A Brief Intro to **Scala**

Informatica III

AA 19/20

Origin

- Started at 2001 by Martin Odersky at EPFL Lausanne, Switzerland
- Scala 2.0 released in 2006
- Current version 2.12.4
- IDE (eclipse based) 4.7.0
- Twitter backend runs on Scala
  - **LinkedIn, Siemens, Sony, …**

Scala

- Statically Typed
- Runs on JVM, full inter-op with Java
- Object Oriented
- Functional
- Dynamic Features

Scala is Practical

- Can be used as drop-in replacement for Java
  - Mixed Scala/Java projects

- Use existing Java libraries

- Use existing Java tools (Ant, Maven, JUnit, etc...)

- Decent IDE Support (NetBeans, IntelliJ, Eclipse)

Java

- What's wrong with Java?
  - Verbose
    - Too much of Thing thing = new Thing();
    - Too much "boilerplate," for example, getters and setters
  - …

- What's right with Java?
  - Very popular
  - Object oriented (mostly), which is important for large projects
  - Strong typing (more on this later)
  - The fine large library of classes
  - **The JVM!** Platform independent, highly optimized

Scala is like Java, except when it isn't

- Java is a *good* language, and Scala is a lot like it
- For each difference, there is a *reason*--none of the changes are "just to be different"
- Scala and Java are (almost) completely interoperable
  - Call Java from Scala? No problem!
  - Call Scala from Java? Some restrictions, but mostly OK.
  - Scala compiles to .class files (a *lot* of them!), and can be run with either the scala command or the java command
- To understand Scala, it helps to understand the reasons for the changes, and what it is Scala is trying to accomplish

Consistency is good

- In Java, every value is an object--unless it's a primitive
  - Numbers and booleans are primitives for reasons of efficiency, so we have to treat them differently (you can't "talk" to a primitive)
- In Scala, all values are objects. Period.
  - The compiler turns them into primitives, so no efficiency is lost (behind the scenes, there are objects like RichInt)

- Java has *operators* (+, <, …) and *methods*, with different syntax
- In Scala, operators are just methods, and in many cases you can use either syntax

Differences with Java - basic

- Scala does not require semicolons to end statements.

- Value types are capitalized: Int, Double, Boolean instead of int, double, boolean.

- Parameter and return types follow, rather than precede as in C.

- Methods must be preceded by def.

- Local or class variables must be preceded by val (indicates an immutable variable) or var (indicates a mutable variable).

- The return operator is unnecessary in a function (although allowed); the value of the last executed statement or expression is normally the function's value.

- Instead of the Java cast operator (Type) foo, Scala uses foo.asInstanceOf[Type], or a specialized function such as toDouble or toInt.

- Instead of Java's import foo.*;, Scala uses import foo._.

- Function or method foo() can also be called as just foo;

Scala is Concise

Type Inference

"capisce" il tipo di una espressione.

Statically typed lo stesso !

```
val sum = 1 + 2 + 3
val nums = List(1, 2, 3)
val map = Map("abc" -> List(1,2,3))
```

Nota: we use scala
worksheet

Explicit Types

```
val sum: Int = 1 + 2 + 3


val nums: List[Int] = List(1, 2, 3)


val map: Map[String, List[Int]] = ...
```

Higher Level

```java
// Java – Check if string has uppercase character
boolean hasUpperCase = false;
for(int i = 0; i < name.length(); i++) {
    if(Character.isUpperCase(name.charAt(i))) {
        hasUpperCase = true;
        break;
    }
}
```

Higher Level

Posso passare come argomento di una funzione una funzione

```scala
// Scala
val hasUpperCase = name.exists(_.isUpperCase)
```

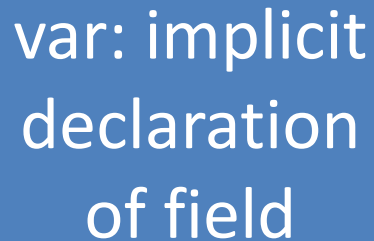Funzione f   Argomento di f

Less Boilerplate

```java
// Java
public class Person {
  private String name;
  private int age;
  public Person(String name, Int age) {  // constructor
    this.name = name;
     this.age = age;
  }
  public String getName() {              // name getter
    return name;
  }
  public int getAge() {                  // age getter
    return age;
  }
  public void setName(String name) {     // name setter
    this.name = name;
  }
  public void setAge(int age) {          // age setter
    this.age = age;
  }
}
```

Less Boilerplate

```scala
// Scala
class Person(var name: String, var age: Int)
```

var: implicit declaration of field

Local or class variables must be preceded by val (indicates an immutable variable) or var (indicates a mutable variable).

