CS 3101-2 - Programming Languages: Scala

Lecture 4: Traits, Case Classes and Pattern Matching

Daniel Bauer (bauer@cs.columbia.edu)

November 12, 2014

Programming in Scala, 2nd edition: Ch. 12/15

1 Pattern Matching and Case Classes

2 Traits

Pattern matching

```
expression match {
    case pattern1 => expression1
    case pattern2 => expression2
    ...
}
```

Pattern matching: Constant patterns

```
scala> val month : Int = 8
month: Int = 8

scala> val monthString: String = month match {
    case 1 => "January"
    case 2 => "February"
    case 3 => "March"
    case 4 => "April"
    case 5 => "May"
    case 6 => "June"
    case 7 => "July"
    case 8 => "August"
}
monthString: String = August
```

- can use any literal or singleton object as pattern.
- Can also use any `val` if its name is upper case.
- compared using equals method.

Collections Containing Mixed Types

- Static type system can be cumbersome to deal with in complex collections (and other data structures).

```
abstract class Publication
class Novel(val author: String, val title: String)
    extends Publication
class Anthology(val title:String)
    extends Publication

val a = new Anthology("Great Poems")
val b = new Novel("The Castle","F. Kafka")

scala> val books = List(a,b)
books: List[Publication] = List(Anthology@2c78beb8,
                                Novel@2a3ec96e)
```

- How to iterate through books and print descriptions?

Case Classes

- Use the case modifier to define case classes.

```
abstract class Publication
case class Novel(title: String, author: String) extends
    Publication
case class Anthology(title: String) extends
    Publication

val a = Anthology("Great Poems")
val b = Novel("The Castle", "F. Kafka")

scala> val books: List[Publication] = List(a,b)
books: List[Publication] = List(Anthology(Great Poems),
                                Novel(The Castle, F. Kafka))
```

Case Classes

- Use the case modifier to define case classes.

```
abstract class Publication
case class Novel(title: String, author: String) extends
    Publication
case class Anthology(title: String) extends
    Publication

val a = Anthology("Great Poems")
val b = Novel("The Castle", "F. Kafka")

scala> val books: List[Publication] = List(a,b)
books: List[Publication] = List(Anthology(Great Poems),
                                 Novel(The Castle, F. Kafka))
```

- Case classes implicitly
  - add a factory method to the companion object for the class – allows initialization without `new`.
  - mark all constructor parameters as `val`s .
  - create an intuitive toString, hashCode, equals method.
  - support pattern matching.

Case Classes and Pattern Matching

```
abstract class Publication
case class Novel(title: String, author: String) extends
    Publication
case class Anthology(title: String) extends
    Publication

val a = Anthology("Great Poems")
val b = Novel("The Castle", "F. Kafka")
val books: List[Publication] = List(a,b)

scala> for (book <- books) {
    val description = book match {
        case Anthology(title) => title
        case Novel(title, author) => title + " by " + author
     }
     println(description)
}
Great Poems
The Castle by F. Kafka
```

Sealed Classes

- A sealed class may not have any subclasses defined outside the same source file.
- Usually 'safe' to use pattern matching on sealed classes:
  - Nobody can define additional sub-classes later, creating unknown match cases.

```
sealed abstract class Publication
case class Novel(title: String, author: String) extends
    Publication
case class Anthology(title: String) extends
    Publication
```

Variable, Concrete Patterns, and Constructor Patterns

```
abstract class Publication {
    val title : String
}
case class Novel(title: String, author: String) extends
    Publication
case class Anthology(title: String) extends
    Publication

val a = Anthology("Great Poems")
val b = Novel("The Castle", "F. Kafka")
val books: List[Publication] = List(a,b)

scala> for (book <- books) {
    val description = book match { // order matters!
        case Novel(title, "F. Kafka") => title + " by Kafka"
        case Novel(title, author) => title + " by " + author
        case other => other.title
     }
     println(description)
}
Great Poems
The Castle by Kafka
```

Wildcard Patterns

Used to ignore parts of patterns. Match anything.

```
abstract class Publication {
    val title : String
}
case class Novel(title: String, author: String) extends
    Publication
case class Anthology(title: String) extends
    Publication

scala> for (book <- books) {
    val description = book match { // order matters!
        case Novel(title, _) => title
        case Anthology(title) => title
        case _ => "unknown publication type"
     }
     println(description)
}
```

