Scala

... a Scalable Language

06.10.2009

XPUG Rhein Main

Mario Gleichmann

Introduction

Mario Gleichmann

site: www.mg-informatik.de

blog: 'brain driven development'
gleichmann.wordpress.com

mail: mario.gleichmann@mg-informatik.de

Motivation

Open your mind ...

- Scala vs. Java 9
- Functional programming for the imperative mind
- Discover the (new) possibilities ...

Motivation

Open your mind ...

- **Scala vs. Java 9**
- Functional programming for the imperative mind
- Discover the (new) possibilities ...

“If **Java** programmers want to use features that aren't present in the language, I think they're probably best off using another language that targets the JVM, such as **Scala** and Groovy“

Joshua Bloch

Motivation

Open your mind ...

- **Scala vs. Java 9**
- Functional programming for the imperative mind
- Discover the (new) possibilities ...

- **Functions & Closures**
- **Extended Type System**
- **Extended Module System**
- **Properties**
- **Essence over Ceremony**
- **Extended Control Structs**

Motivation

Open your mind ...

- **Scala vs. Java 9**
- Functional programming for the imperative mind
- Discover the (new) possibilities ...

> “If i were to pick a language to use today other than **Java**, it would be **Scala**“
>
> James Gosling

Motivation

Open your mind ...

- **Functional programming for the imperative mind**
- Discover the possibilities ...

Imparative programming
is a programming paradigm
that describes **computation**
in terms of **statements**
that change a programs **state**

Motivation

Open your mind ...

- **Functional programming for the imperative mind**
- Discover the possibilities ...

Functional programming
is a programming paradigm
that describes **computation** as the
evaluation of **mathematical functions**
avoiding **state** and **mutable data**

Motivation

Open your mind ...

- **Functional programming for the imperative mind**
- Discover the possibilities ...

Lazy Evaluation

Continuations

Monads

Recursion

Higher Order Functions

Closures

Currying

Immutable Datatypes

Motivation

Open your mind ...

- **Discover the (new) possibilities ...**

Control Structure Abstraction

Composition

Traits

Pattern Matching

Type Variance

Modularity

Type Extentions / Conversions

Motivation

Open your mind !!!

“Scala **taught me to program and reason about programming differently**. I stopped thinking in terms of allocating buffers, structs and objects and of changing those pieces in memory. Instead I learned to think about most of my programs as transforming input to output.

This change in thinking has lead to lower defect rates, more modular code, and more testable code“

David Pollak

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- Functional
- Runs on the JVM

What is Scala ?

A programming language ...

- Martin Odersky (EPFL Switzerland)
  - Pizza, Fummel & Co.
  - Generic Java
  - javac Reference Compiler

- Pure Object Oriented
- Functional
- Statically Typed
- Runs on the JVM

What is Scala ?

A programming language ...

- **Pure Object Oriented**
- Statically Typed
- Functional
- Runs on the JVM

"Everything is an Object"

1 + 2 <=> 1.+(2)

What is Scala ?

A programming language ...

- **Pure Object Oriented**
- Statically Typed
- Functional
- Runs on the JVM

"Everything is an Object"

1 + 2   <=> 1.+(2)

"No primitive Types"

123.hashCode

What is Scala ?

A programming language ...

- **Pure Object Oriented**
- Statically Typed
- Functional
- Runs on the JVM

"Everything is an Object"

1 + 2 <=> 1.+(2)

"No primitive Types"

123.hashCode

"Operations are method calls"

actor ! msg <=> actor.!(msg)

What is Scala ?

A programming language ...

- **Pure Object Oriented**
- Functional
- Statically Typed
- Runs on the JVM

Java

```
public BigInteger factorial( BigInteger n ){

    if( n.equals( BigInteger.ZERO )
        return BigInteger.ONE
    else
        return
            n.multiply(
                factorial( n.subtract( BigInteger.ONE ) ) );
}
```

What is Scala ?

A programming language ...

- **Pure Object Oriented**
- Functional
- Statically Typed
- Runs on the JVM

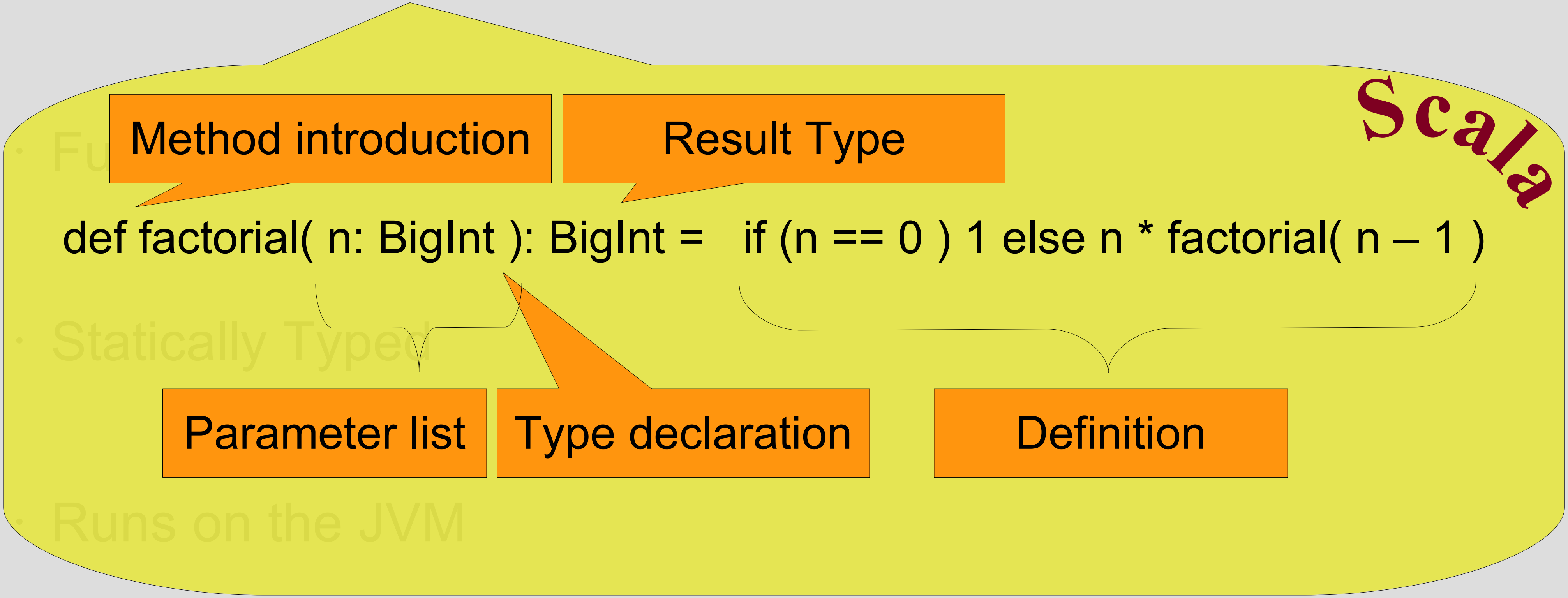
Scala

```
def factorial( n: BigInt ): BigInt =   if (n == 0 ) 1 else n * factorial( n – 1 )
```

What is Scala ?

A programming language ...

- **Pure Object Oriented**



Scala

Method introduction

Result Type

```
def factorial( n: BigInt ): BigInt =   if (n == 0 ) 1 else n * factorial( n – 1 )
```

Parameter list

Type declaration

Definition

- Fu
- Statically Typed
- Runs on the JVM

What is Scala ?

A programming language ...

- **Pure Object Oriented**
- Functional
- Statically Typed
- Runs on the JVM

Scala

```
def factorial( n: BigInt ): BigInt =   if (n == 0 ) 1 else n * factorial( n – 1 )
```

- (almost) everything is an expression

What is Scala ?

A programming language ...

- **Pure Object Oriented**
- Functional
- Statically Typed
- Runs on the JVM

Scala

```
def factorial( n: BigInt ): BigInt =   if (n == 0 ) 1 else n * factorial( n – 1 )
```

- (almost) everything is an expression
- **BigInt** 'integrates' like a Built-In type (but it's not!)

What is Scala ?

A programming language ...

- **Pure Object Oriented**

```
class BigInt( val bigInteger: BigInteger ) extends java.lang.Number{

  override def hashCode(): Int = this.bigInteger.hashCode()

  def +  (that: BigInt): BigInt =
                                  new BigInt( this.bigInteger.add( that.bigInteger ) )
...
}
```

- Functional
- Statically Typed
- Runs on the JVM

What is Scala ?

A programming language ...

- **Pure Object Oriented**

Introduction of class definition

```
class BigInt( val bigInteger: BigInteger ) extends java.lang.Number{

  override def hashCode(): Int = this.bigInteger.hashCode()

  def +  (that: BigInt): BigInt =
                                   new BigInt( this.bigInteger.add( that.bigInteger ) )
...
}
```

What is Scala ?

A programming language ...

- **Pure Object Oriented**

class value parameter
(primary constructor)

```
class BigInt( val bigInteger: BigInteger ) extends java.lang.Number{

  override def hashCode(): Int = this.bigInteger.hashCode()

  def +  (that: BigInt): BigInt =
                                new BigInt( this.bigInteger.add( that.bigInteger ) )
...
}
```

- Functional
- Statically Typed
- Runs on the JVM

What is Scala ?

A programming language ...

- **Pure Object Oriented**

Single Inheritance

```
class BigInt( val bigInteger: BigInteger ) extends java.lang.Number{

  override def hashCode(): Int = this.bigInteger.hashCode()

  def +  (that: BigInt): BigInt =
                                   new BigInt( this.bigInteger.add( that.bigInteger ) )
...
}
```

What is Scala ?

A programming language ...

- **Pure Object Oriented**

Mandatory !

```
class BigInt( val bigInteger: BigInteger ) extends java.lang.Number{

  override def hashCode(): Int = this.bigInteger.hashCode()

  def +  (that: BigInt): BigInt =
                                   new BigInt( this.bigInteger.add( that.bigInteger ) )
...
}
```

- Functional
- Statically Typed
- Runs on the JVM

What is Scala ?

A programming language ...

- **Pure Object Oriented**
- Functional
- Statically Typed
- Run

```
class BigInt( val bigInteger: BigInteger ) extends java.lang.Number{

  override def hashCode(): Int = this.bigInteger.hashCode()

  def +  (that: BigInt): BigInt =
                                   new BigInt( this.bigInteger.add( that.bigInteger ) )
...
}
```

ordinary Method

What is Scala ?

A programming language ...

- **Pure Object Oriented**

```
class BigInt( val bigInteger: BigInteger ) extends java.lang.Number{

  override def hashCode(): Int = this.bigInteger.hashCode()

  def +  (that: BigInt): BigInt =
                                  new BigInt( this.bigInteger.add( that.bigInteger ) )
...
}
```

Class Instantiation

What is Scala ?

A programming language ...

- **Pure Object Oriented**

```
class BigInt( val bigInteger: BigInteger ) extends java.lang.Number{

  override def hashCode(): Int = this.bigInteger.hashCode()

  def +  (that: BigInt): BigInt =
                                new BigInt( this.bigInteger.add( that.bigInteger ) )
...
}
```

Self reference

What is Scala ?

A programming language ...

- **Object Oriented**
- Functional
- Statically Typed
- Runs on the JVM

Example: Implement your own Type

- Stack of Int values
- Immutable (Functional style)

What is Scala ?

A programming language ...

- **Object Oriented**
- Functional
- Statically Typed
- Runs on the JVM

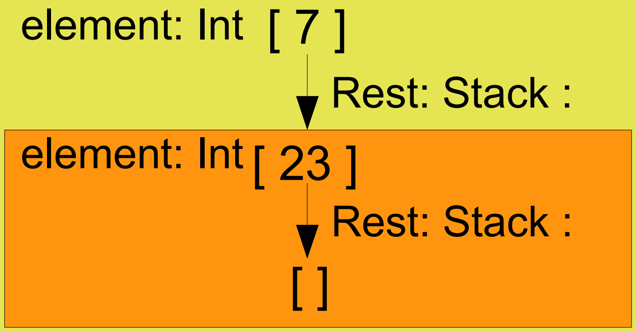
Example: Implement your own Type

- Stack of Int values
- Immutable (Functional style)

Empty Stack :

[ ]

Non Empty Stack :



element: Int [7]

Rest: Stack :

element: Int [23]

Rest: Stack :

[ ]

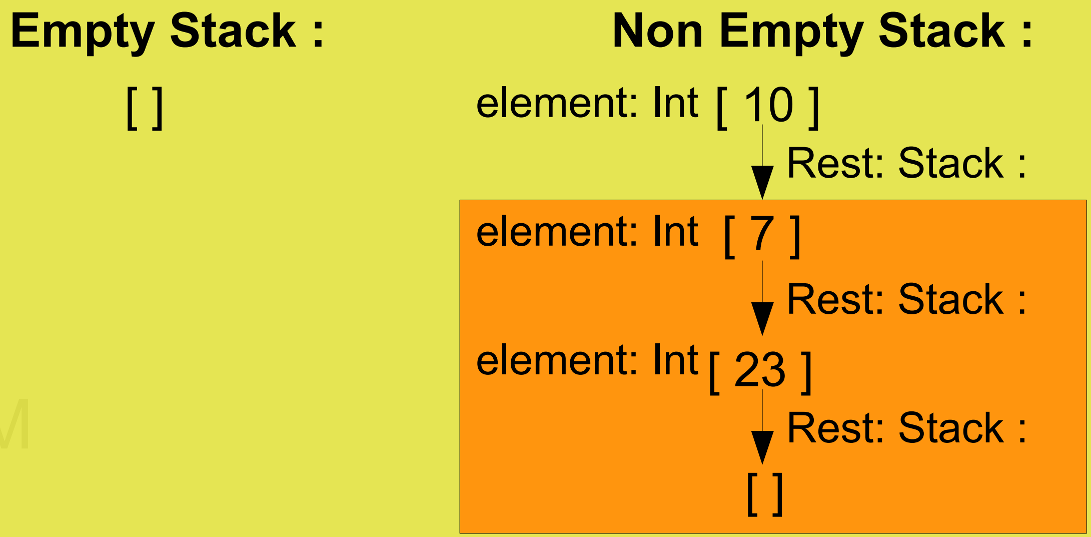
What is Scala ?

A programming language ...

- **Object Oriented**
- Functional
- Statically Typed
- Runs on the JVM

Example: Implement your own Type

- Stack of Int values
- Immutable (Functional style)



Empty Stack :

[ ]

Non Empty Stack :

element: Int [10]

Rest: Stack :

element: Int [7]

Rest: Stack :

element: Int [23]

Rest: Stack :

[ ]

What is Scala ?

A programming language ...

- **Object** Oriented
- Functional
- Statically Typed
- Runs on the JVM

Example: Implement your own Type

```
abstract class IntStack {
  def push(x: Int): IntStack = new NonEmptyIntStack( x, this )
  def isEmpty: Boolean
  def top: Int
  def pop: IntStack
}
```

What is Scala ?

A programming language ...

- **Object Orien**
- Functional
- Statically Typed
- Runs on the JVM

Example: Implement your own Type

Abstract class

```
abstract class IntStack {

  def push(x: Int): IntStack = new NonEmptyIntStack( x, this )

  def isEmpty: Boolean

  def top: Int

  def pop: IntStack
}
```

Abstract method

Abstract method

Abstract method

What is Scala ?

A programming language ...

- **Object Oriented**
- Functional
- Statically Typed
- Runs on the JVM

Example: Implement your own Type

```
class EmptyIntStack extends IntStack {
  def isEmpty = true
  def top = throw new EmptyStackException
  def pop = throw new EmptyStackException
}
```

What is Scala ?

A programming language ...

