

# SCALA

## *Conception Objet Avancée*

Baptiste CARLIER  
Alban DE UBEDA

24 Novembre 2009

La Scala de Milan, en italien Teatro alla Scala (ou simplement la Scala) à Milan, est l'un des plus importants théâtres d'opéra du monde.



**Plan :**

- **Histoire**
- **Opéras créés à la Scala**
- **Chef principal et directeur musical de l'opéra**

# Qu'est-ce que le langage de programmation Scala ?

- Introduction à Scala
- Les caractéristiques du langage et comparaison avec Java
- Quelques exemples de code



# Bienvenue à la programmation Scala

## La popularité des langages de programmation

Qu'est-ce qui fait la popularité d'un langage de programmation ?

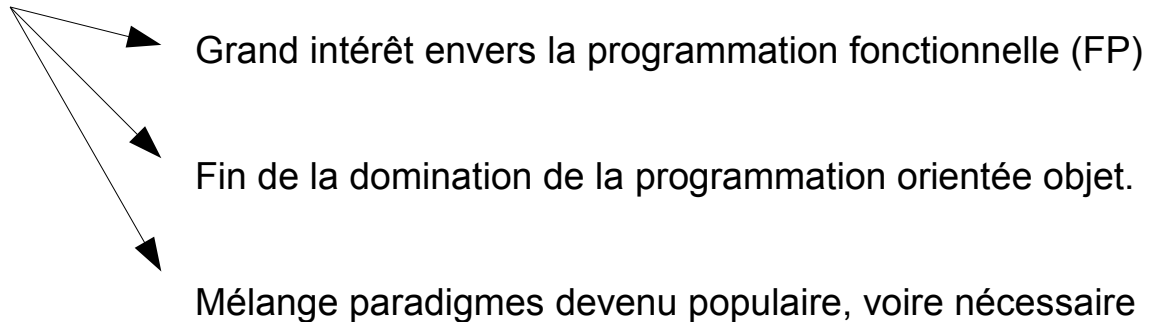
- La plate-forme de référence :
  - Mac OS —▶ Objective-C
  - Windows —▶ C++ / .NET
  - Systèmes embarqués —▶ C / C++
- La mode (voir le fanatisme) :
  - C++, Java, Ruby, ...
- L'adaption du langage aux besoins de son époque.
  - Java ajustait parfaitement le navigateur et applications client riche.
  - Smalltalk capturait l'essence de la programmation orientée objet (POO).

# Bienvenue à la programmation Scala

## La popularité des langages de programmation

Et aujourd'hui ?

- Simultanéité
- Hétérogénéité
- La continuité de services
- ...



# Bienvenue à la programmation Scala

## Petit historique

Scala, un langage de programmation orienté objet et fonctionnel développé à l'EPFL en Suisse.

- Apparue en : 2003
- Auteur : Martin Odersky (Allemand)
- EPFL : École Polytechnique Fédérale de Lausanne
- Paradigmes : Objet, impératif, fonctionnel
- Influencé par : Java, Pizza, Haskell, Erlang, Standard ML, Objective Caml, Smalltalk.

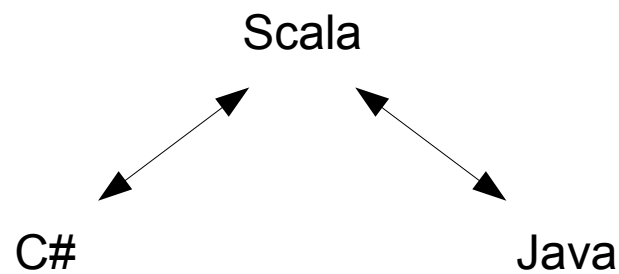
# Caractéristiques de Scala

## Les caractéristiques du langage de programmation Scala

- JVM et CLR

Scala est connue comme une langue JVM, ce qui signifie que Scala génère du bytecode JVM.

Une version .NET de Scala est également en cours de développement qui génère du code CLR octet.



# Caractéristiques de Scala

## Les caractéristiques du langage de programmation Scala

- Typé statiquement

Liaison d'un type à la variable pour la durée de vie de cette variable

Le typage dynamique lie le type à la valeur réelle référencée par une variable.

Scala utilise le typage statique.

# Caractéristiques de Scala

## Les caractéristiques du langage de programmation Scala

- **Multiparadigme**

### *Object Oriented Programming (Java)*

Scala supporte pleinement Programmation Orientée Objet.

