Chapter 13

Packages and Imports

When working on a program, especially a large one, it is important to minimize *coupling*—the extent to which the various parts of the program rely on the other parts. Low coupling reduces the risk that a small, seemingly innocuous change in one part of the program will have devastating consequences in another part. One way to minimize coupling is to write in a modular style. You divide the program into a number of smaller modules, each of which has an inside and an outside. When working on the inside of a module—its *implementation*—you need only coordinate with other programmers working on that very same module. Only when you must change the outside of a module—its *interface*—is it necessary to coordinate with developers working on other modules.

This chapter shows several constructs that help you program in a modular style. It shows how to place things in packages, make names visible through imports, and control the visibility of definitions through access modifiers. The constructs are similar in spirit to constructs in Java, but there are some differences—usually ways that are more consistent—so it's worth reading this chapter even if you already know Java.

13.1 PUTTING CODE IN PACKAGES

Scala code resides in the Java platform's global hierarchy of packages. The example code you've seen so far in this book has been in the *unnamed* package. You can place code into named packages in Scala in two ways. First, you can place the contents of an entire file into a package by putting a package clause at the top of the file, as shown in Listing 13.1.

```
package bobsrockets.navigation
class Navigator
```

Listing 13.1 - Placing the contents of an entire file into a package.

The package clause of Listing 13.1 places class Navigator into the package namedbobsrockets.navigation. Presumably, this is the navigation software developed by Bob's Rockets, Inc.

Note

Because Scala code is part of the Java ecosystem, it is recommended to follow Java's reverse-domain-name convention for Scala packages that you release to the public. Thus, a better name for Navigator's package might becom.bobsrockets.navigation. In this chapter, however, we'll leave off the "com." to make the examples easier to understand.

The other way you can place code into packages in Scala is more like C# namespaces. You follow a package clause by a section in curly braces that contains the definitions that go into the package. This

syntax is called a *packaging*. The packaging shown in Listing 13.2 has the same effect as the code in Listing 13.1:

```
package bobsrockets.navigation {
  class Navigator
}
```

Listing 13.2 - Long form of a simple package declaration.

For such simple examples, you might as well use the syntactic sugar shown in Listing 13.1. However, one use of the more general notation is to have different parts of a file in different packages. For example, you might include a class's tests in the same file as the original code, but put the tests in a different package, as shown in Listing 13.3.

```
package bobsrockets {
  package navigation {

    // In package bobsrockets.navigation
    class Navigator

    package tests {

      // In package bobsrockets.navigation.tests
      class NavigatorSuite
    }
  }
}
```

Listing 13.3 - Multiple packages in the same file.

```
package bobsrockets {
  package navigation {
    class Navigator {
      // No need to say bobsrockets.navigation.StarMap
      val map = new StarMap
    }
    class StarMap
  }
  class Ship {
    // No need to say bobsrockets.navigation.Navigator
    val nav = new navigation.Navigator
  }
  package fleets {
    class Fleet {
      // No need to say bobsrockets.Ship
      def addShip() = { new Ship }
    }
  }
}
```

Listing 13.4 - Concise access to classes and packages.

```
package bobsrockets {
  class Ship
}

package bobsrockets.fleets {
```

```
  class Fleet {
    // Doesn't compile! Ship is not in scope.
    def addShip() = { new Ship }
  }
}
```

Listing 13.5 - Symbols in enclosing packages not automatically available.

```
// In file launch.scala
package launch {
  class Booster3
}

// In file bobsrockets.scala
package bobsrockets {
  package navigation {
    package launch {
      class Booster1
    }
    class MissionControl {
      val booster1 = new launch.Booster1
      val booster2 = new bobsrockets.launch.Booster2
      val booster3 = new _root_.launch.Booster3
    }
  }
  package launch {
    class Booster2
  }
}
```

Listing 13.6 - Accessing hidden package names.

13.2 CONCISE ACCESS TO RELATED CODE

When code is divided into a package hierarchy, it doesn't just help people browse through the code. It also tells the compiler that code in the same package is related in some way to each other. Scala takes advantage of this relatedness by allowing short, unqualified names when accessing code that is in the same package.

Listing 13.4 gives three simple examples. First, as you would expect, a class can be accessed from within its own package without needing a prefix. That's why new StarMap compiles. ClassStarMap is in the same package, bobsrockets.navigation, as the new expression that accesses it, so the package name doesn't need to be prefixed.