- **Object** Oriented
- Functional
- Statically Typed
- Runs on the JVM

Example: Implement your own Type

Inheritance

```
class EmptyIntStack extends IntStack {
  def isEmpty = true
  def top = throw new EmptyStackException
  def pop = throw new EmptyStackException
}
```

Throwing a 'checked' Exception

... but catching is optional

What is Scala ?

A programming language ...

- **Object Oriented**
- Functional
- Statically Typed
- Runs on the JVM

Example: Implement your own Type

```
class EmptyIntStack extends IntStack {
  def isEmpty = true
  def top = throw new EmptyStackException
  def pop = throw new EmptyStackException
}
    val zeroInts = new EmptyIntStack
    var noInts = new EmptyIntStack
```

What is Scala ?

A programming language ...

- **Object Oriented**
- Functional
- Statically Typed
- Runs on the JVM

Example: Implement your own Type

```
class EmptyIntStack extends IntStack {
  def isEmpty = true
  def top = throw new EmptyStackException
  def pop = throw new EmptyStackException
}
```

```
    val zeroInts = new EmptyIntStack
    var noInts = new EmptyIntStack
```

'Immutable' **val**ue

'mutable' **var**iable

What is Scala ?

A programming language ...

- **Object Oriented**
- Functional
- Statically Typed
- Runs on the JVM

Example: Implement your own Type

```
class EmptyIntStack extends IntStack {
  def isEmpty = true
  def top = throw new EmptyStackException
  def pop = throw new EmptyStackExce
}
    val zeroInts = new EmptyIntStack
    var noInts = new EmptyIntStack
```

No need for multiple Instances of EmptyIntStack

What is Scala ?

A programming language ...

- **Object Oriented**
- Functional
- Statically Typed
- Runs on the JVM

Example: Implement your own Type

```
object EmptyIntStack extends IntStack {
  def isEmpty = true
  def top = throw new EmptyStackException
  def pop = throw new EmptyStackException
}
```

- **Singleton Object**

What is Scala ?

A programming language ...

- **Object Oriented**
- Functional
- Statically Typed
- Runs on the JVM

Example: Implement your own Type

```
object EmptyIntStack extends IntStack {
  def isEmpty = true
  def top = throw new EmptyStackException
  def pop = throw new EmptyStackException
}
```

val zeroInts = ~~new~~ EmptyIntStack

var noInts = ~~new~~ EmptyIntStack

What is Scala ?

A programming language ...

- **Object** Oriented
- Functional
- Statically Typed
- Runs on the JVM

Example: Implement your own Type

```
class NonEmptyIntStack( elem: Int, rest: IntStack )
    extends IntStack {

  def isEmpty = false

  def top = elem

  def pop = rest
}
```

What is Scala ?

A programming language ...

- **Object Oriented**
- Functional
- Statically Typed
- Runs on the JVM

Example: Implement your own Type

```
class NonEmptyIntStack( elem: Int, rest: IntStack )
   extends IntStack {

  def isEmpty = false

  def top = elem

  def pop = rest
}
```

var s = EmptyIntStack push 23 push 7 push 10

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

- Using Stack with other types than Int
- Concept of stacking elements is generic

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

- Using Stack with other types than Int
- Concept of stacking elements is generic

=> Generic Types

(Type Parameterization)

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

Type Paramter

```
abstract class Stack[A] {

  def push(x: A): Stack[A] =
                     new NonEmptyStack[A]( x, this )

  def isEmpty: Boolean

  def top: A

  def pop: Stack[A]
}
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

```
class NonEmptyStack[A](
   elem: A,
   rest: Stack[A]  ) extends Stack[A] {

   def isEmpty = false

   def top = elem

   def pop = rest
}
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

```
val s = new EmptyStack[Int]

val t = s push 1 push 2 push 3

t.pop.top                  => 2
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

```
val s = new EmptyStack[Int]

val t = s push 1 push   ush 3

t.pop.top
```

Parameterized Type

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

```
val s = new EmptyStack[Int]

val t = s push 1 push 2 push 3

t.pop.top
```

Mandatory
(no Raw Types !)

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

- A Stack implementation only for numbers

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

- A Stack implementation only for numbers
- Restrict the upper Type to Number

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

- A Stack implementation only for numbers
- Restrict the upper Type to Number
- **Upper Type Bound**

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

- A Stack implementation only for numbers
- Restrict the upper Type to Number
- **Upper Type Bound**

abstract class Stack[**A <: Number**]{ ... }

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

- A Stack implementation only for numbers
- Restrict the upper Type to Number
- **Upper Type Bound**

```
abstract class Stack[A <: Number]{ ... }
```

Defined by the implementor,
not by the user !

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

Is **Stack[String]** a super type of **Stack[Any]** ?

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

Is **Stack[String]** a super type of **Stack[Any]** ?

Java

```
List<String> sList = new ArrayList<String>();

List<Object> oList = sList;
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

Is **Stack[String]** a super type of **Stack[Any]** ?

Java

```
List<String> sList = new ArrayList<String>();

List<Object> oList = sList;
```

Compile Error:
"Type mismatch: cannot convert List<String> to List<Object>"

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

Is **Stack[String]** a super type of **Stack[Any]** ?

Java

```
List<String> sList = new ArrayList<String>();

List<Object> oList = sList;
```

Generic Types are

INVARIANT

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

but ...

Java

```
String[] sArray = new String[]{};

Object[] oArray = sArray;
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

but ...

Java

```
String[] sArray = new String[]{};

Object[] oArray = sArray;
```

Arrays are

COVARIANT

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

but ...

Java

```
String[] sArray = new String[]{};

Object[] oArray = sArray;

oArray[0] = BigDecimal.ONE;
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

but ...

Java

```
String[] sArray = new String[]{};

Object[] oArray = sArray;

oArray[0] = BigDecimal.ONE;
```

ArrayStoreException
at Runtime !

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

Scala

```
abstract class Stack[+A]{

  def push(x: A): Stack[A] =
                    new NonEmptyStack[A]( x, this )
    ...
}
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

Scala

Covariant subtyping

```
abstract class Stack[+A]{

  def push(x: A): Stack[A] =
                    new NonEmptyStack[A]( x, this )
   ...
}
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

Scala

Covariant subtyping

```
abstract class Stack[+A]{

 def push(x: A): Stack[A] =
                    new NonEmptyStack[A]( x, this )
  ...
}
```

Compile Error:

"covariant type A occurs in contravariant position in type A of value x"

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

Scala

```
abstract class Stack[+A]{

  def push[B >: A](x: B): Stack[B] =
                      new NonEmptyStack[B](x, this)
   ...
}
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

Scala

```
abstract class Stack[+A]{
  def push[B >: A](x: B): Stack[B] =
                 new NonEmptyStack[B](x, this)
   ...
}
```

Lower Type Bound

"Parameter **B** is restricted to range only over **supertypes** of type **A**"

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

Scala

```
abstract class Stack[+A]{

  def push[B >: A](x: B): Stack[B] =
                      new NonEmptyStack[B](x, this)
   ...
}
```

```
val s1 = new EmptyStack[Int] push 1 push 2

val s2 = s1 push "x"
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Refined Type System

Scala

```
abstract class Stack[+A]{

  def push[B >: A](x: B): Stack[B] =
                    new NonEmptyStack[B](x, this)
   ...
}

val s1 = new EmptyStack[Int] push 1 push 2

val s2 = s1 push "x"
```

Stack[Int]

Stack[Any]

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Type inference

```
val  creator: String = “Odersky“
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Type inference

```
val  creator: String = “Odersky“

val creator = “Odersky“
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Type inference

```
val  creator: String = “Odersky“

val creator = “Odersky“



def add( a: Int, b: Int ): Int =  a + b
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Type inference

```
val  creator: String = “Odersky“

val creator = “Odersky“



def add( a: Int, b: Int ): Int =  a + b

def add( a: Int, b: Int )  =  a + b
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Implicit Type conversion

“Scalable Language“ - 'a'

=> Sc l ble L ngu ge

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Implicit Type conversion

```
“Scalable Language“ - 'a'


class StringExtension( s: String ){
  def -( sub: Char ) =   s.replace( sub, ' ' )
}

new StringExtension( “Scala“ ).-( 'a' )
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Implicit Type conversion

```
“Scalable Language“ - 'a'


class StringExtension( s: String ){
  def -( sub: Char ) =   s.replace( sub, ' ' )
}

implicit def toStringExtension( s: String ) =
    new StringExtension( s )
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- **Statically Typed**
- Functional
- Runs on the JVM

Implicit Type conversion

“Scalable Language“ - 'a'

Implicit conversion to StringExtension

```
class StringExtension( s: String ){
  def -( sub: Char ) =   s.replace( sub, ' ' )
}

implicit def toStringExtension( s: String ) =
    new StringExtension( s )
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

- Lambda Calculus (A. Church)
- Functions are first class values

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

- Lambda Calculus (A. Church)
- Functions are first class values

Function Literals

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

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

- Lambda Calculus (A. Church)
- Functions are first class values

Function Literals

(x: Int) => x + 1 => λ x . x + 1

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

- Lambda Calculus (A. Church)
- Functions are first class values

Function Literals

$$\underbrace{(\ x\text{: Int}\ )}_{\text{Argument list}}\ \Rightarrow\ \underbrace{x + 1}_{\text{Definition}} \qquad \Rightarrow\ \lambda\, x\ .\ x + 1$$

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

- Lambda Calculus (A. Church)
- Functions are first class values

Function Literals

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

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

- Lambda Calculus (A. Church)
- Functions are first class values

Function Literals

```
( x: Int )  =>  x + 1        =>  λ x . x + 1
 val succ =  ( x: Int )  =>  x + 1
 succ( 7 )                   => 8
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

- Lambda Calculus (A. Church)
- Functions are first class values

Function Literals

```
( x: Int )  =>  x + 1          =>  λ x . x + 1
 val succ =  ( x: Int )  =>  x + 1
 succ( 7 )                     => 8
```

type of succ: (Int) => Int

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

Closures

```
val ages = List( 2, 20, 14, 19, 49, 11, 62 )

var barrier = 18

val minors = {  ( x :Int )  =>   x < barrier  }

val germanMinors = ages.filter( minors )


=> List( 2, 14, 11 )
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

Closures

```
val ages = List( 2, 20, 14, 19, 49, 11, 62 )
var barrier = 18
val minors = {  ( x :Int )  =>   x < barrier  }
val germanMinors = ages.filter( minors )
```

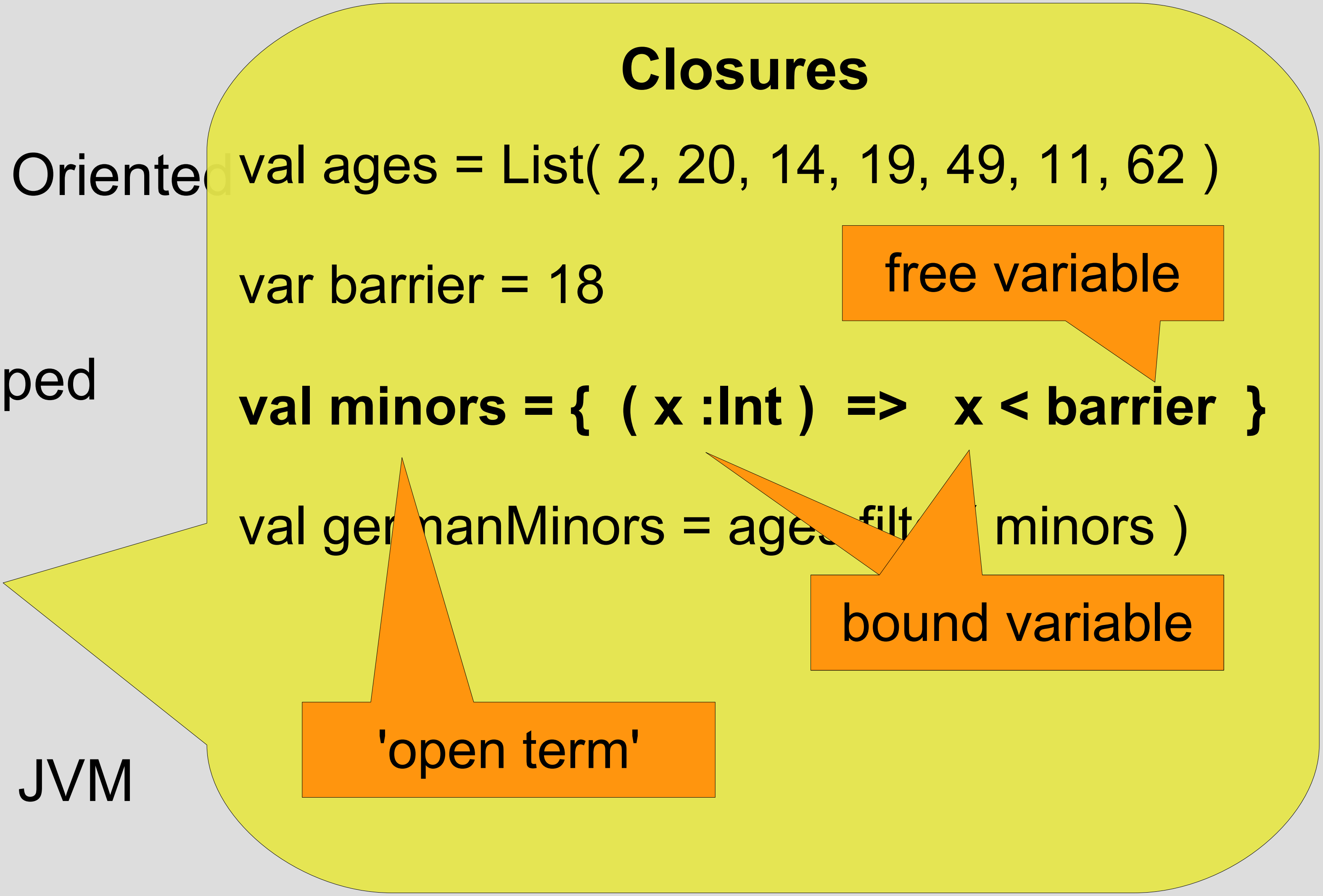
List[Int]   (Type inference)

... accepting a function which
accepts an Int and results to boolean

What is Scala ?

A programming language ...

- Pure Object Oriente
- Statically Typed
- **Functional**
- Runs on the JVM



Closures

```
val ages = List( 2, 20, 14, 19, 49, 11, 62 )

var barrier = 18

val minors = {  ( x :Int )  =>   x < barrier  }

val ger nanMinors = age  filt   minors )
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

