
# multi-paradigm: object-oriented, functional, imperative

Pedro Chambino
Abr 24, 2012

http://p.chambino.com/slides/scala

## History

Scala is a relatively recent language.

2001
: design started by Martin Odersky at the École Polytechnique Fédérale de Lausanne (Switzerland)

late 2003 - early 2004
: released on the Java platform (and on the .NET platform in June 2004)

March 2006
: version 2.0 released (current version 2.9.2)

17 January 2011
: the Scala team won a 5 year research grant of over €2.3 million from the European Research Council

## Hello World

```
# hello.scala
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

or

```
# hello.scala
object HelloWorld extends App {
  println("Hello, world!")
}
```

```
  > scalac hello.scala          # compile it
  > scala HelloWorld            # execute it
```

## Hello World

```
# hello.scala
println("Hello, world!")
```

```
  > scala hello.scala              # script it
```

## Read-Eval-Print Loop

```
  > scala                          # interactive shell
scala> 1+1
res0: Int = 2
```

# POJO

Java

```
class Person {
    private String firstName;
    private String lastName;
    private int    age;

    public Person(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName  = lastName;
        this.age       = age;
    }

    public void setFirstName(String firstName) { this.firstName = firstName; }
    public void String getFirstName() { return this.firstName; }
    public void setLastName(String lastName) { this.lastName = lastName; }
    public void String getLastName() { return this.lastName; }
    public void setAge(int age) { this.age = age; }
    public void int getAge() { return this.age; }
}
```

Scala

```
class Person(var firstName: String, var lastName: String, var age: Int)
```

# Interaction with Java

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._

object FrenchDate {
  def main(args: Array[String]) {
    val now = new Date
    val df = getDateInstance(LONG, Locale.FRANCE)
    println(df format now)
  }
}
```

Powerful imports

- Import multiple classes from same package with curly braces
- Import wildcard is _ instead of * because * is a valid scala identifier
- Also imports are relative!

Syntactic Sugar

- Methods with zero or one argument can use the infix syntax:
  - `df format now` equals `df.format(now)`
  - `new Date` equals `new Date()`

## Variable Declaration and Inferring Type Information

```
val x = 0
var y = 1
var z: Any = 2
x = 3                       // error: reassignment to val
y = 4
y = "5"                     // error: type mismatch
z = "6"
lazy val l = println("!!!")
```

- The `val` keyword is similar to Java's `final` and doesn't allow reassignment
- The `var` keyword allows reassignment however the `y` variable inferred the type `Int`
- The `Any` type is the root of the Scala type hierarchy
- The `lazy` keyword allows the evaluation of a `val` to be delayed until it's necessary

## Inferring Type Information and Generics

### Java

```java
Map<Integer, String> intToStringMap = new HashMap<Integer, String>();
```

### Scala

```scala
val intToStringMap: Map[Int, String] = new HashMap
val intToStringMap2 = new HashMap[Int, String]
```

- Scala uses `[...]` for generic types parameters
- Removes the need for declaring generic types parameters twice

## Everything is an object

```
1 + 2 * 3 / x
```

consists of method calls and is equivalent to:

```
(1).+(((2).*(3))./(x))
```

this means that **+**, **-**, ***** and **/** are valid identifiers in Scala

> “I call it my billion-dollar mistake. It was the invention of the null reference in 1965. (...) I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. (...)”
>
> — Tony Hoare

# Option, Some, and None: Avoiding nulls

```
val stateCapitals = Map(
  "Alabama" -> "Montgomery",
  "Wyoming" -> "Cheyenne")

println( "Alabama: " + stateCapitals.get("Alabama") )
println( "Unknown: " + stateCapitals.get("Unknown") )
println( "Alabama: " + stateCapitals.get("Alabama").get )
println( "Wyoming: " + stateCapitals.get("Wyoming").getOrElse("Oops!") )
println( "Unknown: " + stateCapitals.get("Unknown").getOrElse("Oops2!") )

// *** Outputs ***
// Alabama: Some(Montgomery)
// Unknown: None
// Alabama: Montgomery
// Wyoming: Cheyenne
// Unknown: Oops2!
```

