

Scala Language



Dessì Massimiliano

Linux Day Cagliari
24 ottobre 2009

Author

Software Architect and Engineer

ProNetics / Sourcesense

Presidente

JugSardegna Onlus

Fondatore e coordinatore

SpringFramework Italian User Group

Committer - Contributor

OpenNMS - MongoDB - MagicBox

Autore

Spring 2.5 Aspect Oriented Programming



Topics

Quanti core per processore ci sono sul server
su cui metti le tue applicazioni ?

La tua applicazione gestisce la concorrenza ?

Il tuo codice è scritto pensando alla scalabilità ?

Consideri questi aspetti quando scrivi il tuo codice ?

Se hai risposto affermativamente
fermati ad ascoltare questa presentazione

SCAlable LAnguage

E' un linguaggio funzionale
ad oggetti puro.

Ogni cosa è un oggetto.

Non ci sono primitive.

Gira sulla Java Virtual Machine

Funzionale vs Imperativo

I linguaggi funzionali dicono
cosa deve fare il programma

Java e i linguaggi Object Oriented
sono imperativi,
dicono **come** deve farlo

SCALA - best of both

In Scala, paradigma funzionale e Object oriented
sono complementari.

Invita ad utilizzare l'approccio funzionale,
ma non vincola le scelte.

Tutto è un oggetto
funzioni comprese.

Scala

Scala può essere utilizzato sia come linguaggio di scripting,
sia come linguaggio ad alta concorrenza

Possiede le caratteristiche di un linguaggio dinamico

Con la velocità di un linguaggio *statico* e *tipizzato*, grazie
alla inferenza si può scrivere codice
molto più conciso

Scala Inference

Scala è un linguaggio
low-ceremony, static typing

```
var anno: Int = 2009  
var altroAnno = 2009  
var saluto = "Ciao mondo"
```

quando definiamo il tipo
compare su un solo lato
nella assegnazione

```
var builder = new StringBuilder("hello" )
```


Scala Inference

Scala attraverso l' inferenza, quando
non specificato, comprende
anche il tipo di ritorno dei metodi

```
def metodoUno () { 2 }  
def metodoDue () = { 4 }  
def metodoTre () = 6  
def metodoQuattro : Double = 8
```

Java

```
public class Persona {  
  
    private String nome, cognome;  
    private int eta;  
  
    public Persona(String nome, int eta, String cognome) {  
        this.nome = nome;  
        this.eta = eta;  
        this.cognome = cognome;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
    public int getEta() {  
        return eta;  
    }  
  
    public void setEta(int eta) {  
        this.eta = eta;  
    }  
  
    public String getCognome() {  
        return cognome;  
    }  
}
```

Scala

```
case class Persona(val nome: String, var eta: Int, val cognome: String)
```

Il compilatore scala se non dichiariamo altro,
genera di default (case Class):

toString,

hashCode,

equals,

Costruttore con i parametri dichiarati

getter dei parametri definiti **val**

setter e getter dei parametri definiti **var**

Scala

```
scalac Person.scala  
javap -private Person
```

Compiled from "Person.scala"

```
public class org.magicbox.domain.Person extends java.lang.Object implements  
scala.ScalaObject,scala.Product,java.io.Serializable{  
    private final java.lang.String cognome;  
    private int eta;  
    private final java.lang.String nome;  
    public org.magicbox.domain.Person(java.lang.String, int, java.lang.String);  
    private final boolean gdl$1(java.lang.String, int, java.lang.String);  
    public java.lang.Object productElement(int);  
    public int productArity();  
    public java.lang.String productPrefix();  
    public boolean equals(java.lang.Object);  
    public java.lang.String toString();  
    public int hashCode();  
    public int $tag();  
    public java.lang.String cognome();  
    public void eta_$eq(int);  
    public int eta();  
    public java.lang.String nome();  
}
```

Scala

Se invece vogliamo anche la convenzione classica JavaBean

get<nome>

set<nome>

Utilizziamo l'annotazione

```
@scala.reflect.BeanProperty
```

```
case class Person(@scala.reflect.BeanProperty val nome: String,  
                  @scala.reflect.BeanProperty var eta: Int,  
                  @scala.reflect.BeanProperty val cognome: String)
```

Scala

scalac Person.scala
javap -private Person

Compiled from "Person.scala"

```
public class org.magicbox.domain.Person extends java.lang.Object implements
scala.ScalaObject,scala.Product,java.io.Serializable{
    private final java.lang.String cognome;
    private int eta;
    private final java.lang.String nome;
    public org.magicbox.domain.Person(java.lang.String, int, java.lang.String);
    private final boolean gdl$1(java.lang.String, int, java.lang.String);
    public java.lang.String getCognome();
    public void setEta(int);
    public int getEta();
    public java.lang.String getNome();
    public java.lang.Object productElement(int);
    public int productArity();
    public java.lang.String productPrefix();
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
    public int hashCode();
    public int $tag();
    public java.lang.String cognome();
    public void eta_$eq(int);
    public int eta();
    public java.lang.String nome();
}
```

Val e Var

Scala favorisce l' immutabilità.