Closures

```
val ages = List( 2, 20, 14, 19, 49, 11, 62 )

var barrier = 18

val minors = {  ( x :Int )  =>   x < barrier  }

val germanMinors = ages.filter( minors )
```

capturing

- **bound within lexical scope of function**

=> open term is closed

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

Closures

```
val ages = List( 2, 20, 14, 19, 49, 11, 62 )

var barrier = 18

val minors = {  ( x :Int )  =>   x < barrier  }

val germanMinors = ages.filter( minors )

barrier = 21

val usMinors = ages.filter( minors )
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

Closures

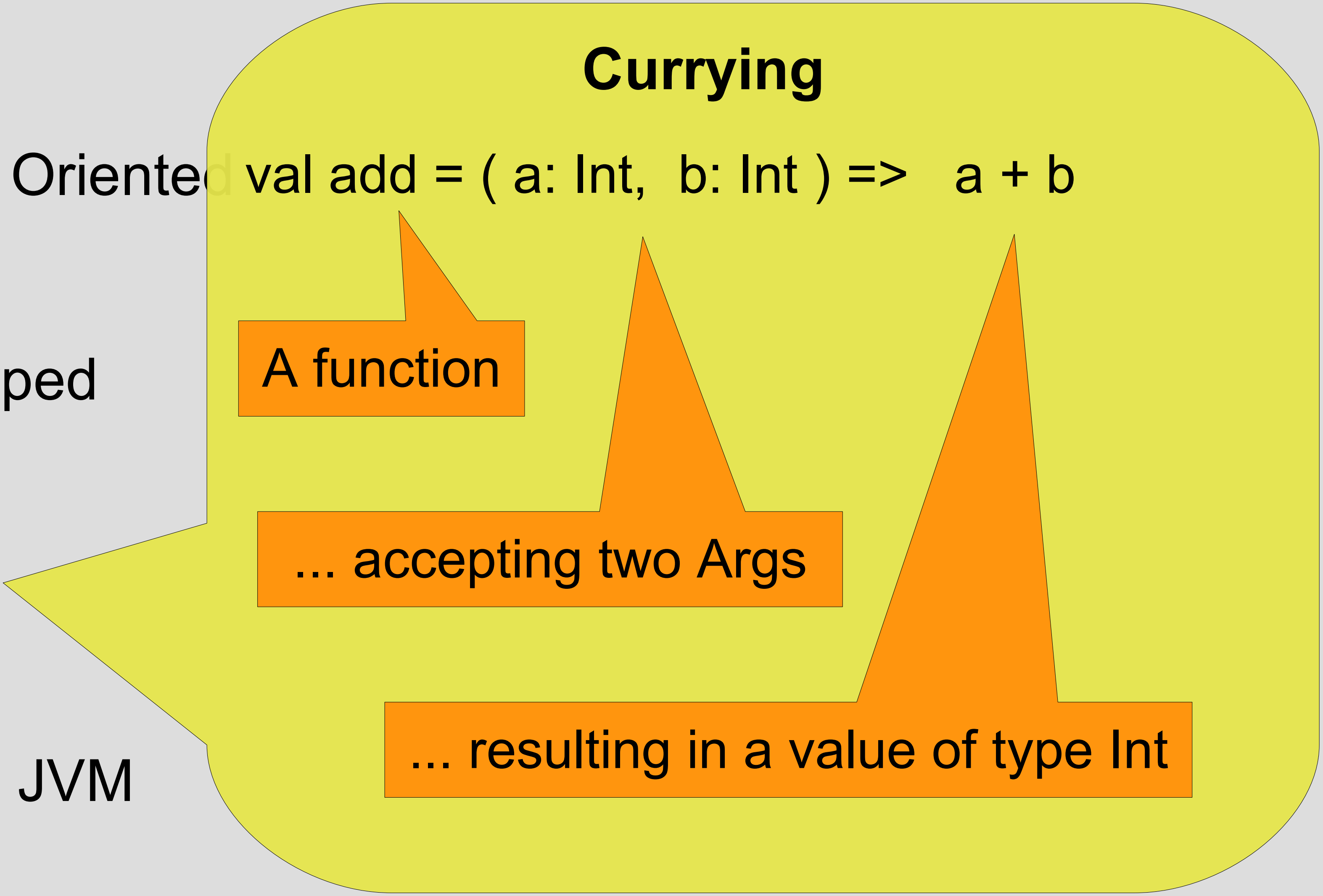
```
val ages = List( 2, 20, 14, 19, 49, 11, 62 )

var barrier = 18

val minors = {  ( x :Int )  =>   x < barrier  }

val german

barrier = 21

val usMinors = ages.filter( minors )

     => 2, 20, 14, 19, 11
```

'dynamic' bound

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM



Currying

```
val add = ( a: Int,  b: Int ) =>   a + b
```

A function

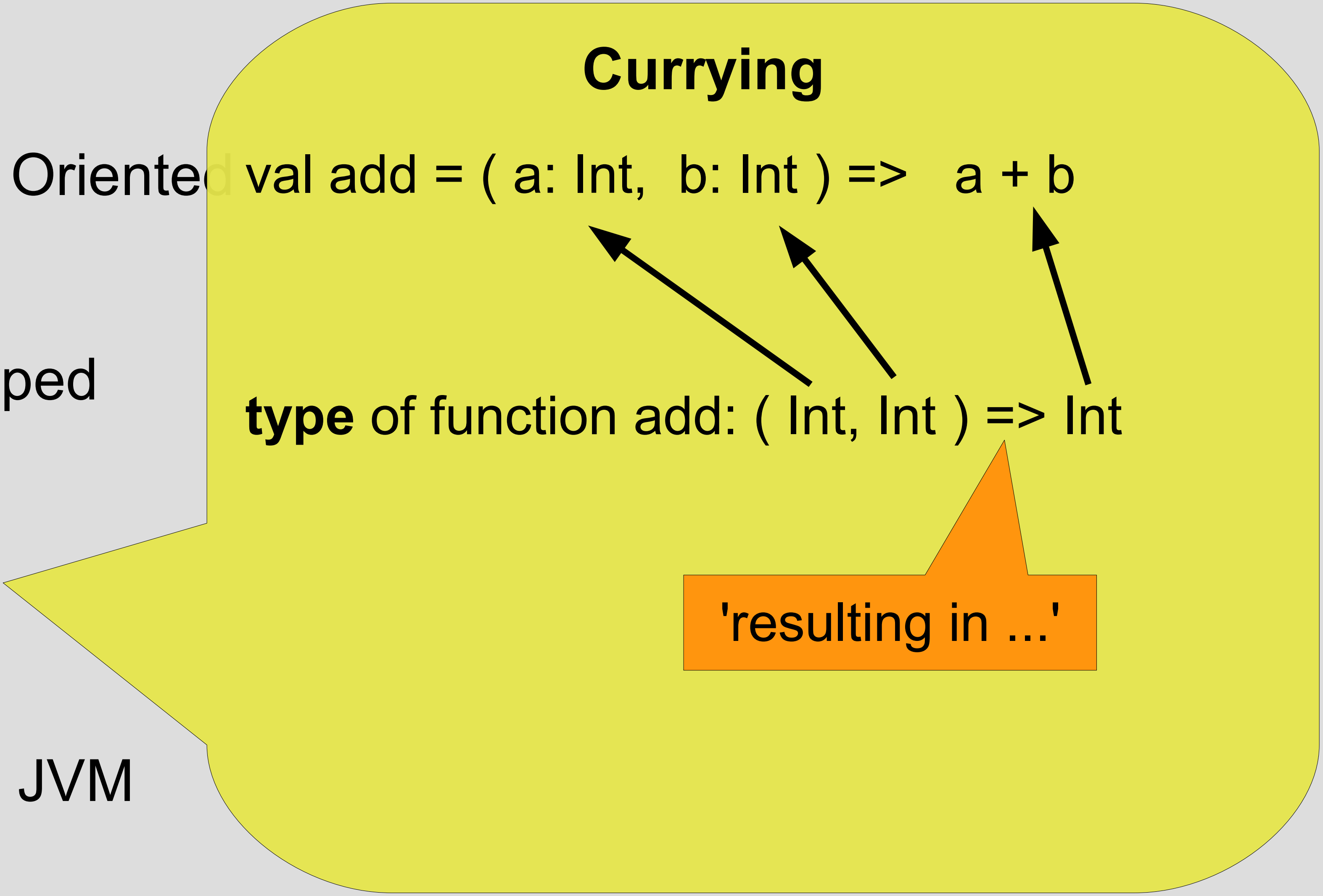
... accepting two Args

... resulting in a value of type Int

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM



Currying

```
val add = ( a: Int,  b: Int ) =>   a + b
```

type of function add: (Int, Int) => Int

'resulting in ...'

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

Currying

```
val add = ( a: Int,  b: Int ) =>   a + b
```

Quiz:

"transform into a function which is accepting only one single Argument after another"

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

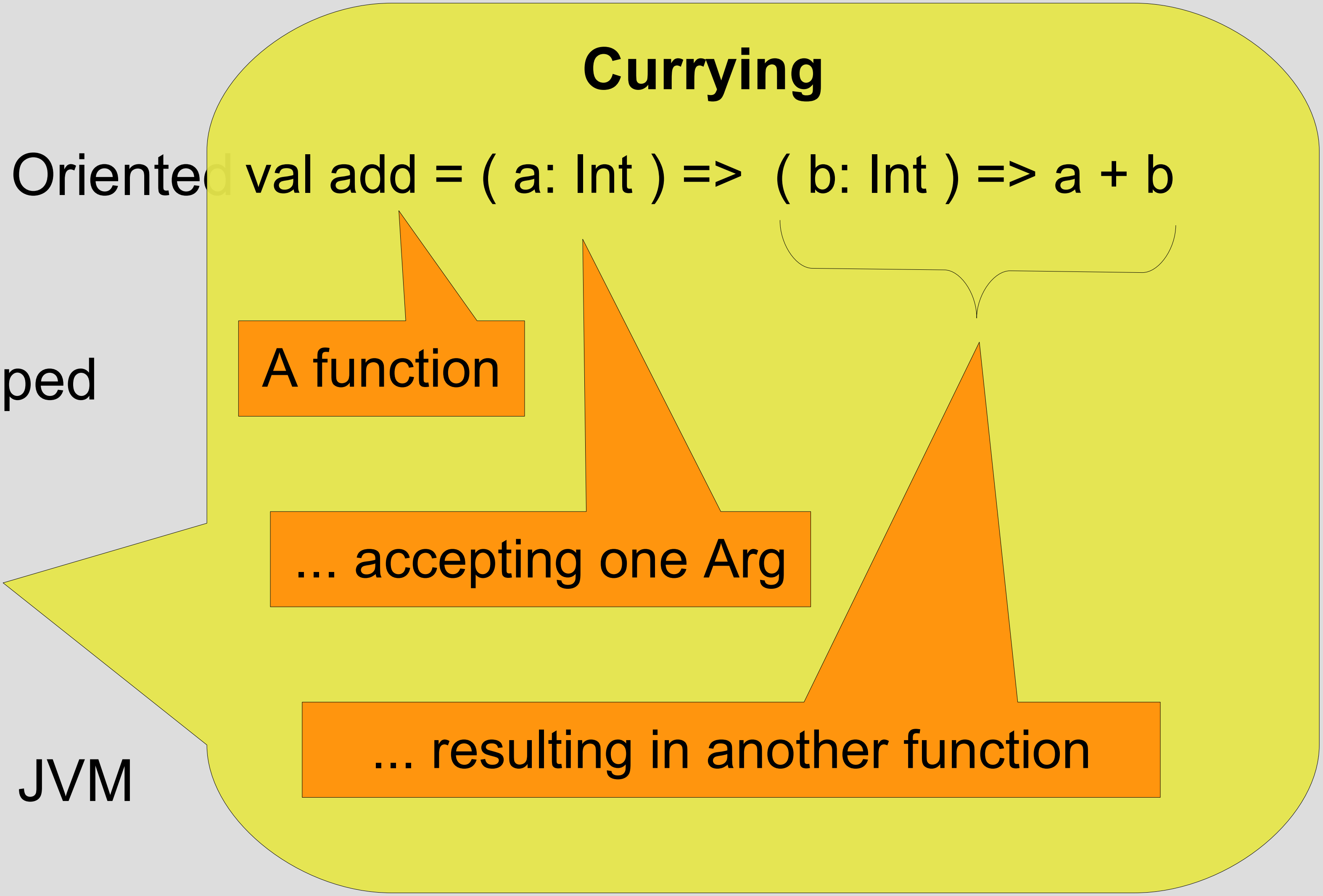
Currying

```
val add = ( a: Int ) =>  ( b: Int ) => a + b
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM



Currying

```
val add = ( a: Int ) =>  ( b: Int ) => a + b
```

A function

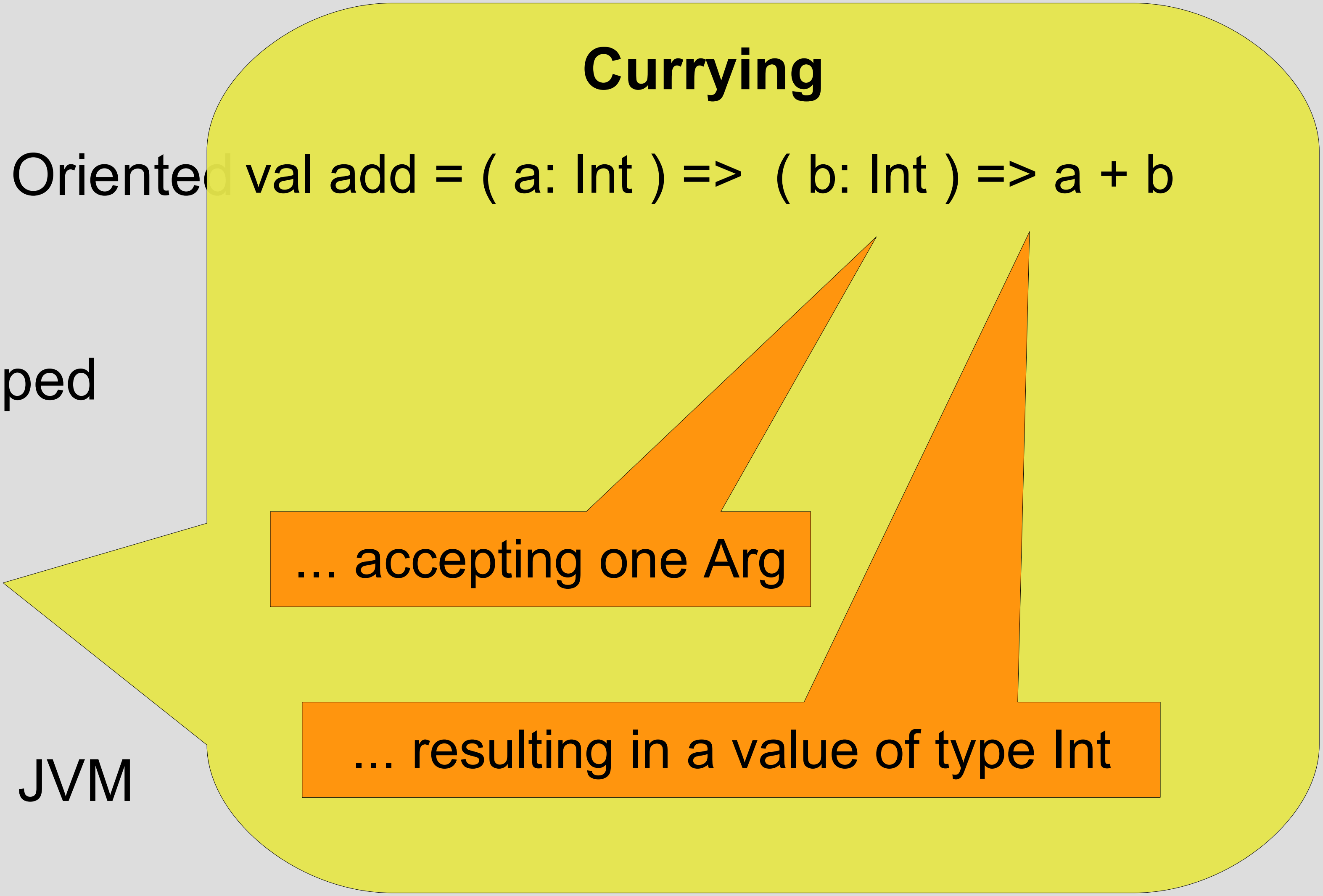
... accepting one Arg

... resulting in another function

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM



Currying

```
val add = ( a: Int ) =>  ( b: Int ) => a + b
```

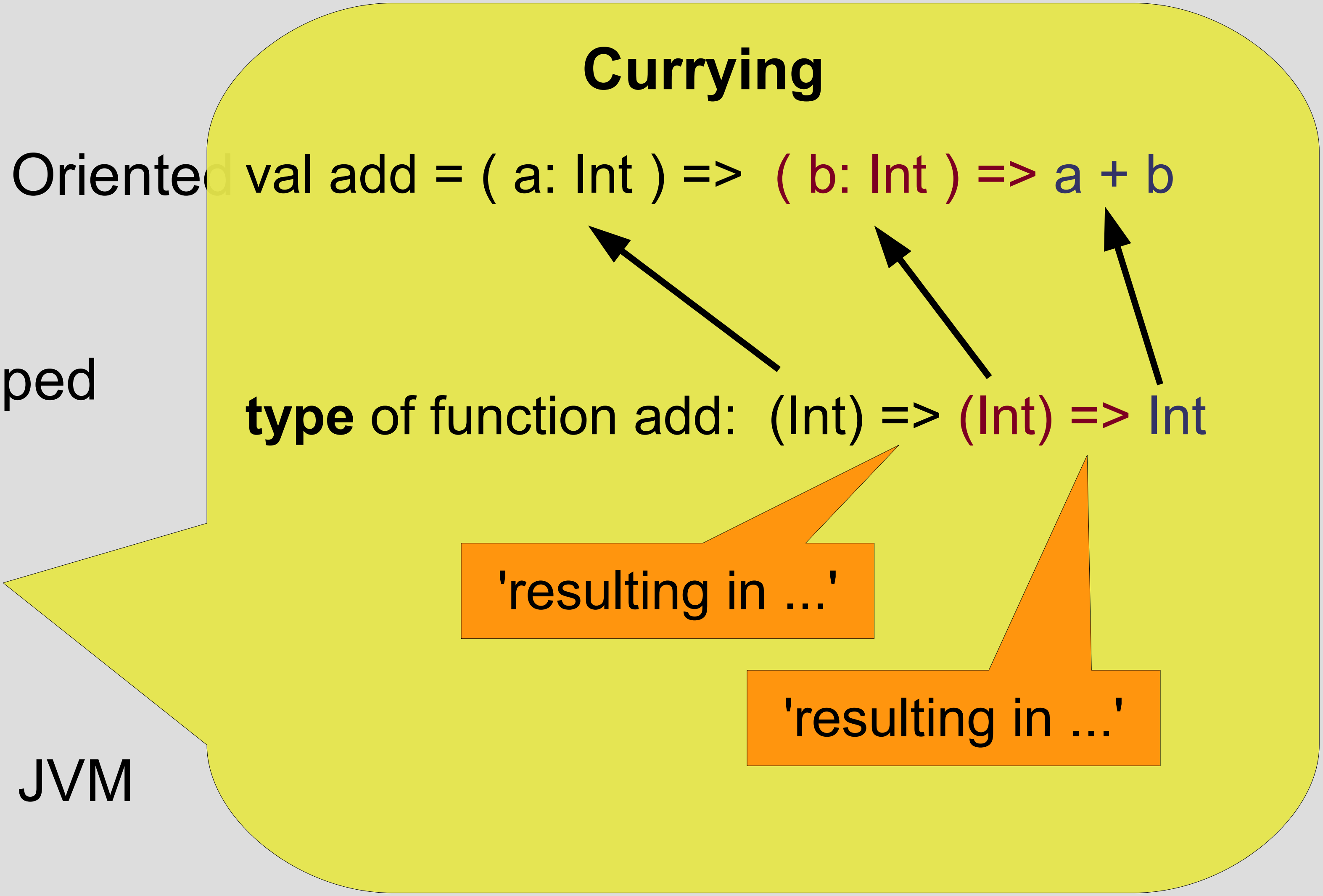
... accepting one Arg

... resulting in a value of type Int

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM



Currying

```
val add = ( a: Int ) =>  ( b: Int ) => a + b
```

type of function add: (Int) => (Int) => Int

'resulting in ...'

'resulting in ...'

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

Currying

```
val add = ( a: Int ) =>  ( b: Int ) => a + b


val succ = add( 1 )


succ( 7 )              => 8
```

What is Scala ?

A programming language ...

- Pure Object Oriente
- Statically Typed
- **Functional**
- Runs on the JVM

Currying

```
val add = ( a: Int ) =>  ( b: Int ) => a + b
```

A function ...

```
val succ = add( 1 )
```

... accepting one Arg

... resulting in another function

```
succ( 7 )              => 8
```

... accepting one Arg

... resulting in a value of type Int

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

Curried Methods

```
def mult( a: Int )( b: Int ) = a * b
```

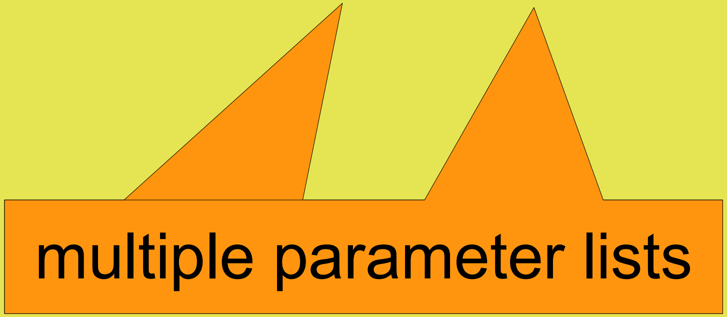
What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

Curried Methods

```
def mult( a: Int )( b: Int ) = a * b
```



multiple parameter lists

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

Curried Methods

```
def mult( a: Int )( b: Int ) = a * b
```

multiple parameter lists

Signature: mult (Int) (Int) : Int

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

Curried Methods

```
def mult( a: Int )( b: Int ) = a * b

val double =  mult( 2 ) _
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

Curried Methods

```
def mult( a: Int )( b: Int ) = a * b

val double =  mult( 2 ) _
```

Partially applied

2nd Arg unapplied

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

Curried Methods

```
def mult( a: Int )( b: Int ) = a * b

val double =  mult( 2 ) _
```

Coercion into a **function** of type **(Int) => Int**

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

Curried Methods

```
def mult( a: Int )( b: Int ) = a * b

val double =  mult( 2 ) _


double( 6 )         => 12
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

Curried Methods

```
val hours = ( 0 to 23 ).toList

def modulo( n: Int )( x: Int ) =  ( x % n ) == 0

hours.filter( modulo( 2 ) _ )
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

Curried Methods

```
val hours = ( 0 to 23 ).toList

def modulo( n: Int )( x: Int ) =  ( x % n ) == 0

hours.filter( modulo( 2 ) _ )
```

List[Int]

expects function of type (Int) => Boolean

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

Curried Methods

```
val hours = ( 0 to 23 ).toList

def modulo( n: Int )( x: Int ) =  ( x % n ) == 0

hours.filter( modulo( 2 ) _ )
```

curried to function of type (Int) => Boolean

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

Curried Methods

```
val hours = ( 0 to 23 ).toList

def modulo( n: Int )( x: Int ) =  ( x % n ) == 0

hours.filter( modulo( 2 ) _ )

=> List( 0, 2, 4, 6, 8, 10, 12, …, 20, 22 )
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- **Functional**
- Runs on the JVM

Curried Methods

```
val hours = ( 0 to 23 ).toList

def modulo( n: Int )( x: Int ) =  ( x % n ) == 0

hours.filter( modulo( 2 ) _ )

=> List( 0, 2, 4, 6, 8, 10, 12, …, 20, 22 )

hours.filter( modulo( 4 )  )

=> List( 0, 2, 4, 8, 12, …, 16, 20 )
```

What is Scala ?

A programming language ...

- **Pure Object Oriented**
- Statically Typed
- **Functional**
- Runs on the JVM

OO + FP Fusion

What is Scala ?

A programming language ...

- **Pure Object Oriented**
- Statically Typed
- **Functional**
- Runs on the JVM

OO + FP Fusion

- Everything is an Object

What is Scala ?

A programming language ...

- **Pure Object Oriented**
- Statically Typed
- **Functional**
- Runs on the JVM

OO + FP Fusion

- Everything is an Object
- Functions are Objects

What is Scala ?

A programming language ...

- **Pure Object Oriented**
- Statically Typed
- **Functional**
- Runs on the JVM

OO + FP Fusion

- Everything is an Object
- Functions are Objects

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

What is Scala ?

A programming language ...

- **Pure Object Oriented**
- Statically Typed
- **Functional**
- Runs on the JVM

OO + FP Fusion

- Everything is an Object
- Functions are Objects

```
val succ = new Function1[Int, Int]{
  override def apply( x: Int ) = x + 1
}
```

What is Scala ?

A programming language ...

- **Pure Object Oriented**
- Statically Typed
- **Functional**
- Runs on the JVM

OO + FP Fusion

- Everything is an Object
- Functions are Objects

```
val succ = new Function1[Int, Int]{
  override def apply( x: Int ) = x + 1
}

succ( 7 )
```

What is Scala ?

A programming language ...

- **Pure Object Oriented**
- Statically Typed
- **Functional**
- Runs on the JVM

OO + FP Fusion

- Everything is an Object
- Functions are Objects

```
val succ = new Function1[Int, Int]{
  override def apply( x: Int ) = x + 1
}

succ.apply( 7 )
```

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- Functional
- **Runs on the JVM**

What is Scala ?

A programming language ...

... so does Groovy, Clojure, JRuby ...

- Pure Object Oriented
- Statically Typed
- Functional
- **Runs on the JVM**

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- Functional
- **Runs on the JVM**

... so does Groovy, Clojure, JRuby ...

- Dynamically typed (MOP & Co)
- Significant Performance Overhead !

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- Functional
- **Runs on the JVM**

... so does Groovy, Clojure, JRuby ...

- Dynamically typed (MOP & Co)
- Significant Performance Overhead !
- **Scala is statically typed !**

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- Functional
- **Runs on the JVM**

... so does Groovy, Clojure, JRuby ...

- Dynamically typed (MOP & Co)
- Significant Performance Overhead !

- **Scala is statically typed !**
- Compiles to Bytecode
- Seamless Java Interoperability

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- Functional
- **Runs on the JVM**

... so does Groovy, Clojure, JRuby ...

- Dynamically typed (MOP & Co)
- Significant Performance Overhead !

- **Scala is statically typed !**
- Compiles to Bytecode
- Seamless Java Interoperability
- Performance on par with Java

What is Scala ?

A programming language ...

- Pure Object Oriented
- Statically Typed
- Functional
- **Runs on the JVM**

> “I can honestly say if someone had shown me the **Programming in Scala** book ... back in 2003 I'd probably have never created Groovy“
>
> James Strachan

Characteristics

- Expressive
- High Level
- Concise
- Extensible
- Pragmatic

Characteristics

- **Expressive**
- High Level
- Concise
- Extensible
- Pragmatic

```
case class Person( name: String, age: Int )

var persons = List( Person( "Hans", 11 ),
                    Person( "Hugo", 19 ),
                    Person( "Helga", 16 ),
                    Person( "Heinz", 38 )  )

val (adults, minors) = persons.partition( _.age > 18 )
```

Can you spot the intention ?

Characteristics

- **Expressive**
- High Level
- Concise
- Extensible
- Pragmatic

```
case class Person( name: String, age: Int )

var persons = List( Person( "Hans", 11 ),
                    Person( "Hugo", 19 ),
                    Person( "Helga", 16 ),
                    Person( "Heinz", 38 )  )

val (adults, minors) = persons.partition( _.age > 18 )
```

"Split Persons into minors and adults by their age"

Characteristics

- **Expressive**
- High Level
- Concise
- Extensible
- Pragmatic

```
case class Person( name: String, age: Int )

var persons = List( Person( "Hans", 11 ),
                    Person( "Hugo", 19 ),
```

Results into a Tuple2[List[Person],List[Person]]

```
val (adults, minors) = persons.partition( _.age > 18 )
```

"Split Persons into minors and adults by their age"

Characteristics

- **Expressive**
- High Level
- Concise
- Extensible
- Pragmatic

```
case class Person( name: String, age: Int )

var persons = List( Person( "Hans", 11 ),
                    Person( "Hugo", 19 ),
```

Results into a Tuple2[List[Person],List[Person]]

```
val (adults, minors) = persons.partition( _.age > 18 )
```

Pattern Matching:
bound to single elements of a Tuple2

Characteristics

- **Expressive**
- High Level
- Concise
- Extensible
- Pragmatic

```
case class Person( name: String, age: Int )

var persons = List( Person( "Hans", 11 ),
                    Person( "Hugo", 19 ),
                    Person( "Helga", 16 ),
                    Person( "Heinz", 38 )  )

val (adults, minors) = persons.partition( _.age > 18 )
```

"Split Persons into minors and adults by their age"

```
adults      => List(Person(Hugo,19), Person(Heinz,38))
minors      => List(Person(Hans,11), Person(Helga,16))
```

Characteristics

- **Expressive**
- High Level
- Concise
- Extensible
- Pragmatic

```
val bookPrices = Map(
            “Prag. Programmer“ ->  20 USD,
            “Systems Thinking“  ->  30 EUR,
            “Code Complete“      ->  25 USD )

bookPrices += ( “Clean Code“ ->  20 EUR )

println(  bookPrices( “Systems Thinking“ )  )

for(  (book, price)  <-  bookPrices ){
    if( price in EUR )  println( book )
}
```

Characteristics

- **Expressive**
- High Level
- Concise
- Extensible
- Pragmatic

Create a new Map

```
val bookPrices = Map(
                "Prag. Programmer“ ->  20 USD,
                “Systems Thinking“  ->  30 EUR,
                “Code Complete“     ->  25 USD )

bookPrices += ( “Clean Code“ ->  20 EUR )

println(  bookPrices( “Systems Thinking“ )  )

for(  (book, price)  <-  bookPrices ){
     if( price in EUR )  println( book )
}
```

Characteristics

- **Expressive**
- High Level
- Concise
- Extensible
- Pragmatic

```
val bookPrices = Map(
                  “Prag. Programmer“ ->  20 USD,
                  “Systems Thinking“  ->  30 EUR,
                  “Code Complete“       ->  25 USD )

bookPrices += ( “Clean Code“ ->  20 EUR )


println(  bookPrices( “Systems Thinking“ )  )


for(  (book, price)  <-  bookPrices ){
     if( price in EUR )  println( book )
}
```

Create a new Map

Implicit Conversion into a Tuple2

Characteristics

- **Expressive**
- High Level
- Concise
- Extensible
- Pragmatic

```
val bookPrices = Map(
                    “Prag. Programmer“ ->  20 USD,
                    “Systems Thinking“  ->  30 EUR,
                    “Code Complete“      ->  25 USD )

bookPrices += ( “Clean Code“ ->  20 EUR )


println(  bookPrices( “Systems Thinking“ )  )


for(  (book, price)  <-  bookPrices ){
      if( price in EUR )  println( book )
}
```

Create a new Map

Implicit Conversion into a Currency

Characteristics

- **Expressive**
- High Level
- Concise
- Extensible
- Pragmatic

```
val bookPrices = Map(
```

Compagnion Object

```
object Map{
 def apply[A, B]( elems: (A, B)* ) : Map[A, B] = ...
 ...
}
```

```
println(  bookPrices( “Systems Thinking“ )  )

for(  (book, price)  <-  bookPrices ){
     if( price in EUR )  println( book )
}
```

Characteristics

- **Expressive**
- High Level
- Concise
- Extensible
- Pragmatic

```
val bookPrices = Map(
```

Map instance

```
class <xxx>Map[A, B]  extends  Map[A, B] {
  override def apply(key: A): B = ...
  ...
}
```

```
println(  bookPrices( “Systems Thinking“ )  )

for(  (book, price)  <-  bookPrices ){
     if( price in EUR )  println( book )
}
```

Characteristics

- Expressive
- **High Level**
- Concise
- Extensible
- Pragmatic

Characteristics

- Expressive
- **High Level**
- Concise
- Extensible
- Pragmatic

Upper Case in given name ? Java

```
boolean hasUpperCase = false;
for( int i=0; i < name.length; i++ ){
    if( Character.isUpperCase( name.charAt( i ) {
        hasUpperCase = true;
        break;
    }
}
```

Characteristics

- Expressive
- **High Level**
- Concise
- Extensible
- Pragmatic

Upper Case in given name ?