A possible implementation of `get` that could be used by a concrete subclass of Map:

```
def get(key: A): Option[B] = {
  if (contains(key))
    new Some(getValue(key))
  else
    None
}
```

# Functions are objects too

```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }

  def timeFlies() {
    println("time flies like an arrow...")
  }

  def main(args: Array[String]) {
    oncePerSecond(timeFlies)
    // or
    oncePerSecond(() =>
      println("time flies like an arrow..."))
  }
}
```

- `() => Unit` declares a funciton that receives zero arguments and returns nothing
- `Unit` type is similar to Java or C/C++ `void` type
- `() => println("time flies like an arrow...")` declares an anonymous function

# Method Default and Named Arguments

```scala
def joiner(strings: List[String], separator: String = " "): String =
    strings.mkString(separator)

println(joiner(List("Programming", "Scala")))
println(joiner(strings = List("Programming", "Scala")))
println(joiner(List("Programming", "Scala"), " "))
println(joiner(List("Programming", "Scala"), separator = " "))
println(joiner(strings = List("Programming", "Scala"), separator = " "))
```

- In contrast with Java, Scala allows default arguments
- Named arguments allows to specify parameters in any order
- Named arguments allows to document each parameter when calling the method

# Currying

```scala
def concat(s1: String)(s2: String) = s1 + s2
// alternative syntax:
// def concat(s1: String) = (s2: String) => s1 + s2

val hello = concat("Hello ")(_)
println(hello("World")) // => Hello World

// transforms a normal function into a curried function:
def concat2(s1: String, s2: String) = s1 + s2
val curriedConcat2 = Function.curried(concat2 _)
```

- Curried functions are named after mathematician Haskell Curry (from whom the Haskell language also get its name)
- Using the alternative syntax, the `(_)` is optional

# Scala `for` comprehensions

```
val dogBreeds = List("Doberman", "Yorkshire Terrier", "Dachshund",
                     "Scottish Terrier", "Great Dane", "Portuguese Water Dog")

for (breed <- dogBreeds)
  println(breed)

// *** Filtering ***

for (
  breed <- dogBreeds
  if breed.contains("Terrier");
  if !breed.startsWith("Yorkshire")
) println(breed)

// *** Yielding ***

val filteredBreeds = for {
  breed <- dogBreeds
  if breed.contains("Terrier")
  if !breed.startsWith("Yorkshire")
  upcasedBreed = breed.toUpperCase()
} yield upcasedBreed
```

No `break` or `continue`!

# Mixins

```
trait Observer[S] {
  def receiveUpdate(subject: S);
}

trait Subject[S] {
  this: S =>
  private var observers: List[Observer[S]] = Nil
  def addObserver(observer: Observer[S]) = observers = observer :: observers

  def notifyObservers() = observers.foreach(_.receiveUpdate(this))
}
```

- Like Java's `interface` but with implementations
- Self-type annotations (`this: S =>`) removes the need for casting when implementing an `Observer`
- `::` is a method of `List` that adds an element to the beginning of list
- All methods that end in `:` when using the infix syntax have to be called in reverse order
- `Nil` is an empty list

# Using Traits

```
class Account(initialBalance: Double) {
  private var currentBalance = initialBalance
  def balance = currentBalance
  def deposit(amount: Double)  = currentBalance += amount
  def withdraw(amount: Double) = currentBalance -= amount
}

class ObservedAccount(initialBalance: Double)
  extends Account(initialBalance)
  with Subject[Account]
{
  override def deposit(amount: Double) = {
    super.deposit(amount)
    notifyObservers()
  }
  override def withdraw(amount: Double) = {
    super.withdraw(amount)
    notifyObservers()
  }
}

class AccountReporter extends Observer[Account] {
    def receiveUpdate(account: Account) =
        println("Observed balance change: "+account.balance)
}
```

# Case Classes

```
// Represent expressions like: (x+x)+(7+y)
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree
```

- The new keyword is not mandatory to create instances of these classes
- Getter functions are automatically defined for the constructor parameters
- Default definitions for methods equals and hashCode are provided, which work on the structure of the instances and not on their identity
- A default definition for method toString is provided, and prints the value in a "source form" (e.g. the tree for expression `x+1` prints as `Sum(Var(x),Const(1))`)
- Instances of these classes can be decomposed through pattern matching

