Chapter 12
Packages & Encapsulation

12.1 Introduction

This chapter discusses the encapsulation and packaging features of Scala. The concept of packages is discussed, along with some concrete examples. It then illustrates how the encapsulation facilities can allow quite fine-grained control over the visibility of the elements of your programs.

12.2 Packages

You can bring a set of related classes together in a single compilation unit by defining them all within one file. By default, this creates an implicit (unnamed) package; classes can access variables and methods that are only visible in the current package. However, only one of the classes can be publicly visible (the class with the same name as the file). A much better approach is to group the classes together into an explicit, named package.

Packages are encapsulated units that can possess classes, interfaces and sub-packages. Packages are extremely useful:

- They allow you to associate related classes and interfaces.
- They resolve naming problems that would otherwise cause confusion.
- They allow some privacy for classes, methods and variables that should not be visible outside the package. You can provide a level of encapsulation such that only those elements that are intended to be public can be accessed from outside the package.

The Scala libraries provide a number of packages, some of which are inherited from the underlying Java runtime. In general, you use these packages as the basis of your programs.

12.2.1 Declaring a Package

An explicit package is defined by the `package` keyword at the start of the file in which one or more classes (or interfaces) are defined:

```
package benchmarks
package com.jjh.transport
```

Package names should be unique to ensure that there are no name conflicts. Scala does not require, although it is common to find that a naming convention is adopted across projects. This naming convention is derived from the Java world in which a package name is made up of a number of components separated by a full stop. The start of such a name is often your organisations domain in reverse; this ensures uniqueness across all software systems.

The package name components actually correspond to the location of the files. Thus if the files in a particular package are in a directory called benchmarks, within a directory called tests, then the package name is given as:

```
package tests.benchmarks
```

Notice that this assumes that all files associated with a single package are in the same directory. It also assumes that files in a separate package will be in a different directory. Any number of files can become part of a package, however, any one file can only specify a single package. Also note that any number of directories can make up the package (particularly if they are arranged in different jar files).

All components in the package name are relative to the contents of the `CLASSPATH` variable. This environment variable tells the Scala compiler where to start looking for class definitions. Thus, if the `CLASSPATH` variable is set to `C:\jjh\Scala` then the following path is searched for the elements of the package:

```
c:\jjh\Scala\tests\benchmarks
```

All the files associated with the `tests.benchmarks` package should be in the benchmarks directory.

12.2.2 Additional Package Definitions Options

12.2.2.1 Package per File

The simplest way to define a package is to use a single package statement at the start of a file. It must be the first line of Scala (other than any comments in the file). It defines the whole contents of the file as being part of that package:

package com.jeh.transport

12.2.2.2 Chained Package Definitions

A further package definition approach is Scala is what is called *chaining* package definitions together. This allows multiple package declarations to be specified with subsequent package declarations being chained to the earlier declaration. For example, the following defines a package x.y containing the class Ship:

```
package x
package y

class Ship {

}
```

The style presented here indicates how packaging chaining can be used with the package declaration at the start of a file. In terms of package chaining, it is a style that you should be familiar with as you may encounter it in examples presented on the web. However it is not a style that is generally used. The style of nested packages which leads to package name chaining is more common, although the most common from of package is a single one line declaration at the start of the file.

12.2.2.3 Nested Package Definoitions

Scala packages can also be nested one inside another. In this case the scope of one packaged needs to be indicated via the presence of curly braces '{…}'. For example,

```
package test {
...
}
```

Actually the curly braces can always be used with a package definition it is just that if they are omitted it is assumed that the whole of the file represents the contents of the same package.