Case classes: A more complex example

```
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
```

Case classes: A more complex example

```
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
```

- Use case classes to easily describe structured/nested expressions.
- = method on case classes works with nested expressions.

```
scala> val expr = BinOp("+", Number(1), UnOp("-",Number(5)))
expr:  BinOp = BinOp(+,Number(1.0),UnOp(-,Number(5.0)))

scala> expr.left == Number(1.0)
res0: Boolean = true

scala> expr.right == UnOp("-",Number(5))
res1: Boolean = true
```

Simplyfing Nested 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"))))
res0: Expr = Var(x)
```

Patterns Outside of `match` Expressions

- Patterns can be used in `val`/`var` assignments to extract information from complex objects.

```
scala> val exp = new 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)
```

Matching Tuples

- A tuple is a fixed-length sequence of values.

```
scala> val x : (Int, Int) = (24,42)
x: (Int, Int) = (24,42)
```

Matching Tuples

- A tuple is a fixed-length sequence of values.

```
scala> val x : (Int, Int) = (24,42)
x: (Int, Int) = (24,42)
```

- Using tuples in pattern matching:

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


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

Matching Tuples

- A tuple is a fixed-length sequence of values.

```
scala> val x : (Int, Int) = (24,42)
x: (Int, Int) = (24,42)
```

- Using tuples in pattern matching:

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


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

- tuple pattern without `match` to "unpack" values:

```
scala> val (number, string) = (12, "hi")
number: Int = 12
string: String = hi
```

Matching Lists

```
scala> val x = List(1,2,3)
x: List[Int] = List(1, 2, 3)

scala> x match {
         case List(a,_*) => "head of list is "+a
      }
head of list is 1
```

Matching Lists

```
scala> val x = List(1,2,3)
x: List[Int] = List(1, 2, 3)

scala> x match {
         case List(a,_*) => "head of list is "+a
      }
head of list is 1

scala> x match {
          case List(_,_,_) => "three element list"
     }
res40: String = three element list
```

Matching Lists

```
scala> val x = List(1,2,3)
x: List[Int] = List(1, 2, 3)

scala> x match {
         case List(a,_*) => "head of list is "+a
      }
head of list is 1

scala> x match {
          case List(_,_,_) => "three element list"
     }
res40: String = three element list

scala> x match {
          case List(_,_) => "three element list"
     }
scala.MatchError: List(1, 2, 3) ...
```

Maps

- Collection that relates unique keys to values.
- Keys need to be immutable and hashable.

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

Patterns in For Expressions

Scala's maps are iterables over `(key, value)` pairs.

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

scala> for ((country,city) <- capitals)
          println("Capital of " + country + ": " + city)
Capital of Japan: Tokyo
Capital of France: Paris
```

Patterns in For Expressions

Scala's maps are iterables over `(key, value)` pairs.

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

scala> for ((country,city) <- capitals)
          println("Capital of " + country + ": " + city)
Capital of Japan: Tokyo
Capital of France: Paris

scala> val capitals =List(("Japan","Tokyo"), 25)
capitals: List[Any] = List((Japan,Tokyo), 25)

scala> for ((country,city) <- capitals) // filtering
         println("Capital of " + country + ": " + city)
Capital of Japan: Tokyo
```

The `Option` type

- sealed abstract class `Option` has two subtypes
  - `Some[A](x :A )`
  - `None`
- `Option` is often used if a function call might not return a value (Java would return `null`, Python None).
- Example: `Map.get(key)` returns `Some(value)` if `key` is in the map, otherwise `None`