# Case Classes and Pattern Matching

```
type Environment = String => Int

def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
//case l Sum r => eval(l, env) + eval(r, env) // alternative syntax
  case Var(n) => env(n)
  case Const(v) => v
}
```

- The keyword `type` declares an alias for a specified type
- The `abstract class Tree` could be declared `sealed` like: `sealed abstract class Tree` which would allow the compiler to verify if the pattern matching was exhaustive
- A `sealed` class can only be inherited by classes declared in the same file

# Pattern Matching

```scala
def countScalas(list: List[String]): Int = {
  list match {
    case "Scala" :: tail => countScalas(tail) + 1
    case _ :: tail       => countScalas(tail)
    case Nil             => 0
  }
}
val langs = List("Scala", "Java", "C++", "Scala", "Python", "Ruby")
val count = countScalas(langs)
println(count)    // => 2
```

## Matching on Type

```scala
val sundries = List(23, "Hello", 8.5, 'q')

for (sundry <- sundries) {
  sundry match {
    case n: Int if (n > 0) => println("got a Natural: " + n)
    case i: Int => println("got an Integer: " + i)
    case s: String => println("got a String: " + s)
    case f: Double => println("got a Double: " + f)
    case other => println("got something else: " + other)
  }
}
```

# Actor Model of Concurrency

```
import scala.actors.Actor._
val echoActor = actor {
  loop {
    receive {
      case msg => println("received: "+msg)
    }
  }
}
echoActor ! "hello"
echoActor ! "world!"
```

- The `actor` method creates and starts a new `Actor`
- Messages can be sent to actors using the `!` method
- Messages can be any kind of object
- Scala encourages the use of immutable objects, specially in concurrent programming since it removes the need for semaphores

# XML

```
val name = "Bob"

val bobXML =
<person>                    /// <person>
  <name>{name}</name>       ///   <name>Bob</name>
</person>                   /// </person>

bobXML \ "name"           // => <name>Bob</name>
bobXML \ "name" text      // => Bob
(bobXML \ "name").text    // => Bob
```

## Summary

- A more succinct syntax
- Interoperability with Java libraries
- More flexible method names and invocation syntax
- Better mechanisms for avoiding `null`'s
- Tuples
- First-class functions and closures
- A true mixin model
- Pattern matching
- Better separation of mutable vs. immutable objects
- A workable concurrency model

## Who uses Scala?

Foursquare
- uses Lift (web application framework) and Scala

Twitter
- backend was converted from Ruby to Scala for performance benefits

LinkedIn
- uses Scalatra (micro web application framework) to power its Signal API

The Guardian (newspaper's website guardian.co.uk)
- switched from Java to Scala

And much more...

# Bibliography


- **"Introducing Scala."** . Web. 21 Apr. 2012. <http://www.scala-lang.org>.
- **"Scala (programming Language)."** . Wikimedia Foundation, 21 Apr. 2012. Web. 21 Apr. 2012. <http://en.wikipedia.org/wiki/Scala_(programming_language)>.
- Wampler, Dean, and Alex Payne. **"Programming Scala."** . 2008. Web. 21 Apr. 2012. <http://programming-scala.labs.oreilly.com>.
- Wampler, Dean. **"The Seductions of Scala, Part I."** . 03 Aug. 2008. Web. 21 Apr. 2012. <http://blog.objectmentor.com/articles/2008/08/03/the-seductions-of-scala-part-i>.
- Wampler, Dean. **"The Seductions of Scala, Part II - Functional Programming."** . 06 Aug. 2008. Web. 21 Apr. 2012. <http://blog.objectmentor.com/articles/2008/08/05/the-seductions-of-scala-part-ii-functional-programming>.
- Wampler, Dean. **"The Seductions of Scala, Part III - Concurrent Programming."** . 14 Aug. 2008. Web. 21 Apr. 2012. <http://blog.objectmentor.com/articles/2008/08/14/the-seductions-of-scala-part-iii-concurrent-programming>.