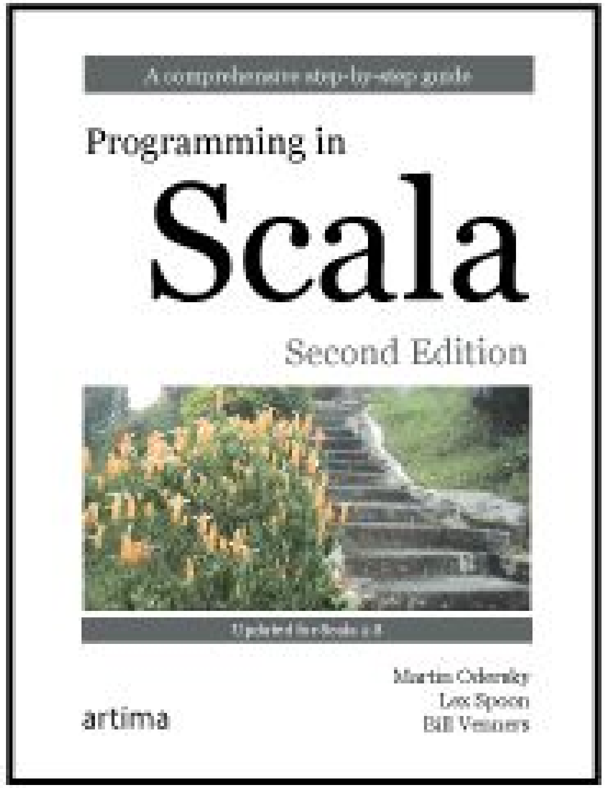
Stairway to Scala - Flight 12

Case classes and pattern matching

Bill Venners

Dick Wall



A comprehensive step-by-step guide

Programming in

Scala

Second Edition

Updated for Scala 2.8

artima

Martin Odersky
Lex Spoon
Bill Venners

escalatesoft.com

Flight 12 goal

Learn about case classes, match expressions, and patterns.

Defining case classes

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
    left: Expr, right: Expr) extends Expr
```

What you get:
1. A factory method

```
scala> val v = Var("x")
v: Var = Var(x)

scala> val op = BinOp("+", Number(1), v)
op: BinOp = BinOp(+,Number(1.0),Var(x))
```

What you get:

2. Parametric fields

```
scala> v.name
res0: String = x

scala> op.left
res1: Expr = Number(1.0)
```

What you get:

3. equals/hashCode/toString

```
scala> println(op)
BinOp(+,Number(1.0),Var(x))

scala> op.right == Var("x")
res3: Boolean = true
```

What you get:
4. copy

```
scala> op.copy(operator = "-")
res4: BinOp = BinOp(-,Number(1.0),Var(x))
```

Simplifying expressions

```
def simplifyTop(expr: Expr): Expr = expr match {
  case UnOp("-", UnOp("-", e))  => e   // Double negation
  case BinOp("+", e, Number(0)) => e   // Adding zero
  case BinOp("*", e, Number(1)) => e   // Multiplying by one
  case _ => expr
}

scala> simplifyTop(UnOp("-", UnOp("-", Var("x"))))
res4: Expr = Var(x)
```

Wildcard patterns

```
expr match {
  case BinOp("+", _, _) =>
     println(expr + "is a binary addition")
  case BinOp(_, _, _) =>
     println(expr + "is a binary operation")
  case _ =>
     println("It's something else")
}
```

Constant patterns

```
def describe(x: Any) = x match {
  case 5 => "five"
  case true => "truth"
  case "hello" => "hi!"
  case Nil => "the empty list"
  case _ => "something else"
}
```

Variable patterns

```
def describe(x: Any) =
  x match {
    case 0 => "zero"
    case somethingElse => s"not zero: $somethingElse"
  }
```

Lower case variables,
Upper case constants

```
scala> import math.{E, Pi}
import math.{E, Pi}

scala> E match {
          case Pi => "strange math? Pi = " + Pi
          case _ => "OK"
        }
res11: java.lang.String = OK
```

Constructor patterns

```
op match {
  case BinOp("+", e, Number(0)) =>
     println("a deep match")
  case _ =>
     println("no match")
}
```

Sequence patterns

```
def seek(x: Any) =
  x match {
    case List(0, _, _) => println("found it")
    case _ =>
  }

def find(x: Any) =
  x match {
    case List(0, _*) => println("found it")
    case _ =>
  }
```

Sequence patterns, :: style

```
def process(x: Any) =
  x match {
    case a :: b :: c :: rest => println("at least 3")
    case a :: Nil => println("just 1")
    case head :: tail => println("at least 1")
    case Nil => println("empty")
  }
```

Tuple patterns

