need to be able to represent extremely large numbers with perfect accuracy.[1]

Another simple example is to write a function that will "count" down from a certain number to zero, printing out each number as we count. First, we need to identify our base case, the point where we stop counting. Since we are counting down to zero, if the value is ever below zero then we are done and should not print anything.

In the recursive case, we will have to handle all the rest of the values greater than or equal to zero and up to that certain number `n`. Counting down from `n` is done by counting/printing the `n`, then counting down from `n-1`. Converting this into code looks like the following:

```
def countDown(n: Int): Unit = {
  if (n >= 0) {
    println(n)
    countDown(n - 1)
  }
}
```

The way this code is written, the base case does nothing so we have an `if` statement that will cause the function to finish without doing anything if the value of `n` is less than 0. You can call this function passing in different values for `n` to verify that it works.

[1]The `Double` type can represent numbers much larger than a `Long`, but it only keeps 15-16 digits of accuracy.

Now let us try something slightly different. What if we want to count from one value up to another? In some ways, this function looks very much like what we already wrote. There are some differences though. In the last function we were always counting down to zero so the only information we needed to know was what we were counting from. In this case, we need to be told both what we are counting from and what we are counting to. That means that our function needs two parameters passed in. This might look like the following.

```
def countFromTo(from: Int, to: Int): Unit = {
  println(from)
  if (from != to) {
    countFromTo(from + 1, to)
  }
}
```

This function will work fine under the assumption that we are counting up. However, if you call this with the intention of counting down so that `from` is bigger than `to`, you have a problem. To see why, let us trace through this function. First, let us see what happens if we call it with 2 and 5, so we are trying to count up from 2 to 5. We will leave out the method name and just put the values of the arguments as that is what changes.

```
(2,5) => prints 2

  ↓

(3,5) => prints 3

  ↓

(4,5) => prints 4

  ↓

(5,5) => prints 5
```

The last call does not call itself because the condition `from!=to` is `false`.

Now consider what happens if we called this function with the arguments reversed. It seems reasonable to ask the function to count from 5 to 2. It just has to count down. To see what it really does, we can trace it.

```
(5,2) => prints 5

  ↓

(6,2) => prints 6

  ↓

(7,2) => prints 7

  ↓

(8,2) => prints 8

  ↓

...
```

This function will count for a very long time. It is not technically infinite recursion because the `Int` type only has a finite number of values. Once it counts above $2^{31} - 1$ it wraps back around to -2^{31} and counts up from there to 2 where it will stop.

Looking at the code and the trace you should quickly see that the problem is due to the fact that the recursive call is passed a value of `from + 1`. So the next call is always using a value one larger than the previous one. What we need is to use `+1` when we are counting up and `-1` when we are counting down. This behavior can be easily added by replacing the 1 with an `if` expression. Our modified function looks like this.

```
def countFromTo(from: Int, to: Int): Unit = {
  println(from)
  if (from != to) {
    countFromTo(from + (if (from < to) 1 else -1), to)
  }
}
```

Now when the `from` value is less than the `to` value we add 1. Otherwise, we will add -1. Since we do not get to that point if the two are equal, we do not have to worry about that situation. You should enter this function in and test it to make sure that it does what we want.

B.3 Recursion with Collections

Hopefully you are starting to get the hang of basic recursion and identifying the base case as well as the recursive case. Our next challenge is looking at how to use recursion with collections. Let us consider the problem of finding the smallest element of an array. I am sure you feel quite comfortable solving this problem using looping constructs such as `for` and `while`, but how would you solve this using a recursive function? Assume that we have the following array.

```
scala> val arr=Array(4,2,9,8,3,5)
arr: Array[Int] = Array(4, 2, 9, 8, 3, 5)
```

We know that we need to check each element of the array from arr(0) until the end of the array which is arr(5). We are going to have our function accept two parameters, the `Array`, `arr` that we are working with and the current index, `Index`, that we are considering. We are going to have the index count up, so it stops when it runs out of elements in the array. That is our base case when `index >= arr.length`. In that situation, we want to return a really large number, one that could not possibly be the minimum value. We will use `Int.MaxValue` for this.

If we have not yet gone beyond the end of the array, that is to say that `index` is less than `arr.length`, then we want to know if `arr(index)` is less than the smallest of all the elements from `index + 1` to the end of the array. We get that second value with the recursive call. Putting these together, our smallest function could be written like this.

```
def smallest(arr: Array[Int], index: Int): Int = {
  if (index < arr.length) {
    val minimum = smallest(arr, index + 1)
    if (arr(index) <= minimum) {
      arr(index)
    } else {
      minimum
    }
  } else {
    Int.MaxValue
  }
}
```

We can test our function like this.

```
scala> println(smallest(arr,0))
```

This function will find that 2 is the smallest number in our array. Note that if called this function with an empty integer array, then the value of 2147483647 would have been returned which is the value of `Int.MaxValue`.

That gave us a bit of exposure of how to use recursion with `Array`s, but what about another major collection type: `List`s? How would we implement a recursive function to find the smallest element in a `List`? This is done using a fundamentally different approach. We could write it using indexing the way we just did with the `Array`, but such an approach is slow when working with `List`s. The fast way to go through a `List` is to recurse on the `tail`, so we are calling the function on smaller parts of the original `List`. This approach is actually perfectly suited for recursion. One way we could write that recursive function is like this.

```
  def smallest(lst: List[Int]): Int = {
    if (lst.isEmpty) {
      Int.MaxValue
    } else {
      val minimum = smallest(lst.tail)
      if (lst.head <= minimum) {
        lst.head
      } else {
        minimum
      }
    }
  }
```

Here, we first check to see if the `List` is empty, which is our base case. If it is not empty, we look for the smallest value in the remaining portion of the `List` using `lst.tail`. Next we check if the first element of the list, which we find using `lst.head`, is less than the minimum element from the rest of the `List` and return the smaller of the two values.

We could have also written a recursive implementation on a `List` using pattern matching. That code would look like this.

```
def smallest2(lst: List[Int]): Int = lst match {
  case Nil => Int.MaxValue
  case h :: t => h min smallest2(t)
}
```

This code also uses the `min` method on the `Int` type to make the code shorter. You could replace the inner most `if` in the previous two versions with a call to `min` as well.

B.4 User Input

To expand our use of recursion, let us consider getting input from the user. We can start by writing a function called `sumInputInts` which will sum a specific number of values. We pass in an integer that represents how many integers we want the user to input, and the function will return the sum of those values. How can we define such a function recursively? If we want to sum up 10 numbers, we could say that sum is the first number, plus the sum of 9 others. The base case here is when the number of numbers we are supposed to sum gets below 1. In that case, then the sum is zero. Let us see what this would look like in code.

```
def sumInputInts(num: Int): Int = {
  if (num > 0) {
    readInt() + sumInputInts(num - 1)
  } else {
    0
  }
}
```

The `if` is being used as an expression here. It is the only expression in the function so it is the last one, and it will be the result value of the function. If `num` is not greater than zero, then the function's value is zero. If it is greater than zero, the function will read in a new value and return the sum of that value and what we get from summing one fewer values.

What if we do not know in advance how many values we are going to sum? We need to ask the user when they are done giving values to sum. An easy way to do this would be to only allow the user to sum positive values and stop as soon as a non-positive value is entered. This gives us a function that does not take any arguments and returns an integer for the sum of the numbers entered before the non-positive value. Such a function could be written as follows.

```
def sumInputPositive(): Int = {
  val n = readInt()
  if (n > 0) {
    n + sumInputPositive()
  } else {
    0
  }
}
```

This time we read a new value before we determine if we will continue or stop. The decision is based on that value, which we store in the variable `n`. This function does a good job of letting us add together an arbitrary number of user inputs, but it has a significant limitation, it only works with positive values. That is because we reserve negative values as the stop condition.

One way that this could be handled is to continue to read input until the user types something like “quit”. In this case, we will need to use a `readLine()` instead of a `readInt` and then we should be able to convert the `String` to an `Int` when it needs to be added to the total. That can be done as shown here.

```
def sumUntilQuit(): Int = {
  val n = readLine()
  if (n != "quit") {
    n.toInt + sumUntilQuit()
  } else {
    0
  }
}
```

Now we have a version of the function which will read one integer at a time until it gets the word “quit”.