Questo significa assegnare i valori definitivi al momento della creazione per non avere “side-effects”.

Per ottenere questo risultato utilizziamo la keyword `val`.

Quando usiamo `val`, il compilatore creerà una variabile final.

Se invece abbiamo necessità di poter riassegnare dei valori utilizziamo `var`, ma è consigliabile utilizzare

il più possibile `val` per scrivere codice in stile funzionale

Singleton

Posso avere una sola istanza (per classloader)
con la keyword `Object`,
l'oggetto è già pronto all'uso senza essere istanziato.

```
class Marcatore private(val color: String) {  
    override def toString() : String = "colore marcatore " + color  
}  
  
object Marcatore {  
    private val marcatori = Map(  
        "red" -> new Marcatore("red" ),  
        "blue" -> new Marcatore("blue" ),  
        "green" -> new Marcatore("green" ))  
  
    def getMarcatore(colore: String) =  
        if (marcatori.contains(colore)) marcatori(colore) else null  
}
```

In questo caso `object` Marcatore agisce anche da Companion
Object di `class` Marcatore

Gerarchia e alcune keyword

`Any` è il tipo base

Classi figlie di `Any` sono `AnyVal` e `AnyRef`

`AnyVal` mappa i tipi primitivi Java

`AnyRef` mappa i tipi reference

Alla fine della gerarchia abbiamo `Nothing`

`Unit` corrisponde al void

Traits

Sono come le interfacce Java ma con una parziale implementazione

Ci consentono di avere i benefici della ereditarietà multipla, ovvero avere classi con più funzionalità, ma senza i corrispettivi problemi.

Vengono usate anche per la creazione di DSL

Traits

```
trait Amico {  
    val nome: String  
    def ascolta() = println("Il tuo amico " + nome + " ti ascolta" )  
}  
  
class Umano(val nome: String) extends Amico  
  
class Uomo(override val nome: String) extends Umano(nome)  
  
class Donna(override val nome: String) extends Umano(name)  
  
class Animale  
  
trait Verso {  
    val verso: String  
    def tono() = println("sgnanfuz" )  
}  
  
class Cane(val nome: String) extends Animale with Amico with Verso {  
    override def ascolta = println(nome + " ascolta in silenzio" )  
    override def tono = println("basso" )  
    val verso = "bau bau"  
}
```

Pattern Matching e XML

Scala utilizza il pattern matching
in maniera molto estesa, l'utilizzo
più comune è quello

con gli Actor che vedremo nelle pagine seguenti.

Per l'XML Scala fornisce il supporto nativo per la dichiarazione,
lettura, lettura da file e ovviamente creazione/scrittura.

Collection

Scala fornisce collection basate su quelle Java, sia di tipo immutabile che mutabile.

I tipi principali sono map, sequenze e liste, ma tutte arricchite nelle funzionalità.

Anche nel caso delle collection lo scopo è fornire collezioni per l'alta concorrenza.

Function value

Possiamo passare una funzione come parametro

```
def store(arr: Array[Int], start: Int, funzione: (Int, Int) => Int) :  
Int = {  
    var precedente = start  
    arr.foreach(element => precedente = funzione(precedente, element) )  
    precedente  
}
```

Passando funzioni anonime diverse possiamo avere:

```
scala> val array = Array(2, 3, 5, 1, 6, 4, 13)
```

```
scala> val somma = store(array, 0, (precedente, elem) => precedente + elem)  
somma: Int = 34
```

```
scala> val max = store(array, Integer.MIN_VALUE, (precedente, elem) =>  
Math.max(precedente, elem))  
max: Int = 13
```

Function value

Possiamo anche ordinare meglio riusando le funzioni

```
def somma(precedente: Int, elem : Int) = { precedente + elem }
```

```
def max(precedente: Int, elem : Int) = { Math.max(precedente, elem) }
```

```
scala> val sum = store(array, 0, somma)
```

```
sum: Int = 34
```

```
scala> val massimo = store(array, Integer.MIN_VALUE, max)
```

```
massimo: Int = 13
```

Concorrenza



Scala Concurrency

Scala rende semplice la programmazione multithread.

