Chapter 29

Modular Programming Using Objects

In Chapter 1, we claimed that one way Scala is a scalable language is that you can use the same techniques to construct small as well as large programs. So far in this book we've focused primarily on *programming in the small*: designing and implementing the smaller program pieces out of which you can construct a larger program.[1] The other side of the story is *programming in the large*: organizing and assembling the smaller pieces into larger programs, applications, or systems. We touched on this subject when we discussed packages and access modifiers in Chapter 13. In short, packages and access modifiers enable you toorganize a large program using packages as *modules*, where a module is a "smaller program piece" with a well defined interface and a hidden implementation.

While the division of programs into packages is already quite helpful, it is limited because it provides no way to abstract. You cannot reconfigure a package two different ways within the same program, and you cannot inherit between packages. A package always includes one precise list of contents, and that list is fixed until you change the code.

In this chapter, we'll discuss how you can use Scala's object-oriented features to make a program more modular. We'll first show how a simple singleton object can be used as a module. Then we'll explain how you can use traits and classes as abstractions over modules. These abstractions can be reconfigured into multiple modules, even multiple times within the same program. Finally, we'll show a pragmatic technique for using traits to divide a module across multiple files.

29.1 THE PROBLEM

As a program grows in size, it becomes increasingly important to organize it in a modular way. First, being able to compile different modules that make up the system separately helps different teams work independently. In addition, being able to unplug one implementation of a module and plug in another is useful, because it allows different configurations of a system to be used in different contexts, such as unit testing on a developer's desktop, integration testing, staging, and deployment.

For example, you may have an application that uses a database and a message service. As you write code, you may want to run unit tests on your desktop that use mock versions of both the database and message service, which simulate these services sufficiently for testing without needing to talk across the network to a shared resource. During integration testing, you may want to use a mock message service but a live developer database. During staging and certainly during deployment, your organization will likely want to use live versions of both the database and message service.

Any technique that aims to facilitate this kind of modularity needs to provide a few essentials. First, there should be a module construct that provides a good separation of interface and implementation. Second, there should be a way to replace one module with another that has the same interface without changing or recompiling the modules that depend on the replaced one. Lastly, there should be a way to wire modules together. This wiring task can by thought of as *configuring* the system.

One approach to solving this problem is *dependency injection*, a technique supported on the Java platform by frameworks such as Spring and Guice, which are popular in the enterprise Java community.[2] Spring, for example, essentially allows you to represent the interface of a module as a Java interface and implementations of the module as Java classes. You can specify dependencies between modules and "wire" an application together via external XML configuration files. Although you can use Spring with Scala and thereby use Spring's approach to achieving system-level modularity of your Scala programs, with Scala you have some alternatives enabled by the language itself. In the remainder of this chapter, we'll show how to use objects as modules to achieve the desired "in the large" modularity without using an external framework.

29.2 A RECIPE APPLICATION

Imagine you are building an enterprise web application that will allow users to manage recipes. You want to partition the software into layers, including a *domain layer* and an*application layer*. In the domain layer, you'll define *domain objects*, which will capture business concepts and rules, as well as encapsulate state that will be persisted to an external relational database. In the application layer, you'll provide an API organized in terms of the services the application offers to clients (including the user interface layer). The application layer will implement these services by coordinating tasks and delegating the work to the objects of the domain layer.[3]

You want to be able to plug in real or mock versions of certain objects in each of these layers, so that you can more easily write unit tests for your application. To achieve this goal, you can treat the objects you want to mock as modules. In Scala, there is no need for objects to be "small" things, no need to use some other kind of construct for "big" things like modules. One of the ways Scala is a scalable language is that the same constructs are used for structures both small and large.

For example, since one of the "things" you want to mock in the domain layer is the object that represents the relational database, you'll make that one of the modules. In the application layer, you'll treat a "database browser" object as a module. The database will hold all of the recipes that a person has collected. The browser will help search and browse that database, for example, to find every recipe that includes an ingredient you have on hand.