What if we have a situation where we would like to return more than one value? For example, we might need to calculate an average, which needs to know both the sum and the number of value. We can use a tuple to do that. We will write a new function called `sumAndCount`, which returns a tuple that has the sum of all the numbers entered as well as the count of how many there were. We will base this off the last version of `sumUntilQuit` so there are no restrictions on the numbers the user can input. Such a function might look like the following:

```
def sumAndCount(): (Int, Int) = {
  val n = readLine()
  if (n != "quit") {
    val (s, c) = sumAndCount()
    (s + n.toInt, c + 1)
  } else {
    (0, 0)
  }
}
```

If you load this function into the REPL and call it, you can enter a set of numbers and see the return value. If, for example, you enter 3, 4, 5, and 6 on separate lines followed by “quit”, you will get this:

```
res0: (Int, Int) = (18,4)
```

This looks a lot like what we had before, only every line related to the return of the function now has a tuple for the sum and the count. We see it on the first line for the result type. We also see it on the last line of both branches of the `if` expression for the actual result values. The last place we see it is in the recursive branch where the result value from the recursive call is stored in a tuple.

B.5 Abstraction

What if, instead of taking the sum of a bunch of user inputs, we want to take a product? What would we change in `sumAndCount` to make it `productAndCount`? The obvious change is that we change addition to multiplication in the recursive branch of the `if`. A less obvious change is that we also need the base case to return 1 instead of 0. So our modified function might look like this.

```
def productAndCount(): (Int, Int) = {
  val n = readLine()
  if (n != "quit") {
    val (s, c) = productAndCount()
    (s * n.toInt, c + 1)
  } else {
    (1, 0)
  }
}
```

This is almost exactly the same as what we had before. We just called it a different name and changed two characters in it. We know that we should avoid code duplication. One way to avoid this is to include abstraction. We look for ways to make the original code more flexible so it can do everything we want.

In order to abstract these functions to make them into one, we focus on the things that were different between them and ask if there is a way to pass that information in as arguments to a version of the function that will do both. For this, the changing of the name is not important. What is important is that we changed the operation we were doing and the base value that was returned. The base value is easy to deal with. We simply pass in an argument to the method that is the value returned at the base. That might look like this.

```
def inputAndCount(base: Int): (Int, Int) = {
  val n = readLine()
  if (n != "quit") {
    val (s, c) = inputAndCount(base)
    (s * n.toInt, c + 1)
  } else {
    (base, 0)
  }
}
```

The argument base is passed down through the recursion and is also returned in the base case. However, this version is stuck with multiplication so we have not gained all that much.

Dealing with the multiplication is a bit harder. For that we need to think about what multiplication and addition really are and how they are used here. Both multiplication and addition are operators. They take in two operands and give us back a value. When described that way, we can see they are like functions. What we need is a function that takes two `Int`s and returns an `Int`. That function could be multiplication or addition and then the `inputAndCount` function would be flexible enough to handle either a sum or a product. It might look like this.

```
def inputAndCount(base: Int, func: (Int, Int) => Int): (Int, Int) = {
  val n = readLine()
  if (n != "quit") {
    val (s, c) = inputAndCount(base, func)
    (func(s, n.toInt), c + 1)
  } else {
    (base, 0)
  }
}
```

The second argument to `inputAndCount`, which is called `func`, has a more complex type. It is a function type. It is a function that takes two `Int`s as arguments and returns an `Int`. As with `base`, we pass `func` through on the recursive call. We also used `func` in place of the `*` or the `+` in the first element of the return tuple in the recursive case. Now instead of doing `s + n.toInt` or `s * n.toInt`, we are doing `func(s, n.toInt)`. What that does depends on the function that is passed in.

To make sure we understand this process we need to see it in action. Let us start by doing a sum and use the longest, easiest to understand syntax. We define a function that does addition and pass that in. For the input we type in the numbers 3, 4, and 5 followed by "quit". Those values are not shown by the REPL.

```
scala> def add(x: Int, y: Int): Int = x + y
add: (x: Int,y: Int)Int

scala> inputAndCount(0, add)
res3: (Int, Int) = (12,3)
```

In the call to `inputAndCount` we used the function `add`, which was defined above it, as the second argument. Using a function defined in this way forces us to do a lot of typing. This is exactly the reason Scala includes function literals. You will recall that a function literal allows us to define a function on the fly in Scala. The normal syntax for this looks a lot like the function type in the definition of `inputAndCount`. It uses a `=>` between the parameters and the body of the function. Using a function literal we could call `inputAndCount` without defining the `add` function. That approach looks like this.

```
scala> inputAndCount(0, (x, y) => x + y)
res4: (Int, Int) = (12,3)
```

One thing to notice about this is that we did not have to specify the types on `x` and `y`. That is because Scala knows that the second argument to `inputAndCount` is a function that takes two `Int` values. As such, it assumes that `x` and `y` must be of type `Int`.

There is an even shorter syntax for declaring function literals that only works in certain situations. That was the syntax that uses _ as a placeholder for the arguments. This syntax can only be used if each argument occurs only once and in order. That is true here, so we are allowed to use that shorthand. Doing so simplifies our call all the way down to this.

```
scala> inputAndCount(0, _ + _)
res5: (Int, Int) = (12,3)
```

Of course, the reason for doing this was so that we could also do products without having to write a second function. The product function differed from the sum function in that the base case was 1 and it used `*` instead of `+`. If we make those two changes to what we did above, we will see that we have indeed created a single function that can do either sum or product.

```
scala> inputAndCount(1, _ * _)
res6: (Int, Int) = (60,3)
```

The `inputAndCount` function is what we call a higher order function. That is because it is a function that uses other functions to operate. We provide it with a function to help it do its job.

Tail Recursive Functions

Every call to a function typically takes a little memory on what is called the stack to store the arguments, local variables, and information about where the call came from. If a recursive function calls itself many times, this can add up to more than the amount of memory set aside for this purpose. When this happens, you get a stack overflow and your program terminates.

Scala will optimize this away for recursive functions that have nothing left to do after the recursive call. You can convert a recursive function that is not tail recursive to one that is by passing in additional arguments. To see this, we can demonstrate how this would be done on the `sumAndCount` function on page 638. The original version is not tail recursive because it does addition to both the sum and the count after the recursive call. Just doing one of those would prevent the function from being tail recursive. To change this, we can pass in the sum and count as arguments. The produces the following code.

```
def sumAndCountTailRec(sum: Int, count: Int): (Int, Int) = {
  val n = readLine()
  if (n != "quit") {
    sumAndCountTailRec(sum + n.toInt, count + 1)
  } else {
    (sum, count)
  }
}
```

Note that when you call this version of the function, you have to provide initial values for `sum` and `count`, both of which should be zero.

If a function really needs to be tail recursive, you can tell Scala this with an annotation. To do this, you first need to `import annotation.tailrec`, then you can put `@tailrec` in front of any function that must be tail recursive. If Scala cannot make the function tail recursive, it will give an error.

B.6 Matching

We saw a simple usage of pattern matching when we wrote our recursive function for finding the smallest element in a `List`. Another example to demonstrate recursion using `match` instead of an `if` is to re-write `countDown`.

```
def countDown(n: Int): Unit = n match {
  case 0 =>
  case i =>
    println(i)
    countDown(i - 1)
}
```

This function is not quite the same as what we had with the `if` because it only stops on the value 0. This makes it a little less robust, but it does a good job of illustrating the syntax and style of recursion with a `match`. The recursive argument, the one that changes each time the function is called, is the argument to `match`. There is at least one case that does not involve a recursive call and at least one case that does.

A more significant example would be to rewrite the `inputAndCount` function using a `match`.

```
def inputAndCount(base: Int, func: (Int, Int) => Int): (Int, Int) = readLine()
    match {
  case "quit" =>
```

```
    (base, 0)
  case n =>
    val (s, c) = inputAndCount(base, func)
    (func(s, n.toInt), c + 1)
}
```

Here the call to `readLine` is the argument to match. This is because there is not a standard recursive argument for this function. The decision of whether or not to recurse is based on user input, so the user input is what we match on.

B.7 Looking Ahead

We have only used recursion in this appendix to repeat tasks, as a model for repetition. The real power of recursion comes from the fact that it can do a lot more than just repetition. Recursive calls have memory. They do not know just what they are doing now, they remember what they have done. This really comes into play when a recursive function calls itself more than once.