```
 def tupleDemo(expr: Any) =
   expr match {
     case (a, b, c)  =>  println("matched " + a + b + c)
     case _ =>
   }

scala> tupleDemo(("a ", 3, "-tuple"))
matched a 3-tuple
```

Typed patterns

```
def generalSize(x: Any) = x match {
  case s: String => s.length
  case m: Map[_, _] => m.size
  case _ => -1
}
```

```
scala> generalSize("abc")
res16: Int = 3

scala> generalSize(Map(1 -> 'a', 2 -> 'b'))
res17: Int = 2

scala> generalSize(math.Pi)
res18: Int = -1
```

Type check and cast (poor style)

```
expr.isInstanceOf[String]

expr.asInstanceOf[String]

if (x.isInstanceOf[String]) {
  val s = x.asInstanceOf[String]
  s.length
} else ...
```

Type erasure means more matches

```
scala> def isIntIntMap(x: Any) = x match {
         case m: Map[Int, Int] => true
         case _ => false
       }
warning: there were unchecked warnings; re-run with
-unchecked for details
isIntIntMap: (x: Any)Boolean

<console>:5: warning: non variable type-argument Int in
type pattern is unchecked since it is eliminated by erasure
         case m: Map[Int, Int] => true
                 ^
scala> isIntIntMap(Map("abc" -> "abc"))
res20: Boolean = true
```

Variable binding

```
def matchExpr(expr: Expr) =
  expr match {
    case UnOp("abs", e @ UnOp("abs", _)) => e
    case _ => "huh?"
  }


// case UnOp("abs", UnOp("abs", v)) => UnOp("abs", v)
```

Pattern guards

Won't work to change (e + e) to (e * 2):

```
scala> def simplifyAdd(e: Expr) = e match {
         case BinOp("+", x, x) => BinOp("*", x, Number(2))
         case _ => e
       }
<console>:11: error: x is already defined as value x
         case BinOp("+", x, x) => BinOp("*", x, Number(2))

 scala> def simplifyAdd(e: Expr) = e match {
         case BinOp("+", x, y) if x == y =>
          BinOp("*", x, Number(2))
         case _ => e
       }
simplifyAdd: (e: Expr)Expr
```

Pattern guard examples

```
// match only positive integers and zero
case n: Int if n >= 0 => ...

// match only strings starting with the letter 'a'
case s: String if s.headOption == Some('a') => ...
```

Sealed classes

```
sealed abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String,
    left: Expr, right: Expr) extends Expr

def describe(e: Expr): String = e match {
  case Number(_) => "a number"
  case Var(_)    => "a variable"
}
```

```
warning: match is not exhaustive!
missing combination          UnOp
missing combination         BinOp
```

The Option type

```
scala> val capitals =
        Map("France" -> "Paris", "Japan" -> "Tokyo")
capitals: scala.collection.immutable.Map[
        java.lang.String,java.lang.String]
 = Map(France -> Paris, Japan -> Tokyo)

scala> capitals get "France"
res23: Option[java.lang.String] = Some(Paris)

scala> capitals get "North Pole"
res24: Option[java.lang.String] = None
```

Deconstructing Option

```
scala> def show(x: Option[String]) = x match {
         case Some(s) => s
         case None => "?"
       }
 show: (x: Option[String])String

 scala> show(capitals get "Japan")
 res25: String = Tokyo

 scala> show(capitals get "France")
 res26: String = Paris

 scala> show(capitals get "North Pole")
 res27: String = ?
```

Patterns in variable definitions

```
scala> val myTuple = (123, "abc")
myTuple: (Int, java.lang.String) = (123,abc)

scala> val (number, string) = myTuple
number: Int = 123
string: java.lang.String = abc

scala> val exp = BinOp("*", Number(5), Number(1))
exp: BinOp = BinOp(*,Number(5.0),Number(1.0))

scala> val BinOp(op, left, right) = exp
op: String = *
left: Expr = Number(5.0)
right: Expr = Number(1.0)
```

Patterns in for expressions

```
scala> for ((country, city) <- capitals)
     println(s"The capital of $country is $city")
The capital of France is Paris
The capital of Japan is Tokyo
```

Case objects

```
sealed abstract class Op
case object NoOp extends Op
case class Do(it: String) extends Op
case object Stop extends Op

def doIt(x: Op) = x match {
  case NoOp => "Ain’t nothing going on."
  case Do(it) => s"Doing $it"
  case Stop => "That’s it. We’re done."
}
```

Exercises for Flight 12