G22.2110-003 Programming Languages - Fall 2012

Week 13 - Part 1

Thomas Wies

New York University

Review

Last lecture

- Object Oriented Programming

Outline

Today:

- Scala

Sources:

- Programming in Scala, Second Edition by Martin Odersky, Lex Spoon and Bill Venners, Artima, 2010.
- `http://www.scala-lang.org`

The Scala Language

What is Scala?

- language for scalable component software
- developed by Martin Odersky's group at EPFL, Switzerland
- influenced by ML/Haskell, Java, and other languages
- unifies object-oriented and functional programming
- interoperates with Java and .NET
- is gaining momentum in industry (cf. `http://www.typesafe.org`)

Why SCALA?

- Runs on the Java Virtual Machine
  - can use any JAVA code in SCALA (and vice versa)
  - similar efficiency
- Much shorter code
  - 50% reduction in most code over JAVA
  - local type inference
- Fewer errors
  - strongly and statically typed
  - encourages state-less programming style
- Clean language design
  - uniform object model
- More flexibility
  - easily extensible (operator overloading, implicit type conversions, user-defined control constructs)
  - mix-in composition of classes

Getting Started in Scala

- `scala` - runs compiled Scala code (like `java`)
  if no arguments are supplied, starts the Scala interpreter
- `scalac` - compiles Scala code (like `javac`)
- `sbt` - build tool for larger project (incremental compilation, dependency and library management, ...)

There are plugins available for popular IDEs such as Eclipse.

For more information visit `http://www.scala-lang.org`

Scala Basics

- Use `var` to declare variables:

```
var x = 3
x += 4
```

- Use `val` to declare values:

```
val x = 3
x += 4 // error: reassignment to val
```

- Scala is statically typed:

```
var x = 3
x = "Hello World!" // error: type mismatch
```

- Explicit type annotations:

```
val x: Int = 3
```

Tuples

```
scala> (3,'c')
res0: (Int, Char) = (3,c)

scala> (3,'c')._1
res1: Int = 3

scala> (3,'c')._2
res2: Char = c

scala> val (i,c) = (3,'c')
i: Int = 3
c: Char = c
```

Function Definitions (Part 1)

Use `def` to declare functions:

```
def max(x: Int, y: Int): Int = {
  if (x < y) {
    return y;
  } else {
    return x;
  }
}
```

or shorter:

```
def max(x: Int, y: Int) = if (x < y) y else x
```

Function definitions can also be nested.

Function Definitions (Part 2)

Scala supports

- tail-call elimination (in most cases):

```
def gcd(x: Int, y: Int): Int =
  if (y == 0) x else gcd(y, x % y)
```

- lambda abstraction (*function literals*):

```
val sum = (x: Int, y: Int) => x + y
```

- currying and partial function application:

```
def curriedSum (x: Int)(y: Int): Int = x + y
```

```
scala> val add3 = curriedSum(3)_
add3: Int =>Int = <function1>
scala> add3(2)
res0: Int = 5
```

Higher-Order Functions

Functions can take functions as arguments:

```
def compose(f: Int => Int, g: Int => Int) =
  (x: Int) => f(g(x))
```

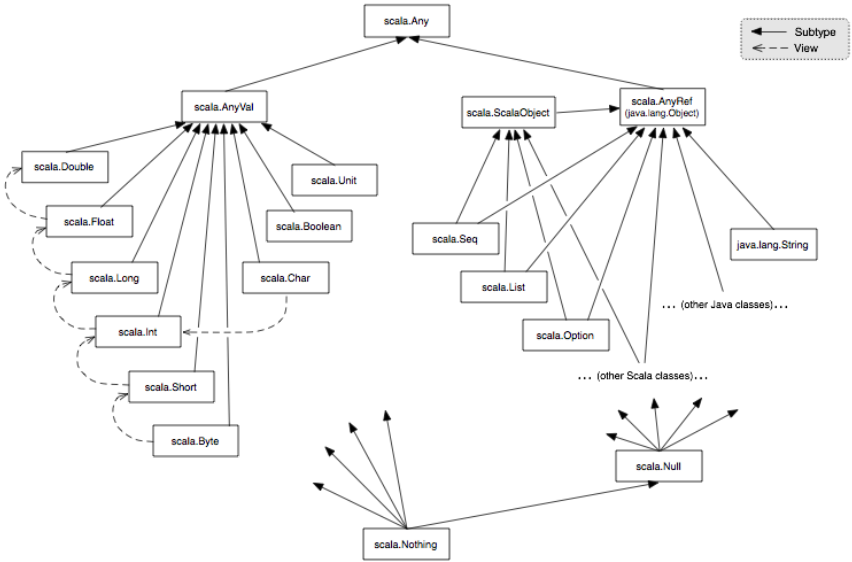
Examples:

- Maps
  `List(1,2,3).map((x: Int) => x + 10).foreach(println)`
  or more readable:
  `List(1,2,3) map (_ + 10) foreach (println)`
- Filtering
  `1 to 100 filter (_ % 7 == 3) foreach (println)`

Scala is Object-Oriented

- Uniform object model
  - every value is an object (including primitive values and functions)
  - every application of an operator is a method call
    `1 + 2` is short for `1.+(2)`
- No static class members (instead: *singleton objects*)
- Single inheritance
- Dynamic method binding by default
- Traits, mix-in composition, and views give more flexibility.