The first thing to do is to model foods and recipes. To keep things simple, a food will just have a name, as shown in Listing 29.1. A recipe will have a name, a list of ingredients, and some instructions, as shown in Listing 29.2.

```
package org.stairwaybook.recipe

abstract class Food(val name: String) {
  override def toString = name
}
```

Listing 29.1 - A simple Food entity class.

```
package org.stairwaybook.recipe

class Recipe(
  val name: String,
  val ingredients: List[Food],
  val instructions: String
) {
  override def toString = name
}
```

Listing 29.2 - Simple Recipe entity class.

The Food and Recipe classes shown in Listings 29.1 and 29.2 represent *entities* that will be persisted in the database.[4] Listing 29.3 shows some singleton instances of these classes, which can be used when writing tests.

```
package org.stairwaybook.recipe

object Apple extends Food("Apple")
object Orange extends Food("Orange")
object Cream extends Food("Cream")
object Sugar extends Food("Sugar")

object FruitSalad extends Recipe(
  "fruit salad",
  List(Apple, Orange, Cream, Sugar),
  "Stir it all together."
)
```

Listing 29.3 - Food and Recipe examples for use in tests.

```
package org.stairwaybook.recipe

object SimpleDatabase {
  def allFoods = List(Apple, Orange, Cream, Sugar)

  def foodNamed(name: String): Option[Food] =
    allFoods.find(_.name == name)

  def allRecipes: List[Recipe] = List(FruitSalad)
}

object SimpleBrowser {
  def recipesUsing(food: Food) =
    SimpleDatabase.allRecipes.filter(recipe =>
      recipe.ingredients.contains(food))
}
```

Listing 29.4 - Mock database and browser modules.

Scala uses objects for modules, so you can start modularizing your program by making two singleton objects to serve as mock implementations of the database and browser modules during testing. Because it is a mock, the database module is backed by a simple in-memory list. Implementations of these objects are shown in Listing 29.4. You can use this database and browser as follows:

```
scala> val apple = SimpleDatabase.foodNamed("Apple").get
apple: Food = Apple

scala> SimpleBrowser.recipesUsing(apple)
res0: List[Recipe] = List(fruit salad)
```

To make things a little more interesting, suppose the database sorts foods into categories. To implement this, you can add a FoodCategory class and a list of all categories in the database, as shown in Listing 29.5. Notice in this example that the private keyword, so useful for implementing classes, is also useful for implementing modules. Items marked private are part of the implementation of a module, and thus are particularly easy to change without affecting other modules.

At this point, many more facilities could be added, but you get the idea. Programs can be divided into singleton objects, which you can think of as modules. This is not big news, but it becomes very useful when you consider abstraction (which we'll cover next).

```
package org.stairwaybook.recipe

object SimpleDatabase {
  def allFoods = List(Apple, Orange, Cream, Sugar)

  def foodNamed(name: String): Option[Food] =
    allFoods.find(_.name == name)

  def allRecipes: List[Recipe] = List(FruitSalad)

  case class FoodCategory(name: String, foods: List[Food])

  private var categories = List(
    FoodCategory("fruits", List(Apple, Orange)),
    FoodCategory("misc", List(Cream, Sugar)))

  def allCategories = categories
}

object SimpleBrowser {
  def recipesUsing(food: Food) =
    SimpleDatabase.allRecipes.filter(recipe =>
      recipe.ingredients.contains(food))

  def displayCategory(category: SimpleDatabase.FoodCategory) = {
    println(category)
  }
}
```

Listing 29.5 - Database and browser modules with categories added.

29.3 ABSTRACTION

Although the examples shown so far did manage to partition your application into separate database and browser modules, the design is not yet very "modular." The problem is that there is essentially a "hard link" from the browser module to the database modules:

```
SimpleDatabase.allRecipes.filter(recipe => ...
```

Because the SimpleBrowser module mentions the SimpleDatabase module by name, you won't be able to plug in a different implementation of the database module without modifying and recompiling the browser module. In addition, although there's no hard link from theSimpleDatabase module to the SimpleBrowser module,[5] there's no clear way to enable the user interface layer, for example, to be configured to use different implementations of the browser module.