Scala

```
val hasUpperCase =
        name.exists( c: Char => c.isUpperCase )
```

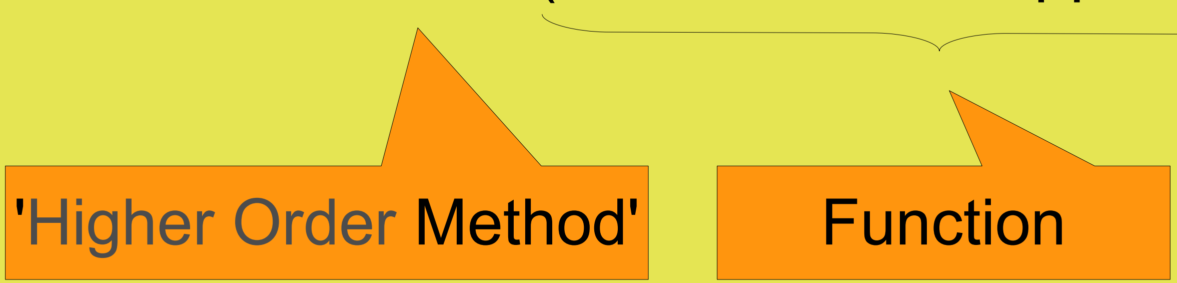
Characteristics

- Expressive
- **High Level**
- Concise
- Extensible
- Pragmatic

Scala

Upper Case in given name ?

```
val hasUpperCase =
        name.exists( c: Char => c.isUpperCase )
```



'Higher Order Method'

Function

Characteristics

- Expressive
- **High Level**
- Concise
- Extensible
- Pragmatic

Upper Case in given name ?

Scala

```
val hasUpperCase =
        name.exists( c: Char => c.isUpperCase )
```

'Higher Order Method'

Function

```
val hasUpperCase =
        name.exists( c => c isUpperCase )
```

Type inference

Characteristics

- Expressive
- **High Level**
- Concise
- Extensible
- Pragmatic

Upper Case in given name ?

Scala

```
val hasUpperCase =
        name.exists( c: Char => c.isUpperCase )
```

'Higher Order Method'

Function

```
val hasUpperCase =
        name.exists( _ isUpperCase )
```

parameter shortcut

Characteristics

- Expressive
- **High Level**
- Concise
- Extensible
- Pragmatic

Find maximal Distance

Java

```
List<Integer> distances =
                        new ArrayList<Integer>();

distances.add( 12 );

distances.add( 17 ); ...


Integer maxDistance = 0;

for( Integer distance : distances ){

    if( distance > maxDistance ){

        maxDistance = distance
    }
}
```

Characteristics

- Expressive
- **High Level**
- Concise
- Extensible
- Pragmatic

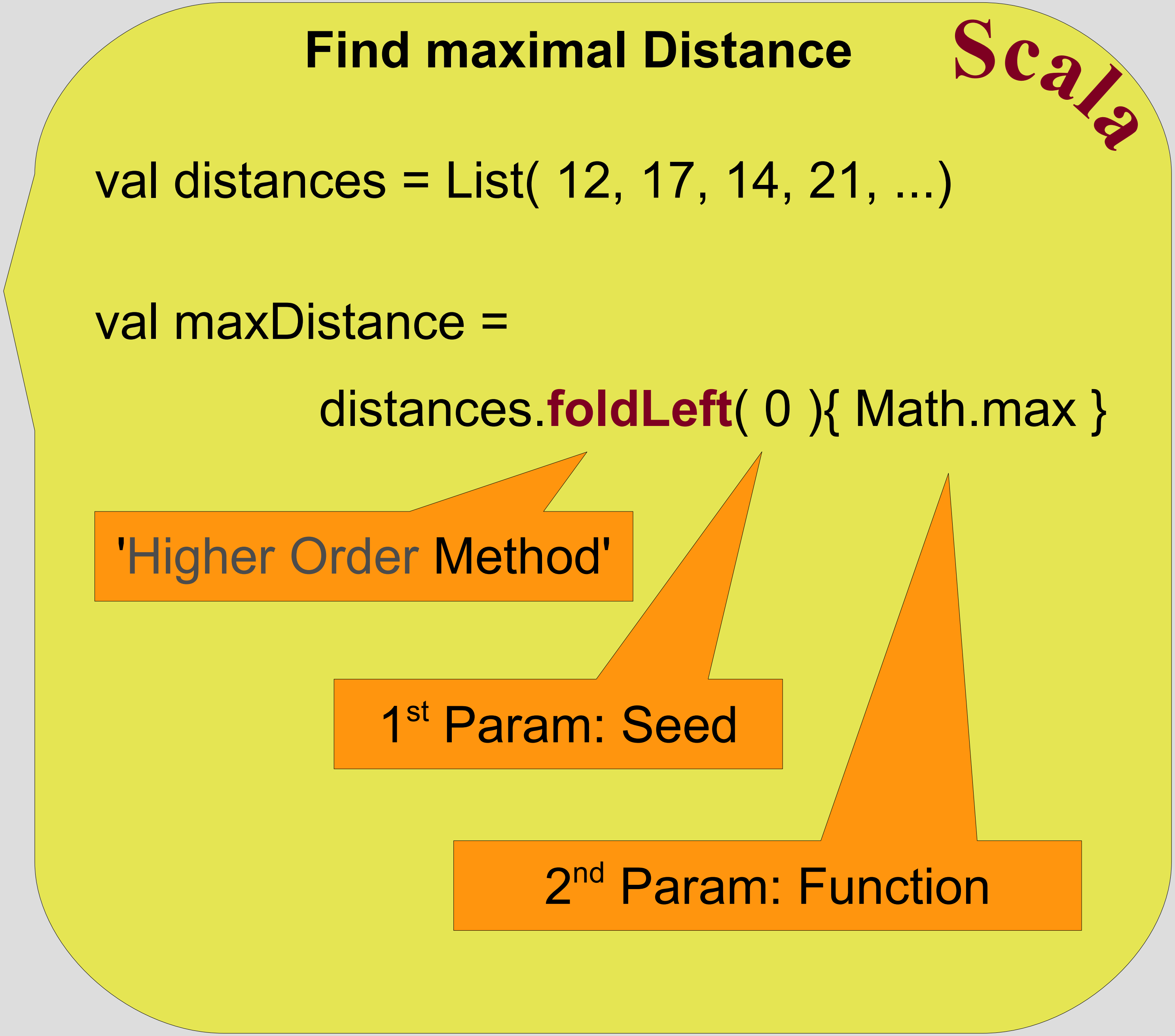
Find maximal Distance

Scala

```
val distances = List( 12, 17, 14, 21, ...)

val maxDistance =
            distances.foldLeft( 0 ){ Math.max }
```

Characteristics

- Expressive
- **High Level**
- Concise
- Extensible
- Pragmatic



Find maximal Distance

Scala

```
val distances = List( 12, 17, 14, 21, ...)

val maxDistance =
            distances.foldLeft( 0 ){ Math.max }
```

'Higher Order Method'

1st Param: Seed

2nd Param: Function

Characteristics

- Expressive
- **High Level**
- Concise
- Extensible
- Pragmatic

Find maximal Distance

Scala

```
val distances = List( 12, 17, 14, 21, ...)

val maxDistance =
            distances.foldLeft( 0 ){ Math.max }
```

```
(x: Int, y: Int ) => Math.max( x, y )
```

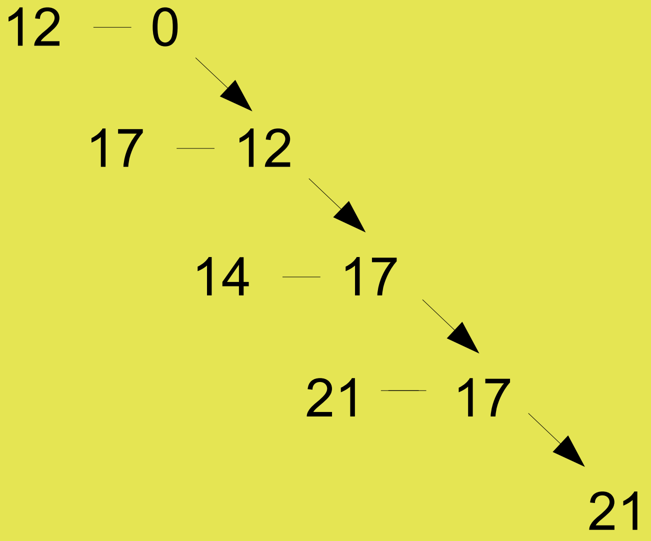
Characteristics

- Expressive
- **High Level**
- Concise
- Extensible
- Pragmatic

Find maximal Distance

Scala

```
val distances = List( 12, 17, 14, 21, ...)

val maxDistance =
            distances.foldLeft( 0 ){ Math.max }
```



Characteristics

- Expressive
- **High Level**
- Concise
- Extensible
- Pragmatic

Declarative style !

Scala

Check for prime number

```
def isPrime( candidate: Int ) = {

  (2  to  candidate/2 )

      .forall( number => candidate % number != 0 )
}
```

Characteristics

- Expressive
- **High Level**
- Concise
- Extensible
- Pragmatic

Scala

Declarative style !

Check for prime number

```
def isPrime( candidate: Int ) = {
  (2  to  candidate/2 )
     .forall( number => candidate % number != 0 )
}
```

Range

Predicate (Function)

Higher Order ('Quantor') Method

Characteristics

- Expressive
- **High Level**
- Concise
- Extensible
- Pragmatic

Scala

Declarative style !

Check for prime number

```
def isPrime( candidate: Int ) = {
  (2  to  candidate/2 )
     .forall( number => candidate % number != 0 )
}
```

- Remember: Everything is an Expression

Characteristics

- Expressive
- **High Level**
- Concise
- Extensible
- Pragmatic

Scala

Declarative style !

Check for prime number

```
def isPrime( candidate: Int ) = {
 (2  to  candidate/2 )
     .forall( number => candidate % number != 0 )
}
```

- Remember: Everything is an Expression
- No Assignments

Characteristics

- Expressive
- **High Level**
- Concise
- Extensible
- Pragmatic

Scala

Declarative style !

Check for prime number

```
def isPrime( candidate: Int ) = {
  (2  to  candidate/2 )
     .forall( number => candidate % number != 0 )
}
```

- Remember: Everything is an Expression
- No Assignments

  => almost Functional Style

Characteristics

- Expressive
- High Level
- **Concise**
- Extensible
- Pragmatic

Characteristics

- Expressive
- High Level
- **Concise**
- Extensible
- Pragmatic

Java

```
class Person{
    private String name;
    private int age;

    public Person( String name ){
        this.name = name
    }

    public String getName(){
        return this.name;
    }

    public int getAge(){
        return this.age;
    }

    public void setAge( int age ){
        this.age = age;
    }
}
```

Characteristics

- Expressive
- High Level
- **Concise**
- Extensible
- Pragmatic

Scala

```
class  Person(  val name:  String  ){
     var age: Int
}
```

Characteristics

- Expressive
- High Level
- **Concise**
- Extensible
- Pragmatic

Scala

```
class  Person(  val name:  String  ){
     var age: Int
}
```

- Class value parameter(s)
- Default Visibility: **private**

Characteristics

- Expressive
- High Level
- **Concise**
- Extensible
- Pragmatic

Scala

```
class  Person(  val name:  String  ){
     var age: Int
}
```

- read-only and public

Characteristics

- Expressive
- High Level
- **Concise**
- Extensible
- Pragmatic

Scala

```
class  Person(  val name:  String  ){
     var age: Int
}
```

- Class property
- Default visibility: **public**
- readable and writeable **var**iable

Characteristics

- Expressive
- High Level
- **Concise**
- Extensible
- Pragmatic

Scala

```
class  Person(  val name:  String  ){
     var age: Int
}

val friend = new Person( “Joe“ )

friend.age = 30

println(  friend.name )
```

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Adding new Control Structures Java

Resource Control

```
Reader reader = new BufferedReader( ... );
try{
      System.out.println(  reader.readLine()  );
}
finally{
      reader.close();
}
```

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Adding new Control Structures Java

Resource Control

```
Reader reader = new BufferedReader( ... );
try{
      System.out.println(  reader.readLine()  );
}
finally{
      reader.close();
}
```

Resource control !

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Adding new Control Structures Scala

Resource Control

```
using ( new BufferedReader( ... )  ) {
      reader =>  println(  reader.readLine()  );
}
```

" Loan Pattern "

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Adding new Control Structures Scala

1st Parameter: Resource under control

```
using ( new BufferedReader( ... )  ) {
      reader =>  println(  reader.readLine()  );
}
```

2nd Parameter: Function, using Resource

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Adding new Control Structures Scala

Resource Control

```
def using( reader: Reader )
             ( block: Reader => Unit ) {
     try{
          block( reader )
     }
     finally{
          reader.close
     }
}
```

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Adding new Control Structures Scala

1st Parameter: Resource under control

```
def using( reader: Reader )
             ( block: Reader => Unit ) {
     try{
          block( reader )
     }
     finally{
          reader.close
     }
}
```

2nd Parameter: Function, using Reader

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Adding new Control Structures Scala

Resource Control

```
def using( reader: Reader )
             ( block: Reader => Unit ) {
     try{
          block( reader )
     }
     finally{
          reader.close
     }
}
```

calling the function, passing the reader

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Adding new Control Structures Scala

Resource Control

```
def using( reader: Reader )
             ( block: Reader => Unit ) {
     try{
          block( reader )
     }
     finally{
          reader.close
     }
}
```

- **Resource control completely separated**

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Adding new Control Structures Scala

Resource Control

```
def using( reader: Reader )
                ( block: Reader => Unit ) {
      try{
            block( reader )
      }
      finally{
            reader.close
      }
}
```

- **Resource control completely separated**
- **Reusable with any Reader**

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Adding new Control Structures Scala

Resource Control

```
def using( reader: Reader )
              ( block: Reader => Unit ) {
     try{
          block( reader )
     }
     finally{
          reader.close
     }
}
```

- **Resource control completely separated**
- **But there's still a more generic way !**

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Scala

Adding new Control Structures

Resource Control

```
def using [ T <: { def close() } ]
                ( resource: T )
                ( block: T => Unit ) {
      try{
            block( resource )
      }
      finally{
            resource.close()
      }
}
```

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Adding new Control Structures Scala

Resource Control

```
def using [ T <: { def close() } ]
            ( resource: T )
            ( block: T => Unit ) {
    try{
        block( resource )
    }
    finally{
        resource.close()
    }
}
```

Structural Type

Any Type
which offers
a close() method

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Adding new Control Structures

Scala

Resource Control

```
def using [ T <: { def close() } ]
              ( resource: T )
              ( block: T => Unit ) {
    try{
        block( resource )
    }
    finally{
        resource.close()
    }
}
```

... statically typed

'Duck Typing'

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Adding new Control Structures

Scala

Write your own 'Loop – Unless'

```
def loop( body:  => Unit ): LoopUnlessCond =
     new LoopUnlessCond( body )


protected class LoopUnlessCond( body: => Unit ) {

   def unless( cond: => Boolean ) {
     body
     if ( !cond ) unless( cond )
   }
 }
```

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Adding new Control Structures

Scala

By-name parameter (Function without Arg)

```
def loop( body:  => Unit ): LoopUnlessCond =
      new LoopUnlessCond( body )
```

Function as class parameter

```
protected class LoopUnlessCond( body: => Unit ) {
    def unless( cond: => Boolean ) {
      body
      if ( !cond ) unless( cond )
    }
  }
```

calling the Function

calling the Function
(evaluating the condition)

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Adding new Control Structures Scala

Write your own 'Loop – Unless'

```
var i = 10

loop {
    println("i = " + i)
    i -= 1
} unless ( i == 0 )
```

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Adding new Control Structures Scala

Write your own 'Loop – Unless'

```
var i = 10

loop {
    println("i = " + i)
    i -= 1
} unless ( i == 0 )
```

By-name parameter
instead of

```
loop { () =>
        ...
} unless( .. )
```

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Adding new Control Structures

Scala

Write your own 'Loop – Unless'

```
var i = 10

loop {
    println("i = " + i)
    i -= 1
} unless ( i == 0 )
```

By-name parameter instead of

unless(**() =>** ...)

