Scala

—

The State of The Art of Statically Typed Languages

Johan Östlund

johano@dsv.su.se

The Scala Programming Language

- A research language by Martin Odersky
- Eclipse plugin - completely rewritten
- Scala has an interactive interpreter
- Multi-paradigm language, in some sense
- Smooth integration of object-oriented and functional style
  - Every value is an object
  - Every function is a value
- Runs in JVM and on CLR
- Integrates with existing classes in the Java API and resources on the CLR

http://scala-lang.org/

Scala Rationale

- Blend object-oriented style and functional programming style in one language
  - Object-orientation is not as useful for many new programming models (e.g., XML)
  - Programmers know object-orientation
  - Functional programming and concurrency play well together
  - Functional programming plays well with tree-transformation - XSLT
- Enable static typing in this context
  - Unlike *e.g.*, Smalltalk, Python and Ruby

Brevity

- How many lines (or keystrokes) to express a certain concept
    - *"A Person has a first name, a last name, and a spouse. Persons always have a first and last name, but may have a spouse. Persons know how to say hi, by introducing themselves and their spouse."*
- Stolen from Ted Neward

Java

```
public class Person
{
    private String lastName;
    private String firstName;
    private Person spouse;

    public Person(String fn, String ln, Person s)
    {
        lastName = ln; firstName = fn; spouse = s;
    }
    public Person(String fn, String ln)
    {
        this(fn, ln, null);
    }

    public String getFirstName()
    {
        return firstName;
    }

    public String getLastName()
    {
        return lastName;
    }

    public Person getSpouse()
    {
        return spouse;
    }
    public void setSpouse(Person p)
    {
        spouse = p;
            // We ignore sticky questions of reflexivity and
            // changing last names in this method for simplicity
    }

    public String introduction()
    {
        return "Hi, my name is " + firstName + " " + lastName +
            (spouse != null ?
            " and this is my spouse, " + spouse.firstName + " " +
            spouse.lastName + "." : ".");
    }
}
```

C#

```
public class Person
{
    private string lastName;
    private string firstName;
    private Person spouse;

    public Person(string fn, string ln, Person s)
    {
        lastName = ln; firstName = fn; spouse = s;
    }
    public Person(string fn, string ln)
        : this(fn, ln, null)
    {
    }

    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }

    public string LastName
    {
        get { return lastName; }
    }

    public Person Spouse
    {
        get { return spouse; }
        set { spouse = value; }
    }

    public string Introduction()
    {
        return "Hi, my name is " + firstName + " " + lastName +
            (spouse != null ?
            " and this is my spouse, " + spouse.firstName + " " +
            spouse.lastName + "." :
            ".");
    }
}
```

Ruby

```
class Person
  def initialize(firstname, lastname, spouse = nil)
   @firstname, @lastname, @spouse = firstname, lastname, spouse
  end

  attr_reader :lastName
  attr_accessor :firstName, :spouse

  def introduction
    "Hello, my name is #{@firstName} #{@lastName}" + (@spouse ?
    " and this is my spouse, #{@spouse.firstName} #
      {@spouse.lastName}" : "")
  end
end
```

Scala

```
class Person(ln : String, fn : String, var s : Person)
{
    def lastName = ln;
    def firstName = fn;
    def spouse = s;

    def spouse(sp : Person) = s = sp;

    def this(ln : String, fn : String) = { this(ln, fn, null); }

    def introduction() : String =
        return "Hi, my name is " + firstName + " " + lastName +
            (if (spouse != null) " and this is my spouse, " +
                 spouse.firstName + " " + spouse.lastName + "."
                 else ".");
}
```

Scala

```
class Person(ln : String, fn : String,
                        var s : Person)
{
    def lastName = ln;
    def firstName = fn;
    def spouse = s;

    def spouse(sp : Person) = s = sp;

    def this(ln : String, fn : String) =
                     { this(ln, fn, null); }

    def introduction() : String =
        return "Hi, my name is " +
                firstName + " " + lastName +
                (if (spouse != null)
                " and this is my spouse, " +
                 spouse.firstName + " " +
                 spouse.lastName + "."
                 else ".");
}
```

Mixin Inheritance

- Units of composeable behavior, *cf.* Objective-C's categories, but may contain member fields, and method adoption is different
    - Can avoid multiple inheritance problems

```
trait Similarity extends Comparable
{
   def compareTo(x: Any): Int

   def equals(x: Any) = compareTo(x) == 0

   def lessThan(x: Any) = compareTo(x) < 0

   def greaterThan(x: Any) = compareTo(x) > 0

   def lessThanOrEquals(x: Any) =
                             compareTo(x) <= 0

   def greaterThanOrEquals(x: Any) =
                             compareTo(x) >= 0

   def between(x: Similarity, y: Similarity) =
       x.lessThan(this) && this.lessThan(x)
}
```

Mixins (cont'd)

- Create a class from extending one class with another

```
class Staple(height: Int)
{
    def compareTo(s : Any): Int =
        height - s.asInstanceOf[Staple].height
}

class ComparableStaple(height : Int)
        extend Staple(height) with Similarity
```

Structural Types

```
class Person(n: String, a: Int)
{
  def name = n
  def age = a
}

class Car(n: String, c: String)
{
  def name = n
  def color = c
}

def printName(named: { def name: String }) =
  Console.println(named.name)

printName(new Person("Amy", 13))
printName(new Car("Volvo", "Red"))
```