Scala Class Hierarchy



A simple class in JAVA

```
class Point {
  private double x;
  private double y;

  public Point (double xc, double yc) {
    x = xc;
    y = yc;
  }

  public double getX () = { return x; }
  public double getY () = { return y; }

  public String toString() {
    return "Point(" + x + "," + y + ")";
  }
}
```

... and its Scala pendant

```
class Point(xs: Double, ys: Double) {
  val x = xs
  val y = ys

  override def toString =
    "Point(" + x + "," + y + ")"
}
```

```
scala> val p = new Point(1,2)
p : Point = Point(1.0,2.0)
scala> val px = p.x
px : Double = 1.0
```

Notice:

- Classes can take arguments.
- The compiler automatically generates a primary constructor.
- All members are public by default.

Access Modifiers

- A member labeled `private` is visible only inside the class that contains that member definition.
- A member labeled `protected` is only accessible in subclasses of the class in which the member is defined.
- Every member not labeled `private` or `protected` is public. There is no explicit modifier for public members.
- Access modifiers can be augmented with qualifiers for more fine-grained access control.

```
package geometry;
class P {
  private[this] val f: Int
    // visible only in the same instance
  private[geometry] val g: Int
    // same as package visibility in Java
}
```

Auxiliary Constructors

Auxiliary constructors are defined using `def this(...)`

```
class Point(val x: Double, val y: Double) {
  def this() = this(0,0)

  override def toString =
    "Point(" + x + "," + y + ")"
}
```

```
scala> val p = new Point()
p : Point = Point(0.0,0.0)
```

Defining Operators

Method names can be operators and can be overloaded:

```
class Point(val x: Double, val y: Double) {
  def this() = this(0,0)

  override def toString =
    "Point(" + x + "," + y + ")"

  def +(other: Point) =
    new Point(x + other.x, y + other.y)
}
```

```
scala> val p = new Point(1,2)
p : Point = Point(1.0,2.0)
scala> val q = p + p
q : Point = Point(2.0,4.0)
```

Operator precedence is predefined, e.g. `*` binds stronger than `+`.

Singleton Objects

SCALA has *singleton objects* instead of static members.

A singleton object definition looks like a class definition, except that the keyword `class` is replace by `object`.

```
object Main {
  def main(args: Array[String]) {
    println("Hello,␣world!")
  }
}
```

or shorter:

```
object Main extends App {
  println("Hello,␣world!");
}
```

Companion Objects

A singleton object of the same name as a class is called the *companion object* of that class. They can access each others private members.

```
class CheckSumAccumulator {
  private var sum = 0
  def add(b: Byte) { sum += b }
  def checksum(): Int = ~(sum & 0xFF) + 1
}

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
    }
}
```

Functions as Objects

Instances of classes and objects that define the `apply` method can be used like functions.

```
object max {
  def apply(x: Int, y: Int) =
    if(x < y) then y else x
}
```

```
scala> max(1,2)
res0: Int = 2
```

In particular, a declaration of a function literal

```
val inc = (x: Int) => x + 1
```

is expanded by the compiler to

```
object inc extends Function1 {
  def apply(x: Int) = x + 1
}
```

Factory Methods

Companion objects can be used to define *factory methods* that construct instances of the companion class.

```
class Point(val x: Double, val y: Double) {
   ...
}

object Point {
  def apply() = new Point(0,0)
  def apply(x: Double, y: Double) = new Point(x,y)
}
```

```
scala> val p = Point(1,2)
p: Point = Point(1.0,2.0)
```

Implicit Parameters

Parameters of methods can be declared *implicit*. If a call to a method misses arguments for its implicit parameters, such arguments are automatically provided.

```
def speakImplicitly (implicit greeting : String) =
  println(greeting)
```

```
scala> speakImplicitly("Goodbye␣world")
Goodbye world
```

An appropriate implicit value must be in scope:

```
scala> speakImplicitly
:6: error: no implicit argument matching parameter type
String was found.
scala> implicit val hello = "Hello␣world"
hello: java.lang.String = Hello world
scala> speakImplicitly
Hello world
```

Views

An implicit value of a function type `S => T` is called a *view*.

```
implicit def pairToPoint(p: (Double, Double)) =
  new Point(p._1, p._2)
```

```
scala> (1,2) + (3,3)
res0: Point(4.0,5.0)
```

If the compiler encounters a type mismatch it searches for an appropriate view to convert the mismatched type.