Below is a little bit of code. You will notice that it is nearly identical to the `countDown` function that we wrote near the beginning of this chapter. Other than changing the name of the method the only difference is that the two lines inside the `if` have been swapped. Put this function into Scala. What does it do? More importantly, why does it do that?

```
def count(n: Int): Unit = {
  if (n >= 0) {
    count(n - 1)
    println(n)
  }
}
```

B.8 End Material

B.8.1 Summary of Concepts

- The concept of recursion comes from mathematics where it is used to refer to a function that is defined in terms of itself.
  - All recursive functions must have at least one base case that does not call itself in addition to at least one recursive case that does call itself.
  - Lack of a base case leads to infinite recursion.
- In a programming context, a recursive function is one that calls itself.
  - Recursion is used in this chapter to provide repetition.
  - Typically an argument is passed in to the function, and on each subsequent call the argument is moved closer to the base case.

  - If the argument is not moved toward the base case it can result in an infinite recursion.

- Recursion can also be done on user input when we do not know in advance how many times something should happen.
  - Reading from input is a mutation of the state of the input.
  - Functions that use this might not take an argument if they read from standard input.
  - The base case occurs when a certain value is input.

- It is inefficient to make copies of code that only differ in slight ways. This can often be dealt with by introducing an abstraction on the things that are different in the different copies.
  - With functions, this is done by passing in other arguments that tell the code what to do in the parts that were different in the different copies.
  - Values can easily be passed through with parameters of the proper value type.
  - Variations in functionality can be dealt with using parameters of function types. This makes the abstract versions into higher-order functions.

- There is another type of conditional construct in Scala called `match`.
  - A `match` can include one or more different `case`s.
  - The first `case` that matches the initial argument will be executed. If the `match` is used as an expression, the value of the code in the `case` will be the value of the `match`.
  - The `case`s are actually patterns. This gives them the ability to match structures in the data and pull out values.
    - Tuples can be used as a pattern.
    - Lowercase names are treated as `val` variable declarations and bound to that part of the pattern.
    - You can use _ as a wildcard to match any value that you do not need to give a name to in a pattern.
  - After the pattern in a `case` you can put an `if` guard to further restrict that `case`.

- The `try`/`catch` expression can be used to deal with things going wrong.
  - The block of code after `try` will be executed. If an exception is thrown, control jumps from the point of the exception down to the `catch`.
  - The `catch` will have different `case`s for the exceptions that it can handle.
  - It only makes sense for this to be an exception if all paths produce the same type.

B.8.2 Self-Directed Study

Enter the following statements into the REPL and see what they do. Try some variations to make sure you understand what is going on. Note that some lines read values so the REPL will pause until you enter those values. The outcome of other lines will depend on what you enter.

```
scala> def recur(n:Int):String = if (n<1) "" else readLine()+recur(n-1)
scala> recur(3)
scala> def recur2(n:Int,s:String):String = if (n<1) s else recur2(n-1,s+readLine())
scala> recur2(3,"")
scala> def log2ish(n:Int):Int = if (n<2) 0 else 1+log2ish(n/2)
scala> log2ish(8)
scala> log2ish(32)
scala> log2ish(35)
scala> log2ish(1100000)
scala> def tnp1(n:Int):Int = if (n<2) 1 else
  1+(if (n%2==0) tnp1(n/2) else tnp1(3*n+1))
scala> tnp1(4)
scala> tnp1(3)
scala> def alpha(c:Char):String = if (c>'z') "" else c+alpha((c+1).toChar)
scala> alpha('a')
```

Appendix C

Quick Preview of Java

Scala has many of its roots in the Java programming language beyond compiling to the Java Virtual Machine (JVM) and having the ability to call on Java library functions. For this reason, it should not be too hard to pick up Java after you have learned Scala. This appendix is designed to help you in doing exactly that. To accomplish that goal, we will run through a number of different simple examples of Java code and look at the ways in which they are different from Scala code. If you have been using either Eclipse or IntelliJ, you can switch to doing Java development and create a Java project to work in.

C.1 Hello World

We will start with Java that same way we started with Scala, using the quintessential "Hello World" application. Here is what that code looks like in Java. It looks much like a Scala application. In Java, there is no scripting model, nor is there a REPL.[1] As such, this is as simple as it gets.

```
package javacode;

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

Just from this little example though, there are a number of things that are obviously different. First, Java does not infer semicolons. You have to type them. In this code we see one semicolon at the end of the package declaration and another at the end of the print statement.

Going into the `class`, you see that it starts with the `public` keyword. Scala does not have a `public` keyword as that is the default visibility. Java also has keywords for `private` and `protected` visibility. In Java `public` visibility is not the default visibility. Instead, the default in Java is called package private. That is what we would get in Scala if we said

[1]There are plans to add a REPL to Java 9.

private[javacode] because this class is in the javacode package. Java does not give you the fine grained control over visibility that Scala does, so the public, private, protected, and package private options are all you have.

In Java it is very strongly recommended that data members never be made public. Unlike Scala, you cannot change a public data member into appropriate methods to include checking of values without breaking code. As such, the default for all data members should absolutely be private when you are coding in Java. Part of the reason for this is the fact that method calls in Java always require parentheses and member data access never has them. So in Java, the syntax always tells you if you are using a method or a data member. Scala went the other way so that the implementation can be changed without breaking code.

The class declaration has curly braces after it. Curly braces in Java group code just like in Scala, but they are a group statement, not an expression. In Java, a code block using curly braces will not give you a value.

Looking inside the class at the one method, the declaration of main includes a number of differences as well. The first is the static keyword, which does not exist in Scala. Java does not have singleton objects. The closest approximation to them in Java is static data members and methods. The static keyword in Java implies that the thing being declared is associated with the class instead of being associated with the instances of that class. They are very much like the members of a companion object except that there is no companion object encapsulating them which reduces their flexibility a bit. The main method has to be static in Java for the same reason that the main method had to go in a singleton object in Scala. Without that, it could not be invoked without some class being instantiated first.

After the static keyword is another word we did not see in Scala, void. This is a type in Java that represents nothing. The equivalent of void in Scala is Unit. The difference is that there is no instance of void while there is an instance of Unit represented by (). Also note that void comes before the name of the method, not at the end after a colon as types did in Scala. All Java types precede the names they are associated with. One advantage of this is that Java does not need the val, var, or def keywords for those declarations. The type tells the language you are declaring something. The down side is that you have to specify a type. Java does not do type inference for you the way that Scala does.

After the method name, main, is String[] args. Based on your knowledge of Scala you should realize that args is the name of a formal parameter and you can figure out that String[] specified a type that is equivalent to the Scala type Array[String]. We will come back to the Java array syntax in a bit, but the short version here is that Java uses square brackets only for arrays, not type parameters, and you specify a type that is an array of another type by putting square brackets after the type. The formal argument shows again that the types in Java precede the names they are associated with.

Inside the main method is a single call to println. By default, there is nothing set up in Java to bring the println method into scope. Instead, you specify the full name, System.out.println. Breaking this down, System is a class in java.lang, which is imported by default. There is a static member in System named out that is a java.io.PrintStream which has println methods for different types.

C.2 Arrays, Primitives, and More

The next example is a bit more significant. This creates an array of 1000 integers, fills it with the first 1000 prime numbers, then adds them up and prints the sum.

