Programmazione OO in SCALA

Info III

Elenco degli argomenti sul syllabus

Basic scala

Basic Scala

- Use `var` to declare variables

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

- Use `val` to declare values

```
val y = 3;
y += 4; // error
```

- Notice no types, but it is statically typed

```
var x = 3;
x = “hello world”; // error
```

- Type annotations:

```
var x : Int = 3;
```

Defs, Vals, and Vars

Three types of identifier definitions:

`def` defines functions with parameters; RHS expression evaluated each time called

`val` defines unchanging values; RHS expression evaluated immediately to initialize

`var` defines storage location whose values can be changed by assignment statements; RHS expression evaluated immediately to initialize

Variables & values, type inference

```
var msg = "Hello“      // msg is mutable
msg += " world"
msg = 5;               // compiler error
```

Variables & values, type inference

```
val msg = "Hello world“     // msg is immutable
msg += " world“             // compiler error
```

```
val n : Int = 3     // explicit type declaration
var n2 : Int = 3
```

Immutability

- Why?
  - Immutable objects are automatically thread-safe

    (you don't have to worry about object being changed by another thread)
  - Compiler can reason better about immutable values -> optimization

  - Steve Jenson from Twitter: *"Start with immutability, then use mutability where you find appropriate."*

Calling Java from Scala

- Any Java class can be used seamlessly

```
import java.io._
val url = new URL("http://www.scala-lang.org")
```

demo

Methods

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


// equivalent:
def neg(x : Int) : Int = -x
def neg(x : Int) : Int = { return -x; }
```

Types

- Int, Double, String, Char, Byte, BigInt, …
  - wrappers around Java types

OO programming in Scala

Scala object system

- Class-based
- Single inheritance
- Can define singleton **objects** easily (no need for static which is not really OO)
- Traits, compound types, and views allow for more flexibility

Defining Hello World

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hey world!")
  }
}
```

- Singleton object named `HelloWorld` (also replaces `static` methods and variables)
- Method `main` defined (procedure)
- Parameter `args` of type `Array[String]`
- `Array` is generic class with type parameter

Classes

```
/** A Person class.
 * Constructor parameters become
 * public members of the class.*/
class Person(val name: String, var age: Int) {
  if (age < 0) {
      throw …
  }
}

var p = new Person(“Peter", 21);
p.age += 1;
```

Constructor

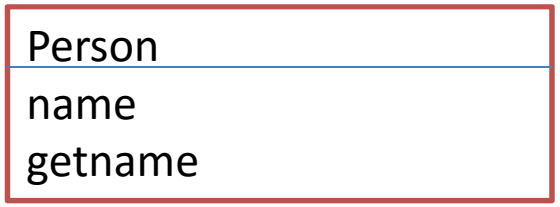
In Scala the *primary constructor* is the class' body and it's parameter list comes right after the class name.

In Scala we create variables (fields) either using the *val* keyword or the *var* keyword. Using *val* we get a read-only variable that's immutable.

```
class Person(n: String){
  val name = n;
  def getname() = name
}
```

```
class Person(val name:
                    String){
    def getname() = name
}
```

```
class Person(var name:
                    String){
    def getname() = name
}
```



Auxiliary constructors

….

Objects

- Scala's way for "statics"
  - not quite – see next slide
  - (in Scala, there is no *static* keyword)

- "Companion object" for a class
  - = object with same name as the class

demo

An analog to a companion object in Java is having a class with static methods. In Scala you would move the static methods to a Companion object.

Objects

```
// we declare singleton object "Person"
// this is a companion object of class Person
object Person {
  def defaultName() = "nobody"
}
class Person(val name: String, var age: Int) {
  def getName() : String = name
}

// surprise, Person is really an object
val singleton : Person = Person;
```

Extending classes

```
class Point(xc: Int, yc: Int) {
  val x: Int = xc
  val y: Int = yc
  def move(dx: Int, dy: Int): Point = new Point(x + dx, y + dy)
}
class ColorPoint(u: Int, v: Int, c: String) extends Point(u, v){
  val color: String = c
  def compareWith(pt: ColorPoint): Boolean =
    (pt.x == x) && (pt.y == y) && (pt.color == color)
  override def move(dx: Int, dy: Int): ColorPoint =
    new ColorPoint(x + dy, y + dy, color)
}
```

ColorPoint adds a new method compareWith - Scala allows member definitions to be *overridden*; Piu' o meno le stesse regole di Java (covarianza del tipo ritornato).
Subclasses define subtypes; this means in our case that we can use ColorPoint objects whenever Point objects are required.

Abstract classes

```
abstract class Greeter {
  val message: String //abstract
  def SayHi() = println(message)
}
class BerghemGreeter extends Greeter {
  val message = "alura"
}
object prova {
  val greeter = new BerghemGreeter() > greeter  : BerghemGreeter = BerghemGreeter@141a32f
  greeter.SayHi()                    > alura
}
```

Traits

- Similar to interfaces in Java
- They may have implementations of methods
- But can’t contain state
- Can be multiply inherited from
  - Scala’s solution to the Diamond Problem is actually fairly simple: it considers the order in which traits are inherited. If there are multiple implementors of a given member, the implementation in the supertype that is furthest to the right (in the list of supertypes) “wins.” Of course, the body of the class or trait doing the inheriting is further to the right than the entire list of supertypes, so it “wins” all conflicts, should it provide an overriding implementation for a member.

Traits example

```
trait Similarity {
  def isSimilar(x: Any): Boolean
}

class Student extends Similarity {
  def isSimilar(x: Student) = true
}
```

Scala class hierarchy