Characteristics

- Expressive
- High Level
- Concise
- **Extensible**
- Pragmatic

Scala

Adding new Control Structures

```
def using( reader: Reader )
           ( block: Reader => Unit ) {
    try{
        block( resource )
    }
    finally{
        resource.close
    }
}
```

Your keyword is my library !

Characteristics

- Expressive
- High Level
- Concise
- Extensible
- **Pragmatic**

Characteristics

- Expressive
- High Level
- Concise
- Extensible
- **Pragmatic**

Scala

```
def booksAsXml =
    <books>
        <book category="IT">
            <isbn>{ book.isbn }</isbn>
            <author>{ book.author }</author>
            ...
        </book>
        ...
    </books>
```

Characteristics

- Expressive
- High Level
- Concise
- Extensible
- **Pragmatic**

Scala

Parameterless Method

Direct XML Generation

```
def booksAsXml =
    <books>
        <book category="IT">
            <isbn>{ book.isbn }</isbn>
            <author>{ book.author }</author>
            ...
        </book>
        ...
    </books>
```

Embedding

Characteristics

- Expressive
- High Level
- Concise
- Extensible
- **Pragmatic**

Scala

Pattern Matching

```
def printAuthors {
   booksAsXml match
      case <books>{ books @ _* }</books> =>
         for( book  <-  books )
            println( “Author:“ + ( book \ “author“ ).text )
}
```

XPath – like Method

(Some) Features

- Composition
- Pattern Matching
- Modules
- Monads

Composition

- Feature Mixin
- Composable Types
- Enrichment
- Stackable Behaviour

Composition

- **Feature Mixin**
- Composable Types
- Enrichment
- Stackable Behaviour

```
trait Singer{               trait Flyer{
   def sing = …                def fly = …
}                           }
```

Composition

- **Feature Mixin**
- Composable Types
- Enrichment
- Stackable Behaviour

```
trait Singer{
   def sing = …
}
```

```
trait Flyer{
   def fly = …
}
```

Separation of independent facets

Composition

- **Feature Mixin**
- Composable Types
- Enrichment
- Stackable Behaviour

```
trait Singer{
   def sing = …
}
```

```
trait Flyer{
   def fly = …
}
```

...can be mixed into any type independently

class Bird **extends Flyer with Singer** {…}

```
val myBird = new Bird

myBird.sing

myBird.fly
```

Composition

- **Feature Mixin**
- Composable Types
- Enrichment
- Stackable Behaviour

```
trait Singer{                trait Flyer{
   def sing = …                 def fly = …
}                            }
```

orthogonal / independently to any type hierarchy

```
class Plane extends Flyer {…}

trait Superstar extends Human
                          with Singer
                          with Dancer
                          with ...
...
```

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

abstract class Spaceship{ def engage }

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

```
abstract class Spaceship{  def engage }
```

abstract Method (without definition)

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

```
abstract class Spaceship{  def engage }

trait CommandoBridge{
    def engage { for( _ <- 1 to 3 ){ speedUp } }
    def speedUp
}
```

abstract Method (without definition)

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

```
class Spaceship{  def engage }

trait CommandoBridge{
    def engage { for( _ <- 1 to 3 ){ speedUp } }
    def speedUp
}

trait PulseEngine{
    val maxPulse: Int
    var currentPulse = 0;
    def speedUp {
       if( currentPulse < maxPulse )
           currentPulse += 1  }
  }
```

abstract value

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

```
class StarCruiser extends Spacecraft
                             with CommandoBridge
                             with PulseEngine{

   val maxPulse = 200
}
```

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

```
class StarCruiser extends Spacecraft
                          with CommandoBridge
                          with PulseEngine{

   val maxPulse = 200
}


class Shuttle extends Spacecraft
                     with ControlCabin
                     with PulseEngine{

    val maxPulse = 50

    def increaseSpeed = speedUp
}
```

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

```
class StarCruiser extends Spacecraft
                          with CommandoBridge
                          with PulseEngine{
```

```
)
```

```
trait PulseEngine{
   def speedUp = ...
}
```

```
trait ControlCabin{
   def increaseSpeed
}
```

```
class Shuttle ext    ds Spacecraft
                 w  h ControlCabin
                 with PulseEngine{

     val maxPulse = 50

     def increaseSpeed = speedUp
}
```

'wiring'

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

```
trait WarpEngine{
   val maxWarp: Int
   var currentWarp = 0;

   def toWarp( x: Int ) {
    if( x < maxWarp ) currentWarp = x }
}
```

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

```
trait WarpEngine{

    val maxWarp: Int

    var currentWarp = 0;

    def toWarp( x: Int ) {
     if( x < maxWarp ) currentWarp = x }
}

  class Explorer extends Spacecraft
                              with CommandoBridge
                              with WarpEngine{

    val maxWarp = 10

    def speedUp = toWarp( currentWarp + 1 )
  }
```

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

```
trait WarpEngine{
    val maxWarp: Int
    var currentWarp = 0;

    def toWarp( x: Int )
     if( x < maxWarp ) c
}

  class Explorer extends Spacecraft
                              with CommandoBridge
                              with WarpEngine{

    val maxWarp = 10

    def speedUp = toWarp( currentWarp + 1 )
  }
```

```
trait CommandoBridge {
    def speedUp
}
```

'wiring'

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

```
class Jet extends Airplane with WarpEngine{
    val maxWarp = 4
   ...
}
```

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

```
class Jet extends Airplane with WarpEngine{
    val maxWarp = 4
   ...
}
```

WarpEngine is meant to be used only within Spaceships !!!

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

```
class Jet extends Airplane with WarpEngine{
    val maxWarp = 4
   ...
}

trait WarpEngine{
    this: Spacecraft =>
    ...
}
```

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

```
class Jet extends Airplane with WarpEngine{
    val maxWarp = 4
    ...
}

trait WarpEngine{
    this: Spacecraft =>
    ...
}
```

selftype declaration :

"can only be mixed into something which is at least of type Spacecraft"

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

```
class Jet extends Airplane with WarpEngine{
    val maxWarp = 4
   ...
}

trait WarpEngine{
    this: Spacecraft =>

}
```

Compiler Error:

"illegal inheritance: Jet does not conform to WarpEngine's selftype WarpEngine with Spacecraft"

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

```
def inspection( craft: ControlCabin
                                       with PulseEngine ) {

        craft.increaseSpeed

        assert( craft.currentPulse > 0 )
}
```

Assert that ControlCabin is wired with PulseEngine

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

```
def inspection( craft: ControlCabin
                       with PulseEngine ) {

    craft.increaseSpeed

    assert( craft.currentPulse > 0 )
}
```

Mixed in by **ControlCabin**

Mixed in by **PulseEngine**

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

```
def inspection( craft: ControlCabin
                        with PulseEngine ) {

     craft.increaseSpeed

     assert( craft.currentPulse > 0 )
}
```

'Compound type'

Intersection of object types

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

"Dependency Injection"

```
trait DBProvider {
  def mydatabase : ObjectContainer
}

class CafeDAO{
  self: DBProvider =>
  val db = mydatabase

  def findByName(..)
  ...
}
```

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

"Dependency Injection"

```
trait DBProvider {
  def mydatabase : ObjectContainer
}

class CafeDAO{
  self: DBProvider =>
  val db = mydatabase

  def findByName(..)
  ...
}
```

Self type

Can only be instantiated with a mixed in DBProvider

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

"Dependency Injection"

```
trait DBProvider {
  def mydatabase : ObjectContainer
}

class CafeDAO{
  self: DBProvider =>
  val db = mydatabase

  def findByName(..)
  ...
}
```

Self type

Can only be instantiated with a mixed in DBProvider

Get the database from the mixed in DBProvider

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

"Dependency Injection"

```
trait ProdDatabase extends DBProvider{
  def mydatabase = Db4o openFile "prodCafe.yap"
}

trait TestDatabase extends DBProvider{
  def mydatabase = Db4o openFile "testCafe.yap"
}
```

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

"Dependency Injection"

```
trait ProdDatabase extends DBProvider{
  def mydatabase = Db4o openFile "prodCafe.yap"
}

trait TestDatabase extends DBProvider{
  def mydatabase = Db4o openFile "testCafe.yap"
}

...
val cafeDaoTestee =
      new CafeDAO with TestDatabase
```

Composition

- Feature Mixin
- **Composable Types**
- Enrichment
- Stackable Behaviour

"Dependency Injection"

```
trait ProdDatabase extends DBProvider{
  def mydatabase = Db4o openFile "prodCafe.yap"
}

trait TestDatabase extends DBProvider{
  def mydatabase = Db4o openFile "testCafe.yap"
}

...
val cafeDaoTestee =
     new CafeDAO with TestDatabase
```

'Dynamic Mixin'

Single instance gets TestDatabase mixed in

Composition

- Feature Mixin
- Composable Types
- **Enrichment**
- Stackable Behaviour

```
trait RichCollection[+T] {
    def foreach( f: T => Unit )

    def exist ( predicate: T => Boolean ): Boolean = {
      foreach{ elem => if( predicate(elem) ) return true }
      false
    }

    def foldLeft[B]( seed: B)( f: (B,T) => B ) = {
      var res = seed
      foreach{ elem => res = f(res, elem) }
      res
    }
    ...
}
```

Composition

- Feature Mixin
- Composable Types
- **Enrichment**
- Stackable Behaviour

```
trait RichCollection[+T] {
    def foreach( f: T => Unit )

    def exist ( predicate: T => Boolean ): Boolean = {
      foreach{ elem => if( predicate(elem) ) return true }
      false
    }

    def foldLeft[B]( seed: B)( f: (B,T) => B ) = {
      var res = seed
      foreach{ elem => res = f(res, elem) }
      res
    }
    ...
}
```

'contract'

Composition

- Feature Mixin
- Composable Types
- **Enrichment**
- Stackable Behaviour

```
trait RichCollection[+T] {
    def foreach( f: T => Unit )

    def exist ( predicate: T => Boolean ): Boolean = {
      foreach{ elem => if( predicate(elem) ) return true }
      false
    }

    def foldLeft[B]( seed: B)( f: (B,T) => B ) = {
      var res = seed
      foreach{ elem => res = f(res, elem) }
      res
    }
    ...
}
```

'contract'

forall, filter, partition, size, ...

Composition

- Feature Mixin
- Composable Types
- **Enrichment**
- Stackable Behaviour

```
trait RichCollection[+T] {
  def foreach( f: T => Unit )

  def exist ( predicate: T => Boolean ): Boolean = {
    foreach{ elem => if( predicate(elem) ) return true }
    false
  }

  def foldLeft[B]( seed: B)( f: (B,T) => B ) = {
    var res = seed
    foreach{ elem => res = f(res, elem) }
    res
  }
  ...
}
```

Implement one ...

... receive many

Composition

- Feature Mixin
- Composable Types
- **Enrichment**
- Stackable Behaviour

```
abstract class Stack[+A] extends Object
                                       with RichCollection[A] {

  def push[B >: A](x: B): Stack[B] = ...

  def isEmpty: Boolean

  def top: A

  def pop: Stack[A]

  def foreach( f: A => Unit ) {
     if( ! isEmpty ){
        f( top )
        pop.foreach( f )
     }
  }
}
```

Composition

- Feature Mixin
- Composable Types
- **Enrichment**
- Stackable Behaviour

```
val s = new EmptyStack[Int] push 1 push 2 push 3

s.exist( _ >= 2 )           => true

s.foldLeft(0)( _ + _ )      => 6

s.filter( _ >= 2 )          => List( 2, 3 )
```

Composition

- Feature Mixin
- Composable Types
- **Enrichment**
- Stackable Behaviour

```
val jSet = new java.util.HashSet[Int]
                                with RichCollection[Int] {

   def foreach( f: Int => Unit ) {
        val elems = iterator
        while( elems.hasNext ){ f( elems.next ) }
   }
}


jSet.exist( _ >= 2 )                 => true

jSet.foldLeft(0)( _ + _ )            => 6

jSet.filter( _ >= 2 )                => List( 2, 3 )
```

Composition

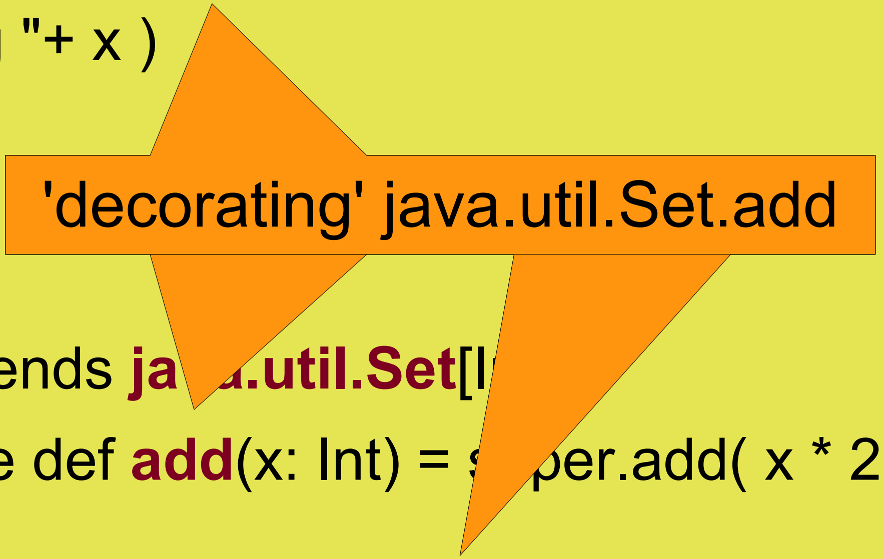
- Feature Mixin
- Composable Types
- Enrichment
- **Stackable Behaviour**

```
trait Logging[A] extends java.util.Set[A]{
  abstract override def add(x: A) = {
    println( "adding "+ x )
    super.add( x )
  }
}

trait Doubling extends java.util.Set[Int]{
  abstract override def add(x: Int) = super.add( x * 2 )
}

trait Incrementing extends java.util.Set[Int]{
  abstract override def add(x: Int) = super.add( x + 1 )
}
```

Composition

- Feature Mixin
- Composable Types
- Enrichment
- **Stackable Behaviour**

```
trait Logging[A] extends java.util.Set[A]{
  abstract override def add(x: A) = {
    println( "adding "+ x )
    super.add( x )
  }
}

trait Doubling extends ja a.util.Set[I
  abstract override def add(x: Int) = s  per.add( x * 2 )
}

trait Incrementing extends java.util.Set[Int]{
  abstract override def add(x: Int) = super.add( x + 1 )
}
```



'decorating' java.util.Set.add

Composition

- Feature Mixin
- Composable Types
- Enrichment
- **Stackable Behaviour**

```
val jSet = new java.util.HashSet[Int]
                              with Logging[Int]
                              with Incrementing
                              with Doubling

jSet  add  1
jSet  add  2
jSet  add  3


   =>  adding 3
       adding 5
       adding 7
```

Composition

- Feature Mixin
- Composable Types
- Enrichment
- **Stackable Behaviour**

```
val jSet = new java.util.HashSet[Int]
                                        with Logging[Int]
                                        with Doubling
                                        with Incrementing

jSet  add  1
jSet  add  2
jSet  add  3


    =>  adding 4
        adding 6
        adding 8
```