```
package javacode;

import static java.lang.System.out;

public class PrimeArray {
   private static boolean isPrime(int n) {
      for(int i = 2; i*i<=n; ++i) {
         if(n%i==0) return false;
      }
      return true;
   }

   public static void main(String[] args) {
      int[] primes = new int[1000];
      int pos = 0;
      int i = 2;
      while(pos<primes.length) {
         if(isPrime(i)) {
            primes[pos] = i;
            pos++;
         }
         i++;
      }
      int sum = 0;
      for(int j=0; j<primes.length; ++j) {
         sum += primes[j];
      }
      out.println(sum);
   }
}
```

To shorten the call to `System.out.println` a bit, the file has a `static import` at the top. Normal `import`s in Java can only import the contents of packages. The `static import` also has the ability to import `static` members. Note that the Scala version does not distinguish between types of uses. Also, Java `import` statements need to be at the top of the file and they lack much of the flexibility of those in Scala. If you want to `import` all of the contents of a package in Java use `*` instead of `_`.

The `class` in this file has two methods in it called `isPrime` and `main`. Both are `static`. The `main` method has to be `static` so that it functions as an entry point into an application. The `isPrime` method needs to be `static` here because only a `static` method can be called from another `static` method without instantiating a `class` to call it on. It also should be `static` because it does not use any information from any instance of the `class`. In Scala terms, it is a method that should be in the companion object.

Looking in the `isPrime` method shows a few more differences between Java and Scala. There are two more types that appear here, `boolean` and `int`. These should look familiar to you. They are just like what you are used to in Scala except for the fact that they start with a lowercase letter. You might have noticed this about the `void` type as well, though it is not true about the `String`. This is more than just a matter of capitalization. It has semantic implications as well. Java is not as object-oriented as Scala. The instances of the basic types in Java are not objects. They are called primitives and they do not have any methods. This is why their types start with lowercase letter. `String` is a `class` and instances of `String` are objects. That is not true for `int` or `boolean`.

The `for` loop should stand out as another significant difference. The `for` loop is Scala is technically a for-each loop that runs through the contents of a collection. Java's basic `for` loop is more of a dressed up `while` loop. In the parentheses there are three parts, separated by semicolons. The first part is an initializer where you set things up. This happens once, right when the loop is reached. If variables are declared there, as is the case in this example, that variable has a scope only through the loop.

The second part is a condition, just like what you would put in a `while` loop. It is pre-check and the loop will iterate until the condition is false. This is why the Java `for` loop can be described as a dressed-up `while` loop. The way in which the end case is determined is just like a `while` loop. The dressing is what comes before and after the semicolons around the condition.

The last part of what appears in the parentheses of a `for` loop is an iterator. This code happens after the body of the loop is evaluated and before the next condition check. In this case, the expression is `++i`. This is a pre-increment of the variable `i`. The `++` operator in Java does an increment by 1. It can be placed before or after the variable. When it is before, the result of that expression is the incremented value. When placed after the expression the result is the value before the increment.

There is another type of `for` loop in Java which functions as a for-each loop. The syntax is `for(`*type name*`:`*iterable*`)`, where *type* is the type in the collection, *name* is the variable name you want, and *iterable* is an object of type `Iterable`.

One last thing to note about the `for` loop is that it is never an expression. It never gives back a value. There is no equivalent to `yield`. The `while` loop in Java looks just like what you are used to in Scala.

The `isPrime` method also uses the `return` keyword. This is a valid usage in Scala, but you typically do not need it because every block is an expression with a value equal to that of the last statement in the block. In Java, blocks are not expressions, so every method that returns a value needs to specifically include a `return` statement at the end. If a `return` statement is reached inside the method, it will terminate the execution of that method and immediately jump back out to where it was called from.

The `main` method demonstrates Java's array syntax. The `primes` variable has the type `int[]`. You can see the array itself is built using the `new` syntax. Inside of the `while` loop you can see that the indexing of the array is also done using square brackets, passing in the index you want to access. Arrays in Java are objects, so you can ask them how long they are using `length`. Note that `length` here is a data member. You know this in Java because there are no parentheses after it. The `length` of a `String`, however, is gotten with a method and must always have empty parentheses after it.

C.3 File Names and Interfaces

To illustrate a few more features of Java we will use another favorite example, the bank account. We begin by introducing a supertype for bank accounts.

```
package javacode;
```

```
public interface BankAccount {
   String getName();
   int getBalance();
}
```

This supertype is made using a Java construct called an `interface`. You can think of an `interface` as being like a `trait` that has no data members and where everything is abstract. Java 8 added the ability to include default implementations of methods to `interface`s. These behave roughly the same way as methods put in traits in Scala, though method resolution rules in Java are a bit simpler to deal with the fact that this capability was added after the language was 20 years old. Putting data in interfaces does not do what you would normally expect. It creates `static` data members, which probably is not what you really wanted so you should likely not put data members in your `interface`s. Like Scala, Java only allows single inheritance of `class`es.

This `interface` will be inherited by a `CheckingAccount class`. Note that when inheriting from an `interface` in Java you use the `implements` keyword. When you inherit from a `class` you use the same `extends` keyword that you are used to from Scala. Multiple interfaces can follow `implements` and they are separated by commas instead of `with`.

```
package javacode;

public class CheckingAccount implements BankAccount {
   private final String name;
   private int balance = 0;

   public CheckingAccount(String n,int b) {
      name = n;
      balance = b;
   }

   @Override
   public String getName() {
      return name;
   }

   @Override
   public int getBalance() {
      return balance;
   }

   public boolean deposit(int amount) {
      if(amount>=0) {
         balance += amount;
         return true;
      } else {
         return false;
      }
   }

   public boolean withdraw(int amount) {
      if(amount>=0 && amount<=balance) {
         balance -= amount;
         return true;
```

```
        } else {
            return false;
        }
    }
}
```

There are many things in this class that are worth noticing. Before looking at the details though, we should note something that is not at all obvious here. The `BankAccount interface` is located in a file called "BankAccount.java". The `CheckingAccount class` is in a separate file called "CheckingAccount.java". In Scala it was recommended that separate `class`, `trait`, and `object` declarations be put in separate files with names that match. In Java, this is not just recommended, it is required. There can be only one top level `public` declaration per file in Java, and its name must match the name of the file.

So what are some of the differences between this code and what you would have written in Scala? For one thing, the `name` member is preceded by the `final` keyword. In Scala, `final` means that something cannot be overridden in a subtype. When applied to a method, it has that same meaning in Java. However, when used in front of a variable declaration, it means that the value of that variable cannot be changed after it is set. A `final` variable in Java is like a `val` declaration in Scala. Without `final`, it is like you are using `var`.

There is something else worth noting about `name`, it is not given a value. In Scala, any member without a value is considered abstract. This is not the case in Java. You can declare variables either locally or as `class` members without providing an initial value. For local variables, Java will check that they are not used before initialization and give you an error if that happens. For member variables there is no error. Instead, they are given a default value. For primitives the default is 0, 0.0, or false, depending on the exact type. For object types, typically called reference types in Java, they are given the value of `null`. This is a very common source of errors in Java and that is one of the reasons Scala forces you to initialize variables.

C.4 Constructors and `@Override`

With this example it also starts to become obvious that there are no arguments to the `class`. Argument lists are not required for Scala, but in this situation they would clearly have been useful. There are no argument lists for `class`es in Java. To get information into an object when it is instantiated in Java, you write CONSTRUCTORS. A constructor is a method that gets called when an object is instantiated. By default, Java creates a constructor that takes no arguments. Constructors that take no arguments are called "default constructors", even if you write them. If you include any constructor of your own, the default one is no longer created for you. A constructor looks like a method except that it has no return type, not even `void`, and the name matches the name of the class. In `CheckingAccount` you can see that there is a constructor directly after the two lines declaring the member data. This constructor takes initial values for the member data and sets them.

There are some things worth noting here. First, writing a little `class` in Java with member data that can be set at construction is much more verbose than it is in Scala. The `class` needs to contain declarations of the data members and a constructor that sets them instead of just listing them with `val` or `var` in an argument list. One could argue that the payoff of this is that the syntax for multiple constructors is a bit cleaner.

After the constructor are the two methods required by the supertype along with two others for doing deposits and withdraws. These should all be fairly straightforward at this point. The only thing really new is the `@Override` annotation. Scala made `override` a keyword in the language. Java did not do that originally, but added the annotation in later. This is not required in Java, but for the same reasons it is required in Scala, it is strongly recommended that you include it in Java. Eclipse will add it for you if you have it put in methods required by a supertype.

C.5 Generics and Polymorphism

To make the accounts useful, they need to be put into a bank. This is something that could involve a large amount of code, but we will stick to a very short example to demonstrate the basic concepts. Here is simple `Bank class` that keeps track of accounts using a Java collection.

