

Chapter 19

Further Traits

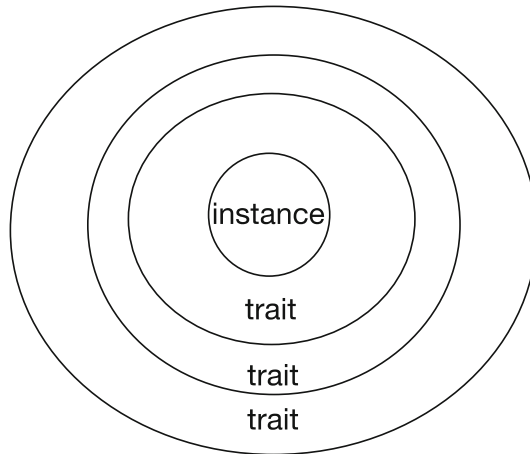


19.1 Introduction

This chapter looks at some of the more advanced features associated with the use of traits. The chapter looks at the way in which Traits can be dynamically wrapped around an instance providing a form of aspect-oriented programming (AOP) known as stackable traits. We then look at the role Traits can play in developing reusable behaviour that simplifies the development of new types. Universal Traits used with Value Types are then discussed. The chapter concludes by considering the way in which traits can be used to define a restricted set of values for a given type.

19.2 Stackable Modifications

Traits can be stacked one on top of another when an instance of a class is created. Each stacked trait can override the behaviour of the trait it is stacked on top of. This allows the trait to either replace some of the behaviour of that trait, or wrap additional behaviour around that trait. This idea is illustrated in the following figure.



As an example, the following code defines an abstract trait `AbstractProcessor`. It is abstract because it defines an abstract method `update` that takes an `Int` and returns `Unit`. This trait is mixed into the class `BasicProcessor`. This class defines the `update` method as setting the `amount` property defined on the class.

```
package com.jjh.scala.processor

trait AbstractProcessor {
  def update(x: Int)
}

class BasicProcessor(var amount: Int)
  extends AbstractProcessor {
  def update(x: Int) = amount = x
  override def toString = "BasicProcessor: " + amount
}

object Test extends App {
  val p = new BasicProcessor(0)
  p.update(5)
  println(p)
}
```

The simple test harness class creates a new `BasicProcessor` using the initial value 0 and then updates it to 5 and prints out the result. The result of executing this program is:

```
BasicProcessor: 5
```

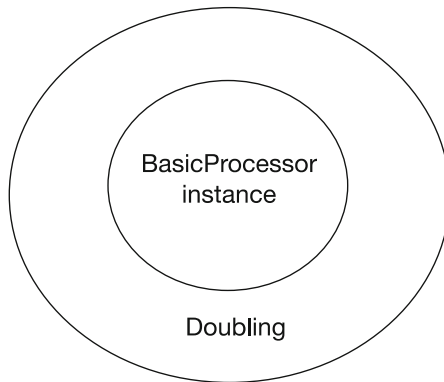
We could now define a Trait *Doubling* that also extends the `AbstractProcessor` trait. Note that the `update` method in this trait states that it overrides any other definitions but that it is also abstract. That is, it expects to build on something that it will be mixed into at a later date which is expected to be able to provide the

remainder of the behaviour of the update method. This allows it to invoke the super.update method. Here super means the next version of update in the stack of traits wrapping the concrete instance of a class (or indeed that concrete instance).

```
trait Doubling extends AbstractProcessor {
  abstract override def update(x: Int) {
    super.update(2 * x)
  }
}
```

Note if the method is not marked as abstract, it cannot invoke the super version of the *update* method.

This trait can be mixed into the BasicProcessor when that BasicProcessor is instantiated, which results in the Doubling trait wrapping around the BasicProcessor as shown below.



The following listing illustrates how the Test application is modified to *stack* the Doubling trait onto the BasicProcessor:

```
object Test extends App {
  val p = new BasicProcessor(0) with Doubling
  p.update(5)
  println(p)
}
```

The result of running this program is:

BasicProcessor: 10

As you can see from this, the value 10 is now being stored in the Basic Processor—thus the Doubling trait version of update is being invoked which results in the integer being doubled before being passed on down the line to the update method defined on the BasicProcessor instance.