Curley braces are normally used to when defining one or more nested packages so that the scoep of one package can be represented. For example:

```
package test {
  ...
  package demo {
    ...
  }
  ...
}
```

The above dots imply that there are members defined in the package test and members defined in the package test.demo. It is also clear from this that in Scala you can therefore have more than one package in a single file. In fact you have multiple packages for example:

```
package test {
  ...
  package demo {
    ...
  }
  ...
 package util{
    ...
  }
  ...
  }
```

The above example would have three packages in a single file, these packages would be:

- Package test
- Package test.demo
- Package test.util

Note that we could further nest packages so that package demo could have a further nested package print:

```
package test {
  ...
  package demo {
    ...
    package print {
      ...
    }
  }
  ...
 package util {
    ...
  }
  ...

}
```

As a concrete example of this consider the following listing:

```
package com.jeh.transport {

  package personal {
    class Bike
  }

  case class Car

  package group {
    class TaxiFleet {
      val c = Car()
      }
    }
  }
```

This example defines the package com.jeh.transport as the top level package (note that it is perfectly legal to name a package with multiple elements and then to provide nested packages that build on that namespace. The top level package contains two nested packages personal and group. The full name of these packages is:

- com.jeh.transport.personal
- com.jeh.transport.group

If you were importing these packages to use in your own code then these are the names that you would have to use.

An interesting set of questions to ask is what is the scope or visibility of the classes:

- Car defined in com.jeh.transport
- Bike defined in com.jeh.transport.personal
- TextFleet defined in com.jeh.transport.personal

The answers are that:

- the Car class is directly visible in com.jeh.transport and in the nested packages personal and group. This is why it can be directly referenced within the class TaxiFleet.
- The Bike is only visible directly within the package personal.
- The TaxiFleet is only directly visible within the nested package group.

From this we can see one of the key aspects of packages—helping to organise our code elements (and to restrict the default namespaces of such elements).

However, this approach is not without its problems and it can actually led to namespace issues of its own. For example, consider the following listing:

```
package engine {
  class Petrol1
}

package family {
  package economy {
    package engine {
     class Petrol2
    }
    class Control {
      val b1 = new engine.Petrol2
      val b2 = new economy.engine.Petrol2
      val b3 = new family.engine.Petrol3
      // val b4 = new engine.Petrol1
      val b5 = new _root_.engine.Petrol1
    }
  }
  package engine {
    class Petrol3
  }
}
```

This example has the following packages with the following contents:

- Package engine with the class Petrol1
- Package family with two nested packages economy and engine
- Package family.economy with the class Control and a nested package engine
- Package family.economy.engine with the class Petrol2
- Package family.engine.Petrol3

All this looks fine except when you realise that currently there is no way for the commented out line

val b4 = new engine.Petrol1

to compile? Why is this? It is because in Scala when you reference a class or a package Scala always attempts to din the most local version of that class or package. For the class Control the package engine which is *nearest* to it in terms of=name space is the package engine defined within the package family and as it is a nested package within family as is the package economy, there is no need for code within either package to have to mention the root package family in order to access each other (it is implied by their nested status). However, as there is an external package called engine also present this means that there is a name conflict between the two packages engine.

To get around this problem, Scala provides a special root package reference which can be used to indicate that you do not want to use the locally scoped package but to start at the root location of all package names and find a package from there. This root package reference is referred to by pre-fixing a package name with '_root_', for example:

```
val b5 = new _root_.engine.Petrol1
```

This ensures that the search for the package engine starts at the root of all packages rather than looking locally. This approach works as *root* is essentially a special package that pre-fixes all packages.

12.2.3 An Example Package

As an example, the files for the `com.jeh.lights` package are stored within a directory called `lights`, within a directory called `jeh`, within the `com` directory. The directory contains three classes that make up the contents of the `lights` package: `Light`, `WhiteLight` and `ColoredLight`. The header for the `Light.Scala` file contains the following code:

```
package com.jeh.lights

abstract class Light {

}
```

The `WhiteLight.Scala` and `ColoredLight.Scala` files are similar, for example:

```
package com.jeh.lights

class ColouredLight extends Light { }
```

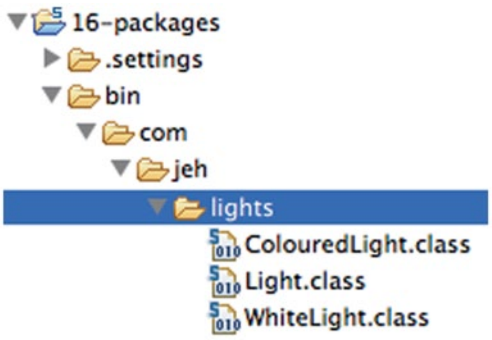
And

```
package com.jeh.lights

class WhiteLight extends Light { }
```

Note in the above example we have placed each class in a separate file, however we could have defined all three classes in the same source file and this would have produced the same set of .class files as are described below.