```
package javacode;

import java.util.*;

public class Bank {
    private List<BankAccount> accounts = new ArrayList<BankAccount>();

    public void makeNewCheckingAccount(Scanner sc) {
        System.out.println("Who is the new checking account for?");
        String name = sc.next();
        System.out.println("What is the starting balance in cents?");
        int bal = sc.nextInt();
        accounts.add(new CheckingAccount(name,bal));
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Bank bank = new Bank();
        bank.makeNewCheckingAccount(sc);
        for(BankAccount ba:bank.accounts) {
            System.out.println(ba.getName()+" "+ba.getBalance());
        }
    }

}
```

The accounts are specifically stored in a `java.util.List`. The closest equivalent to this in Scala is the `mutable.Buffer`. Like the `Buffer`, the `List` is implemented using both an array and a linked list. You can see here that we have gone with the array based implementation.

The fact that Java used square brackets for arrays means that something else must be used for type parameters. These are called GENERICS in Java and they are put inside of angle braces, the characters we typically read as less than and greater than. As with Scala, you can make Java `class`es or methods generic. The generic type is typically inferred for methods, but not for `class`es.

If you want to make a collection or just a reference that can hold “anything” in Java, the common supertype is `Object`. Technically, `Object` is equivalent to `AnyRef`. The fact that primitives are not objects means they are not instances of any `class`. They are not involved in supertype/subtype relationships. To get around this, Java includes wrapper types like `java.lang.Integer` and does something called autoboxing to wrap up primitives in a way that makes life easier on the programmer. Technically Scala was doing this same thing for efficiency, but it hides the primitives from the programmer completely.

C.6 Functional Aspects

Java 8 added a number of more functional aspects to the Java programming language. There are now lambda expressions that use `->` instead of `=>`, but they have a somewhat different nature that fits in better with the longer history of Java not being a functional language. You can use a lambda expression anyplace that expects an instance of an `interface` that includes only one method without a default implementation. The lambda expression compiles to an anonymous inner class of that `interface` with the code being the implementation of that one method. The lambda expressions in Java are also not full closures, so if you try to use outside variables in them, you will get errors. This goes back to the fact that anonymous inner classes in Java have always had the restriction that they can only refer to local variables that are `final`. This restriction was not lifted for lambda expressions.

Java 8 also added other features to take advantage of the lambda expressions. In particular, they added "streams" to the collections library.[2] You can get a stream from a collection like an `ArrayList` by calling `stream()`. The code related to streams is in `java.util.stream`. Methods that you are used to using, like `map`, `filter`, and `foreach` are all in the `java.util.stream.Stream` type. This library is a bit less straightforward to use than the Scala collections. In addition to calling `stream()` to get a stream that has these methods at the beginning, if you want values back you also have to call `collect` when you are done and use one of several styles of "Collectors" to get back to having a normal value. The collectors are all methods in `java.util.stream.Collectors`. You can also make streams execute in parallel by calling the `parallel` method on a `Stream`.

Even with Java 8, Java is not as functional as Scala. There is no pass-by-name semantics in Java. There are no curried methods, and you do not really have higher order methods. Other features like lazy evaluation, which are discussed in appendix D are also missing. Still, Java is moving toward being more functional. This is a trend seen across most languages. With lambdas being added to Java 8 and C++11, virtually all languages in the top 15 of most language rankings, with the exception of C, support some level of functional/declarative programming.

C.7 Much More

There is a lot more to Java than what has been presented here. In some ways, Java is fairly simple and straightforward, but in other ways it is not. The language specification for Java is more than three times longer than that for Scala, and Java has significantly more keywords than Scala. Fortunately, there are many resources you can use to help you learn

[2]Yes, the term stream is used in far too many contexts. Do not confuse this with the I/O streams that were covered in chapter 10.

Java that are available online along with many books on the subject. What is more, through exploring the Java API to use with Scala programs, you already have something of a head start on the learning process.

Appendix D

Advanced Scala

Scala is an interesting language in that it has a fairly simple specification with few keywords, yet it allows for remarkable power. This book has avoided going into detail on a number of different features of the language. After all, the goal was to teach you object oriented programming and abstraction, not to be an expert in Scala. The purpose of this appendix is to give you a brief introduction to many of the features that were either glossed over or left out completely. For more details you can search for the Scala language specification. Language specifications are notoriously hard to read so you might want to consider "Programming in Scala, 3rd Edition" by Odersky et al. [10] for a comprehensive and up to date coverage of the Scala language.

D.1 `abstract type` Members

Back in chapter 1 we briefly introduced the concept of a `type` declaration that can be used to give a different name to a type. A straightforward use of this is the ability to give shorter and more meaningful names to tuples that were grouping data. Of course, in most cases this is better done with `case class`es which provide meaningful names on data groups along with stronger type checking.

The main benefit of `type` declarations in Scala comes about when you put `abstract type` declarations in a supertype. You can then use that type through the code and know that it will be bound to something meaningful in the subtypes. You can also provide type bounds on the abstract type using `<:` or `>:` after the name you are giving the `type` and before the type that provides the bounds. Without bounds, you can only use the type in a way that is safe with `Any`.

D.2 implicits

There have been a few points in the book where `implicit` conversions and parameters have been mentioned. The idea of an implicit conversion is that if you call a method on an object that is not defined on that object or if you pass an object to a method that is not of the correct type, Scala will check to see if there is an `implicit` conversion in scope that would make it work. Implicit parameters allow you to have arguments passed to methods without the user having to actually write them out each time.

D.2.1 Basic `implict` Conversions

The `implicit` conversions provide a lot of power to Scala. Both the `Array` and `String` types are basically Java types. They do not really have methods like `map` and `filter`. Despite this, you have been able to call `map` and `filter` and both `Array`s and `String`s. The reason this has worked is that there are `implicit` conversions that are imported by default that convert those types to `ArrayOps` and `StringOps` respectively and those types have the methods we associate with Scala collections.

You can make your own `implicit` conversions by having a "function" in scope that takes the type you want to convert from, returns the type you want to convert to, and includes the `implicit` keyword before `def`. The "function" was put in quotes because in general this is actually a method, but it needs to be brought into scope so that no object name has to be provided.

To illustrate the creation of an `implicit` conversion as well as motivate their existence, consider the following `class` and companion `object`.

```
class Vect3D(val x: Double, val y: Double, val z: Double) {
  def +(v: Vect3D) = Vect3D(x + v.x, y + v.y, z + v.z)
  def -(v: Vect3D) = Vect3D(x - v.x, y - v.y, z - v.z)
  def *(c: Double) = Vect3D(c * x, c * y, c * z)
  def /(c: Double) = Vect3D(c / x, c / y, c / z)
  def dot(v: Vect3D) = x * v.x + y * v.y + z * v.z
  def cross(v: Vect3D) = Vect3D(y * v.z - z * v.y, z * v.x - x * v.z, x * v.y - y *
      v.x)
}

object Vect3D {
  def apply(x: Double, y: Double, z: Double) = {
    new Vect3D(x, y, z)
  }
}
```

This `class` represents vectors in 3-space with some appropriate operations. Most of the things that you would want to do are present here. There is one thing that is missing though. If `v` is a `Vect3D`, you can do `v*3` to scale the components of the vector up by 3. However, you cannot do `3*v`. To see why, remember that `v*3` is seen by Scala as `v.*(3)`. It is calling the `*` method on the `v` instance of the `Vect3D class`. That works because such a method has been defined. On the other hand, 3*v is trying to call a *(Vect3D) method on a number. That method was not part of the standard libraries so it does not work.

The solution to this is an implicit conversion. While Double does not have the needed * method, we can convert the Double to some other type that does. To make this work, we can add the following code to the companion object.

```
implicit def doubleToScaling(c:Double):VectScaler = new VectScaler(c)