This can be taken further; we could define additional traits such as Filtering and Incrementing that either filter the value to be updated or add one to the value being updated. For example,

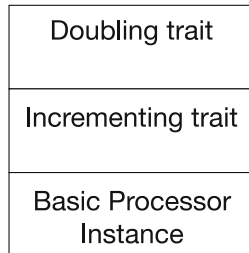
```
trait Incrementing extends AbstractProcessor {
  abstract override def update(x: Int) {
    super.update(x + 1)
  }
}
```

```
trait Filtering extends AbstractProcessor {
  abstract override def update(x: Int) {
    if (x > 0) super.update(x)
  }
}
```

We can now combine these traits in various ways and in different orders. For example,

```
object Test extends App {
  val p = new BasicProcessor(0)
    with Incrementing with Doubling
  p.update(5)
  println(p)
}
```

The effect of this is that the Doubling trait is stacked on top of the Incrementing trait, which is stacked on top of the BasicProcessor. This idea is illustrated in the next figure.



The end result of executing the above program is shown below:

BasicProcessor: 11

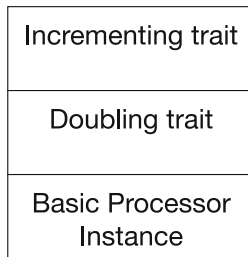
whereas this version results in a different output:

```
object Test extends App {
  val p = new BasicProcessor(0)
    with Doubling with Incrementing
  p.update(5)
  println(p)
}
```

The result this time is:

BasicProcessor: 12

This is because in one example the value is doubled and then incremented by one and in the second example it is incremented first and then doubled. Thus you can see that the order in which the traits are added is significant (with the last trait being the one that is accessed first). Thus the above listing can be represented as shown in the next figure.



Finally, we can also include the Filtering Trait to decide if anything is to be done with a value at all:

```
object Test extends App {
  val p = new BasicProcessor(10)
    with Doubling
    with Incrementing
    with Filtering
  p.update(0)
  println(p)
}
```

where the attempt to update the BasicProcessor instance to Zero is vetoed by the Filtering trait, and thus the output from this program is:

BasicProcessor: 10

Note that the AbstractProcessor could also have been an abstract class rather than a Trait; however, from a design point of view it is cleaner to have the AbstractProcessor as a Trait.

Of course we are not just limited to mixing in stackable traits when an instance of class `I` is created. We could also have defined a new type with the base class and the stackable traits mixed together. For example,

```
class DoublingProcessor extends BasicProcessor with Doubling {...}
```

19.3 Fat Versus Thin Traits

There is a continual tension in software between the richness of an interface offered by a component or library and implementation and maintenance effort required for such an interface. This is because although (in theory) a rich interface is better for client applications, a simpler interface is easier to develop and maintain. Ideally, we want the best of both worlds: minimum effort for the developer of the component and maximum utility for the user of the component. Traits allow methods to be constructed based on existing implementations.

For example, the `Ordered` Trait defined in the `scala.math` package is a trait for data that has a single natural ordering. Class or traits that implement this trait inherit a range of concrete methods such as `<`, `<=`, `>`, `>=` which rely on a method `compare`. However, the method `compare` is an abstract method that is expected to return an integer depending on the value being compared. This method must be provided by a concrete trait, class or object. The definition of the method is:

```
abstract def compare(that: A): Int
```

This method returns the result of comparing the current instance with the operand *that*.

The method returns a value 'x' where:

- $x < 0$ when *this* < *that*
- $x == 0$ when *this* == *that*
- $x > 0$ when *this* > *that*

For example, if we wished to create a new `Balance` class, which supported basic Ordering and comparison type behaviour, we could do this by mixing in the `Ordered` trait, as shown below.

```

package com.jjh.scala.financial

import scala.math.Ordered

class Balance(val amount: Double)
  extends Ordered[Balance] {

  def compare(that: Balance): Int = (this.amount - that.amount).toInt

}

```

The result is that although the code we have written is quite light as have obtained a rich interface. For example the range of operations available on the currency instance are shown in:

```

package com.jjh.scala.financial

object Test extends App {
  val b1 = new Balance(20.0)
  val b2 = new Balance(25.0)
  println(b1.<(b2))
  println(b1.<=(b2))
  println(b1.>=(b2))
  println(b1.>(b2))
  println(b1.compare(b2))
}

```

Thus the `Balance` class has a rich interface but has a simple implementation. The majority of the comparison behaviour is mixed in from the trait (such as the `<`, `>` methods), but they build on a concrete implementation of the `compare` method.

