Tutorial on Writing Modular Programs in Scala

Martin Odersky and Gilles Dubochet

13 September 2006

Welcome to the Scala tutorial at JMLC 2006

A half-day tutorial on the Scala programming language.

- A rapid, no-frills, presentation of Scala as a language for writing modular programs.
- For Java or related programmers.

Be advised, you will work too: this is an interactive hands-on tutorial: get your computer ready!

This afternoon's plan

1. Meeting Scala
2. Pattern matching
3. Functions
4. Mixins
5. Higher-order Functions

Scala vs. Java

At first glance, Scala is similar to Java (or C♯).
or rather everything Java has to offer can be found in Scala.

- Scala is object-oriented, statically typed, throws exceptions, etc.
- Scala's syntax will look familiar to Java programmers.
- Scala compiles to Java bytecode: it runs on any JVM.
- Scala even shares Java's libraries: all classes and methods defined as a Java libraries are transparently accessible from Scala code.

The two following classes have the same behaviour.

In Java:

```
class PrintOptions {
  public static void main(String[] args) {
    System.out.println("Opts␣selected:");
    for (int i = 0; i < args.length; i++)
      if (args[i].startsWith("-"))
        System.out.println(
          "␣"+args[i].substring(1));
  }
}
```

In Scala:

```
class PrintOptions {
  def main(args: Array[String]: Unit) = {
    System.out.println("Opts␣selected:")
    for (val arg <- args)
      if (arg.startsWith("-"))
        System.out.println(
          "␣"+arg.substring(1))
  }
}
```

You might notice some similarities.

Basic differences

This first section will describe some basic differences between Scala and Java you need to be aware of.

These include

- syntactic differences,
- different class member definitions,
- a purely object model and differences in the class model.