When making these modules more pluggable, however, it is important to avoid duplicating code, because much code can likely be shared by different implementations of the same module. For example, suppose you want the same code base to support multiple recipe databases, and you want to be able to create a separate browser for each of these databases. You would like to reuse the browser code for each of the instances, because the only thing different about the browsers is which database they refer to. Except for the database implementation, the rest of the code can be reused character for character. How can the program be arranged to minimize repetitive code? How can the code be made reconfigurable, so that you can configure it using either database implementation?

The answer is a familiar one: If a module is an object, then a template for a module is a class. Just like a class describes the common parts of all its instances, a class can describe the parts of a module that are common to all of its possible configurations.

```
abstract class Browser {
  val database: Database

  def recipesUsing(food: Food) =
    database.allRecipes.filter(recipe =>
      recipe.ingredients.contains(food))

  def displayCategory(category: database.FoodCategory) = {
    println(category)
  }
}
```

Listing 29.6 - A Browser class with an abstract database val.

The browser definition therefore becomes a class, instead of an object, and the database to use is specified as an abstract member of the class, as shown in Listing 29.6. The database also becomes a class, including as much as possible that is common between all databases, and declaring the missing parts that a database must define. In this case, all database modules must define methods for allFoods, allRecipes, and allCategories, but since they can use an arbitrary definition, the methods must be left abstract in the Database class. The foodNamedmethod, by contrast, can be defined in the abstract Database class, as shown in Listing 29.7.

```
abstract class Database {
  def allFoods: List[Food]
  def allRecipes: List[Recipe]

  def foodNamed(name: String) =
    allFoods.find(f => f.name == name)

  case class FoodCategory(name: String, foods: List[Food])
  def allCategories: List[FoodCategory]
```

```
    }
```

Listing 29.7 - A Database class with abstract methods.

The SimpleDatabase object must be updated to inherit from the abstract Database class, as shown in Listing 29.8.

```
    object SimpleDatabase extends Database {
      def allFoods = List(Apple, Orange, Cream, Sugar)

      def allRecipes: List[Recipe] = List(FruitSalad)

      private var categories = List(
        FoodCategory("fruits", List(Apple, Orange)),
        FoodCategory("misc", List(Cream, Sugar)))

      def allCategories = categories
    }
```

Listing 29.8 - The SimpleDatabase object as a Database subclass.

Then, a specific browser module is made by instantiating the Browser class and specifying which database to use, as shown in Listing 29.9.

```
    object SimpleBrowser extends Browser {
      val database = SimpleDatabase
    }
```

Listing 29.9 - The SimpleBrowser object as a Browser subclass.

You can use these more pluggable modules the same as before:

```
  scala> val apple = SimpleDatabase.foodNamed("Apple").get
  apple: Food = Apple

  scala> SimpleBrowser.recipesUsing(apple)
  res1: List[Recipe] = List(fruit salad)
```

Now, however, you can create a second mock database, and use the same browser class with it, as shown in Listing 29.10:

```
    object StudentDatabase extends Database {
      object FrozenFood extends Food("FrozenFood")

      object HeatItUp extends Recipe(
        "heat it up",
        List(FrozenFood),
        "Microwave the 'food' for 10 minutes.")

      def allFoods = List(FrozenFood)
      def allRecipes = List(HeatItUp)
      def allCategories = List(
        FoodCategory("edible", List(FrozenFood)))
    }

    object StudentBrowser extends Browser {
      val database = StudentDatabase
    }
```

Listing 29.10 - A student database and browser.

29.4 SPLITTING MODULES INTO TRAITS

Often a module is too large to fit comfortably into a single file. When that happens, you can use traits to split a module into separate files. For example, suppose you wanted to move categorization code out of the main Database file and into its own. You could create a trait for the code as shown in (Listing 29.11).

```
    trait FoodCategories {
      case class FoodCategory(name: String, foods: List[Food])
      def allCategories: List[FoodCategory]
    }
```

Listing 29.11 - A trait for food categories.