19.4 Universal Traits

Scala's rules for inheritance do not permit Value Classes to mix in traits that extend from `AnyRef`. Prior to Scala 2.10, all traits eventually extended `AnyRef`, and thus traits could not be mixed into a Value Class. However, since Scala 2.10 traits can optionally extend `Any` instead of `AnyRef`. This must be specified explicitly when the trait is defined. Such a Trait is known as a Universal Trait as it can be mixed into all types of classes from reference types to Value Classes. This permits Value Classes to extend traits (as long as they are Universal Traits).

When a Universal Trait is mixed into a Value Class, then they allow inheritance of methods for the Value Class but they do not incur the overhead of heap allocation and referencing.

The following trait `Printable` is a Universal Trait as it explicitly specifies the parent type as `Any`. It is then used with the Value Class Wrapper (which merely wraps around the underlying type `Int`) and extends `AnyVal` and mixes in `Printable`.

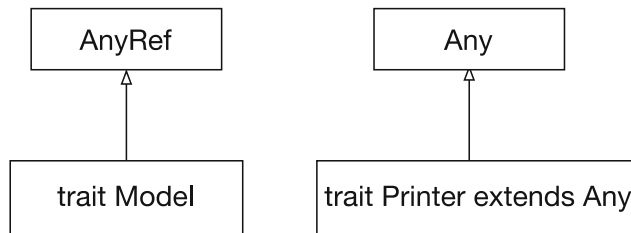
```
package com.jjh.scala.universal

/**
 * Universal Trait - does no allocation
 * and extends Any.
 */
trait Printable extends Any {
  def print(): Unit = println(this)
}

/**
 * Value class that wraps an Int.
 */
class Wrapper(val underlying: Int)
  extends AnyVal with Printable

/**
 * Test App
 */
object Main extends App {
  val p: Printable = new Wrapper(32)
  p.print()
}
```

Note that if you do not explicitly specify `Any` as the super type of a Trait, then that trait still defaults to extending `AnyRef`. Thus in the next figure, the trait `Model` is a Trait as it (by default) extends `AnyRef` and the trait `Printer` is a Universal Trait as it explicitly extends `Any`.



This also has some implications for further inheritance. If we have a Universal Trait `Equals` (which explicitly extends `Any`) and a subtrait `Ordered` that extends `Equals`, then the effect is that the trait `Ordered` *by default* extends the class `AnyRef` and *mixes in* the trait `Equals`. The end result is that this is not a Universal Trait:


```
trait Equals extends Any { ... }
```

```
trait Ordered extends Equals{ ... }
```

To turn Ordered into a Universal Trait then you must explicitly specify that Any is the super class of Ordered as follows:

```
trait Ordered extends Any with Equals {...}
```

The trait Ordered is now explicitly a Universal Trait.

19.5 Traits for a Data Type

Although Scala has an enumeration type, this implies a specific ordering whereas in some cases we merely want to define a set of associated values. For example, if we wished to create a set of values for traffic lights, then we might wish to create values for Red, Yellow and Green. However, there is no specific ordering, just these values. We could use a trait to help define the objects used to represent the traffic light colours. For example,

```
package com.jeh.scala.traits
trait TrafficLight
case object Red extends TrafficLight
case object Yellow extends TrafficLight
case object Green extends TrafficLight
```

In this case, the trait TrafficLight has been defined but contains no data elements or behaviour (other than the defaults inherited from AnyRef). It is then used to create a set of objects, Red, Yellow and Green. Note that these are case *objects* indicating that all the values associated with TrafficLight are defined in the same file and thus Red, Yellow and Green can be used safely within pattern matching statements with the compiler able to indicate if all conditions are being accounted for.