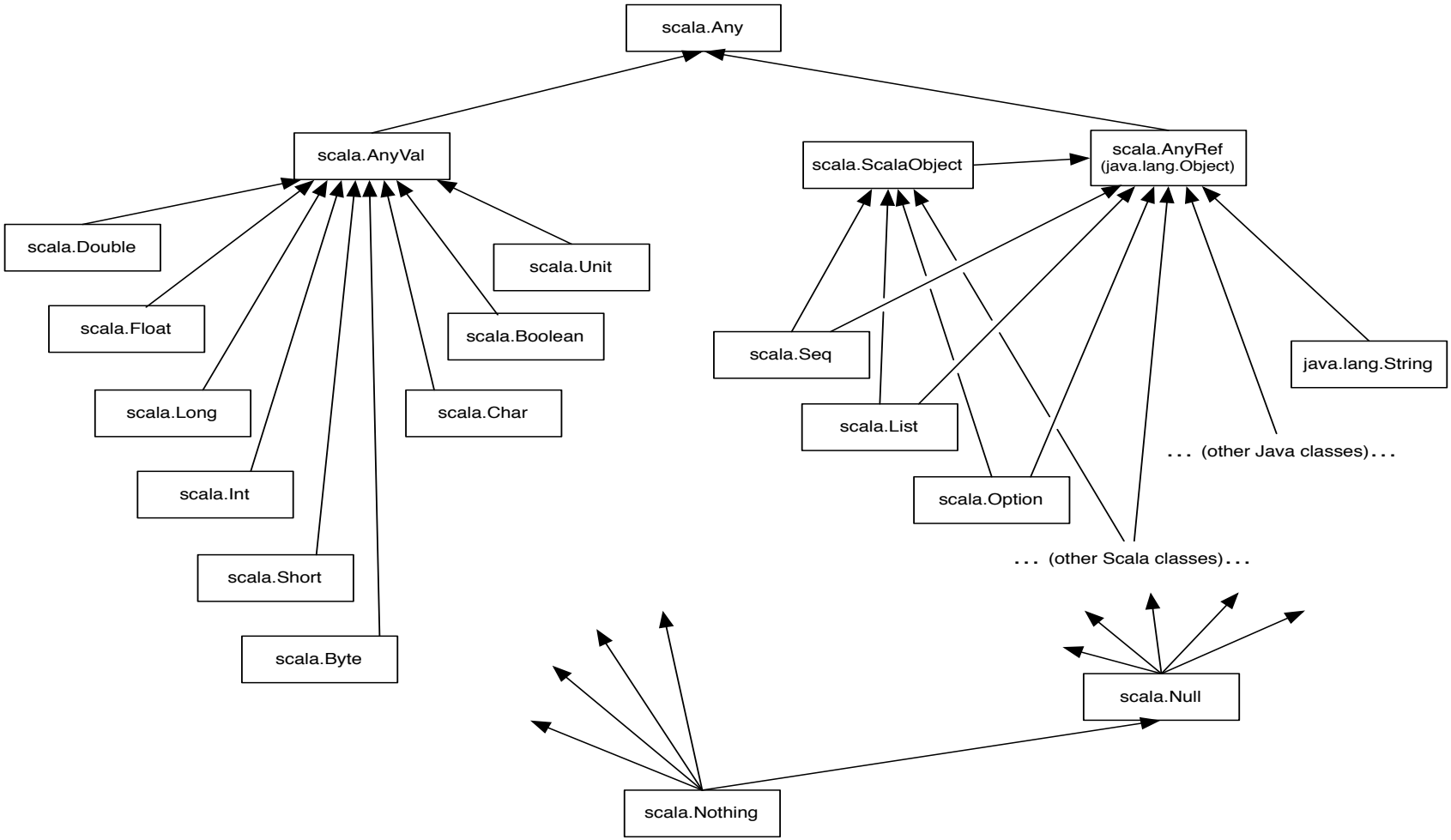
Syntactic differences

- Scala uses an *identifier-colon-type* notation for member or parameter definitions.
  `int age (String name)` becomes
  `def age (name: String): Int`
- Semi-colons are optional.
- There is no pre-defined syntax for `for` loops. *Comprehensions* are provided instead.
- Blocks such as `{...}` are required only to group statements. Single expressions can be defined outside a block.
  `def x = if (p) a else b` is a legal declaration.
- All definitions can be arbitrarily nested.

Everything is an object

- All native types (`int`, `double`, `bool`) are classes, define methods etc., but
  - they are not passed as references,
  - they are subclasses of `AnyVal` (as opposed to other classes that extend `AnyRef`).
- Arrays are objects and array-specific syntax does not exist (c.f. API).
- The `void` pseudo-type is replaced by the `Unit` class. Instances of `Unit` can be created with `()`.

A new class hierarchy



The object value

Objects can even be created as such, without defining a class.

```
object Scala extends Language { val creator = LAMP }
```

- Objects replace the singleton pattern,
- There are no static members, instead objects can be created directly.
  An object with the same name as a class is called *companion module* and can access private class members.

Richer class members

Java only allows fields and methods in classes. Scala has a richer semantic for class members.

- `def` defines a method. Parameters are allowed, but optional.
  `def f(a: Int): String` or `def f: String` are legal definitions.
- `val` defines a constant value. A value can also override a (non parameterized) `def`.
  This is required for writing "functional" (i.e. invariant) classes.
- `var` defines a variable, like a Java field.
- `object` and `class` are legal members in a class.
- `type` members also exist but will not be covered in this tutorial.

Class body as constructor

In Java, constructors are special (smalltalk-ish) methods. Scala has a different approach.

- The class or object body is executed at instance creation.
- Class declarations have parameters.

```
class Human (soul: Soul) {
  soul.insufflate(creator.getLife)
}
val me = new Human(new Soul)
```

This is the primary constructor, others can be defined as

```
def this (...) = ...
```

Hands-on

You will now test some of the described concepts yourself.

- Installing and running Scala.
- Creating classes and objects.
- Using constructors.
- And generally getting familiar with the Scala syntax.

The accompanying documentation describes your task in more detail.

Class hierarchies as ADT

OO languages use class hierarchies for representing data types.

- Content is encapsulated in the object and accessed through methods.

Algebraic data types are a common concept in functional languages.

- Data is accessed through *decomposing* the value by pattern matching.

ADT and class hierarchies have complementary strength and weaknesses.

- ADTs allow easy extension of operations supported by the data
- while class hierarches allow easy addition of data variants.

Scala case classes

ADTs can be encoded using `case` classes.