Now class Database can mix in the FoodCategories trait instead of defining FoodCategory andallCategories itself, as shown in Listing 29.12:

```
    abstract class Database extends FoodCategories {
      def allFoods: List[Food]
      def allRecipes: List[Recipe]
      def foodNamed(name: String) =
        allFoods.find(f => f.name == name)
    }
```

Listing 29.12 - A Database class that mixes in the FoodCategories trait.

You might try and divide SimpleDatabase into two traits, one for foods and one for recipes. This would allow you to define SimpleDatabase as shown in Listing 29.13:

```
    object SimpleDatabase extends Database
        with SimpleFoods with SimpleRecipes
```

Listing 29.13 - A SimpleDatabase object composed solely of mixins.

The SimpleFoods trait could look as shown in Listing 29.14:

```
    trait SimpleFoods {
      object Pear extends Food("Pear")
      def allFoods = List(Apple, Pear)
      def allCategories = Nil
    }
```

Listing 29.14 - A SimpleFoods trait.

So far so good, but unfortunately, a problem arises if you try to define a SimpleRecipes trait like this:

```
  trait SimpleRecipes { // Does not compile
    object FruitSalad extends Recipe(
      "fruit salad",
      List(Apple, Pear),  // Uh oh
      "Mix it all together."
    )
    def allRecipes = List(FruitSalad)
  }
```

The problem is that Pear is located in a different trait from the one that uses it, so it is out of scope. The compiler has no idea that SimpleRecipes is only ever mixed together with SimpleFoods.

However, there is a way you can tell this to the compiler. Scala provides the *self type* for precisely this situation. Technically, a self type is an assumed type for this whenever this is mentioned within the class. Pragmatically, a self type specifies the requirements on any concrete class the trait is mixed into. If you have a trait that is only ever used when mixed in with another trait or traits, then you can specify that those other traits should be assumed. In the present case, it is enough to specify a self type of SimpleFoods, as shown in Listing 29.15:

```
trait SimpleRecipes {
  this: SimpleFoods =>

  object FruitSalad extends Recipe(
    "fruit salad",
    List(Apple, Pear),   // Now Pear is in scope
    "Mix it all together."
  )
  def allRecipes = List(FruitSalad)
}
```

Listing 29.15 - A SimpleRecipes trait with a self type.

Given the new self type, Pear is now available. Implicitly, the reference to Pear is thought of asthis.Pear. This is safe, because any *concrete* class that mixes in SimpleRecipes must also be a subtype of SimpleFoods, which means that Pear will be a member. Abstract subclasses and traits do not have to follow this restriction, but since they cannot be instantiated with new, there is no risk that the this.Pear reference will fail.

29.5 RUNTIME LINKING

Scala modules can be linked together at runtime, and you can decide which modules will link to which depending on runtime computations. For example, Listing 29.16 shows a small program that chooses a database at runtime and then prints out all the apple recipes in it:

```
object GotApples {
  def main(args: Array[String]) = {
    val db: Database =
      if(args(0) == "student")
        StudentDatabase
      else
        SimpleDatabase

    object browser extends Browser {
      val database = db
    }

    val apple = SimpleDatabase.foodNamed("Apple").get

    for(recipe <- browser.recipesUsing(apple))
```

```
            println(recipe)
        }
    }
```

Listing 29.16 - An app that dynamically selects a module implementation.

Now, if you use the simple database, you will find a recipe for fruit salad. If you use the student database, you will find no recipes at all using apples:

```
$ scala GotApples simple
fruit salad
$ scala GotApples student
$
```

CONFIGURING WITH SCALA CODE

You may wonder if you are not backsliding to the hard links problem of the original examples in this chapter, because the GotApples object shown in Listing 29.16 contains hard links to both StudentDatabase and SimpleDatabase. The difference here is that the hard links are localized in one file that can be replaced.