The directory containing the compiled (byte code) version of the `lights` package is shown below:



```
16-packages
  .settings
  bin
    com
      jeh
        lights
          ColouredLight.class
          Light.class
          WhiteLight.class
```

The `CLASSPATH` variable (set up by the Scala IDE) includes the path `bin` directory of the current project, so the package specification, `com.jeh.lights`, completely specifies the location of the byte code files.

12.2.4 Accessing Package Elements

There are two ways to access an element of a package. One is to name the element in the package fully; this is referred to as the fully qualified class name. For example, we can specify the `Light` abstract class by giving its full designation:

```
class NewLight extends com.jeh.lights.Light {...}
```

This tells the Scala compiler exactly where to look for the definition of the class `Light`. However, this is laborious if we refer to the `Light` class a number of times.

The alternative is to import the `Light` class, which makes it available to the package within which we are currently working:

```
import com.jeh.lights.Light
class NewLight extends Light
```

However, in some situations, we wish to import a large number of elements from another package. Rather than generate a long list of import statements, we can import all the elements of a package at once using the '_' wildcard. For example,

```
import com.jeh.lights._
```

This imports all the elements of the `com.jeh.lights` package into the current package. Notice that this can slow down the compilation time (although it has no effect on the run time performance). Also note that this only imports the contents of the `com.jeh.lights` package—it has no affect on any sub packages of lights. Also note that if you are a Java programmer that the wild card here is '_' and not '*' as it is in Java. Also note that we do not include the (optional) ';' statement terminator in Scala—you can use the ';' to terminate both the package declaration and the import statements, it is just that it is considered superfluous and thus not good style.

It is also possible to import all the methods or functions defined on a type using the name of the type followed by the '_' wild card, for example:

```
import transport.Car._  // import all members from Car
```

To summarize then it is possible to import the whole contents of a package, a single type from a package, use an alias with a type and to import the functionality for a given type.

12.2.5 An Example of Using a Package

The `lights` package described above has been used within a code outside the package. This application defined in the com.jeh.test packages uses the `ColoredLight` class. It therefore imports it into the current package. For example:

```
package com.jeh.test

import com.jeh.lights.ColouredLight

object LightTest extends App {
  val l = new ColouredLight()
  println(l)
}
```

Notice that we have chosen to import the ColoredLight class explicitly rather than the whole package. Also note that we can import any number of classes, objects, traits, types etc. as required into a single file but that these imports are only in scope for the current file.

12.3 Import Options

Scala actually has a wider set of import options than Java. In Java an import can only be specified at the top of a file after any (optiona0 package declaration and before any other Java declarations (such as a class or interface). In Scala an import can appear anywhere and affects the scope within which it was specified. Thus imports can appear in a:

- Package
- Class
- Method or function
- Package object

For example, the following example illustrates importing a set of functions define don the object utill.PrintUtil into a method so that they can be accessed directly within that method (but only that method):

```
package banking

class Bank {
  def print(acc: Account) {
    import util.PrintUtils._
    printAccount(acc)
  }
}
```

PrintUtil is a singleton object defined in the package util, for example:

```
package util

object PrintAccount {
  def printAccount(acc: Account) {
        println(acc.name + ": " + number)
  }
}
```

Scala also allows you to import more than just classes, objects or traits. You can import the methods on instances of a given class. For example, in the following example the method printShip *imports* the methods defined on the parameter car so that it does not need to prefix model and spec with car (e.g. car.model and car.spec) thus making the code simpler:

```
class Car(val model: String, val spec: String)

object Test extends App {

  // Main behaviour
  val c = new Car("Ford", "SE")
  printCar(c)

  // Support method
  def printCar(car: Car) {
    import car._
    println(model + ": " + spec)
  }
}
```

12.4 Additional Import Features

It is also possible to provide an alias as part of an import, for example:

```
import transport.{Car => Audi}
```

This indicates that the type transport.Car will be alias to (and accessible via) Audi in the current context (e.g. the current file).