Less Boilerplate

```scala
// Scala
class Person(var name: String, private var _age:
  Int) {
  def age = _age              // Getter for age
  def age_=(newAge:Int) {  // Setter for age
    println("Changing age to: "+newAge)
    _age = newAge
  }
}
```

Variables and **Val**ues

l-value vs r-value

```
// variable
var foo = "foo"
foo = "bar"   // okay


// value
val bar = "bar"
bar = "foo"   // nope
```

null

- "I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."

--Tony Hoare

null in Scala

- In Java, any method that is *supposed* to return an object *could* return null
  - Here are your options:
    - Always check for null
    - Always put your method calls inside a try...catch
    - Make sure the method can't possibly return null
- Yes, Scala has null--but only so that it can talk to Java
- In Scala, if a method *could* return "nothing," write it to return an Option object, which is either Some(*theObject*) or None
  - This forces you to use a match statement--but only when one is really needed!

Esempio uso None

```scala
def toInt(in: String): Option[Int] = {
  try {
    Some(Integer.parseInt(in.trim))
  } catch {
    case e: NumberFormatException => None
  }
}
```

Referential transparency

- In Scala, variables are really functions
  - Huh?
- In Java, if age is a public field of Person, you can say:
  david.age = david.age + 1;
  but if age is accessed via methods, you would say:
  david.setAge(david.getAge() + 1);

- In Scala, if age is a public field of Person, you can say:
  david.age = david.age + 1;
  but if Person defines methods age and age_=, you would say (the same!)
  david.age = david.age + 1;

- In other words, if you want to access a piece of data in Scala, you don't have to know whether it is computed by a method or held in a simple variable
  - This is the principle of uniform access
  - Scala *won't let you* use parentheses when you call a function with no parameters

Scala is **O**bject **O**riented
Scala is **Functional**

What is Multiparadigm Programming?

- Definition:
- A multiparadigm programming language provides "a framework in which programmers can work in a variety of styles, freely intermixing constructs from different paradigms." [Tim Budd]

- Programming paradigms:
- imperative versus declarative (e.g., functional, logic)
- other dimensions – object-oriented, component-oriented, concurrency-oriented, etc.

Why Learn Multiparadigm Programming?

- Tim Budd:

"Research results from the psychology of programming indicate that expertise in programming is far more strongly related to the number of different programming styles understood by an individual than it is the number of years of experience in programming."

- The "goal of multiparadigm computing is to provide … a number of different problem-solving styles" so that a programmer can "select a solution technique that best matches the characteristics of the problem".

Why Teach Multiparadigm Programming?

- Contemporary imperative and object-oriented languages increasingly have functional programming features, e.g.,
  - higher order functions (closures)
  - list comprehensions