class VectScaler(c:Double) {
  def *(v:Vect3D):Vect3D = Vect3D(c*v.x,c*v.y,c*v.z)
}
```

The first line of this is the implicit conversion from Double to the type VectScaler. Below that is the definition of VectScaler. It is a simple class that contains only the method that we need here for multiplication against a Vect3D. To use this, you need to import that method into the local scope. This can be done with import Vect3D._.

To make this process easier, implicit classes were added to Scala. This allows you to leave off the method doubleToScaling completely and just put the implicit keyword in front of the VectScaler class. Note that implicit classes cannot be at the top level in code, so you would likely put VectScaler inside of an object to make this work. The purpose of imports like scalafx.Includes._ was just to bring in a number of implicit conversions to make it easier to write code using that library.

You should notice that what this did in this example was to basically add a new method to the Double class. The ability to make it appear that a type has methods that were not originally part of it has been called “pimping an interface” because you are adding new features to provide functionality that is more to your liking.

D.2.2 Rules and Limits on implicits

While implicit conversions bring a lot of power, they can make code harder to read and understand if they are abused. To help prevent this, Scala has rather strict rules on when implicit conversions will be invoked. First, they must be in scope. If calling the conversion function would require any specifiers before the name, it will not be used as an implicit in that part of the code. Second, implicit conversions are not nested. If it would require two conversions to get from the declared type to a type that will work, Scala will not do that. Similarly, if there is ambiguity where two different implicit conversions could make the call work, neither will be used and you will get an error.

These various rules and limitations combine to make Scala implicits safer. They should not be invoked in ways that you are not expecting or would find overly confusing. What is more, IDE plug-ins have the ability to show you what implicit conversions are being used at different points in your code so that it is even more clear.

D.2.3 implicit Parameters

It is possible to get Scala to implicitly pass extra parameters into a method as well. This can be done for multiple reasons, including passing implicit type conversions. Scala will only implicitly include the last argument list to a method or function. Typically, methods that have implicit parameters are curried. For the last parameter list to be supplied implicitly the argument list must be labeled as implicit and there must be a value matching each of the types in that parameter list which is in scope and labeled as implicit. Note that this implies that you can make implicit val declarations as well as implicit def declarations. We saw the use of an `implicit val` in chapter 8 when we used the Akka system dispatcher as the execution context for our futures.

If you look in the API, for example at `scala.collection.Seq`, you will see many examples of methods that include `implicit` parameters. In the collections library the `implicit` parameter list is almost always a single parameter of the type `CanBuildFrom`.

D.2.4 The Meaning of <%

When you put `<%` as a bound on a type parameter, Scala switches it to a normal type bound and adds a curried `implicit` parameter to the method. That extra parameter will be filled in with an appropriate `implicit` conversion from the current scope if one is available.

D.3 `sealed classes`

Many functional languages have a construct called algebraic types. These allow you to say that a type can be represented by a small number of different possible structures/values. If all you want are separate values, an `Enumeration` is a good way to get this functionality in Scala. When the different possibilities have different structure to them, the normal object-oriented way to represent that is with inheritance. The one shortcoming of inheritance in this case is that anyone can add other subtypes at a later date, and that might be a problem. If you make a `class final`, no subtypes are allowed. In this case you want to be able to limit the subtypes to a specific number.

This mechanism is provided in Scala with the `sealed` keyword. A `sealed class` can be extended, but all the subtypes have to appear in the same file as the original declaration. That gives you much stronger control over what subtypes exist.

There is another benefit to a class being `sealed` in the form of error checking for `match` expressions. If you do a `match` on an expression whose type is a `sealed class`, Scala will produce syntax errors if you do not have `case`s that can match all subtypes. This completeness checking makes it so that you should not get `MatchException`s when a type comes in that you were not expecting.

The standard usage of a `sealed class` is to use a `sealed abstract class` or a `sealed trait` that is completely empty at the top of the hierarchy, then have various `case class`es or `case objects` which extend the first `class/trait`. The use of `case` types makes it easier to take full advantage of the pattern matching, but does require that your types be immutable.

D.4 Covariant, Contravariant, and Invariant Types

Chapter 12 brought up the concept of covariant types in relation to immutable linked lists. This deals with how Scala treats type parameters when it comes to determining subtype relationships for the main type. Imagine you have the following declarations.

```
class Stuff[A](val a:A)
class MoreStuff[A](a:A, val b:A) extends Stuff[A](a)
```

You can see from this that `MoreStuff` is a subtype of `Stuff`. However, this ignores the fact that `Stuff` and `MoreStuff` are not well defined types on their own. In reality, you have to provide a type parameter to have a well defined type. In this case we can say that `MoreStuff[String]` is a subtype of `Stuff[String]`. The question is, is `MoreStuff[String]` and subtype of `Stuff[AnyRef]`? You might be tempted to say that it is, but as written, it is not. If you try to pass an instance of `MoreStuff[String]` into a function/method that accepts `Stuff[AnyRef]`, Scala will complain. If you put the types above into the REPL, you can quickly test that with the following.

```
scala> def hasNull(s:Stuff[AnyRef]) = s.a==null
hasNull: (s: Stuff[AnyRef])Boolean

scala> val s = new Stuff("Hi")
s: Stuff[String] = Stuff@c8d310f

scala> hasNull(s)
<console>:11: error: type mismatch;
 found   : Stuff[String]
 required: Stuff[AnyRef]
Note: String <: AnyRef, but class Stuff is invariant in type A.
You may wish to define A as +A instead. (SLS 4.5)
             hasNull(s)
                     ^

scala> val ms = new MoreStuff("hi","mom")
ms: MoreStuff[String] = MoreStuff@35e20aca

scala> hasNull(ms)
<console>:12: error: type mismatch;
 found   : MoreStuff[String]
 required: Stuff[AnyRef]
Note: String <: AnyRef, but class Stuff is invariant in type A.
You may wish to define A as +A instead. (SLS 4.5)
             hasNull(ms)
                     ^
```

This happens because the default behavior is for type parameters to be invariant. That means that types with different type parameters are always unrelated, regardless of the inheritance relationships between the main types or the type parameters.

The type `MoreStuff` does not even have to be involved. You might expect that `Stuff[String]` should be a subtype of `Stuff[AnyRef]`. However, if `A` is invariant that is not the case either, as shown here.

```
scala> val s = new Stuff("Hi")
s: Stuff[String] = Stuff@c8d310f

scala> hasNull(s)
<console>:11: error: type mismatch;
 found   : Stuff[String]
 required: Stuff[AnyRef]
Note: String <: AnyRef, but class Stuff is invariant in type A.
You may wish to define A as +A instead. (SLS 4.5)
             hasNull(s)
                     ^
```

As the error message above indicates, this could be fixed by using the type parameter `+A` instead of just `A`. That makes the type parameter covariant. When the type parameter is covariant, the full types will be subtypes if the parameters are subtypes and the main types are the same or subtypes. In this particular example just adding a `+` will make this work.

```
class Stuff[+A](val a:A)
```

With this one character, everything else from above will work.

So you might wonder why invariance is the default instead of covariance. To make that clear, consider the following change to `Stuff` and a function that works on it.

```
class Stuff[A](var a:A)
def changeStuff(s:Stuff[AnyRef]) = s.a = List(1,2,3)
```

Now that the `a` member is a `var`, things are a bit different. Consider what happens if Scala let you pass in an instance of `Stuff[String]`. In that object, `a` has to be a `String`. This function would try to assign to it a `List[Int]`, an operation that should clearly fail. The switch to a `var` makes covariance for this `class` unsafe. Scala can even tell you that.

```
class Stuff[+A](var a:A)
<console>:7: error: covariant type A occurs in contravariant position in type A of
    value a_=
      class Stuff[+A](var a:A)
           ^
```

As the error message indicates, it is the assignment method into the `var` that is really causing the problem. This also shows you that Scala can tell if it is safe for a type to be covariant or contravariant. It is always safe to be invariant. That is why invariant is the default. The algorithm that Scala uses to determine what is safe is beyond the scope of this appendix, but a quick rule of thumb is that when the type is passed into a method that is a contravariant position. When they are returned from methods that is a covariant position. For a type parameter to be either covariant or contravariant, it cannot occur in a position of the other end. Since a public `var` implicitly makes one of each, any type of a public `var` has to be invariant.

Contravariance is the opposite of covariance. When the type parameter is `-A`, then `Stuff[Type1]` is a subtype of `Stuff[Type2]` only if `Type2` is a subtype of `Type1`. This probably seems counter intuitive, but there are times when it is very useful. The most obvious example is the `Function1` type in the Scala library. This type represents a function that takes one argument and returns a value.

```
trait Function1 [-T1, +R]
```

This is basically what we have been writing as `(T1) => R`. Here the return type is covariant and the input type is contravariant. To understand why this is, imagine a place where you say you need a function of the type `(Seq[Int]) => Seq[Int]`. What other types would work there? For the return type, anything that returns a subtype of `Seq[Int]` will work fine. For the the input type, you can use any function that takes a supertype of `Seq[Int]` will work because you know it will be safe to pass it a `Seq[Int]`.

D.5 Extractors

Pattern matching has been used in many places in this book and hopefully you have become aware of the power that it can provide. One thing you might have noticed that was missing was the ability for you to make your own `class`es capable of doing pattern matching. You can make your own types that do pattern matching by writing extractors. An extractor is an `object` that includes either the `unapply` or the `unapplySeq` methods. Most of the time you will do this in a companion `object` that also has an `apply` method so that the usage resembles that of `case class`es.

D.5.1 `unapply`

You should put an `unapply` method in an `object` when you know exactly how many fields should be pulled out of the pattern. The parameter for `unapply` is the value that would be matched against the pattern. It should have a type that is appropriate for what you want to be working with. The result type of `unapply` depends on how many fields it should give back.

- No values results in `Boolean`. It is `true` if there was a match and `false` if there was not.
- One value of type `A` results in `Option[A]`. The result will be `None` if there was a match or `Some` with the value of the result if there was a match.
- For two or more values the result type should be an `Option` of a tuple with the proper number of types. No match will result in `None` while a match will result in `Some` of a tuple filled with the proper values.

To illustrate this, we could put an `unapply` method in the `Vect3D` companion `object` that was used earlier to show the use of `implicit` conversions.