Family Polymorphism

```
class Graph
{
    class Node
    {
        def connect(n : Node) =
                    new Edge(n, this)
    }
}

val g1 = new Graph
var n1 = new g1.Node
var n2 = new g1.Node

val g2 = new Graph
var n3 = new g2.Node

var e1 = n1.connect(n2)
var e2 = n1.connect(n3); // illegal
```

Currying

- Applying functions to arguments—one at a time

```
def method(x: Int)(y: Int) =
  Console.println(x + y)

val n = method(4700) _
n(11); // prints 4711
```

Higher-order functions

```
class Decorator(left: String, right: String)
{
    def layout[A](x: A) =
        left + x.toString() + right
}

def apply(func: Int => String, v: Int) =
        func(v)

val decorator = new Decorator("[", "]")

Console.println(apply(decorator.layout, 4711))
```

Anonymous functions

- Sometimes it's convenient to be able to create functions "on-the-fly" without having to name them

```
var f = (x: Int) => x + 1

Console.println(f(10)) // prints 11
```

Anonymous functions

```
def map[T,U](xs: List[T], f: T => U): List[U] =
  for (x <- xs) yield f(x)

map(List(1, 2, 3),
   (x: Int) => x * 2) foreach { println }


def filter[T](xs: List[T], f: T => Boolean):
 List[T] =
  for (x <- xs if f(x)) yield x

filter(List(1, 2, 3),
      (x: Int) => x % 2 != 0 ) foreach
                                    { println }
```

Local Type Inference

```
object InferenceTest1 extends Application {
  val x = 1 + 2 * 3;
                // the type of x is Int

  val y = x.toString();
                // the type of y is String

  def succ(x: Int) = x + 1;
                // method succ returns Int values
}
class MyPair[A, B](x: A, y: B);

object InferenceTest2 {
  def id[T](x: T) = x;

  val p = new MyPair(1, "scala") ;
                    // type: MyPair[Int, String]

  val q = id(1);           // type: Int
  val s = id(“Hello”);    // type: String

  var f = (x: Int) => x + 1 // type: Int => Int
}
```

Compound Types

```
class Colored(c: Int)
{
  def color = c;
}

class Weighted(w: Int)
{
  def weight = w;
}

class Example
{
  def example(arg: Colored with Weighted) =
    {
        Console.println(arg.color)
        Console.println(arg.weight)
    }
}
```

Pattern Matching

```
case class Point(Int x, Int y);

def method(a: Any): String =
    a match
    {
        case 1 => "One"
        case 2 => "Two"
        case "Hello" => a.toString() +
                                ", World!"
        case s: String => s
        case b: BaseballPlayer => ...
        case Point(0,0) => ″Origin″
        case Point(a,b) => a + ″, ″ + b
        case _ => "Uknown"
    }

Console.println(method(2))
// prints "Two"
Console.println(method(“Hello”))
// prints "Hello, World!"
```

XML Support

- XML is a literal, just like strings
  - Can query XML-data in standard fashion
  - Can even pattern match against XML

```
val page =
     <html>
       <head><title>Title</title></head>
       <body>
         Hello, World<br />
       </body>
     </html>;
Console.println(page);
```

```
x match {
  case <entry>{ _* }<foo/></entry> => true;
  case _ => false;
}
```

Closures

- Blocks of code to be passed to functions for execution at a later time

```
def whileLoop(cond: => Boolean)
             (body: => Unit): Unit =
  if (cond)
    {
      body
      whileLoop(cond)(body)
    }

var i = 5
whileLoop (i > 0) {
  Console.println(i)
  i = i - 1
}
```

Actors

- Scala's concurrency model is based on actors
- Actors are basically concurrent processes that communicate by exchanging messages
- Actors can be seen as a form of active objects
- Messages can be sent asynchronously to actors

Actors

```
case object Ping
case object Pong
case object Stop

import scala.actors.Actor
import scala.actors.Actor._

class Ping(count: Int, pong: Actor)
                              extends Actor {
  def act() {
    var pingsLeft = count - 1
    pong ! Ping
    while (true) {
      receive{
        case Pong =>
          if (pingsLeft % 1000 == 0)
            Console.println("Ping sends pong")
          if (pingsLeft > 0) {
            pong ! Ping
            pingsLeft -= 1
          } else {
            Console.println("Ping: stop")
            pong ! Stop
            exit()
          }
        }
      }
    }
  }
```

Actors

```
class Pong extends Actor {
  def act() {
    var pongCount = 0
    while (true) {
      receive {
        case Ping =>
          if (pongCount % 1000 == 0)
            Console.println("Pong: ping " +
                                     pongCount)
          sender ! Pong
          pongCount = pongCount + 1
        case Stop =>
          Console.println("Pong: stop")
          exit()
      }
    }
  }
}

val pong = new Pong
val ping = new Ping(10000, pong)
ping.start
pong.start
```

Actors

- But there’s of course a more terse way to create actors

```
import scala.actors.Actor._

case object Msg

val a = actor {
  receive
  {
    case Msg => Console.println("A msg")
    case _   => Console.println(“Huh?”)
  }
}

Console.println("Hello1");
a ! Msg
Console.println("Hello2");
```