- Case classes are like normal classes.
- Instance constructors can be recovered by pattern matching.
- Structural equality is used for comparison.
- The `new` keyword is optional for instance creation.

`case class ClockTime (hour: Int, min: Int)` is a valid case class definition. `ClockTime(10,30)` creates an instance.

Scala's pattern matching

A case class can be decomposed using a `match` construct, like the following.

```
time match {
  case ClockTime(hour, min) => ...
  case SwatchTime(beats) => ...
  case Sunset => ...
  case Sunrise => ...
}
```

All lower-case identifiers in the pattern will bind the decomposed value and are available on the right-hand side of the pattern.

Order is important: a first-match policy is used.

Constant values can be used in patterns to restrict matching.

`case ClockTime(10, min)` will only match any time in the 10th hour (and bind minutes to `min`).

`case "ten o'clock"` will match the ten o'clock string.

A name starting with a capital letter will also be treated as a constant.

`case ClockTime(Ten, min)` will behave as above if `Ten == 10`.

Richer conditions can be defined with *guards*.

`case ClockTime(hour, min) if hour > min` is a guard.

When no pattern matches a value, the `match` statement throws a `MatchError`.

A default case can be added, using the wildcard pattern.
`case _` will match any value.

Wildcards can also be used as a component of a pattern.
`case SwatchTime(_)` will match any time defined in Swatch-beat time.

Pattern matching without case classes

Scala's patterns even extend to non-case classes.
`case x: String` will bind `x` (of type string) to any value of type string.

Of course, deconstruction isn't available on type patterns.
Instead, this is a rich way to do type casts or type tests.

Hands-on

You will now test some of the described concepts yourself.

- Matching on case-class ADTs.
- Matching on values.

Functions

Scala supports lightweight syntax for anonymous functions.
`(x: Int) => x + 1` defines a successor function on integers.

Functions are first-class values, and can be stored or passed.

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

`succ(44)` applies the successor function an returns 45.

Lifting functions

A method can easily be transformed into a function

- by not providing it with its parameters,
- and by flagging it with a `&`.

```
class Number (value: Int) {
  def add (other: Number) = ...
}
```

Can be used as a function value in unrelated code

```
val addOne = &new Number(1).add
```

Functions as objects

As mentioned earlier, Scala is purely object-oriented.
Since functions are values, they must be objects too.

- A function is instance of class `Function0` or `Function1` or ...
- There exists one function class for all number of parameters.
- A class (or object) implementing `FunctionX` must define an `apply` method with the correct number of parameters.

```
object addOne extends Function1[Int, Int] {
  def apply(num: Int): Int = num + 1
}
```

`addOne(23)` will return 24.

The type of function

A shorthand syntax for writing the type of functions also exists.

- `Function0[Int]` becomes `() => Int`
- `Function1[String, Person]` becomes `String => Person`
- `Function2[Int, Int, Int]` becomes `(Int, Int) => Int`
- `Int => Int => Int` is `Function1[Int, Function1[Int, Int]]`

Taking advantage of function

A functional programming style offers real benefits for modular programs.

- A module can be parameterized by *function*, not only by state.
- Functions can be passed from module to module.

Scala's functions-as-objects allow an easy integration of functions in a traditional OO environment.

Functional programming style

Writing in functional style can be difficult for seasoned OO programmers.

- Behaviour is no longer attached to an object but moves freely.
- State becomes less important: there are no methods depending on it.
- Immutable objects become natural: why deal with state when a function can simply return a new object?

In other words, use state sparingly in Scala, functions and immutable objects (think `val`) help structure messy code.

Hands-on

You will now test some of the described concepts yourself.

- Use functions as values.
- Define anonymous functions.
- Turn a class into a function.

Modular systems with single inheritance

In single class inheritance languages

- merging behaviours of different classes (or modules) is tricky,
- adaptor code is required,
- which make the relation brittle.

Often, module reengineering is required.

Java's *interfaces* provide some help, but are clearly insufficient.