I thread comunicano tra loro
utilizzando un modello ad eventi
per inviarsi oggetti immutabili come messaggi.

Actors

Gli actors sono unità di esecuzione
threadless e stackless
che processano messaggi (eventi)
in maniera seriale.

Un actor riceve i messaggi nella sua mailbox,
li processa uno alla volta
in maniera asincrona
prendendoli dalla mailbox.

Actors

Un actor non esponendo nessuno stato
e venendo modificato o interrogato
attraverso i messaggi,
che sono processati in maniera seriale
non ha bisogno di lock sul suo stato interno
ed è perciò thread safe

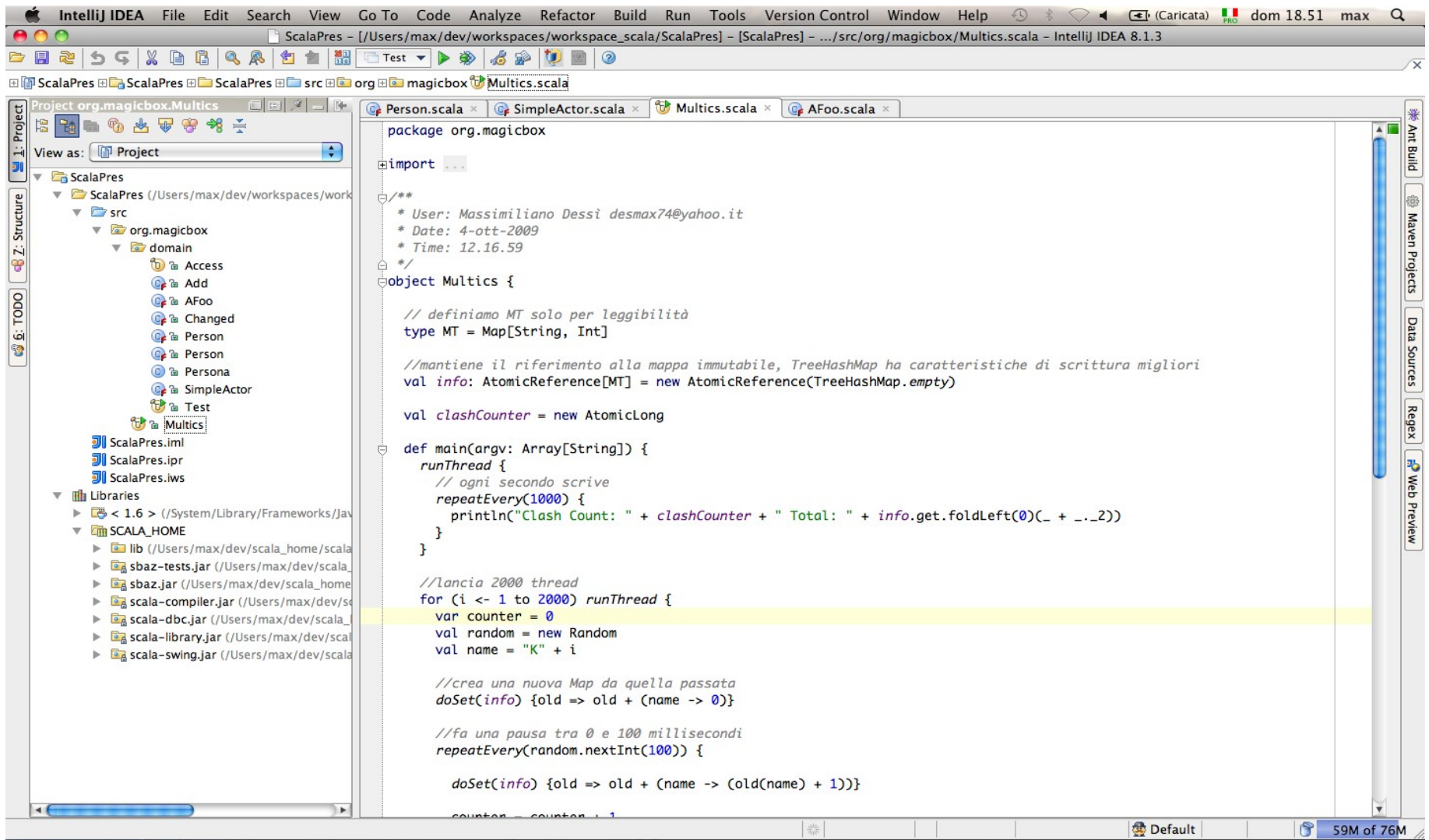
Actor

```
class FooActor extends Actor {  
  
  def act() = {  
    var control = true  
    while (control) {  
      react {  
        case "mailbox" => println("Mailbox: " + mailbox.toString)  
        case "quit" => {  
          println("spengo questo FooActor")  
          control = false  
        }  
        case i: Int => println("Ho ricevuto un Int: " + i.toString)  
        case _ => println("Nulla di particolare.")  
      }  
    }  
  }  
}
```

