Chapter 18
Traits

18.1 Introduction

In the previous chapters there have already been several references to these things called Traits. For example, we have encountered the App trait numerous times. However we have avoided the question "What are traits?".

In this chapter we will look at the core concepts behind traits and attempt to explain how and why you might want to use them. The next chapter will then explore some more advanced uses and concepts within the area of Traits.

18.2 What are Traits?

Let us start of with what Traits are not:

- They are not Classes or Objects.
- They are not Abstract Classes.
- They are not (Java and C#) Interfaces.
- They cannot be instantiated directly.

However, Traits are part of the type system in Scala. That is a Trait defines a type just as a class defines a type. Thus *vals* and *vars*, properties parameters, return types etc. can all be of a type defined by a Trait and can reference instances (or indeed objects) that mix in a Trait.

They are also a fundamental unit of reuse within the Scala language. If you use Traits appropriately you will find that they allow you to reuse code very elegantly and in a much cleaner way than is possible with Java (or C#) interfaces and abstract classes.

Traits allow you to define methods, functions and properties. They also allow you to define abstract methods, functions and properties. They allow type declarations and can use a self-reference to indicate the types that they expect to be used with. When you define a method, a function or a property they can be made available to the type (Class or Object) that the Trait is used with.

DOI 10.1007/978-3-319-06776-6_18, © Springer International Publishing Switzerland 2014

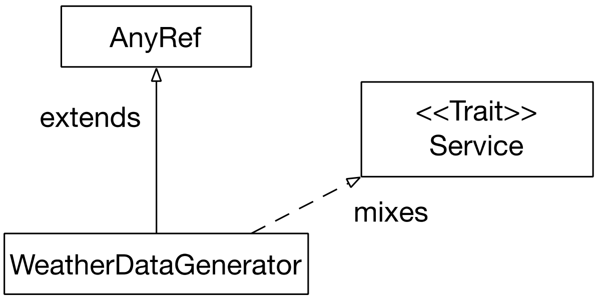


Fig. 18.1 Mixing a Trait into a class

Adding a Trait to a class or object is referred to as *mixing in* a Trait to that type. In fact a class or object can *mix in* any number of traits allowing for a great deal of flexibility and reuse of Traits. This idea is illustrated in Fig. 18.1. In this diagram the class WeatherDataGenerator extends the class AnyRef (the default super class) and mixes in the Trait Service. Thus the WeatherDataGenerator is a reference type and a Service.

You will find many texts refer to a class or object as inheriting form a Trait (partly due to the syntax used with Traits) however I will restrict this terminology to the relationship between one Trait and a sub Trait that directly inherits the features of that Trait. Thus inheritance is possible between Traits and we can use terms such as Super Trait, Trait and Sub Trait to refer to the inheritance relationships between traits. This idea is illustrated in Fig. 18.2.

Note that in Fig. 18.2 the root of the Trait hierarchy is shown as extending *AnyRef*. Prior to Scala 2.10 this was the only option—all traits extended *AnyRef* however, since Scala 2.10 you can also make a Trait extend *AnyVal* that is used for Universal Traits that are discussed later in the chapter.

Also note that a trait can extend one Trait but can mix in any additional traits required using with. This Traits support multiple inheritance rather than single inheritance. It is thus legal to write:

```
trait MyReader extends Immutable with Decorator {
    ...
}
```

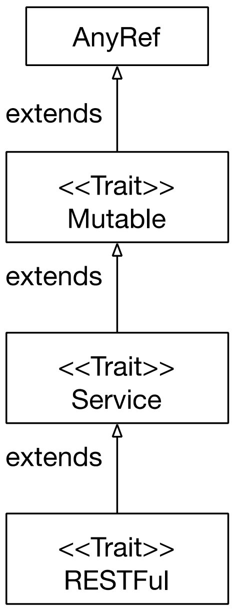
In summary, a Trait is a reuseable type that can be combined with Classes and Objects to provide a very significant form of code reuse in Scala.

