

Stairway to Scala - Flight 3

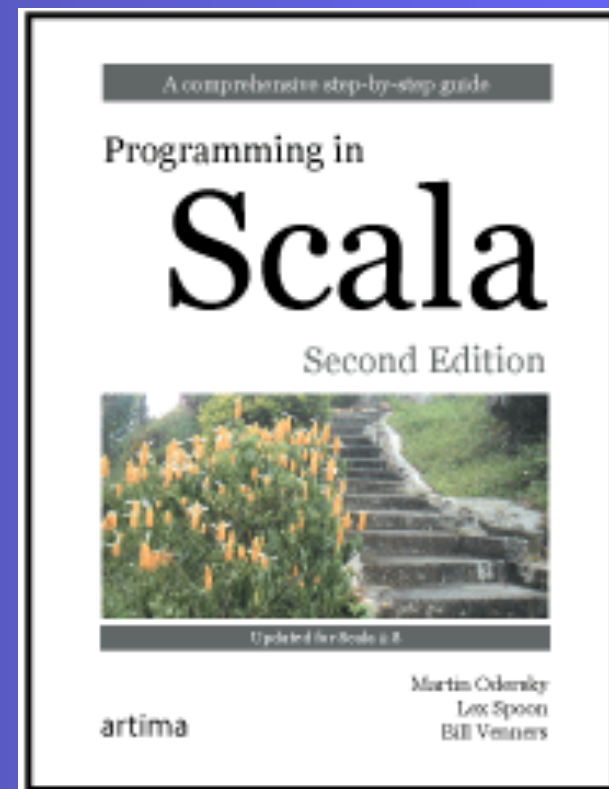
Classes and objects

Bill Venners

Dick Wall

www.artima.com

Copyright (c) 2010 Artima Inc. All Rights Reserved.



Flight 3 goal

Introduce Classes, Methods, Constructors,
Companion Objects, Operators, Fields,
Self References, Overloaded Methods and
Implicit Conversion
(Chapters 4 and 6)

Defining a class

```
class ChecksumAccumulator {
```

```
    private var sum = 0
```

```
    def add(b: Byte): Unit = {  
        sum += b  
    }
```

```
    def checksum(): Int = {  
        return ~(sum & 0xFF) + 1  
    }  
}
```

Defining a procedure

```
class ChecksumAccumulator {
```

```
    private var sum = 0
```

```
    def add(b: Byte) {  
        sum += b  
    }
```

```
    def checksum(): Int = {  
        return ~(sum & 0xFF) + 1  
    }  
}
```

A more concise class definition

```
// In file ChecksumAccumulator.scala  
class ChecksumAccumulator {  
  private var sum = 0  
  def add(b: Byte) { sum += b }  
  def checksum(): Int = ~(sum & 0xFF) + 1  
}
```

A companion object

```
// In file ChecksumAccumulator.scala
```

```
import scala.collection.mutable.Map
```

```
object ChecksumAccumulator {
```

```
  private val cache = Map[String, Int]()
```

```
  def calculate(s: String): Int =
```

```
    if (cache.contains(s))
```

```
      cache(s)
```

```
    else {
```

```
      val acc = new ChecksumAccumulator
```

```
      for (c <- s)
```

```
        acc.add(c.toByte)
```

```
      val cs = acc.checksum()
```

```
      cache += (s -> cs)
```

```
      cs
```

Using a singleton object

`ChecksumAccumulator.calculate("Every value is an object.")`



A Scala application

```
// In file Summer.scala
import ChecksumAccumulator.calculate

object Summer {
  def main(args: Array[String]) {
    for (arg <- args)
      println(arg + ": " + calculate(arg))
  }
}
```


Compiling and running

```
$ scalac ChecksumAccumulator.scala Summer.scala
```

```
$ fsc ChecksumAccumulator.scala Summer.scala
```

```
$ scala Summer of love
```

```
of: -213
```

```
love: -182
```

The primary constructor

class Rational(n: Int, d: Int)

```
public class Rational { // This is Java
    private int n;
    private int d;
    public Rational(int n, int d) {
        this.n = n;
        this.d = d;
    }
}
```