It is also possible to indicate what should not be imported, for example:

import transport.{Car=>_, _}

This import indicates that everything should be imported from the package transport with the exception of Car. This is because the first part of the contents of the curly brackets '{…}' indicates what not to import, e.g. Car=>_ and the second part is the wild card that indicates what should be imported (that is the second '_').

A further way in which the types to import can be specified is via the curly bracket '{..}' notation. This can be used to reduce the number of import statements when several (but not all) of the types in a particular package should be imported. This is written in the following way:

```
import java.sql.{Connection, DriverManager, ResultSet}
```

Note that this statement imports the Connection, DriverManager and ResultSet types from the java.sql package.

12.5 Package Objects

A package can also (optionally) have a *package object* associated with it. A package object is an object that is part of the package (and has the same name as the package) that can be used to hold utility functions or methods. Any members defined in the package object are considered to be top-level members of the package and can be accessed by other members of the package directly.

As an example of a package object consider the following listing:

```
package com.jeh

package object banking {
  def printAccount(acc: Account) {
    import acc._
    println(name + ": " + number)
  }
}
```

This defines a package object for the package com.jeh.banking. Note that it is defined by a keyword for the package level above banking (com.jeh) with a package object definitions for the banking element of the package. This banking package object defines a single utility method printAccount that can be used by any other members of the com.jeh.banking package to print out bank account information. For example:

```
package com.jeh.banking

case class Account(val name: String, val number: Int)

object TestAccount extends App {
  val acc = Account("John", 1234)
  printAccount(acc)
}
```

You can also use the printAccount method in other packages by importing it. For example the following code is in a separate package com.jeh.test. It imports both the Account case class and the printAccount method from the com.jeh.banking package. Notice that from this you cannot see that com.jeh.banking is both a package and a package object. We can then use the Account class to create an account instance and print its details via printAccount (the utility methods define don the package object).

```
package com.jeh.test

import com.jeh.banking.printAccount
import com.jeh.banking.Account

object Test extends App {
   val acc = Account("John", 123)
   printAccount(acc)
}
```

12.6 Key Scala Packages

There are very many packages in Scala but the core or central ones are:

- *scala*—the core types
- *scala.collection* provide basis of the Scala collections (data structures) frameworks
- *scala.collection.immutable* provides the definitions for the immutable versions of the collection classes in Scala
- *scala.collection.mutable* provides definitions for the mutable versions of the collection classes in Scala.
- *scala.actors* provides the actor based concurrency types
- *scala.io* provides for input and output type definitions
- *scala.math* which provides basic mathematical functions and additional numeric types
- *scala.sys* which provides types for interacting with other processes and the operating system.
- *scala.util.matching* which provides pattern matching in text using regular expressions.
- *scala.xml* containing types to be used when parsing, manipulating and serializing XML structures.

12.7 Default Imports

There are also a set of default imports that are imported into every Scala file, these are:

- The java.lang package
- The scala package
- The `Predef` object.

The core java.lang package is imported as it provides some of the basic concepts that underpin the Scala (and Java) runtime such as the definition of a String.

The scala package contains definitions for the core Scala types and as such it is always available in any Scala code without the need for an explicit import.

The `Predef` object in Scala provides type aliases for commonly used Scala types (such as the immutable collection classes), some simple functions for Console I/O (such as println), basic assertions (such as require) and some implicit conversion routines. The inclusion of the `Predef` object reduces the amount of explicit code that needs to be written in Scala.

12.8 Encapsulation

In Scala, you have a great deal of control over how much encapsulation is imposed on a class, a trait and an object. You achieve it by applying *modifiers* to classes, objects and trait properties, methods and functions. Some of these modifiers refer to the concept of a package and others to the type itself.