18.3 Defining a Trait

A trait is defined by the keyword *trait*, this if followed by the name of the Trait with the body of the trait defined between curly brackets '{.}'.

```
access-modifier trait <name> extends <suber trait> { .. }
```

Fig. 18.2 Trait inheritance hierarchy



Trait definitions can have any number or combination of concrete methods, functions, properties, field, type definitions. It can also have any number or combination of abstract methods, functions and properties. However, Traits do not have constructors (either primary or auxiliary) and may or may not have the extends keyword to indicate inheritance.

A simple example of a trait definition is presented below.

```
trait Model {
  var value: Any

  def printValue() {
    println(value)
  }

  def printer(): Unit
}
```

This trait, called *Model,* defines

- a method *printValue()* that returns *Unit*,
- an abstract property *value* which can hold *Any* type of element but currently is not initialized (and is thus abstract),
- an abstract method *printer()* which returns Unit.

Note that this is not an abstract class (a concept which Scala also supports); it is a Trait that can optionally have concrete (fully defined) methods, functions and properties or abstract methods functions and properties. Any abstract member of the trait will have to be defined either in a sub Trait of this trait, or in a class or object with which this trait is mixed.

Note that in this example all the members (that is the property value and the two methods) are public as this is the default in Scala. However, any member can have either of the encapsulation modifiers applied to them. Thus the methods or the property could be private or protected (and may be qualified).

18.4 Using a Trait

How do you use the trait presented in the last section? You cannot instantiate it directly, you cannot invoke the behaviour defined in it directly; you must mix it into a class or an object.

That is, classes and objects mix in traits such as Model. Any class or object that does this obtains the method *printValue* and must implement the abstract members *value* and *printer*. The key word used to do this differs depending on whether the class or object being used currently extends a named class or not.

If the class or object does not extend a named class then the keyword *extends* is used to indicate the first trait to mix in. If *extends* is already being used to extend a named class then the keyword with is used to indicate the first trait to mix in. Any subsequent traits are always indicated via the keyword *with*.

Thus the generic syntax is actually:

Class <class-name> extends <class or if no class a Trait> with <Trait> with <Trait> …

Object <object-name> extends <class or if no class a Trait> with <Trait> with <Trait> …

The following list illustrates several different combinations:

- `class Course extends Model`—a class with an extends that indicates a trait Model to mix in.
- `class Course extends AnyRef with Model`—a class which explicitly extends the default AnyRef class and mixes in the trait Model.
- `class Course extends Programme with Model`—a class which extends Programme and mixes in the trait Model. Note that without looking at Programme we can not tell whether it is a class or a Trait.
- `class Course extends Model with AwardCeremony`—this is a class with multiple traits. Both Model and AwardCeremony are traits but the first trait is preceeded by extends where as subsequent Traits are preceeded by with.
- `class Course extends Programme with Model with AwardCeremony`. This class extends Programme (a class) and mixes in multiple traits Model and AwardCeremony.

Note that we have used classes above, but the same can be done with Objects. For example:

- `object Course extends Model`—an object that mixes in a single Trait Model.
- `object Course extends Programme with Model`—an object that extends Programme and mixes in Model.
- `object Course extends Model with AwardCeremony`—an object mixing in Model and AwardCeremony.
- `object Course extends Programme with Model with AwardCeremony`—an object that extends Programme and mixes in Model and AwardCeremony.

As a concrete example of this consider the following listing:

```
object Course extends Model {
  var value: Any = "Hello World"
  def printer(): Unit = {println("Hello")}
}
```

This object mixes in the Model trait (even though it uses the extends keyword) and implements the abstract value property and the abstract printer method. It therefore meets the *contract* of the Model trait (which required the two abstract members to be implemented by an *concrete* class or object).

It is now possible to treat the Course object as either a Course or as a Model, for example:

```
object TraitTest extends App {
  val c = Course;
  c.printValue()
  val m: Model = c
  println(m.value)
}
```

This simple test application assigns the Course object to the *val* 'c' and then invokes the *printValue* method on 'c' (of course we could also have invoked it directly on the Course object). We then assign c to the *val* 'm' that is of type Model (we have explicitly stated that in the declaration of m). This is perfectly legal because in terms of the Scala type system, a Course is also a Model.