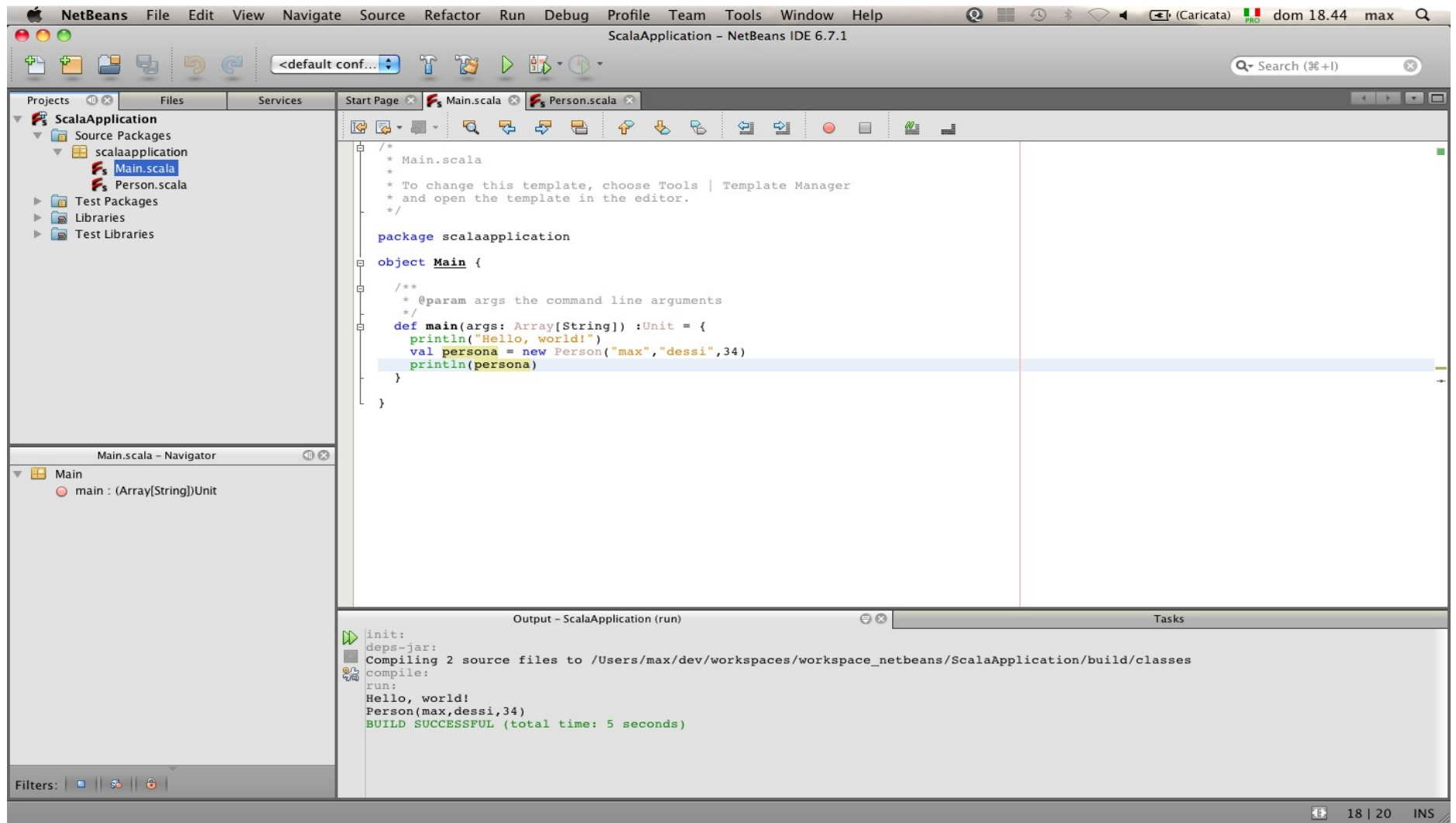
Actor

```
object Test{  
  def main(args: Array[String]) {  
    val foo = new FooActor()  
    foo.start  
    foo ! "mailbox"  
    foo ! 1  
    foo ! 12  
    foo ! 20  
    foo ! "mailbox"  
    System.exit(0)  
  }  
}
```