Every modular application needs some way to specify the actual module implementations to use in a particular situation. This act of "configuring" the application will by definition involve the naming of concrete module implementations. For example, in a Spring application, you configure by naming implementations in an external XML file. In Scala, you can configure via Scala code itself. One advantage to using Scala source over XML for configuration is that the process of running your configuration file through the Scala compiler should uncover any misspellings in it prior to its actual use.

29.6 TRACKING MODULE INSTANCES

Despite using the same code, the different browser and database modules created in the previous section really are separate modules. This means that each module has its own contents, including any nested classes. FoodCategory in SimpleDatabase, for example, is a different class from FoodCategory in StudentDatabase!

```
scala> val category = StudentDatabase.allCategories.head
category: StudentDatabase.FoodCategory =
FoodCategory(edible,List(FrozenFood))

scala> SimpleBrowser.displayCategory(category)
<console>:21: error: type mismatch;
 found   : StudentDatabase.FoodCategory
 required: SimpleBrowser.database.FoodCategory
              SimpleBrowser.displayCategory(category)
                                            ^
```

If instead you prefer all FoodCategorys to be the same, you can accomplish this by moving the definition of FoodCategory outside of any class or trait. The choice is yours, but as it is written, each Database gets its own, unique FoodCategory class.

Since the two FoodCategory classes shown in this example really are different, the compiler is correct to complain. Sometimes, though, you may encounter a case where two types are the same but the compiler can't verify it. You will see the compiler complaining that two types are not the same, even though you as the programmer know they are.

In such cases you can often fix the problem using *singleton types*. For example, in theGotApples program, the type checker does not know that db and browser.database are the same. This will cause type errors if you try to pass categories between the two objects:

```
object GotApples {
  // same definitions...
  for (category <- db.allCategories)
    browser.displayCategory(category)
  // ...
}
GotApples2.scala:14: error: type mismatch;
 found   : db.FoodCategory
 required: browser.database.FoodCategory
        browser.displayCategory(category)
                                ^
one error found
```

To avoid this error, you need to inform the type checker that they are the same object. You can do this by changing the definition of browser.database as shown in Listing 29.17:

```
object browser extends Browser {
  val database: db.type = db
}
```

Listing 29.17 - Using a singleton type.

This definition is the same as before except that database has the funny-looking type db.type. The ".type" on the end means that this is a *singleton type*. A singleton type is extremely specific and holds only one object; in this case, whichever object is referred to by db. Usually such types are too specific to be useful, which is why the compiler is reluctant to insert them automatically. In this case, though, the singleton type allows the compiler to know that db andbrowser.database are the same object—enough information to eliminate the type error.

29.7 CONCLUSION

This chapter has shown how to use Scala's objects as modules. In addition to simple static modules, this approach gives you a variety of ways to create abstract, reconfigurable modules. There are actually even more abstraction techniques than shown, since anything that works on a class also works on a class used to implement a module. As always, how much of this power you use should be a matter of taste.

Modules are part of programming in the large, and thus are hard to experiment with. You need a large program before it really makes a difference. Nonetheless, after reading this chapter you know which Scala features to think about when you want to program in a modular style. Think about these techniques when you write your own large programs and recognize these coding patterns when you see them in other people's code.

Footnotes for Chapter 29:

[1] This terminology was introduced in DeRemer, *et. al.*, "Programming-in-the-large versus programming-in-the-small." [DeR75]

[2] Fowler, "Inversion of control containers and the dependency injection pattern." [Fow04]

[3] The naming of these layers follows that of Evans, *Domain-Driven Design*. [Eva03]

[4] These entity classes are simplified to keep the example uncluttered with too much real-world detail. But transforming these classes into entities that could be persisted with Hibernate or the Java Persistence Architecture, , for example, would require only a few modifications, such as adding a private Long id field and a no-arg constructor, placingscala.reflect.BeanProperty annotations on the fields, specifying appropriate mappings via annotations or a separate XML file, and so on.

[5] This is good, because each of these architectural layers should depend only on layers below them.