Of course we are not limited to do this with objects, we could also use a class declaration, for example:

```
class Degree extends Model {
  var value: Any = "B.Sc."
  def printer(): Unit = {println("Degree Award")}
}
```

This class mixes in the Model trait into the class Degree and as before this means that it must implement the value property and the printer method (otherwise we would be defining an *abstract* class which would require us to prefix the class keyword with the keyword abstract and would mean that we could not instantiate the class).

The following listing shows how you can instantiate the *Degree* class and treat it as a *Degree* type or a *Model* type (or indeed an *AnyRef* or *Any* type):

```
object DegreeTest extends App {
   val d = new Degree()
   d.printer
   val m: Model = d
   m.printer
}
```

18.5 Abstract Trait Members

The members of a trait (that is the properties, types, methods and functions defined within a trait) can be abstract (as illustrated in the Model trait. The following Sample trait illustrates an abstract type T, an abstract method transform and two abstract properties; a read only (val) initial and a read-write (var) current.

```
package com.jjh.scala.abs

trait Sample {
  type T
  def transform(x: T): T
  val initial: T
  var current: T
}
```

The implementing class (or object) must provide implementations for all the abstract members:

```
class Car extends Sample {
  type T = String
  def transform(x: T) = x + x
  val initial = "first"
  var current = initial
}
```

This class defines a concrete type String for the type T (which can then be used within the rest of the trait). It also provides an implementation for the transform method that takes a parameter of type T (String) and concatenates it to itself. It also sets initial to the string “first” and current to the value set in initial.

18.6 Dynamic Binding of Traits

There is another way in which you can combine a trait with a class. It is possible to mix in a trait at the point that an instance of a given class is created. For example, given the class Person that contains a *name* property and an override of the *toString* method defined as follows:

```
class Person(val name: String) {
  override def toString = "Person[" + name + "]"
}
```

We can also define a trait Logger that defines a single method log that prints out a “Created” string and runs the Log method whenever anything it is mixed with is instantiated (the invocation of the Log method):

```
trait Logger {
  log
  def log = println("Created")
}
```

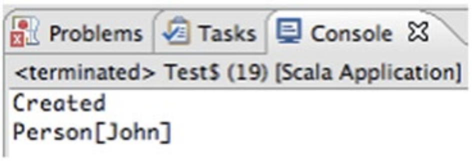
The trait Logger can be mixed into an instance of the class Person dynamically when that instance is created by combining the call to *new* with an additional reference to the trait via the *with* keyword. The syntax for this is:

```
new <classname>(<params>) with <traitname>
```

For example:

```
object Test extends App {
  val p = new Person("John") with Logger
  println(p)
}
```

As you can see form this we are creating a new instance of Person and mixing in Logger at the same time. Thus the instance referenced by the val ‘p’ is both a Person type and a Logger type. It combines the behaviour and data in these types together. The result of running this application is shown below:



```
<terminated> Test$ (19) [Scala Application]
Created
Person[John]
```

From this you can see that when we created the new instance the log method was run as part of the freestanding code executed when the instance was created. This resulted in the String “Created” being printed out. When the method println was invoked on the contents of ‘p’ the resulting version of toString (invoked by println) was that define din the Person class and hence Person[John] was printed out. Thus we have indeed combined to two together at the point of instantiation.

18.7 Sealed Traits

A sealed trait is a trait that can only be used within the file that it is defined within. That is only the classes and objects within the same file as the sealed trait can mix in that trait. It can also only be extended by traits in the same file. For example:

```
package com.jjh.scala.qanda

sealed trait Answer
object Yes extends Answer
object No extends Answer
```

Note that the trait Answer can be used as the type of a variable or a value within other packages—but it cannot be extended or mixed in elsewhere. Therefore in a package com.jjh.scala.test you can reference the type and use it as the type for vars or vals to hold the singleton instances of Yes and No. For example:

```
package com.jjh.scala.test

import com.jjh.scala.qanda.Answer
import com.jjh.scala.qanda.Yes
import com.jjh.scala.qanda.No

object AnswerTest extends App {
   var a: Answer = Yes
   println(a)
   a = No
   println(a)
}
```

18.8 Marker Traits

A marker trait is a trait that declare no methods, functions, types or properties. Instead it is used to indicate additional semantics of a type (class, object or further traits). For example see the scala.Mutable and scala.Immutable traits; these are marker traits indicating the semantics of mutability and immutability.