Composition

- Feature Mixin
- Composable Types
- Enrichment
- **Stackable Behaviour**

```
val jSet = new java.util.HashSet[Int]
                                       with Logging[Int]
                                       with Doubling
                                       with Incrementing

jSet  add  1
jSet  add  2
jSet  add  3
```

Lineariation

```
    =>  adding 4
        adding 6
        adding 8
```

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

```
EXPRESSION := NUMBER | BINARY_OP

BINARY_OP := ADD | SUB | MULT

ADD := Add( EXPRESSION, EXPRESSION )

SUB := Sub( EXPRESSION, EXPRESSION )

MULT := Mult( EXPRESSION, EXPRESSION )

NUMBER := Number( Int )
```

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

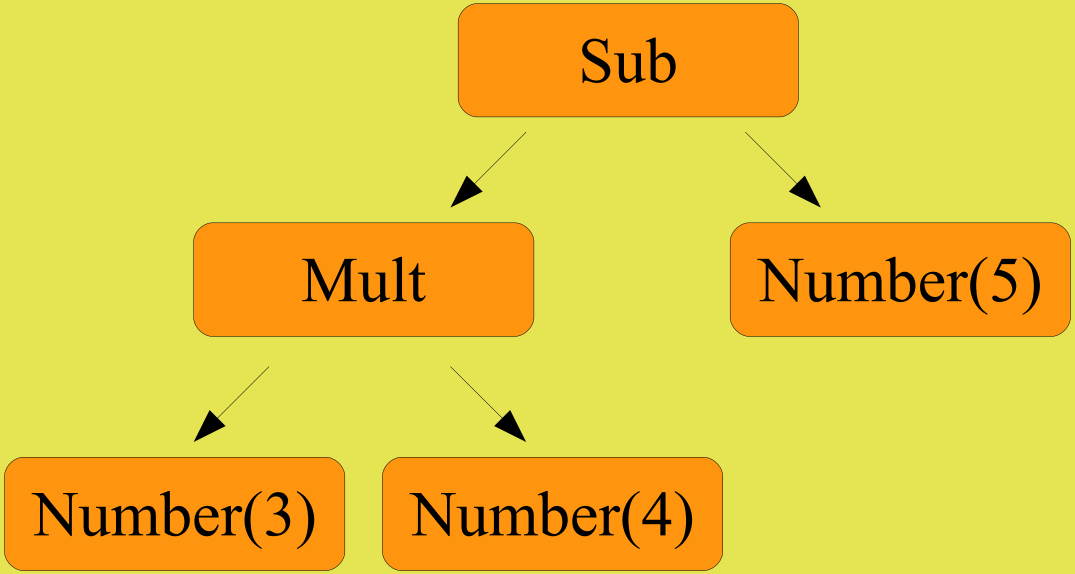
A little 'Expression Language'

```
EXPRESSION := NUMBER | BINARY_OP

BINARY_OP := ADD | SUB | MULT

ADD := Add( EXPRESSION, EXPRESSION )

SUB := Sub( EXPRESSION, EXPRESSION )

MULT := Mult( EXPRESSION, EXPRESSION )

NUMBER := Number( Int )


Sub( Mult( Number( 3 ), Number( 4 ) ), Number( 5 ) )
```

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'



Sub(Mult(Number(3), Number(4)), Number(5))

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

```
abstract class Expression

case class Number( num: Int ) extends Expression

case class BinaryOperator
    ( opCode: String, left: Expression, right: Expression )
        extends Expression

case class Add( s1: Expression, s2: Expression )
    extends BinaryOperator( "+", s1, s2 )

case class Sub( s1: Expression, s2: Expression )
  extends BinaryOperator( "-", s1, s2 )

case class Mult( m1: Expression, m2: Expression )
    extends BinaryOperator( "*", m1, m2 )
```

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

Case class

```
case class Number( num: Int ) extends Expression

case class BinaryOperator
    ( opCode: String, left: Expression, right: Expression )
        extends Expression

case class Add( s1: Expression, s2: Expression )
    extends BinaryOperator( "+", s1, s2 )

case class Sub( s1: Expression, s2: Expression )
  extends BinaryOperator( "-", s1, s2 )

case class Mult( m1: Expression, m2: Expression )
    extends BinaryOperator( "-", m1, m2 )
```

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

Case class

```
case class Number( num: Int ) extends Expression

case class BinaryOperator
   ( opCode: String, left: Expression, right: Expression )
      extends Expression
```

Serve Super Constructor !

```
case class Sub( s1: Expression, s2: Expression )
  extends BinaryOperator( "-", s1, s2 )

case class Mult( m1: Expression, m2: Expression )
   extends BinaryOperator( "-", m1, m2 )
```

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

```
def prettyPrint( expr: Expression ) {
   expr match {
      case Number( x ) => print( x )
      case BinaryOperator( opCode, expr1, expr2 ) => {
                                          print( "(" )
                                             prettyPrint( expr1 )
                                                print( opCode )
                                             prettyPrint( expr2 )
                                          print( ")" ) }
      case _ => print( "unknown" )
   }
}
```

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

Match expr agains 'Patterns'

```
def prettyPrint(
  expr match {
     case Number( x ) => print( x )

     case BinaryOperator( opCode, expr1, expr2 ) => {
                                              print( "(" )
                                                 prettyPrint( expr1 )
                                                    print( opCode )
                                                 prettyPrint( expr2 )
                                              print( ")" ) }

     case _ => print( "unknown" )
  }
}
```

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

Match expr agains 'Patterns'

```
def prettyPrint(
  expr match {
    case Number( x ) => print( x )
```

expr matches case class Number(Int) ?

```
                                   print( "(" )
                                      prettyPrint( expr1 )
                                         print( opCode )
                                      prettyPrint( expr2 )
                                   print( ")" ) }

    case _ => print( "unknown" )
  }
}
```

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

```
def prettyPrint(
  expr match {
    case Number( x ) => print( x )
```

Match expr agains 'Patterns'

Bind class' value parameter in Number(Int) to x

```
                              print( "(" )
                                prettyPrint( expr1 )
                                  print( opCode )
                                prettyPrint( expr2 )
                              print( ")" ) }

    case _ => print( "unknown" )
  }
}
```

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

Match expr agains 'Patterns'

expr matches any case class BinaryOperator(..) ?

```
def prettyPrint(
  expr match {

     case BinaryOperator( opCode, expr1, expr2 ) => {
                                   print( "(" )
                                     prettyPrint( expr1 )
                                       print( opCode )
                                     prettyPrint( expr2 )
                                   print( ")" ) }

     case _ => print( "unknown" )
  }
}
```

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

Match expr agains 'Patterns'

Bind class' value parameters ...

```
def prettyPrint(
   expr match {

      case BinaryOperator( opCode, expr1, expr2 ) => {
                                   print( "(" )
                                      prettyPrint( expr1 )
                                         print( opCode )
                                      prettyPrint( expr2 )
                                   print( ")" ) }

      case _ => print( "unknown" )
   }
}
```

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

```
def prettyPrint(
  expr match {
    case Number( x ) => print( x )

    case BinaryOperator( opCode, expr1, expr2 ) => {
                        print( "(" )
                          prettyPrint( expr1 )
                            print( opCode )
                          prettyPrint( expr2 )
                        print( ")" ) }

    case _ => print( "unknown" )
  }
}
```

Match expr agains 'Patterns'

Block instead of
a single expression

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

Match expr agains 'Patterns'

```
def prettyPrint(

  expr match {

     case Number( x ) => print( x )

     case BinaryOperator( opCode, expr1, expr2 ) => {

                                 print( "(" )
                                    prettyPrint( expr1 )
                                       print( opCode )
                                    prettyPrint( expr2 )
                                 print( ")" ) }

     case _ => print( "unknown" )
  }
}
```

Matches against 'everything'

(Some) Features

- Composition
- **Pattern Matching**

A little 'Expression Language'

```
val expr =
    Sub( Mult( Number( 3 ), Number( 4 ) ), Number( 5 ) )


prettyPrint( expr )

          => ( ( 3 * 4 ) - 5 )
```

- Modules
- Monads

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

```
val expr =
   Sub( Mult( Number( 3 ), Number( 4 ) ), Number( 5 ) )
```

Compagnion object.apply()
for every case class provided
(no instantiation using *new* necessary)

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

```
def simplify( expr: Expression ): Expression = {

  expr match {

    case Add( e, Number( 0 ) ) => e
    case Add( Number( 0 ), e ) => e

    case Mult( e, Number( 0 ) ) => Number( 0 )
    case Mult( Number( 0 ), e ) => Number( 0 )
    case Mult( e, Number( 1 ) ) => e
    case Mult( Number( 1 ), e ) => e

    case Sub( Number( x ), Number( y ) ) if x == y => Number( 0 )
    case Sub( Add( e, Number( x ) ), Number( y ) ) if x == y => e

    case Mult( e1, e2 ) =>  Mult( simplify( e1 ), simplify( e2 ) )

    case _ => expr
  }
}
```

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

```
def simplify( expr: Expression ): Expression = {

  expr match {

    case Add( e, Number( 0 ) ) => e
    case Add( Number( 0 ), e ) => e
```

Matches only against Number(0)

```
    case Mult( e, Number( 1 ) ) => e
    case Mult( Number( 1 ), e ) => e

    case Sub( Number( x ), Number( y ) ) if x == y => Number( 0 )
    case Sub( Add( e, Number( x ) ), Number( y ) ) if x == y => e

    case Mult( e1, e2 ) =>  Mult( simplify( e1 ), simplify( e2 ) )

    case _ => expr
  }
}
```

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

```
def simplify( expr: Expression ): Expression = {

  expr match {

    case Add( e, Number( 0 ) ) => e
    case Add( Number( 0 ), e ) => e

    case Mult( e, Number( 0 ) ) => Number( 0 )
    case Mult( Number( 0 ), e ) => Number( 0 )
```

Guard: Matches only if bound x equals bound y

```
    case Sub( Number( x ), Number( y ) ) if x == y => Number( 0 )
    case Sub( Add( e, Number( x ) ), Number( y ) ) if x == y => e

    case Mult( e1, e2 ) =>  Mult( simplify( e1 ), simplify( e2 ) )

    case _ => expr
  }
}
```

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

```
val expr =
  Mult( Sub( Add( Number( 1 ), Number( 4 )  ), Number( 4 ) ),
        Sub( Number(3), Number(2)  )  )

prettyPrint( expr )

     => ( ( ( 1 + 4 ) - 4 ) * ( 3 - 2 ) )
```

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

```
val expr =
  Mult( Sub( Add( Number( 1 ), Number( 4 )  ), Number( 4 ) ),
         Sub( Number(3), Number(2)  )  )

prettyPrint( expr )

        => ( ( ( 1 + 4 ) - 4 ) * ( 3 - 2 ) )


val sExpr = simplify( expr )

prettyPrint( sExpr )

        => ( 1 * ( 3 - 2 ) )
```

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

```
val expr =
 Mult( Sub( Add( Number( 1 ), Number( 4 )  ), Number( 4 ) ),
       Sub( Number(3), Number(2)  )  )

prettyPrint( expr )

        => ( ( ( 1 + 4 ) - 4 ) * ( 3 - 2 ) )


val sExpr = simplify( expr )

prettyPrint( sExpr )

        => ( 1 * ( 3 - 2 ) )
```

Mult(Number(1), expr) should be expr !!!

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

```
def simplify( expr: Expression ): Expression = {

  expr match {

    case ...

    case Mult( e1, e2 ) =>  {
      val se1 = simplify( e1 )
      val se2 = simplify( e2 )

      if( se1 != e1 || se2 != e2 ) simplify( Mult( se1, se2 ) )
      else Mult( se1, se2 )
   }
}
```

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

```
def simplify( expr: Expression ): Expression = {

  expr match {

    case ...

    case Mult( e1, e2 ) =>  {
      val se1 = simplify( e1 )
      val se2 = simplify( e2 )

      if( se1 != e1 || se2 != e2 ) simplify( Mult( se1, se2 ) )
      else Mult( se1, se2 )
    }
  }
}
```

(not) equals() on every case class provided !

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

A little 'Expression Language'

```
val expr =
  Mult( Sub( Add( Number( 1 ), Number( 4 )  ), Number( 4 ) ),
        Sub( Number(3), Number(2)  )  )


val sExpr = simplify( expr )

prettyPrint( sExpr )
```

=> (3 - 2)

(Some) Features

- Composition
- **Pattern Matching**
- Modules
- Monads

Some Pattern 'types'

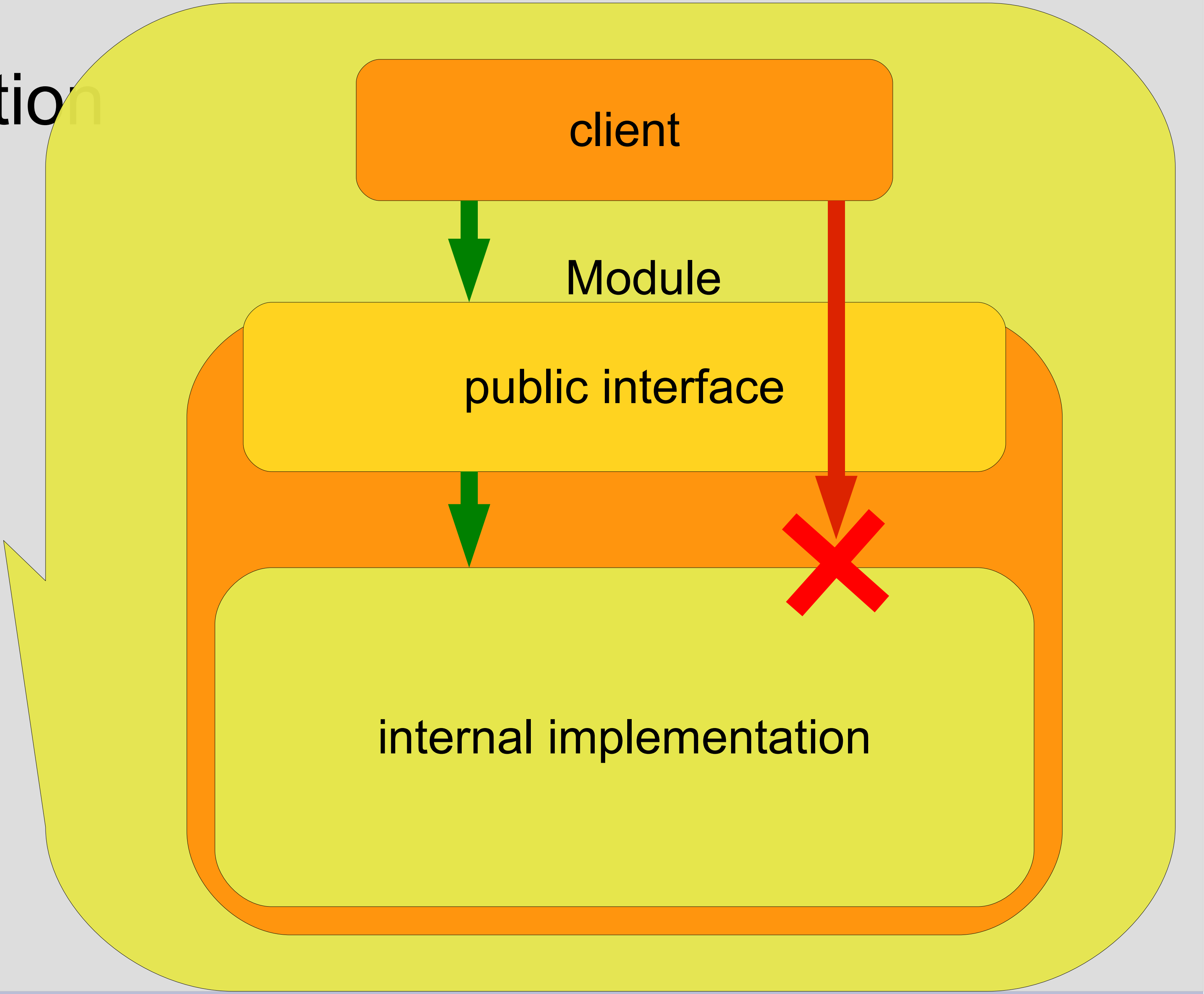
```
def matchAny( a: Any ) : Any {

    a match {
            case 1                        => “one“
            case “two“                    => 2
            case i: Int                   => “scala.Int“
            case <tag>{ t }</tag>         => t
            case head::tail               => head
            case ( x, y )                 => “tuple“
            case _                        => “anything else“
    }
}
```