Second, a package itself can be accessed from its containing package without needing a prefix. In Listing 13.4, look at how class Navigator is instantiated. The new expression appears in package bobsrockets, which is the containing package of bobsrockets.navigation. Thus, it can access package bobsrockets.navigation as simply navigation.

Third, when using the curly-braces packaging syntax, all names accessible in scopes outside the packaging are also available inside it. An example in Listing 13.4 is the way addShip()creates a new Ship. The method is defined within two packagings: an outer one for bobsrockets, and an inner

one for bobsrockets.fleets. Since Ship is accessible in the outer packaging, it can be referenced from within addShip().

Note that this kind of access is only available if you explicitly nest the packagings. If you stick to one package per file, then—like in Java—the only names available will be the ones defined in the current package. In Listing 13.5, the packaging of bobsrockets.fleets has been moved to the top level. Since it is no longer enclosed in a packaging for bobsrockets, names frombobsrockets are not immediately in scope. As a result, new Ship gives a compile error. If nesting packages with braces shifts your code uncomfortably to the right, you can also use multiple package clauses without the braces.[1] For instance, the code below also defines class Fleet in two nested packages bobrockets and fleets, just like you saw it in Listing 13.4:

```
package bobsrockets
package fleets
class Fleet {
  // No need to say bobsrockets.Ship
  def addShip() = { new Ship }
}
```

One final trick is important to know. Sometimes, you end up coding in a heavily crowded scope where package names are hiding each other. In Listing 13.6, the scope of classMissionControl includes three separate packages named launch! There's one launch inbobsrockets.navigation, one in bobsrockets, and one at the top level. How would you reference each of Booster1, Booster2, and Booster3?

Accessing the first one is easiest. A reference to launch by itself will get you to packagebobsrockets.navigation.launch, because that is the launch package defined in the closest enclosing scope. Thus, you can refer to the first booster class as simply launch.Booster1. Referring to the second one also is not tricky. You can write bobrockets.launch.Booster2 and be clear about which one you are referencing. That leaves the question of the third booster class: How can you access Booster3, considering that a nested launch package shadows the top-level one?

To help in this situation, Scala provides a package named _root_ that is outside any package a user can write. Put another way, every top-level package you can write is treated as a member of package _root_. For example, both launch and bobsrockets of Listing 13.6 are members of package _root_. As a result, _root_.launch gives you the top-level launch package, and_root_.launch.Booster3 designates the outermost booster class.

13.3 IMPORTS

In Scala, packages and their members can be imported using import clauses. Imported items can then be accessed by a simple name like File, as opposed to requiring a qualified name likejava.io.File. For example, consider the code shown in Listing 13.7.

```
package bobsdelights

abstract class Fruit(
  val name: String,
  val color: String
```

```
)

object Fruits {
  object Apple extends Fruit("apple", "red")
  object Orange extends Fruit("orange", "orange")
  object Pear extends Fruit("pear", "yellowish")
  val menu = List(Apple, Orange, Pear)
}
```

Listing 13.7 - Bob's delightful fruits, ready for import.

An import clause makes members of a package or object available by their names alone without needing to prefix them by the package or object name. Here are some simple examples:

```
// easy access to Fruit
import bobsdelights.Fruit

// easy access to all members of bobsdelights
import bobsdelights._

// easy access to all members of Fruits
import bobsdelights.Fruits._
```

The first of these corresponds to Java's single type import and the second to Java's *on-demand* import. The only difference is that Scala's on-demand imports are written with a trailing underscore (_) instead of an asterisk (*). (After all, * is a valid identifier in Scala!) The third import clause above corresponds to Java's import of static class fields.

These three imports give you a taste of what imports can do, but Scala imports are actually much more general. For one, imports in Scala can appear anywhere, not just at the beginning of a compilation unit. Also, they can refer to arbitrary values. For instance, the import shown in Listing 13.8 is possible:

```
def showFruit(fruit: Fruit) = {
  import fruit._
  println(name + "s are " + color)
}
```

Listing 13.8 - Importing the members of a regular (not singleton) object.

Method showFruit imports all members of its parameter fruit, which is of type Fruit. The subsequent println statement can refer to name and color directly. These two references are equivalent to fruit.name and fruit.color. This syntax is particularly useful when you use objects as modules, which will be described in Chapter 29.