Marker traits can be used where:

- it is useful to semantically indicate a role or concept that other entities may play with the application. However, these entities may be of varying types (from classes, to objects to further traits) and may inherit behaviour from various different places in the type hierarchy.
- semantically there is a common concept, but there is little or no common behaviour or data representation between the concrete implementations of the generic domain concept.
- client classes may need to know something about the type of an object without actually needing to know the specific type (at least at the interface level).

Using a trait, as the basis of a marker, is particularly convenient in Scala as a type may mix in any number of traits. For example, the following code defines two marker traits, one called Decorator and one called Service.

```
package com.jjh.scala.marker

trait Decorator

trait Service
```

Any type can implement one or more traits, thus any type can implement a marker trait and any other traits as required. For example:

```
trait MyReader extends Immutable with Decorator {
def read: Int
}
```

Semantically this tells us that *MyReader* is a type of Decorator and that it is Immutable.

18.9 Trait Dependencies

When you mix a trait into another type you may want to be able to invoke functionality form the host type. This can be done by defining a *self* reference. A *self reference* ties one type to another type which will be provided at a future point in time. This means that the *this* value can be used to access another types behaviour and data. That is the type it will be mixed into must be of a particular kind and thus the trait you are defining can rely on certain data or behaviour being provided by the host type in the trait.

For example, let us assume that we have defined a class Service:

```
class Service {
  def printer: Unit = println("Service Hello")
}
```

We will then define a class Client that takes an Adaptor type. The method *doWork* invokes the method invoke on the adapter.

```
class Client(adapter: Adaptor) {
  def doWork = adapter.invoke
}
```

Let us assume that we have initially written the Adaptor type as follows:

```
trait Adaptor {
  def invoke: Unit
}
```

However, we would like to use an instance of Service with the Client class. But currently the Service type does not implement an invoke method and the Adapter trait defines an abstract invoke method.

Ideally we want to link the Adaptor to the Service types. There are various ways in which we could do this but the core issue is that currently there is no link between the Service type and the Client type. To solve this problem we will use a self reference in the Adapter type. This allows us to define a trait that can only be used with Service types (and subtypes)

This trait defines a method invoke to use the method printer (provided by the Service type). Thus we can guarantee that the trait Adapter will be used with a Service or a subtype of Service (whether that is a class or an object). Therefore the method printer will be available to wherever the Adapter trait is used.

```
trait Adaptor {
  self: Service =>
  def invoke = printer
}
```

We can then mix this trait into a Service either dynamically at the point of use or statically with the Service in a new type. For example, the following example dynamically binds Adaptor into Service when an instance of the Service class is created. It then passes this to the Client class as a (compatible) parameter. We can then invoke the doWork method.

```
object Test extends App {
  val adpator = new Service() with Adaptor
  val client = new Client(adpator)
  client.doWork
}
```

Alternatively we could have defined a new sub type (AdaptedService) of the class Service that also mixes in the Adapter trait.

```
class AdaptedService extends Service with Adaptor
```

We can now use instances of this type with the client—the only difference to the previous example is that we have statically defined a new type that combines the Service class with the Adaptor trait that can be reused in multiple situations.

```
object Test2 extends App {
  val adpator = new AdaptedService()
  val client = new  Client(adpator)
  client.doWork
}
```

18.10 To Trait or not to Trait

It is worth considering when you should define a trait and when you might define a class. The general set of guidelines on when something should be a trait include:

- If behaviour or data will not be reused—then implement that behaviour or data as a concrete class.
- If behaviour or data might be reused in multiple, unrelated classes, then make it a trait.
- If efficiency is of ultimate importance lean towards a class.
- If it is a reusable concept for a root of a class hierarchy use an abstract class.
- If you want to use it in Java code use an abstract class.
- If you want to model domain concepts to be implemented by different classes in different ways then use Traits.