Tout est vraiment un objet. Pas de types primitifs, comme Java.

Au lieu de cela tous les types numériques sont de vrais objets.

# Caractéristiques de Scala

## Les caractéristiques du langage de programmation Scala

- **Multiparadigme**

### Programmation fonctionnelle (C)

Scala supporte aussi la programmation fonctionnelle.

Plus anciens que les langages object-oriented, les langages fonctionnels purs ne permettent pas les État mutable, évitant ainsi la nécessité pour la synchronisation sur l'accès partagé à l'état mutables (*multithread*).

Les fonctions peuvent être assigné à des variables, et donc passé en paramètre à d'autres fonctions etc.. tout comme d'autres variables

Dans Scala tout est un objet, fonctions elles-mêmes sont des objets.

Scala propose également des *closure*, une caractéristique absente des dernières versions de Java.

# Caractéristiques de Scala

## Les caractéristiques du langage de programmation Scala

- **Performance**

Scala bénéficie de toutes les optimisations de performances fournies par les runtimes.

Certaines constructions particulièrement dans la langue et certaines parties de la bibliothèque de peuvent entrainer des résultats sensiblement meilleurs ou pires que d'autres options dans d'autres langues.

# Caractéristiques de Scala

## Les caractéristiques du langage de programmation Scala

- Inférence

En Scala, la déclaration d'une variable se fait de la manière suivante *variable:Type* :

```
var myStr:String = 'xebia'
```

Scala possède le mécanisme d'inférence et donc il est possible d'écrire cette déclaration de la manière suivante :

```
var myStr = 'xebia'
```

La construction d'une liste :

```
val listStr: Array[String] = new Array[String](4)
```

Ici la variable *listStr* est de type *Array[String]* et est initialisé avec une longueur de 4. Néanmoins, Scala permet de simplifier cette déclaration (mécanisme d'inférence) :

```
val listStr = new Array[String](4)
```

# Caractéristiques de Scala

## Les caractéristiques du langage de programmation Scala

- Inférence

C'est la base des simplifications de code de Scala :

```
object InferenceTest1 extends Application {  
    val x = 1 + 2 * 3           // Le type de x est Int  
    val y = x.toString()      // Le type de y est String  
    def succ(x: Int) = x + 1  // La méthode succ renvoie un Int  
}
```

# Par rapport à Java

## Petite comparaison

- Java date de 1995  
Devenu trop complexe ? Ne s'attaque pas adéquatement aux défis de développement les plus récents ?
- La JVM continue de briller.  
Les optimisations effectuées sont extraordinaires
- Utilisation de la JVM avec de nouvelles langues : le chemin à suivre ? Sun s'emploie là dessus.  
Jython et JRuby, qui sont des ports de la JVM: Ruby et Python, respectivement.
- Scala : version plus récente, plus moderne de Java.

# Exemples de code Scala

## Le code

```
class Upper {  
    def upper(strings: String*): Seq[String] = {  
        strings.map((s:String) => s.toUpperCase())  
    }  
}  
  
val up = new Upper  
  
Console.println(up.upper("A", "First", "Scala", "Program"))
```

- *Upper* : nom de la class

# Exemples de code Scala

## Le code

```
class Upper {  
    def upper(strings: String*): Seq[String] = {  
        strings.map((s:String) => s.toUpperCase())  
    }  
}  
  
val up = new Upper  
  
Console.println(up.upper("A", "First", "Scala", "Program"))
```

- *upper* : nom de la fonction

# Exemples de code Scala

## Le code

```
class Upper {  
    def upper(strings: String*): Seq[String] = {  
        strings.map((s:String) => s.toUpperCase())  
    }  
}  
  
val up = new Upper  
  
Console.println(up.upper("A", "First", "Scala", "Program"))
```

- *strings* : nom de la variable

# Exemples de code Scala

## Le code

```
class Upper {  
    def upper(strings: String*): Seq[String] = {  
        strings.map((s:String) => s.toUpperCase())  
    }  
}  
  
val up = new Upper  
  
Console.println(up.upper("A", "First", "Scala", "Program"))
```

- *string\** : type, ici un tableau de string

# Exemples de code Scala

## Le code

```
class Upper {  
    def upper(strings: String*): Seq[String] = {  
        strings.map((s:String) => s.toUpperCase())  
    }  
}  
  
val up = new Upper  
  
Console.println(up.upper("A", "First", "Scala", "Program"))
```

- `seq[String]` : type de retour  
`seq` est un type de collection particulier (ici de string).