SCALA'S FLEXIBLE IMPORTS

Scala's import clauses are quite a bit more flexible than Java's. There are three principal differences. In Scala, imports:

- may appear anywhere
- may refer to objects (singleton or regular) in addition to packages
- let you rename and hide some of the imported members

Another way Scala's imports are flexible is that they can import packages themselves, not just their non-package members. This is only natural if you think of nested packages being contained in their surrounding package. For example, in Listing 13.9, the packagejava.util.regex is imported. This makes regex usable as a simple name. To access the Patternsingleton object from the java.util.regex package, you can just say, regex.Pattern, as shown inListing 13.9:

```
import java.util.regex

class AStarB {
  // Accesses java.util.regex.Pattern
  val pat = regex.Pattern.compile("a*b")
}
```

Listing 13.9 - Importing a package name.

Imports in Scala can also rename or hide members. This is done with an import selector clause enclosed in braces, which follows the object from which members are imported. Here are some examples:

import Fruits.{Apple, Orange}

This imports just members Apple and Orange from object Fruits.

import Fruits.{Apple => McIntosh, Orange}

This imports the two members Apple and Orange from object Fruits. However, the Apple object is renamed to McIntosh, so this object can be accessed with either Fruits.Apple or McIntosh. A renaming clause is always of the form "<original-name> => <new-name>".

import java.sql.{Date => SDate}

This imports the SQL date class as SDate, so that you can simultaneously import the normal Java date class as simply Date.

import java.{sql => S}

This imports the java.sql package as S, so that you can write things like S.Date.

import Fruits.{_}

This imports all members from object Fruits. It means the same thing as import Fruits._.

import Fruits.{Apple => McIntosh, _}

This imports all members from object Fruits but renames Apple to McIntosh.

import Fruits.{Pear => _, _}

This imports all members of Fruits except Pear. A clause of the form "<original-name> => _" excludes <original-name> from the names that are imported. In a sense, renaming something to `_' means hiding it altogether. This is useful to avoid ambiguities. Say you have two packages,Fruits and Notebooks, which both define a class Apple. If you want to get just the notebook named Apple, and not the fruit, you could still use two imports on demand like this:

```
import Notebooks._
import Fruits.{Apple => _, _}
```

This would import all Notebooks and all Fruits, except for Apple.

These examples demonstrate the great flexibility Scala offers when it comes to importing members selectively and possibly under different names. In summary, an import selector can consist of the following:

- A simple name x. This includes x in the set of imported names.
- A renaming clause x => y. This makes the member named x visible under the name y.
- A hiding clause x => _. This excludes x from the set of imported names.
- A *catch-all* `_'. This imports all members except those members mentioned in a preceding clause. If a catch-all is given, it must come last in the list of import selectors.

The simpler import clauses shown at the beginning of this section can be seen as special abbreviations of import clauses with a selector clause. For example, "import p._" is equivalent to "import p.{_}" and "import p.n" is equivalent to "import p.{n}".

13.4 IMPLICIT IMPORTS

Scala adds some imports implicitly to every program. In essence, it is as if the following three import clauses had been added to the top of every source file with extension ".scala":

```
import java.lang._ // everything in the java.lang package
import scala._     // everything in the scala package
import Predef._    // everything in the Predef object
```

The java.lang package contains standard Java classes. It is always implicitly imported in Scala source files.[2] Because java.lang is imported implicitly, you can write Thread instead ofjava.lang.Thread, for instance.

As you have no doubt realized by now, the scala package contains the standard Scala library, with many common classes and objects. Because scala is imported implicitly, you can writeList instead of scala.List, for instance.

The Predef object contains many definitions of types, methods, and implicit conversions that are commonly used on Scala programs. For example, because Predef is imported implicitly, you can write assert instead of Predef.assert.

These three import clauses are treated a bit specially in that later imports overshadow earlier ones. For instance, the StringBuilder class is defined both in package scala and, from Java version 1.5 on, also in package java.lang. Because the scala import overshadows the java.langimport, the simple name StringBuilder will refer to scala.StringBuilder, notjava.lang.StringBuilder.

13.5 ACCESS MODIFIERS

Members of packages, classes, or objects can be labeled with the access modifiers private andprotected. These modifiers restrict access to the members to certain regions of code. Scala's treatment of access modifiers roughly follows Java's but there are some important differences which are explained in this section.

Private members

Private members in Scala are treated similarly to Java. A member labeled private is visible only inside the class or object that contains the member definition. In Scala, this rule applies also for inner classes. This treatment is more consistent, but differs from Java. Consider the example shown in Listing 13.10.

```
class Outer {
  class Inner {
    private def f() = { println("f") }
    class InnerMost {
      f() // OK
    }
  }
  (new Inner).f() // error: f is not accessible
}
```

Listing 13.10 - How private access differs in Scala and Java.