- New explicitly multiparadigm (object-oriented/functional) languages are appearing, e.g.,
  - Scala on the Java platform (and .Net in future)
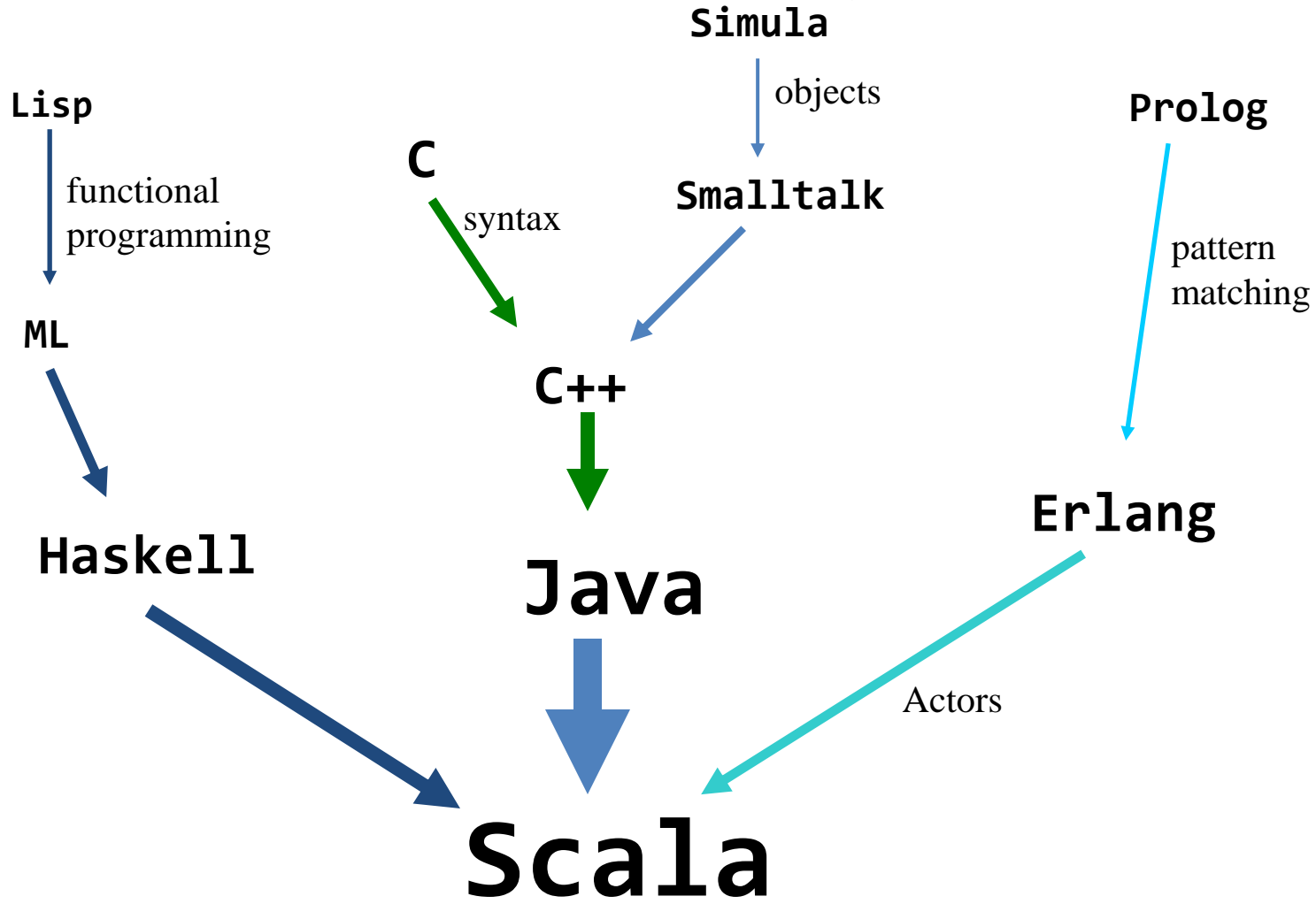 - F# on the .Net platform

Functional languages

- The best-known functional languages are ML, OCaml, and Haskell
- Functional languages are regarded as:
  - "Ivory tower languages," used only by academics (mostly but not entirely true)
  - Difficult to learn (mostly true)
  - The solution to all concurrent programming problems everywhere (exaggerated, but not entirely wrong)
- Scala is an "impure" functional language--you can program functionally, but it isn't forced upon you

Scala as a functional language

- The hope--*my* hope, anyway--is that Scala will let people "sneak up" on functional programming (FP), and gradually learn to use it
  - This is how C++ introduced Object-Oriented programming
- Even a little bit of functional programming makes some things a lot easier
- Meanwhile, Scala has plenty of other attractions

- FP really is a different way of thinking about programming, and not easy to master…
- …but…
- Most people that master it, never want to go back

Genealogy

Simula

objects

Lisp

functional
programming

C

syntax

Smalltalk

Prolog

pattern
matching

ML

C++

Haskell

Java

Erlang

Actors

Scala

Scala is Dynamic

(Okay not really, but it has lots of features typically only found in Dynamic languages)

Read-Eval-Print Loop

bash$ **scala**

Welcome to Scala version 2.8.1.final (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_22).

Type in expressions to have them evaluated.