12.8.1 Scala Visibility Modifiers

By default all the members of a package, a class, an object or a trait are public. Thus the following holds true:

```
package com.jeh.sample

object PublicObject {
  val publicVal
  var publicVar
  def publicMethod = println("Hello")
}
```

That is everything above is publically available, you only need to import the contents of the package com.jeh.sample._ or the object itself com.jeh.sample. PublicObject to be able to access everything. You do not need to use a special keyword public to make it so.

However, not all members of a type should be public, indeed in many cases you specifically do not want them to be publically available. In these cases there are two additional keywords that can be used to control visibility these are private and protected.

This means that you can choose whether these elements of your program are publically visible everywhere (the default), only visible to inherited types (protected) or only visible within the context they are defined (private). Thus these visibility modifiers can be used to restrict the access to (or visibility of) these members to other regions of code. In general to use an access modifier you need to include the appropriate keyword (private or protected) in the definition of the member of a package, class or object.

However, a word of caution is advisable here. Protected and private in Scala are not the same as in Java. For example, protected din Scala means that the member is only available in the current class and subclasses—it is not available in the current package. However, this is a default, both protected and private can be modified to indicate the scope they should be applied to. In the case of private it means that in Scala we can distinguish between private to an instance and private to a class.

12.8.2 Private Modifier

A private member is (by default) on visible to the class or object that it is defined in. Thus in the following example, the method print is only available to methods defined within the class Account:

```
package com.jeh.banking

case class Account(val name: String, val number: Int) {
   private def print = println(name)
}
```

However, an issue is that it is available to all instances of the class Account. Thus johns account can access the private method of the Denise's Account. This is the approach taken by Java and it is the default approach taken by Scala and represents class based privacy. If we want instance based privacy, that is the method print can only be called from within the same instance of the class Account then we need to qualify its scope. This can be done with a scope associated with the keyword in square brackets, for example private[this], for example:

```
package com.jeh.banking

case class Account(val name: String, val number: Int) {
   private[this] def print = println(name)
}
```

In this case private means private to this instance and not the whole class.

Interestingly you can also provide a package name within the square brackets so that you can indicate that a method is private to the package, for example:

```
package com.jeh.banking

case class Account(val name: String, val number: Int) {
   private[banking] def print = println(name)
}
```

In this revised version the method print is private to the package (that is it is available anywhere in the current package). This equates to package visibility in Java.

In fact the qualifier can be any from of scope thus the from private[x] can be used where x is one of an enclosing package, class or singleton object.

Also note that the keyword private can be applied to properties, methods and functions within a class, trait or object.

12.8.3 Protected Modifier

The protected modifier indicates that a member of a class, trait or object is visible within subtypes in any package (by default). For example, given the following definition:

```
package com.jeh.test

class Super {
    protected def print = println("Super")
}

class Sub extends Super {
  print
}

class Other {
  val s = new Super()
  // s.print - error is not visible
}
```

The class Super defines a protected method print. This method is only accessible (visible) in subclass of Super. The class Sub extends Super and therefore can reference the method *print* directly. It happens that this class is defined in the same package as Super but it could have been defined anywhere. However, even though the class Other is defined in the same package as Super and can create an instance of Super, it cannot reference the method print on an instance of Super as it is only visible/ accessible to subclasses of Super.

As with the private access modifier, the protected access modified can be qualified with a scope. For example, we can indicate that the protected member is protected up to a particular scope. Thus the previous example could be redefined such that the qualified test is added to the protected method print:

```
package com.jeh.test

class Super {
    protected[test] def print = println("Super")
}

class Sub extends Super {
  print
}

class Other {
  val s = new Super()
  s.print // No longer an error as it is now visible
}
```

In this way we can indicate that a member should be visible up to a certainly level and after that is only accessible to subclasses. Thus the example above in which we specify protected[test] is the equivalent of Java's version of protected as it indicates that the method *print* is visible in the current package and in an y subclass in any package.

As with the private access modifier the protected modifier can be qualified with a range of scopes. In fact the qualifier can be any from of appropriate scope thus the form protected[x] can be used where x is one of an enclosing package, class or singleton object.

Also note that the keyword *protected* can be applied to properties, methods and functions within a class, trait or object