In Scala, the access (new Inner).f() is illegal because f is declared private in Inner and the access is not from within class Inner. By contrast, the first access to f in class InnerMost is OK, because that access is contained in the body of class Inner. Java would permit both accesses because it lets an outer class access private members of its inner classes.

Protected members

Access to protected members in Scala is also a bit more restrictive than in Java. In Scala, aprotected member is only accessible from subclasses of the class in which the member is defined. In Java such accesses are also possible from other classes in the same package. In Scala, there is another way to achieve this effect[3] so protected is free to be left as is. The example shown in Listing 13.11 illustrates protected accesses.

```
package p {
  class Super {
    protected def f() = { println("f") }
```

```
  }
  class Sub extends Super {
    f()
  }
  class Other {
    (new Super).f()  // error: f is not accessible
  }
}
```

Listing 13.11 - How protected access differs in Scala and Java.

In Listing 13.11, the access to f in class Sub is OK because f is declared protected in Super and Subis a subclass of Super. By contrast the access to f in Other is not permitted, because Other does not inherit from Super. In Java, the latter access would be still permitted because Other is in the same package as Sub.

Public members

Scala has no explicit modifier for public members: Any member not labeled private orprotected is public. Public members can be accessed from anywhere.

```
package bobsrockets

package navigation {
  private[bobsrockets] class Navigator {
    protected[navigation] def useStarChart() = {}
    class LegOfJourney {
      private[Navigator] val distance = 100
    }
    private[this] var speed = 200
  }
}
package launch {
  import navigation._
  object Vehicle {
    private[launch] val guide = new Navigator
  }
}
```

Listing 13.12 - Flexible scope of protection with access qualifiers.

Scope of protection

Access modifiers in Scala can be augmented with qualifiers. A modifier of the form private[X]or protected[X] means that access is private or protected "up to" X, where X designates some enclosing package, class or singleton object.

Qualified access modifiers give you very fine-grained control over visibility. In particular they enable you to express Java's accessibility notions, such as package private, package protected, or private up to outermost class, which are not directly expressible with simple modifiers in Scala. But they also let you express accessibility rules that cannot be expressed in Java.

Listing 13.12 presents an example with many access qualifiers being used. In this listing, classNavigator is labeled private[bobsrockets]. This means that this class is visible in all classes and

objects that are contained in package bobsrockets. In particular, the access to Navigator in objectVehicle is permitted because Vehicle is contained in package launch, which is contained inbobsrockets. On the other hand, all code outside the package bobsrockets cannot access classNavigator.

This technique is quite useful in large projects that span several packages. It allows you to define things that are visible in several sub-packages of your project but that remain hidden from clients external to your project. The same technique is not possible in Java. There, once a definition escapes its immediate package boundary, it is visible to the world at large.

Of course, the qualifier of a private may also be the directly enclosing package. An example is the access modifier of guide in object Vehicle in Listing 13.12. Such an access modifier is equivalent to Java's package-private access.

Table 13.1 - Effects of private qualifiers on LegOfJourney.distance

no access modifier	public access
private[bobsrockets]	access within outer package
private[navigation]	same as package visibility in Java
private[Navigator]	same as private in Java
private[LegOfJourney]	same as private in Scala
private[this]	access only from same object

All qualifiers can also be applied to protected, with the same meaning as private. That is, a modifier protected[X] in a class C allows access to the labeled definition in all subclasses of Cand also within the enclosing package, class, or object X. For instance, the useStarChart method in Listing 13.12 is accessible in all subclasses of Navigator and also in all code contained in the enclosing package navigation. It thus corresponds exactly to the meaning of protected in Java.

The qualifiers of private can also refer to an enclosing class or object. For instance the distancevariable in class LegOfJourney in Listing 13.12 is labeled private[Navigator], so it is visible from everywhere in class Navigator. This gives the same access capabilities as for private members of inner classes in Java. A private[C] where C is the outermost enclosing class is the same as just private in Java.

Finally, Scala also has an access modifier that is even more restrictive than private. A definition labeled private[this] is accessible only from within the same object that contains the definition. Such a definition is called object-private. For instance, the definition of speedin class Navigator in Listing 13.12 is object-private. This means that any access must not only be within class Navigator, it must also be made from the very same instance of Navigator. Thus the accesses "speed" and "this.speed" would be legal from within Navigator.

The following access, though, would not be allowed, even if it appeared inside class Navigator:

```
val other = new Navigator
other.speed // this line would not compile
```

Marking a member private[this] is a guarantee that it will not be seen from other objects of the same class. This can be useful for documentation. It also sometimes lets you write more general variance annotations (see Section 19.7 for details).