```
def unapply(str: String): Option[(Double, Double, Double)] = {
  val s = str.trim
  if (!s.startsWith("(") || !s.endsWith(")")) None
  else {
    val parts = s.substring(1, s.length - 1).split(",")
    if (parts.length != 3) None
    else try {
      Some(parts(0).trim.toDouble, parts(1).trim.toDouble, parts(2).trim.toDouble)
    } catch {
      case e: NumberFormatException => None
    }
  }
}
```

This example will pattern match on a `String` that should have three numbers, separated by commas, inside of parentheses. It gives back three values that are the numbers of the three components. A simple usage of this might look like the following.

```
"(1,4.5,83)" match {
  case Vect3D(x,y,z) => println(x+" "+y+" "+z)
}
```

Note that while you can make `unapply` methods that take different inputs and return whatever, you cannot make methods that differ only in their return type. So in this example, you cannot also have a second version that creates a `Vect3D` instance. You have to pick between the two.

D.5.2 `unapplySeq`

There are also situations where the number of fields in the pattern match can vary from one match to another. Examples of this include matching on `List` or `Array`, or the matching or regular expressions where each group defines a field. To get this functionality, you need to implement the `unapplySeq` method in your `object`. The return type of this method can be an `Option[Seq[A]]`, for whatever type `A` you want. It can also have an `Option` of a tuple where the last element of the tuple is a `Seq[A]`. The first version could match any number of things, but they will all have the same type. The second version can force a certain number of fields of different types at the beginning of the pattern followed by the sequence of unknown length with a single type.

D.6 `lazy` Member Data

`val` declarations in `class`, `trait`, and `object` declarations can be labeled as `lazy`. This changes when the value of that member data will be set. Normally member data is initialized when the object is instantiated. `lazy` members are not initialized until they are used.

This is particularly helpful if the the value requires a significant calculation or requires significant resources such as memory. In that situation, you do not want to go through the effort of making the value unless it is actually needed. If the object can go through its life without ever needing that member, setting it to be `lazy` will make it so those resources are never consumed.

Note that you have to use caution combining `lazy` declarations with multithreading. The mechanism used for determining whether a `lazy val` has been initialized yet is not thread safe and can cause a race condition if the value is first accessed by multiple threads.

D.7 `scala.collection.immutable.Stream`

There is another type in the Scala collections that was not used in this book, the `scala.collection.immutable.Stream` type.[1] `Stream` is a subtype of `Seq` and you access members of it just like you would any other `Seq` using an integer index. What makes `Stream` interesting is that it is a lazy sequence. The last section described how the `lazy` keyword modifies members so that they are not calculated until they are needed. This same idea can be applied to a sequence as well. The values at different indexes are not calculated until they are needed. Once calculated, they are remembered so they do not have to be calculated again, but if you never use an index, it will never be calculated.

At first glance this might seem like just an interesting way to save memory. However,

[1]Yes, another use of the word stream. It should not to be confused with the I/O streams of the Java 8 functional streams.

it can be used for more than that. The fact that the values of a `Stream` are not absolutely calculated and stored means that it is possible to make `Stream`s of infinite length. The Scala API includes an example that builds an infinite `Stream` of prime numbers. If you look at that you will notice that it uses the method `Stream.cons` instead of the `::` method to add elements to the front of the stream. It also adds elements using a recursive definition of everything after the `head`. This is required for an infinite `Stream` because the `tail` of the `Stream` never really exists in full.

If you have a block of code that is going to be working with `Stream`s a lot, you should include `import Stream._`. One of the useful things this brings into scope is an `implicit` conversion from `Stream` to `Stream.ConsWrapper`. This will allow you to use the `#::` and `#:::` operations to cons and concatenate `Stream`s. This allows you to write the following code to define Fibonacci numbers.[2]

```
  import Stream._
  def fibFrom(a: BigInt, b: BigInt): Stream[BigInt] = a #:: fibFrom(b, a + b)
```

To test this you can do the following.

```
fibFrom(1, 1).take(100).toList
```

You can use this type of approach generally for any place that you need to define something that is accurately represented by an infinite sequence. It is worth noting though that you have to be careful with these. Many of the methods in the collection API keep going until they get to the end of the collection. Needless to say, that is not a good thing to do with an infinite collection.

D.8 Nested Types

We have seen quite a few places in this book where `class`es are written inside of other `class`es. What we did not generally do is refer to these types from outside of the outer `class`. This was in large part because none of the code we wrote really needed to do this. In fact, in most places where we did nest types in this way, we made the nested type `private` so that it could not be seen from outside. For example, the `Node` type in a linked list should not be known to outside code as that is an implementation detail. However, there are instances where those nested types need to be used by outside code, and there are some significant details to it that are worth mentioning.

When you put a `class` or `trait` inside of another `class` or `trait`, every instance of that type gets its own version. If you use standard "dot" notation to refer to the type, it needs to be on an object, not the `class`, and the type you get is specific to that object. You can also refer to the general type across all instances with the `class` name followed by a `#`. The following code demonstrates this.

```
class NestedClasses(x: Double, y: Double) {
  val root: Node = new Plus(new Num(x), new Num(y))

  trait Node {
    def eval: Double
```

[2]This code is presented using `Int` in the Scala collections API description given at `http://www.scala-lang.org/`.

```
  }

  class Plus(left: Node, right: Node) extends Node {
    def eval = left.eval + right.eval
  }

  class Num(n: Double) extends Node {
    def eval = n
  }
}

object NestedClass {
  def main(args: Array[String]) {
    val a = new NestedClasses(4, 5)
    val b = new NestedClasses(8.7, 9.3)

    a.root match {
      case n: a.Num => println("A number")
      case n: a.Node => println("Not a number")
    }

    def evalNode(n: NestedClasses#Node) = n.eval
    def evalNodeA(n: a.Node) = n.eval
    def evalNodeB(n: b.Node) = n.eval

    println(evalNode(a.root))
    println(evalNodeA(a.root))
    println(evalNodeB(a.root)) // This is a type mismatch.
    println(evalNode(b.root))
    println(evalNodeA(b.root)) // This is a type mismatch.
    println(evalNodeB(b.root))
  }
}
```

The `class` at the top contains types for a little expression tree. This is greatly simplified for this example. In the companion `object` is a `main` method that makes two instances of the top `class` called `a` and `b`. A `match` on `a.root` shows how you can refer to types inside of the object `a`. If you try to put something in a `case` that is not one of the subtypes of `Node` inside of `a`, Scala gives you an error saying it cannot be that type.

After the `match` are three `def` declarations of local functions. The first is written to take a general `Node` from inside of any instance of `NestedClasses`. The other two are specific to `a` and `b`. There are six calls where these three methods are invoked on the `root` objects of both `a` and `b`. Two of these result in errors as `a.root` is a mismatch with `b.Node` and `b.root` is a mismatch with `a.Node`.

D.9 Self Types

There are times when you will create a `trait` which is intended to be mixed in with some other `trait` or `class`. Occasionally, inside of the mix-in `trait`, you will need to make reference to members of the type it is being mixed in to. Conceptually this makes sense as you know that this will always be safe. However, the compiler does not know that this will be safe. To get around this, you have to have a way to tell the compiler that the particular trait will only be used in situations where it is part of some other type. This can be done using a self type.

The syntax of a self type is as follows.