Type :help for more information.

scala> **class Foo { def bar = "baz" }**

defined class Foo

scala> **val f = new Foo**

f: Foo = Foo@51707653

scala> **f.bar**

res2: java.lang.String = baz

Noi usermo però ScalaIDE

Structural Typing

```
// Type safe Duck Typing
def doTalk(any:{def talk:String}) {
  println(any.talk)
}

class Duck { def talk = "Quack" }
class Dog  { def talk = "Bark"  }

doTalk(new Duck)   →   "Quack"
doTalk(new Dog)    →   "Bark"
```

tipizzazione dinamica dove la semantica di un oggetto è determinata dall'insieme corrente dei suoi metodi e delle sue proprietà anziché dal fatto di estendere una particolare classe o implementare una specifica interfaccia

il duck typing permette il polimorfismo (sottotipazione) senza ereditarietà

Scala has tons of other cool stuff

Default Parameter Values

```
def hello(foo:Int = 0, bar:Int = 0) {
  println("foo: "+foo+"  bar: "+bar)
}
```

```
hello()          →  foo: 0  bar: 0
hello(1)         →  foo: 1  bar: 0
hello(1,2)       →  foo: 1  bar: 2
```

Named Parameters

```
def hello(foo:Int = 0, bar:Int = 0) {
  println("foo: "+foo+"  bar: "+bar)
}
```

```
hello(bar=6)         →   foo: 0  bar: 6
hello(foo=7)         →   foo: 7  bar: 0
hello(foo=8,bar=9)   →   foo: 8  bar: 9
```

Everything Returns a Value

```scala
val a = if(true) "yes" else "no"
```

Non esiste il void

```scala
val b = try{
  "foo"
} catch {
  case _ => "error"
}


val c = {
  println("hello")
  "foo"
}
```

Lazy Vals

```scala
// initialized on first access
lazy val foo = {
  println("init")
  "bar"
}


foo  →   init
foo  →
foo  →
```

Nested Functions

```
// Can nest multiple levels of functions
def outer() {
    var msg = "foo"
    def one() {
        def two() {
            def three() {
                println(msg)
            }
            three()
        }
        two()
    }
    one()
}
```

By-Name Parameters

- … argument is not evaluated at the point of function application, but instead is evaluated at each use within the function.

- Sintassi

  - Pass by value

    def f (x:Int, y:Int) = x;

  - Pass by name

    def f (x: => Int, y: => Int) = x;

By-Name Parameters

- Ci sia una funzione che ha side effect e che restituisce qualcosa

```
def something() = {
    println("callingsomething")
    1
}
```

Poi due funzioni che stampano due volte l'argomento che passo:

```
def callByValue(x: Int) = {
    println("x1=" + x)
    println("x2=" + x)
  }
def callByName(x: => Int) = {
    println("x1=" + x)
    println("x2=" + x)
}
```

- Se chiamo:

```
callByValue(something())
```
Come al solito ho:

- Valuto `something()` e passo il suo valore.

```
calling something
x1=1
x2=1
```

- Se chiamo:

```
callByName(something())
```
Valuto `something()` dentro in callByname:

```
calling something
x1=1
calling something
x2=1
```

Which is faster?

def test (x:Int, y:Int)= x*x

def test (x: =>Int, y:=>Int)= x*x

We want to examine the evaluation strategy and determine which one is faster (less steps) in these conditions:

- test (2,3)

- call by value: test(2,3) -> 2*2 -> 4
- call by name: test(2,3) -> 2*2 -> 4

- test (3+4,8)

- call by value: test (7,8) -> 7*7 -> 49
- call by name: (3+4) (3+4) -> 7(3+4)-> 7*7 ->49
- Here call by value is faster.

- test (7,2*4)

- call by value: test(7,14) -> 7*7 -> 49
- call by name: 7 * 7 -> 49
- Here call by name is faster

- test (3+4, 2*4)

- call by value: test(7,2*4) -> test(7, 8) -> 7*7 -> 49
- call by name: (3+4)(3+4) -> 7(3+4) -> 7*7 -> 49
- The result is reached within the same steps.

Foreach

```scala
val list = List("mff", "cuni", "cz")
```

- Following 3 calls are equivalent

```scala
list.foreach((s : String) => println(s))
list.foreach(s => println(s))
list.foreach(println)
```

Many More Features

- **Actors**
- **Annotations** → `@foo def hello = "world"`
- **Case Classes** → `case class Foo(bar:String)`
- **Currying** → `def foo(a:Int,b:Boolean)(c:String)`
- **For Comprehensions**
  → `for(i <- 1.to(5) if i % 2 == 0) yield i`
- **Generics** → `class Foo[T](bar:T)`
- **Package Objects**
- **Partially Applied Functions**
- **Tuples** → `val t = (1,"foo","bar")`
- **Type Specialization**
- **XML Literals** → `val node = <hello>world</hello>`
- **etc…**