To summarize, Table 13.1 here lists the effects of private qualifiers. Each line shows a qualified private modifier and what it would mean if such a modifier were attached to the distancevariable declared in class LegOfJourney in Listing 13.12.

Visibility and companion objects

In Java, static members and instance members belong to the same class, so access modifiers apply uniformly to them. You have already seen that in Scala there are no static members; instead you can have a companion object that contains members that exist only once. For instance, in Listing 13.13 object Rocket is a companion of class Rocket.

```
class Rocket {
  import Rocket.fuel
  private def canGoHomeAgain = fuel > 20
}

object Rocket {
  private def fuel = 10
  def chooseStrategy(rocket: Rocket) = {
    if (rocket.canGoHomeAgain)
      goHome()
    else
      pickAStar()
  }
  def goHome() = {}
  def pickAStar() = {}
}
```

Listing 13.13 - Accessing private members of companion classes and objects.

Scala's access rules privilege companion objects and classes when it comes to private or protected accesses. A class shares all its access rights with its companion object and *vice versa*. In particular, an object can access all private members of its companion class, just as a class can access all private members of its companion object.

For instance, the Rocket class in Listing 13.13 can access method fuel, which is declared privatein object Rocket. Analogously, the Rocket object can access the private method canGoHomeAgain in class Rocket.

One exception where the similarity between Scala and Java breaks down concerns protectedstatic members. A protected static member of a Java class C can be accessed in all subclasses of C. By contrast, a protected member in a companion object makes no sense, as singleton objects don't have any subclasses.

13.6 PACKAGE OBJECTS

So far, the only code you have seen added to packages are classes, traits, and standalone objects. These are by far the most common definitions that are placed at the top level of a package. But Scala doesn't limit you to just those—Any kind of definition that you can put inside a class can also be at the top level of a package. If you have some helper method you'd like to be in scope for an entire package, go ahead and put it right at the top level of the package.

To do so, put the definitions in a *package object*. Each package is allowed to have one package object. Any definitions placed in a package object are considered members of the package itself.

An example is shown in Listing 13.14. File package.scala holds a package object for packagebobsdelights. Syntactically, a package object looks much like one of the curly-braces packagings shown earlier in the chapter. The only difference is that it includes the objectkeyword. It's a package *object*, not a *package*. The contents of the curly braces can include any definitions you like. In this case, the package object includes the showFruit utility method from Listing 13.8.

Given that definition, any other code in any package can import the method just like it would import a class. For example, Listing 13.14 also shows the standalone object PrintMenu, which is located in a different package. PrintMenu can import the utility method showFruit in the same way it would import the class Fruit.

```
// In file bobsdelights/package.scala
package object bobsdelights {
  def showFruit(fruit: Fruit) = {
    import fruit._
    println(name + "s are " + color)
  }
}

// In file PrintMenu.scala
package printmenu
import bobsdelights.Fruits
import bobsdelights.showFruit

object PrintMenu {
  def main(args: Array[String]) = {
    for (fruit <- Fruits.menu) {
      showFruit(fruit)
    }
  }
}
```

Listing 13.14 - A package object.

Looking ahead, there are other uses of package objects for kinds of definitions you haven't seen yet. Package objects are frequently used to hold package-wide type aliases (Chapter 20) and implicit conversions (Chapter 21). The top-level scala package has a package object, and its definitions are available to all Scala code.

Package objects are compiled to class files named package.class that are the located in the directory of the package that they augment. It's useful to keep the same convention for source files. So you would typically put the source file of the package object bobsdelights of Listing 13.14 into a file named package.scala that resides in the bobsdelights directory.

13.7 CONCLUSION

In this chapter, you saw the basic constructs for dividing a program into packages. This gives you a simple and useful kind of modularity, so that you can work with very large bodies of code without different parts of the code trampling on each other. Scala's system is the same in spirit as Java's packages, but there are some differences where Scala chooses to be more consistent or more general.

Looking ahead, Chapter 29 describes a more flexible module system than division into packages. In addition to letting you separate code into several namespaces, that approach allows modules to be parameterized and inherit from each other. In the next chapter, we'll turn our attention to assertions and unit testing.

Footnotes for Chapter 13:

[1] This style of multiple package clauses without braces is called *chained package clauses*.

[2] Scala also originally had an implementation on .NET., where namespace System, the .NETanalogue of package java.lang, was imported instead.

[3] Using qualifiers, described in "Scope of protection" here.