The implicit conversion is inserted automatically by the compiler.

Views are typically defined in the companion object of the involved types.

Pimp My Library

Implicit conversions can be used to extend easily the functionality of existing libraries.

```
class RichArray[T](a: Array[T]) {
  def append(b: Array[T]): Array[T] = {
    val res = new Array[T](a.length + b.length)
    Array.copy(a, 0, res, 0, a.length)
    Array.copy(b, 0, res, a.length, b.length)
    res
  }
}

implicit def enrichArray[T](a: Array[T]) =
  new RichArray[T](a)

val a = Array(1, 2, 3)
val b = Array(4, 5, 6)
val c = a append b
```

Traits

Traits are like classes except that

1. they do not take parameters (and have no constructors)
2. calls to super in traits are dynamically bound

```
trait Rectangular {
  def topLeft: Point
  def bottomRight: Point
  def left = topLeft.x
  def right = bottomRight.x
  def width = right - left
}

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

Mix-In Composition

Traits can be mixed into classes.

Mix-in composition captures the cases where multiple inheritance is useful while avoiding its pitfalls.

Use case: stackable modifications

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

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

Stackable Modifications

```
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 Filtering with Incrementing...
scala> queue.put(-2); queue.put(0); queue.get()
res0: Int = 1
```

Compound Types

Sometimes it is necessary to express that the type of an object is a subtype of several other types.

In Scala this can be expressed with the help of *compound types*, which are intersections of object types.

```
trait Cloneable extends java.lang.Cloneable {
  override def clone(): Cloneable = {
    super.clone(); this
  }
}
trait Resetable {
  def reset: Unit
}
def cloneAndReset(obj: Cloneable with Resetable):
Cloneable = {
  val cloned = obj.clone()
  obj.reset
  cloned
}
```

Case Classes

SCALA supports ML-style algebraic data types.

They are implemented using *case classes*.

```
sealed abstract class List
case object Nil extends List
case class Cons(hd: Int, tl: List) extends List
```

```
scala> val l: List = Cons(1, Cons(3, Cons(2, Nil)))
l: List = Cons(1, Cons(3, Cons(2, Nil)))
```

Case Classes

Scala supports ML-style algebraic data types.

They are implemented using *case classes*.

```
sealed abstract class List
case object Nil extends List
case class Cons(hd: Int, tl: List) extends List
```

```
scala> val l: List = Cons(1, Cons(3, Cons(2, Nil)))
l: List = Cons(1, Cons(3, Cons(2, Nil)))
```

The compiler generates a factory method for each case class.

Also, case classes override the methods `equals`, `hashCode`, and `toString` with appropriate implementations.

Pattern Matching

Case classes support pattern matching.

```
def filter(p: Int => Boolean, l: List): List =
  l match {
    case Cons(x, t) if p(x) =>
      Cons(x, filter(p, t))
    case Cons(x, t) => filter(p, t)
    case _ => Nil
  }
```

As in ML, patterns are tried in the order in which they are written.

By-Name Parameters

Scala provides *by-name parameters*. An argument that is passed by name is not evaluated at the point of function application, but instead is evaluated at each use within the function.

```
def nano() = {
  println("Getting␣nano␣time")
  System.nanoTime
}
def delayed(t: => Long) = {
  println("In␣delayed␣method")
  println("Param:␣" + t)
  t
}
```

```
scala> println(delayed(nano()))
In delayed method
Getting nano time
Param: 843494419494656 9
Getting nano time
843494419501745 9
```

Writing New Control Structures

The combination of automatic closure construction and by-name parameters allows programmers to make their own control structures.

```
object Main extends App {
  def whileLoop(cond: => Boolean)(body: => Unit) {
    if (cond) {
      body
      whileLoop(cond)(body)
    }
  }

  var i = 10
  whileLoop (i > 0) {
    println(i)
    i -= 1
  }
}
```

Example: `repeat until`

```
object Main extends App {
  def repeat(body: => Unit): RepeatUntilCond =
    new RepeatUntilCond(body)
  protected class RepeatUntilCond(body: => Unit) {
    def until(cond: => Boolean) {
      body
      if (!cond) until(cond)
    }
  }

  var i = 10
  repeat {
    println(i)
    i -= 1
  } until (i == 0)
}
```