The primary constructor (cont.)

```
class Rational(n: Int, d: Int) {  
    println("Created " + n + "/" + d)  
}
```

```
scala> new Rational(1, 2)
```

```
Created 1/2
```

```
res0: Rational = Rational@90110a
```

Overriding toString

```
class Rational(n: Int, d: Int) {  
  override def toString = n + "/" + d  
}
```

```
scala> val x = new Rational(1, 2)  
x: Rational = 1/3
```

Checking preconditions

```
class Rational(n: Int, d: Int) {  
  require(d != 0)  
  override def toString = n + "/" + d  
}
```

Can only access class parameters on this instance

```
class Rational(n: Int, d: Int) { // This won't compile
  require(d != 0)
  override def toString = n + "/" + d
  def add(that: Rational): Rational =
    new Rational(n * that.d + that.n * d, d * that.d)
}
```

```
<console>:11: error: value d is not a member of Rational new
Rational(n * that.d + that.n * d, d * that.d) ^ <console>:11: error:
value d is not a member of Rational new Rational(n * that.d + that.
n * d, d * that.d)
^
```

Adding fields

```
class Rational(n: Int, d: Int) {  
  require(d != 0)  
  val numer: Int = n  
  val denom: Int = d  
  override def toString = numer + "/" + denom  
  def add(that: Rational): Rational =  
    new Rational(  
      numer * that.denom + that.numer * denom,  
      denom * that.denom  
    )  
}
```

Self references

```
def max(that: Rational) =  
  if (this.lessThan(that)) that else this
```


Auxiliary constructors

```
class Rational(n: Int, d: Int) {  
  require(d != 0)  
  val numer: Int = n  
  val denom: Int = d  
  def this(n: Int) = this(n, 1) // auxiliary constructor  
  override def toString = numer + "/" + denom  
  def add(that: Rational): Rational =  
    new Rational(  
      numer * that.denom + that.numer * denom,  
      denom * that.denom  
    )  
}
```

Defining operators

```
class Rational(n: Int, d: Int) {  
  require(d != 0)  
  val numer: Int = n  
  val denom: Int = d  
  def this(n: Int) = this(n, 1) // auxiliary constructor  
  override def toString = numer + "/" + denom  
  def + (that: Rational): Rational =  
    new Rational(  
      numer * that.denom + that.numer * denom,  
      denom * that.denom  
    )  
}
```

Defining operators

```
scala> val x = new Rational(1, 2)
```

```
x: Rational = 1/2
```

```
scala> val y = new Rational(2, 3)
```

```
y: Rational = 2/3
```

```
scala> x + y
```

```
res8: Rational = 7/6
```

Overloading methods

```
class Rational(n: Int, d: Int) {  
  require(d != 0)  
  val numer: Int = n  
  val denom: Int = d  
  def this(n: Int) = this(n, 1) // auxiliary constructor  
  override def toString = numer + "/" + denom  
  def + (that: Rational): Rational =  
    new Rational(  
      numer * that.denom + that.numer * denom,  
      denom * that.denom  
    )  
  def + (i: Int): Rational =  
    new Rational(numer + i * denom, denom)  
}
```

Using overloaded methods

```
scala> val x = new Rational(2, 3)
```

```
x: Rational = 2/3
```

```
scala> x + x
```

```
res13: Rational = 12/9
```

```
scala> x + 1
```

```
res14: Rational = 5/3
```

What about the other way around?

```
scala> 2 + x
```

```
<console>:8: error: overloaded method value + with alternatives:
```

```
(Double)Double <and>
```

```
(Float)Float <and>
```

```
(Long)Long <and>
```

```
(Int)Int <and>
```

```
(Char)Int <and>
```

```
(Short)Int <and>
```

```
(Byte)Int <and>
```

```
(java.lang.String)java.lang.String
```

```
cannot be applied to (Rational)
```

```
2 + x
```

```
^
```

Implicit conversions

```
scala> implicit def intToRational(x: Int) = new Rational(x)
```

```
scala> val r = new Rational(2,3)
```

```
r: Rational = 2/3
```

```
scala> 2 + r
```

```
res16: Rational = 8/3
```

```
intToRational(2) + r
```