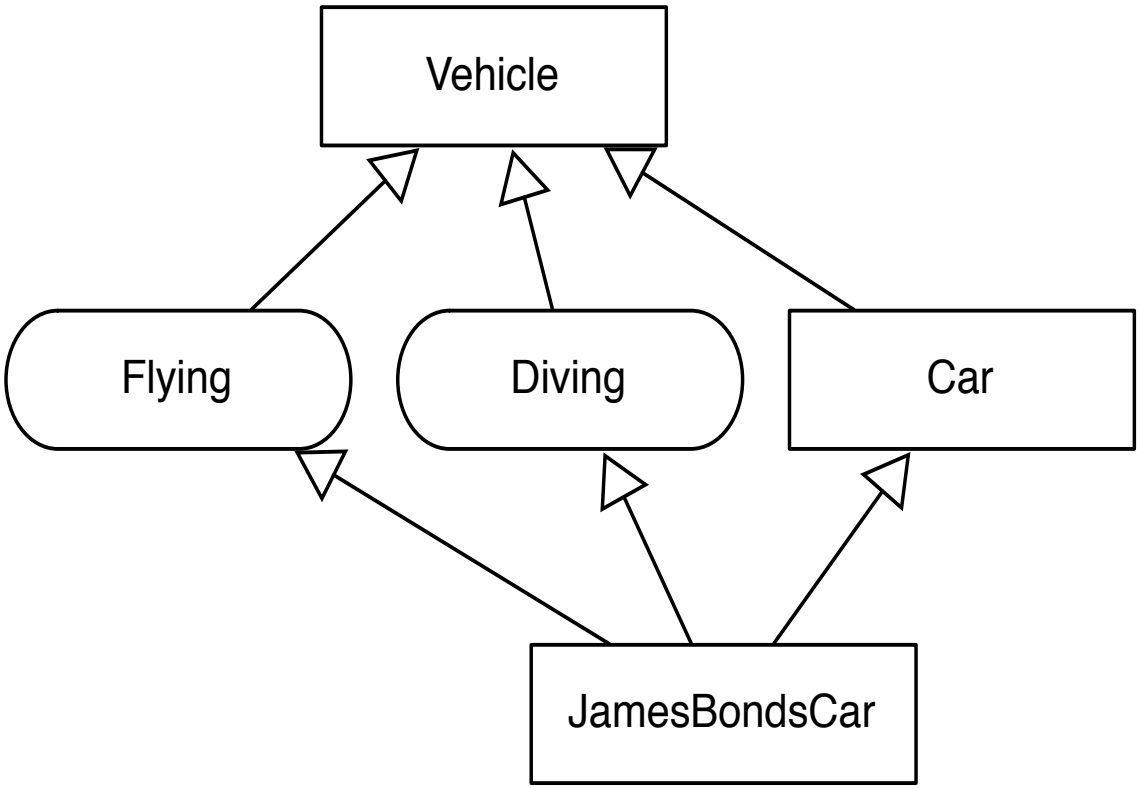
Mixins

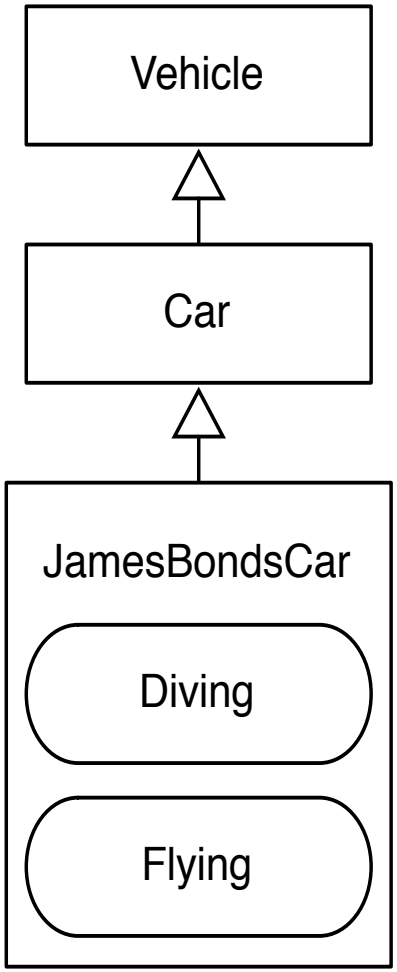
Full multiple inheritance is often too complex to be of great use.

Scala provides *mixins* as a compromise.

- A class can inherit from multiple *traits*.
- A trait is a special kind of class which implements some behaviour.
- There must be a common parent class with the inherited mixin.