```
scala> val capitals = Map("Japan"->"Tokyo",
                          "France"->"Paris")
scala> capitals get "Japan"
res0: Option[String] = Some(Tokyo)

scala> capitals get "Italy"
res1: Option[String] = None
```

Matching Options

```
scala> val capitals = Map("Japan"->"Tokyo",
                           "France"->"Paris")

scala> val countries = List("Japan","Italy","France")

scala> for (c <- countries) {
          val description = capitals.get(c) match {
              case Some(city) => c + ": "+city
              case None => "no entry for "+c
          }
          println(description)
      }
Japan: Tokyo
no entry for Italy
France: Paris
```

Matching Types

- Occasionally we can't assume that all members of a collection will be types of a restricted class hierarchy.
- Example: Collections of type `Any`
- Java uses `instanceof` to explicitly typecheck.

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

Traits vs. Inheritance

- Inheritance means adding to the implementation of a single parent class (or overriding).
- Scala does not support multiple inheritance (unlike e.g. Python), but offers traits.
- Traits are a *'fundamental unit of code reuse'*.
  - Defines methods and attributes that can be re-used by various classes.
  - Classes can mix in any number of traits.
- Similar to Java interfaces.
- No parameters.

```
trait Philosophical {
    def philosophize() {
        println("I consume memory, therefore I am!")
    }
}
```

Defining and Using Traits

```
trait Philosophical {
  def philosophize() {
    println("I consume memory, therefore I am!")
  }
}
trait HasLegs { val legs : Int = 4 }

class Animal

class Frog extends Animal with Philosophical with HasLegs{
  override def toString = "green"
}
scala> val frog = new Frog
frog: Frog = green

scala> frog.philosophize
I consume memory, therefore I am!

scala> frog.legs
res0: Int = 4
```

Using Traits II

- A single Trait can be mixed in using `extends`.

```
trait Philosophical {
    def philosophize() {
        println("I consume memory, therefore I am!")
    }
}
// mix in Philosophical
class Philosopher extends Philosophical
```

```
scala> class Philosopher extends Philosophical
defined class Philosopher

scala> val p = new Philosopher
p: Philosopher = Philosopher@2dc4de05

scala> p.philosophize
I consume memory, therefore I am!
```

Traits are Types

```
trait Philosophical {
    def philosophize() {
        println("I consume memory, therefore I am!")
    }
}
class Animal
class Frog extends Animal with Philosophical {
    val color = "green"
}
```

```
scala> val phil : Philosophical = new Frog() // trait as type
f: Philosophical = Frog@16a15a6e

scala> phil.philosophize
I consume memory, therefore I am!
```

Traits are Types

```
trait Philosophical {
    def philosophize() {
        println("I consume memory, therefore I am!")
    }
}
class Animal
class Frog extends Animal with Philosophical {
    val color = "green"
}
```

```
scala> val phil : Philosophical = new Frog() // trait as type
f: Philosophical = Frog@16a15a6e

scala> phil.philosophize
I consume memory, therefore I am!

scala> phil.color // not accessible because defined on Frog
<console>:12: error: value color is not a member of
            Philosophical
```

Polymorphism with Traits

```
trait Philosophical {
    def philosophize() {
        println("I consume memory, therefore I am!")
    }
}

class Animal
class Frog extends Animal with Philosophical {
    override def toString = "green"
    override def philosophize() {
        println("It ain't easy being " + toString + "!")
    }
}
```

```
scala> val phrog : Philosophical = new Frog()
phrog: Philosophical = green

scala> phrog.philosophize
It ain't easy being green!
```

Thin vs. Rich Interfaces to Classes

Thin Interfaces:

- Minimal functionality, few methods.
- Easy for the developer of the interface.
- Larger burden on client using the class (needs to fill in the gaps or adapt general methods).

Rich Interfaces:

- Many specialized methods.
- Larger burden when implementing the class.
- Convenient for the client.

- Traits can be used to enrich thin interfaces, re-using existing methods.

Thin vs. Rich Interfaces - Example: Rectangular Objects

```
class Point(val x: Int, val y: Int)

class Rectangle(val topLeft: Point, val bottomRight: Point) {
    def left = topLeft.x
    def right = bottomRight.x
    def width = right - left
    // and many more geometric methods...
}
```

- Another class outside of the same type hierarchy with similar functionality:

```
abstract class Widget {
    def topLeft : Point
    def bottomRight : Point

    def left = topLeft.x
    def right = bottomRight.x
    def width = right - left
    // and many more geometric methods...
}
```

Thin vs. Rich Interfaces - Example: Rectangular Objects