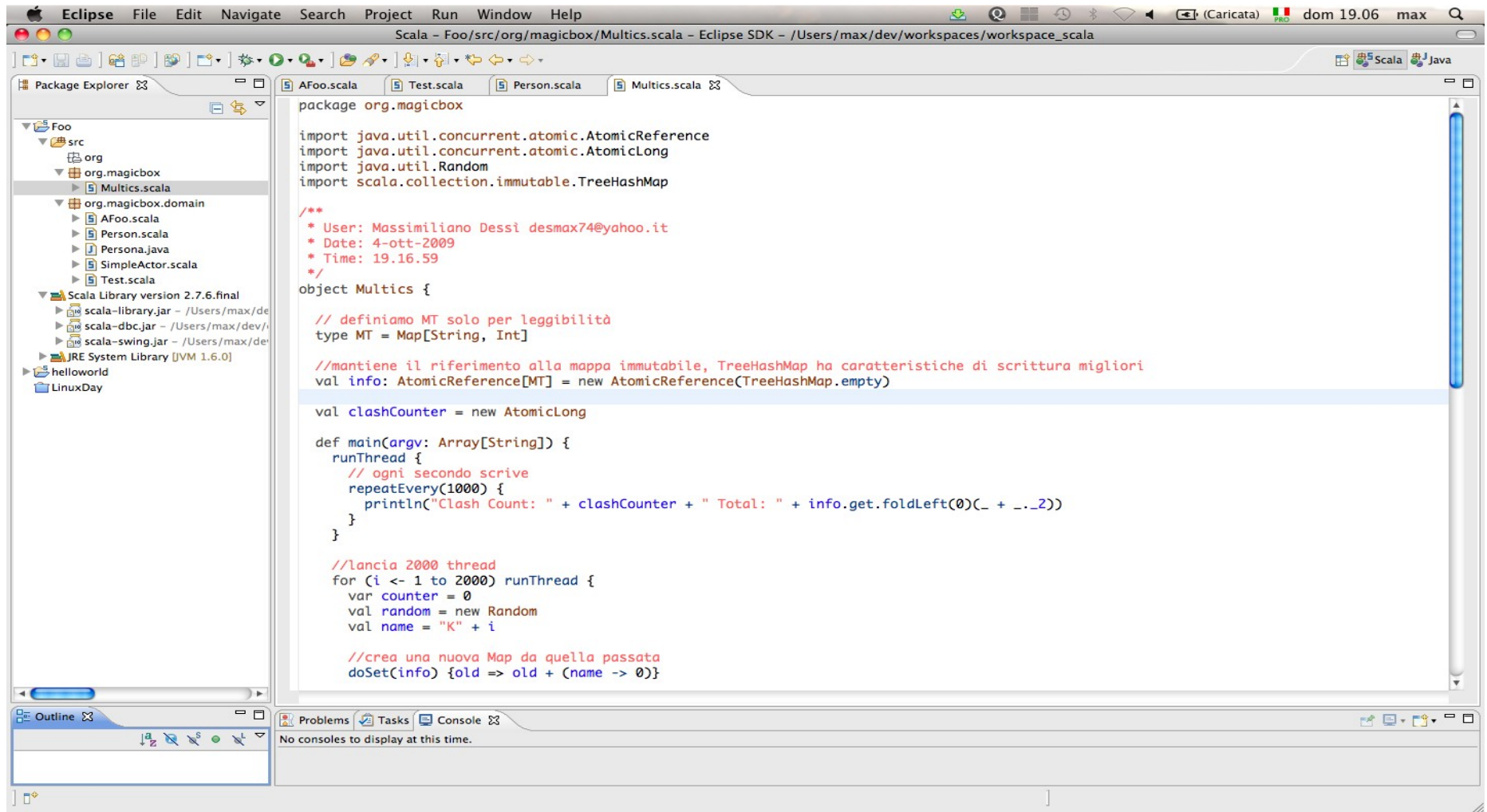
Scala - IntelliJ IDEA



Scala - Netbeans



Scala - Eclipse



Demo concorrenza

Una negozio da barbiere

3 sedie

1 barbiere

n-clienti



Domande ?

Grazie per l'attenzione !

Massimiliano Dessì

desmax74 at yahoo.it

massimiliano.dessi at pronetics.it

<http://twitter.com/desmax74>

<http://jroller.com/desmax>

<http://www.linkedin.com/in/desmax74>

<http://www.slideshare.net/desmax74>

<http://wiki.java.net/bin/view/People/MassimilianoDessi>

<http://www.jugsardegna.org/vqwiki/jsp/Wiki?MassimilianoDessi>