```
trait A {
  this: B =>
  ...
}
```

This tells Scala that any object that includes the `trait A` must include the type `B` above it in the resolution linearization. With this in place, the body of `A` can include references to members of `B`.

One common place that this is used in Scala is in something called the Cake pattern. Interested readers can use that term as a jumping off point for searching for more information.

D.10 Structural Types

There are situations where you do not want to force the use of a subtype of any particular type. Instead, you want the code to be able to use any type as long as it has the methods you need to call. This is where structural types come into play. A structural type is written as curly braces with method signatures separated by semicolons. These are particularly useful when you want to be able to use code written by other people and you cannot force that code to have a certain type, but it does use method names that are consistent with your own code. In that situation, you can specify a type bound to a structural type.

A simple example of this would be the following code which will read all the integer values from any source that has the methods `hasNextInt` and `nextInt`.

```
def structRead[A <: { def hasNextInt(): Boolean; def nextInt(): Int }](source:
    A): List[Int] = {
  var buf = List[Int]()
  while (source.hasNextInt) buf ::= source.nextInt
  buf.reverse
}
```

An example of such a type is `java.util.Scanner`. The use of a structural type makes it so that this code will work with any type that has those two methods, regardless of what it inherits from.

It is worth noting that you cannot normally use structural types with sorts. The reason is that structural types cannot be recursive the way normal type bounds can. So you cannot use a structural type to say that you want a type which can be compared to itself.

As a general rule, you should avoid using structural types unless you really need to. They are inefficient, and most of the time there are better ways of accomplishing what you want to do with them in a manner that uses normal types.

D.11 trait Initialization

At the time that this book was written, you cannot pass arguments to traits the way that you can with classes. There are plans to change this, which will be nice because the inability to pass in arguments can cause problems related to the order of initialization of values when you inherit from traits. Clearly this is not always a problem. Over the course of the book we had many situations where we used inheritance from traits and this was never an issue. The problem arises when there are abstract vals or vars.[3]

To help you understand this, let us go back to our standard vector example, but this time we will make it be a 2D vector trait and define an extra val as the magnitude.

```
trait Vect2D {
  val x: Double
  val y: Double
  val mag = math.sqrt(x * x + y * y)
}

object Vect2DTester extends App {
  val v = new Vect2D {
    val x = 3.0
    val y = 4.0
  }

  println(v.mag)
}
```

In the test code we make a new instance of Vect2D using the anonymous inner class syntax. Looking at this code, you probably expect the print statement at the end to print "5.0". Unfortunately, it does not. If you run this, it prints "0.0". The reason deals with the order of initialization. When you pass arguments into a class, the values you pass in are set before any code in the class is run. On the other hand, when you define abstract values as is done here, the values are assigned after the code in the trait has been executed. So the value of mag is set using default values for x and y, which happen to be zero.

An even more extreme example would be the following.

```
trait AbstractString {
  val str: String
  val strLength = str.length
}

...

  val astr = new AbstractString {
    val str = "hi"
  }
```

Just running this causes a NullPointerException. This is because the default value for any subtype of AnyRef is null. So at the point where strLength is initialized, the value of str is null and we try to call length on it.

There are two ways to get around this. One is to use lazy vals as were described above.

[3]Technically this issue could arise with classes as well, but when using a class you typically would pass in arguments in this type of situation instead of using abstract data members.

In our two examples, if `mag` and `strLength` are declared to be `lazy`, the problems shown in these examples go away because we do not reference either `mag` or `strLength` until after the abstract members have been initialized. Even this is not always sufficient. In that case we have to use pre-initialized fields. The syntax for this puts the initialization in a block of code before the reference to the `trait`. Here are examples showing how we can use it with an anonymous inner class as well as with the declaration of a normal `class`.

```
  val v2 = new {
    val x = 3.0
    val y = 4.0
  } with Vect2D

  println(v2.mag)

  class Vect2DClass(_x: Double, _y: Double) extends {
    val x = _x
    val y = _y
  } with Vect2D {
    // Other stuff we want in the class
  }

  val v3 = new Vect2DClass(3,4)

  println(v3.mag)
```

The two print statements in these examples both print "5.0", which is the behavior that we want.

D.12 Value Classes

Normally, every `class` that you create becomes a subtype of `AnyRef`. Associated with this are some factors such as they are always allocated on the heap as full objects. There is overhead to doing this that can be nice to avoid at times. The subtypes of `AnyVal`, such as `Int` and `Double`, can avoid this overhead for many usages. Value classes are a way that you can create types that become subtypes of `AnyVal` and potentially avoid the overhead of object creation, but there are lots of restrictions on when they can be used.

To make a value class, simply say that your `class` extends `AnyVal`. It must also take a single argument that is a `val` and it can only contain `def`s. Here are two examples and a short application showing their usage.

```
class Mile(val dist: Double) extends AnyVal {
  override def toString = dist+" mile"+(if (dist != 1.0) "s" else "")
  def +(m: Mile) = new Mile(dist + m.dist)
  def toKilometers = new Kilometer(dist * 1.60934)
}

class Kilometer(val dist: Double) extends AnyVal {
  override def toString = dist+" km"
  def +(km: Kilometer) = new Kilometer(dist + km.dist)
  def toMiles = new Mile(dist * 0.621371)
}
```

```
object Distance extends App {
  val marathon = new Mile(26.2)
  val fivek = new Kilometer(5)

  println(marathon + fivek.toMiles)
}
```

Note that even though the code says `new Mile` and `new Kilometer`, this usage will not force the creation of an instance object. Instead, they will compile to primitive doubles in the JVM bytecode. There are a number of other restrictions on value classes, such as the fact that they can only inherit from types that contain only `def` declarations.

This example shows a potential usage of value types. They can add type safety to values that have units without adding extra overhead to processing them. For example, if you just use a `Double` to store your distances and you have one value in units of miles and another in units of kilometers, you can simply add them together and the compiler will let you, even though the result is meaningless. In this code, you cannot add `marathon` and `fivek` together unless you include a conversion on one of them so that the types match.

Some of the restrictions on value classes could be relaxed in a future version of Scala. This is in part because there are plans to add value classes to Java in Java 10, and those value classes would be able to have more than one value in them, opening the door for Scala to do the same in an efficient way on the JVM. This type of feature has already been added to other Scala implementations such as the Scala-native project, which compiles Scala code to the LLVM platform instead of JVM. Once those restrictions have been listed, it is likely that types like the `Vect2D` and `Vect3D` would be ideally implemented at value classes.

D.13 Macros

Beginning with Scala 2.10 the Scala language has support for macros. A macro is a piece of code that can alter the structure of code that it is used with. During the compiling of a program, when a macro is encountered, that macro has the ability to replace the macro code with alternate code, typically something more complex. In Scala, the macros are also type safe.

You can create macros in much the same way that you create standard methods using `def`, but after the equal sign you put the keyword `macro` and the name of a method that actually does the macro expansion. The details of writing macros is well beyond the scope of this appendix, but they are used in a number of features that we have used in the book. In particular, the string interpolation feature of Scala is implemented using macros, as is the printf function. This is significant because it allows those constructs to be type safe. Without macros, the contents of a `String` literal are just characters that the compiler does not really check. For example, the compiler does not normally care if you misspell something in a `String` literal. However, the macro expansion of an interpolated `String` includes checking to make sure that names are declared as well as checks that types are used appropriately.

D.14 Type-Level Programming

Scala has a powerful type system, and at many points in this book we have utilized the capabilities of the type system to make our code more expressive while maintaining type safety. However, it turns out that we have not really even touched on the full power of the type system. The Scala type system is formally "Turing Complete". You might recall from the discussion in section 17.1.4 that recursively enumerable grammar are also Turing Complete. This means that any calculation that is computable can be done using that system. What this means for the Scala type system is that technically you could write all of your programs so that they did the calculations using Scala types and all the work was done by the compiler and all the program did was print out the answer. To get a feel for what type-level programming looks like in Scala, we refer you to the following website, `https://apocalisp.wordpress.com/2010/06/08/type-level-programming-in-scala/`.

It should not take long to convince yourself that type-level programming is much messier and less expressive than normal programming in Scala and that you really do not want to write all of your programs that way. However, the ability to do type-level programming is still beneficial. There are libraries like Shapeless (`https://github.com/milessabin/shapeless`) that use type level programming to enhance the language and add what are called dependent typing capabilities.

D.15 Making Executable JAR Files

Scala programs compile to bytecode, not an actual executable. If you want to be able to give your program to someone else for them to run, you need to package it in a proper form. For code compiled to Java bytecode, that format is an executable JAR file. You can make these using the `jar` command from the JDK. Complete instructions can be found on the book's website as the ideal set of steps is likely to change over time.