# Exemples de code Scala

## Le code

```
class Upper {  
    def upper(strings: String*): Seq[String] = {  
        strings.map((s:String) => s.toUpperCase())  
    }  
}  
  
val up = new Upper  
  
Console.println(up.upper("A", "First", "Scala", "Program"))
```

Ensuite c'est la définition de la méthode :

- *.map* est une énumération. Cela revient à faire un foreach  
Pour chaque sting *s*, on la met donc en majuscule.

# Exemples de code Scala

## Le code

```
class Upper {  
    def upper(strings: String*): Seq[String] = {  
        strings.map((s:String) => s.toUpperCase())  
    }  
}  
  
val up = new Upper  
  
Console.println(up.upper("A", "First", "Scala", "Program"))
```

En Scala, la dernière expression dans la fonction est la valeur de retour, bien que vous pouvez avoir des déclarations *return* d'ailleurs, aussi.

Le mot-clé *return* est facultative ici et il est rarement utilisé, sauf lors du retour de celle du milieu d'un bloc (par exemple, dans un if déclaration).

# Exemples de code Scala

Le code

Compilation :

```
scalac exemple1.scala
```

Le fichier est interprété, ce qui signifie qu'il est compilé et exécuté en une seule étape.

# Exemples de code Scala

## Le code

```
class Upper {  
    def upper(strings: String*): Seq[String] = {  
        strings.map((s:String) => s.toUpperCase())  
    }  
}  
  
val up = new Upper  
  
Console.println(up.upper("A", "First", "Scala", "Program"))
```

Résultat du programme ci-dessus :

*Array(A, FIRST, SCALA, PROGRAM)*

# Exemples de code Scala

Le code

## Autre solution :

```
object Upper {  
    def upper(strings: String*) = strings.map(_.toUpperCase())  
}  
  
println(Upper.upper("A", "First", "Scala", "Program"))
```

Ce programme fait le même résultat, mais est différent syntaxiquement.

# Exemples de code Scala

Le code

## Autre solution :

```
object Upper {  
    def upper(strings: String*) = strings.map(_.toUpperCase())  
}  
  
println(Upper.upper("A", "First", "Scala", "Program"))
```

- *objet* rend *Upper* singleton. Il n'y a donc pas besoin de *new* ici.

Scala peut généralement déduire le type de retour de la méthode, on retire donc le type de retour.

# Exemples de code Scala

Le code

## Autre solution :

```
object Upper {  
    def upper(strings: String*) = strings.map(_.toUpperCase())  
}  
  
println(Upper.upper("A", "First", "Scala", "Program"))
```

$(S: String) \Rightarrow s.toUpperCase()$  a été réduit à l'expression suivante:  
 $_.toUpperCase()$

Le « `_` » agit comme une variable anonyme, car `map` n'a qu'un argument.

# Exemples de code Scala

Le code

## Autre solution :

```
object Upper {  
    def upper(strings: String*) = strings.map(_.toUpperCase())  
}  
  
println(Upper.upper("A", "First", "Scala", "Program"))
```

Enfin, Scala importe automatiquement de nombreuses méthodes pour I/O, donc il n'y a plus besoin de console.

# Exemples de code Scala

## Code avancé

```
package shapes {
  class Point(val x: Double, val y: Double) {
    override def toString() = "Point(" + x + "," + y + ")"
  }
  abstract class Shape() {
    def draw(): Unit
  }
  class Circle(val center: Point, val radius: Double) extends Shape {
    def draw() = println("Circle.draw: " + this)
    override def toString() = "Circle(" + center + "," + radius + ")"
  }
  class Rectangle(val lowerLeft: Point, val height: Double, val width: Double) extends Shape
  {
    def draw() = println("Rectangle.draw: " + this)
    override def toString() = "Rectangle(" + lowerLeft + "," + height + "," + width + ")"
  }
  class Triangle(val point1: Point, val point2: Point, val point3: Point) extends Shape {
    def draw() = println("Triangle.draw: " + this)
    override def toString() = "Triangle(" + point1 + "," + point2 + "," + point3 + ")"
  }
}
```

Le mot clef *override* obligatoire.

Il n'y a pas de mot clef *abstract* pour les méthode abstraite, on sait qu'elles le sont car elle n'ont pas de corps.