```
def Rectangular {
    def topLeft : Point
    def bottomRight : Point

    def left = topLeft.x
    def right = bottomRight.x
    def width = right - left
    // and many more geometric methods...
}

 abstract class Widget extends Rectangular {
    // other methods...
  }

class Rectangle(val topLeft: Point,
               val bottomRight: Point) extends Rectangular {
        // other methods...
 }
```

Modifying Methods with Traits

```
import scala.collection.mutable.ArrayBuffer

abstract class IntQueue {
  def get(): Int
  def put(x: Int)
}

class BasicIntQueue extends IntQueue {
  private val buf = new ArrayBuffer[Int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}
```

Modifying Methods with Traits

```
import scala.collection.mutable.ArrayBuffer

abstract class IntQueue {
  def get(): Int
  def put(x: Int)
}

class BasicIntQueue extends IntQueue {
  private val buf = new ArrayBuffer[Int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}

scala> val queue = new BasicIntQueue
queue: BasicIntQueue = BasicIntQueue@24655f

scala> queue.put(10)
scala> queue.put(20)
```

Modifying Methods with Traits

```
import scala.collection.mutable.ArrayBuffer

abstract class IntQueue {
  def get(): Int
  def put(x: Int)
}

class BasicIntQueue extends IntQueue {
  private val buf = new ArrayBuffer[Int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}

scala> val queue = new BasicIntQueue
queue: BasicIntQueue = BasicIntQueue@24655f

scala> queue.put(10)
scala> queue.put(20)

scala> queue.get()
res0: Int = 10

scala> queue.get()
res1: Int = 20
```

Modifying Methods with Traits

- Traits can modify (override) methods of a base class.
- Add some functionality but then call method of the super class.

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

scala> class MyQueue extends BasicIntQueue with Incrementing
defined class MyQueue
```

Modifying Methods with Traits

- Traits can modify (override) methods of a base class.
- Add some functionality but then call method of the super class.

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

scala> class MyQueue extends BasicIntQueue with Incrementing
defined class MyQueue

scala> val queue = new MyQueue
scala> val queue = new BasicIntQueue with Incrementing
queue: BasicIntQueue with Incrementing = $anon$1@5fa12d
```

Modifying Methods with Traits

- Traits can modify (override) methods of a base class.
- Add some functionality but then call method of the super class.

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

scala> class MyQueue extends BasicIntQueue with Incrementing
defined class MyQueue

scala> val queue = new MyQueue
scala> val queue = new BasicIntQueue with Incrementing
queue: BasicIntQueue with Incrementing = $anon$1@5fa12d

scala> queue.put(10)
scala> queue.get()
res: Int = 21
```

Modifying Methods with Traits

- Multiple traits can be mixed in to stack functionality.
- Methods on `super` are called according to linear order of `with` clauses (right to left).

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

trait Filtering extends IntQueue {
    abstract override def put(x: Int) {
        if (x >= 0) super.put(x)
    }
}
```

Modifying Methods with Traits

- Multiple traits can be mixed in to stack functionality.
- Methods on `super` are called according to linear order of `with` clauses (right to left).

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

trait Filtering extends IntQueue {
    abstract override def put(x: Int) {
        if (x >= 0) super.put(x)
    }
}

scala> val queue = new (BasicIntQueue
                        with Incrementing
                        with  Filtering)
queue: BasicIntQueue with Incrementing with Filtering...
```

Modifying Methods with Traits

- Multiple traits can be mixed in to stack functionality.
- Methods on `super` are called according to linear order of `with` clauses (right to left).

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

trait Filtering extends IntQueue {
    abstract override def put(x: Int) {
        if (x >= 0) super.put(x)
    }
}

scala> val queue = new (BasicIntQueue
                        with Incrementing
                        with  Filtering)
queue: BasicIntQueue with Incrementing with Filtering...

scala> queue.put(-1); queue.put(0);
scala> queue.get()
res: Int = 1
```

Traits or Abstract Classes

Both traits and abstracts classes can have abstract and concrete members.

Traits:

- No constructor paramters or type parameters.
- Multiple traits can be mixed into class definitions.
- Semantics of `super` depends on order of mixins. Can call abstract methods.

Abstract Classes:

- Have constructor parameters and type parameters.
- Work better when mixing Scala with Java.
- `super` refers to unique parent. Can only call concrete methods.