As an example of using these values, the following test harness creates a set of *vals* for each colour and can be used to print out the results and test for equality, etc.

```
object Test extends App {
  val c1 = Red
  val c2 = Yellow
  val c3 = Red

  println(Red)
  println(c1 equals c2)
  println(c1 equals c3)
}
```

This is a commonly recurring idiom in Scala.

19.6 Single Abstract Method (SAM) Traits

A SAM trait is a trait with a single abstract method. SAMs are originally introduced in Java 8 (in the form of single abstract method interfaces also known as functional interfaces) as it started to support the functional programming world.

This feature has been incorporated into the Scala world (essentially since Scala 2.11.5) as it makes Java interoperability easier. Strictly speaking Scala does not need to support the concept of a SAM as it has its own (richer) set of features available. However, it can make working with functional literals easier.

There are a set of constraints that must be met by a trait that wishes to be treated as a SAM; these are:

- It must define a single abstract method (SAM).
- The abstract method must take a single argument list.

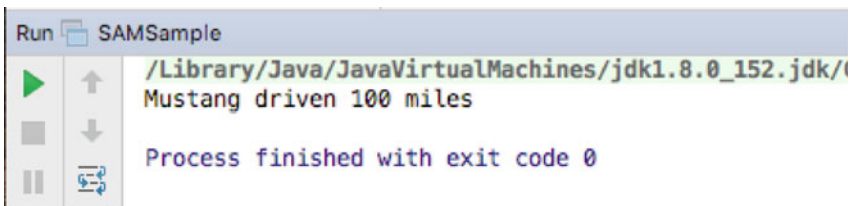
The following example illustrates a SAM trait. It defines a single abstract method `drive` that takes a single `Integer` argument:

```
trait Drivable {
  // exactly one abstract method
  def drive(miles: Int): Unit
}
```

This can then be used to create a concrete implementation of the trait on the fly using what is now as a functional literal or a lambda:

```
object SAMSample extends App {
  val d1: Drivable = (m: Int) => println(s"Mustang driven $m miles")
  d1.drive(100)
}
```

The output from this simple application is



The screenshot shows a console window titled "Run SAMSAMPLE". The output is as follows:

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/
Mustang driven 100 miles

Process finished with exit code 0
```

This means that it is not necessary to define a class or object to implement the abstract method defined in the trait. This reduced the amount of code written and compiled.

A SAM trait can define any number of concrete values and methods (as long as there is only a single abstract method). For example,

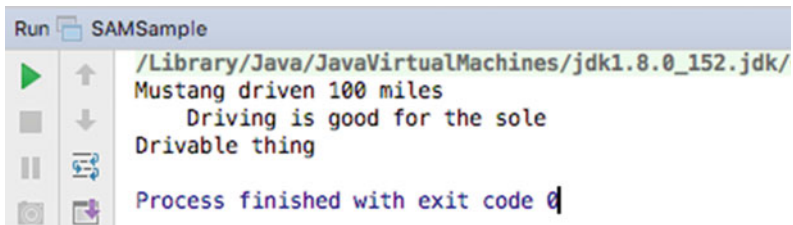
```
trait Drivable {
  // exactly one abstract method
  def drive(miles: Int): Unit
  // Can define concrete properties
  val message = "Driving is good for the sole"
  // Can also define concrete methods
  def getDetails() = "Drivable thing"
}
```

We can now use these properties and methods in our code:

```
object SAMSample extends App {
  val d1: Drivable = (m: Int) => println(s"Mustang driven $m miles
  \n\t${d1.message}")
  d1.drive(100)
  println(d1.getDetails())
}
```

Note that the function we have defined that takes an Integer refers to the variable `d1` that is being used to set up—this is referred to as closure. It works as the variable `d1` will be set up by the time the function is executed.

The output from this is:



```
Run SAMSample
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/
Mustang driven 100 miles
  Driving is good for the sole
  Drivable thing
Process finished with exit code 0
```

We will return to functions later in this book.