Chapter 15
Value Classes

15.1 Introduction

This chapter introduces another type of class in Scala, Value Classes. A Value Class is a type where the actual value being represented by the type class is hold directly by a variable, rather than needing to access that value via a reference (an address in memory). Examples of Value Types include Booelan, Int and Double which can have the values true, false, 32, 45.7, etc. Such values can be held directly by a variable, rather than accessed via a reference. This can be more efficient for simple types like Int.

Value Classes inherit from AnyVal, rather than AnyRef. However, prior to Scala 2.10 *AnyVal* was actually a type of Trait not a Class. This meant that it was not possible to create user-defined Value Types. However in Scala 2.10 *AnyVal* was redefined as an *abstract* class. As it is normal to subclass abstract classes it is now possible to create *user-defined* Value Classes. Thus subclasses of *AnyVal* are user-defined Value Classes.

15.2 Value Classes

Value Classes are treated as special by the Scala compiler. That is, the compiler will determine if it can inline the value to be used directly. This avoids the need to allocate runtime objects and is thus more efficient and faster (as no allocation must be made and no reference must be followed).

To ensure that the compiler can treat a value in this way it is necessary for the programmer to ensure that no object allocation is performed within the type. Thus a Value Type cannot hold within itself a reference to a non-Value Type (such as an instance of the class Person). It must also ensure that the type being defined:

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_15

- must extend AnyVal,
- must be immutable by nature (i.e. it should not change itself but return a new instance whenever a change in value is required),
- must have a single *public val parameter* for the underlying type (i.e. the built-in Value Type being wrapped),
- does not declare any additional fields within itself,
- cannot have any auxiliary constructors,
- cannot define any nested types such as classes, objects or traits,
- are not used in tests used to determine their type or in type-based pattern matching,
- must not override the *equals* or *hashcode* methods,
- cannot have any initialisation statements.

However, they can have

- any methods or functions as required.

15.3 Simple Value Type Example

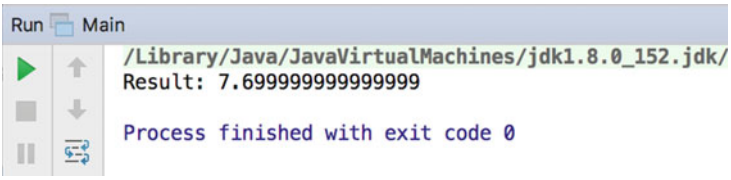
The following Value Type class meets the criteria defined in the last section. That is, the Value Class `Meter` has a single *val* property `value` of type `Double` (which is a built-in Value Type); it extends `AnyVal` directly and provides a method '+'. In addition it exhibits immutability. That is, when the '+' method is invoked, it does not change *value* instead it returns a new instance of the `Meter` class representing the new value:

```
class Meter(val value: Double) extends AnyVal {
  def +(m: Meter) : Meter = new Meter(value + m.value)
}
```

The following simple application illustrates how this class may be used:

```
object Main extends App {
  val x = new Meter(3.4)
  val y = new Meter(4.3)
  val z = x + y
  Console.println("Result: " + z.value)
}
```

In this example, we create two instances of the `Meter` Value Class and store them in the variables `x` and `y`. We then add them together and store the result in `z`. Note that this line looks very much as it would if `x` and `y` held *Int* or *Doubles* and we added them together. The result is then printed out. The effect of running this application is shown below.



Interestingly the compiler actually replaces the references to `Meter` with the primitives held within the Value Class at runtime. Thus there is virtually no overhead in using `Meter` than in using `Double` directly. This raises the question "Why bother?" The answer is twofold:

- `Meter` is more semantically meaningful than *Double*. That is *Double* is a generic way of representing 54 bit real numbers. The Value Class *Meter* represents the concept of a length, i.e. a meter.
- `Meter` also allows methods to be defined that allow semantically meaningful operations to be defined that can also indicate what is being done at a higher level of abstraction than the basic type Double would allow.

15.4 Additional Value Class Concepts

Value Classes are implicitly treated as final classes, thus ensuring that they cannot be extended by other classes. This is important as it restricts the need for polymorphism and thus allows the compiler to inline the values being represented.

Value Classes are implicitly assumed to have structural equality and *hashcode*. That is, their equals and *hashcode* methods are taken to be defined as follows (and this is why you must not redefine them):

```
def equals(other: Any) = other match {
  case that: C => this.u == that.u
  case _ => false
}
def hashCode = u.hashCode
```

Where u equates to the underlying (Value Type) property (such as Double, or Int). In other words if the underliers have the same value, then the Value Types are equal; otherwise, they are not equal. In addition the *hashcode* of a Value Type is the *hashcode* of its underling type.

Value Classes can only mix in Universal Traits. If you try to mix in a trait which is not a Universal Trait, then the class you are defining is not a Value Class but a reference class. A Universal Trait is a special trait which extends the Any type rather than the default AnyRef type.

You can make the Value Class a case class that simplifies the syntax and means that you do not need to use the keyword *new*. This often makes for much more readable and semantically clear Value classes. For example, taking the Meter class defined earlier and changing it into a case class:

```
case class Meter(val value: Double) extends AnyVal {
  def +(m: Meter) : Meter = new Meter(value + m.value)
}
```

This now means that we do not need to use the keyword new, and thus the test application looks less as if we have created instance of a class and more as if Meter was a built-in type:

```
object Main extends App {
  val x = Meter(3.4)
  val y = Meter(4.3)
  val z = x + y
  Console.println("Result: " + z.value)
}
```

15.5 Negating Value Classes

It should be noted that the compiler will not treat a class as a Value Class in some situations. These are presented below:

The compiler will not treat an instance of a Value Class as a value when:

- It is used in an array—it will not inline values into an Array.
- It is used in pattern matching situations such as case statements.
- When used within any polymorphic situation. For example, where the type of the variable relates to a more generic type (such as a trait).