Mix-ins?


Vehicle
Flying
Diving
Car
JamesBondsCar


Vehicle
Car
JamesBondsCar
Diving
Flying

Traits

A trait is defined like a class, but using the `trait` keyword instead.

```
trait Flying extends Vehicle {
  def takeOff = ... // concrete
  def land: Unit // abstract
}
```

All members inherited from `Vehicle` can be used:
this trait will eventually be mixed-in with a class extending `Vehicle`.

Inheriting traits

A trait can be inherited

- when defining a class

```
class JamesBondsCar extends Car with Flying with Diving
```

- or when creating an instance.

```
val jbsCar = new Car with Flying with Diving
```

When a class only extends mixins, it will automatically also extend `AnyRef`.

Requiring a behaviour

When multiple traits are inherited

- they can refer to members of their common super class,
- but not to members of other mixed-in traits.

A trait can *require* another class or trait;
it can only be mixed-in when the requirement is available.

```
trait Reading extends Person requires Seeing
```

Hands-on

You will now test some of the described concepts yourself.

- Define traits for mixin.
- Mixin these traits into instances to inherit behaviour.

Higher-order functions

A higher-order function is a function (or a method) that takes another function as parameter.

```
def order(
  data: List[Thing],
  lessThan: (Thing, Thing) => Boolean
) = ...
```

This method orders a list of things.
But since order on things is not well-defined, order is parameterized as a function.

There is nothing more to it.

Higher-order functions on lists

But the real deal with higher-order functions is their use in lists (and other container structures).

- Lists are the most common *container* structures.
- "For every element of the list do ..." is a natural task.
- This requires the definition of the operation to apply on each element.
- Which means: higher-order function.

Map, Flat map and Filter

The scala list class defines a number of usefull higher-order functions.

For a list of type `A`

- `def map[B](f: (A) => B): List[B]` will apply `f` to all elements of the list, and return the resulting new list.
- `def flatMap[B](f: (A) => List[B]): List[B]` will apply `f` to all elements of the list, and concatenate all resulting lists into one flat list.
- `def filter(p: (A) => Boolean): List[A]` will return a new list containing only those elements that are true for predicate `p`.

For-comprehensions

Java's `for` loops are replaced in Scala with comprehensions.

- A comprehension will loop on all elements of a list.
  `for (val e <- List(1,2,3)) print(e)` prints "123"
- A comprehension can return a new list.
  `for (val e <- List(1,2,3)) yield e * 2` returns `List(2,4,6)`.
- A comprehension can filter its elements.
  `for (val e <- List(1,2,3); e.isEven) yield e` returns `List(2)`.
- A comprehension can have multiple loops.
  `for (val e1 <- List(1,2); val e2 <- List(2,3)) yield e1 * e2` returns `List(2,3,4,6)`.

For-comprehensions are entirely made out of higher-order functions on lists.

A for-comprehension

```
for {
  val i <- 1 to n
  val j <- 1 to i
  isPrime(i+j)
} yield Pair(i, j)
```

and the code it gets turned into.

```
(1 to n).flatMap {
  case i => (1 to i)
    .filter { j => isPrime(i+j) }
    .map { case j => Pair(i, j) }
}
```

Which means they can be used on any class that supports `map`, `flatmap` and `filter`.

Hands-on

You will now test some of the described concepts yourself.

- Using for-loops or other higher-order functions on lists.

As this is the last hands-on section, we will also reuse everything else we discussed in a grand finale.

To conclude: the Scala tutorial at JMLC 2006

We hope you did enjoy this tutorial.

We would like you to be able to take back the following.

- A good idea as to how pattern matching, mixins and first-class functions can improve the modularity of an OO language.
- A feel for the kind of programming that Scala permits, and how useful it is.

To conclude: Scala and modular programs

Scala is a *fairly complex language* with many features.

But this complexity can be *put to good use*

- because it allows modularising more
- in a safer way and
- in a more reusable way.

Scala's seamless and complete integration of functional and OOP features is the key to its success.

Get Scala for yourself

Scala is open source and available free of charge.

For downloads, example code, libraries, discussions, etc., visit the Scala website at `http://scala.epfl.ch`