# Exemples de code Scala

## Code avancé

```
package shapes {
  class Point(val x: Double, val y: Double) {
    override def toString() = "Point(" + x + "," + y + ")"
  }
  abstract class Shape() {
    def draw(): Unit
  }
  class Circle(val center: Point, val radius: Double) extends Shape {
    def draw() = println("Circle.draw: " + this)
    override def toString() = "Circle(" + center + "," + radius + ")"
  }
  class Rectangle(val lowerLeft: Point, val height: Double, val width: Double) extends Shape
  {
    def draw() = println("Rectangle.draw: " + this)
    override def toString() = "Rectangle(" + lowerLeft + "," + height + "," + width + ")"
  }
  class Triangle(val point1: Point, val point2: Point, val point3: Point) extends Shape {
    def draw() = println("Triangle.draw: " + this)
    override def toString() = "Triangle(" + point1 + "," + point2 + "," + point3 + ")"
  }
}
```

La liste des arguments après le nom de classe sont les paramètres du constructeur.

En Scala, le corps d'une classe entière est le constructeur.

Parce qu'il y a le mot-clé *val* avant chaque déclaration de paramètres, ils sont automatiquement convertis en variable en lecture seuls en avec les mêmes noms.

# Exemples de code Scala

## Code avancé

```
package shapes {
  import scala.actors._
  import scala.actors.Actor._
  object ShapeDrawingActor extends Actor {
    def act() {
      loop {
        receive {
          case s: Shape => s.draw()
          case "exit" => println("exiting..."); exit
          case x: Any => println("Error: Unknown message! " + x)
        }
      }
    }
  }
}
```

Dans les import, le « `_` » remplace « `*` »

Lors de chaque passage dans la boucle, la méthode `receive` est appelée. Elle bloque jusqu'à ce qu'un nouveau message arrive.

Le `case` compare par rapport à la source du message.

# Exemples de code Scala

## Code avancé

### Script utilisant ces classes :

```
ShapeDrawingActor.start()  
ShapeDrawingActor! new Cercle (new Point (0.0,0.0), 1.0)  
ShapeDrawingActor! new Rectangle (new Point (0.0,0.0), 2, 5)  
ShapeDrawingActor! new Triangle (new Point (0.0,0.0),  
new Point (1.0,0.0),  
new Point (0.0,1.0))  
ShapeDrawingActor! 3,14159  
ShapeDrawingActor! "Exit"
```

5 messages sont envoyés à l'acteur, en utilisant la syntaxe « *actor ! message* ».

# Exemples de code Scala

Code avancé

## Script utilisant ces classes :

```
ShapeDrawingActor.start()  
ShapeDrawingActor! new Cercle (new Point (0.0,0.0), 1.0)  
ShapeDrawingActor! new Rectangle (new Point (0.0,0.0), 2, 5)  
ShapeDrawingActor! new Triangle (new Point (0.0,0.0),  
new Point (1.0,0.0),  
new Point (0.0,1.0))  
ShapeDrawingActor! 3,14159  
ShapeDrawingActor! "Exit"
```

## Résultat :

```
Circle.draw: Cercle (Point (0.0,0.0), 1.0)  
Rectangle.draw: Rectangle (Point (0.0,0.0), 2.0,5.0)  
Triangle.draw: Triangle (Point (0.0,0.0), Point (1.0,0.0), Point (0.0,1.0))  
Erreur: message inconnu! 3,14159  
ShapeDrawingActor! "Exit"
```

# Qu'est-ce que le langage de programmation Scala ?

# Qu'est-ce que le langage de programmation Scala ?

## Bilan et caractéristiques

# Qu'est-ce que le langage de programmation Scala ?

## Bilan et caractéristiques

- Orienté-objet: toutes les valeurs sont des objets.
- Fonctionnel: les fonctions sont des valeurs.
- Syntaxe proche de celle de Java. Compilé en bytecode Java.
- Statiquement typé.
- Classes génériques.
- Méthodes polymorphiques.
- Abstractions.
- Extensible: une méthode peut être utilisée comme opérateur.
- Compatible avec le runtime Java et le runtime .NET.
- Fonctions imbriquées.
- Pas de terminateur tel que ; à la fin des instructions.
- Pas de break et de continue.

# QUESTION ?

## *Conception Objet Avancée*

Baptiste CARLIER  
Alban DE UBEDA

24 Novembre 2009