(Some) Features

- Composition
- Pattern Matching
- **Modules**
- Monads

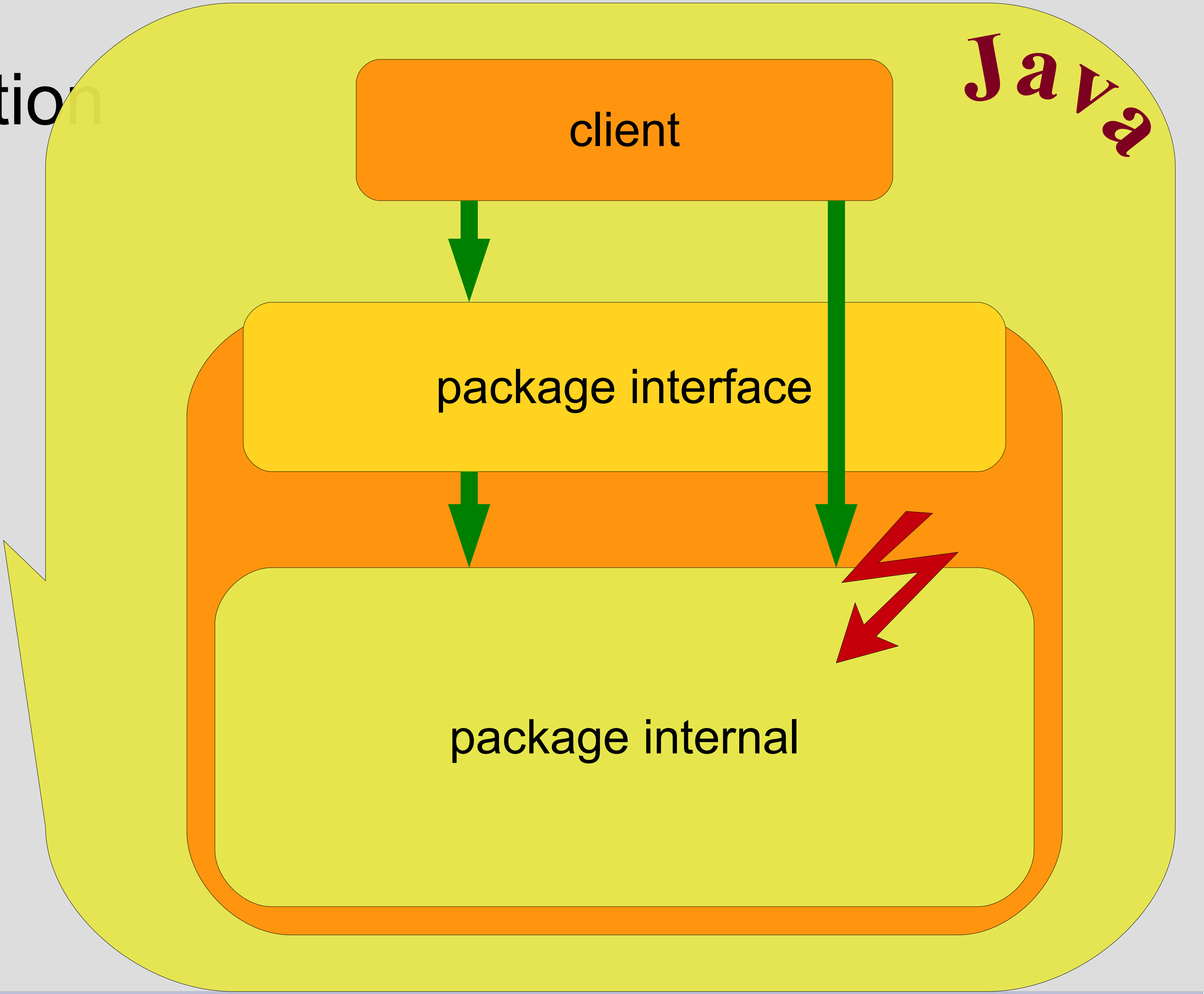
(Some) Features

- Composition
- Pattern Matching
- **Modules**
- Monads



(Some) Features

- Composition
- Pattern Matching
- **Modules**
- Monads



Java

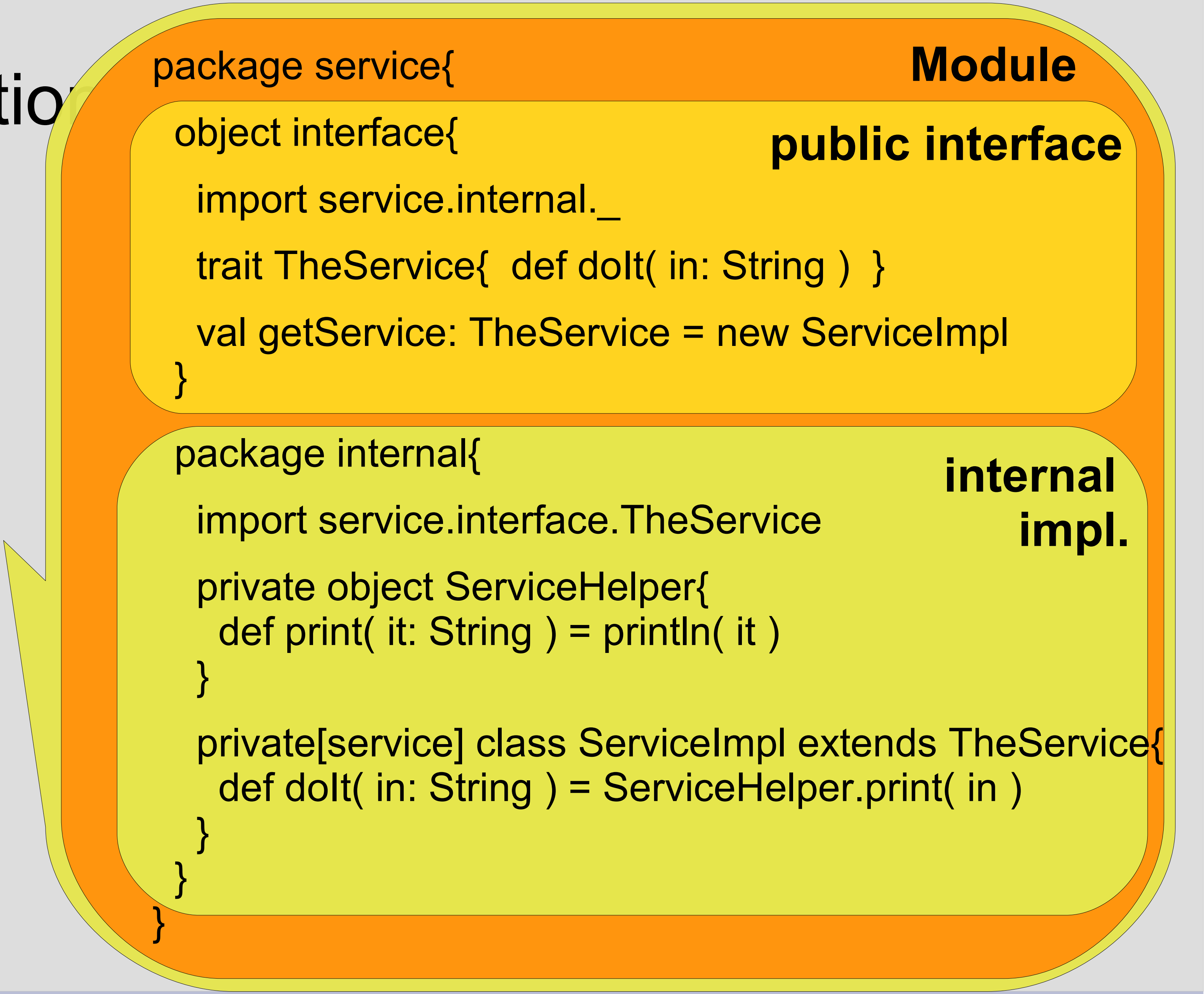
client

package interface

package internal

(Some) Features

- Compositio
- Pattern Matching
- **Modules**
- Monads



```
package service{
 object interface{
  import service.internal._
  trait TheService{  def doIt( in: String )  }
  val getService: TheService = new ServiceImpl
 }
 package internal{
  import service.interface.TheService
  private object ServiceHelper{
   def print( it: String ) = println( it )
  }
  private[service] class ServiceImpl extends TheService{
   def doIt( in: String ) = ServiceHelper.print( in )
  }
 }
}
```

(Some) Features

- Composition
- Pattern Matching
- **Modules**
- Monads

```
package service{
  object interface{
    import service.internal._
    trait TheService{  def doIt( in: String )  }
    val getService: TheService = new ServiceImpl
  }

  package internal{
    import service.interface.TheService
    private object ServiceHelper{
      def print( it: String ) = println( it )
    }
    private[service] class ServiceImpl extends TheService{
      def doIt( in: String ) = ServiceHelper.print( in )
    }
  }
}
```

Service interface

(Some) Features

- Composition
- Pattern Matching
- **Modules**
- Monads

```
package service{
 object interface{
  import service.internal._
  trait TheService{  def doIt( in: String )  }
  val getService: TheService = new ServiceImpl
 }
 package internal{
  import service.interface.TheService
  private object ServiceHelper{
   def print( it: String ) = println( it )
  }
  private[service] class ServiceImpl extends TheService{
   def doIt( in: String ) = ServiceHelper.print( in )
  }
 }
}
```

Nested Package

(Some) Features

- Composition
- Pattern Matching
- **Modules**
- Monads

```
package service{
 object interface{
  import service.internal._
  trait TheService{  def doIt( in: String )  }
  val getService: TheService = new ServiceImpl
 }

 package internal{
  import service.interface.TheService
  private object ServiceHelper{
   def print( it: String ) = println( it )
  }
  private[service] class ServiceImpl extends TheService{
   def doIt( in: String ) = ServiceHelper.print( in )
  }
 }
}
```

Local import

(Some) Features

- Composition
- Pattern Matching
- **Modules**
- Monads

```
package service{
 object interface{
   import service.internal._
   trait TheService{  def doIt( in: String )  }
   val getService: TheService = new ServiceImpl
 }

 package internal{
   import service.interfac
   private object ServiceHelper{
     def print( it: String ) = println( it
   }
   private[service] class ServiceImpl extends TheService{
     def doIt( in: String ) = ServiceHelper.print( in )
   }
 }
}
```

Only visible within this package

(Some) Features

- Composition
- Pattern Matching
- **Modules**
- Monads

```
package service{
  object interface{
    import service.internal._
    trait TheService{  def doIt( in: String )  }
    val getService: TheService = new ServiceImpl
  }

  package internal{
    import service.interface.Th
    private object ServiceHelpe
      def print( it: String ) = p
    }
    private[service] class ServiceImpl extends TheService{
      def doIt( in: String ) = ServiceHelper.print( in )
    }
  }
}
```

Only visible within this package, up to package *service*

(Some) Features

- Composition
- Pattern Matching
- **Modules**
- Monads

```
package client{

 import service.interface._

 object TheClient{
   val theService: TheService = getService
   theService.doIt( "hello" );
 }

}
```

(Some) Features

- Composition
- Pattern Matching
- **Modules**
- Monads

```
package client{

  import service.interface._

  object TheClient{
     val theService: TheService = getService
     theService.doIt( "hello" );
  }

}
```

importing
all members
of the public
interface object

(Some) Features

- Composition
- Pattern Matching
- **Modules**
- Monads

```
package client{

  ~~import service.interface._~~
  import service.internal._

  object TheClient{

    ~~val theService: TheService = getService~~
    val theService = new ServiceImpl

    theService.doIt( "hello" );
  }

}
```

(Some) Features

- Composition
- Pattern Matching
- **Modules**
- Monads

```
package client{

  ~~import service.interface._~~
  import service.internal._

  object TheClient{

     ~~val theService: TheService = getService~~
     val theService = new ServiceImpl

     theService.doIt( "h    o" );
  }

}
```

Compile Error:

"class ServiceImpl cannot be accessed in package service.internal"

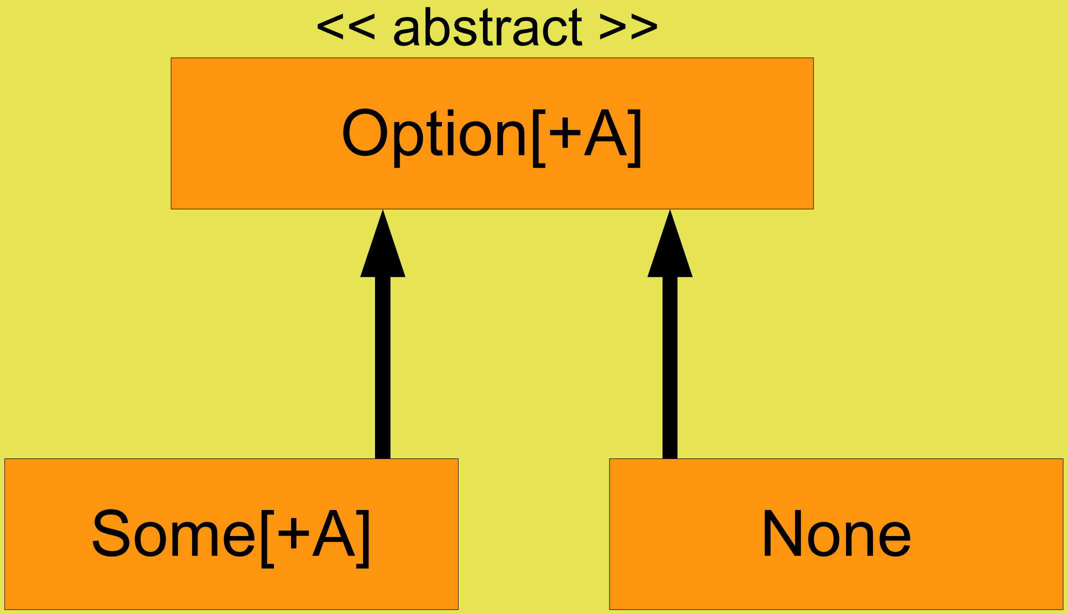
(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option



Handling the **presence** or **absence** of something

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option

```
class CustomerDAO{

  def findCustomer( custId: Long ) : Option<Customer> = {
    ...
    if( found( customer ) ) Some( customer ) else None
  }
}
```

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option

```
class CustomerDAO{

  def findCustomer( custId: Long ) : Option<Customer> = {
    ...
    if( found( customer ) ) Some( customer ) else None
  }
}
```

Explicit Notion, that there may be 'none' result

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option

```
val customerHit = customerDAO.findCustomer( 123 );
...

customerHit match {
        case Some( customer )        => println( customer.name )
        case None                    => println( “not found“ )
}
```

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option

```
val customerHit = customerDAO.findCustomer( 123 );
...

customerHit match {
      case Some( customer )      => println( customer.name )
      case None                  => println( “not found“ )
}
```

Explicit Handling the absensce of a result

Forces 'Awareness'

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option

```
val customerHit = customerDAO.findCustomer( 123 );
...

customerHit match {
      case Some( customer )       => println( customer.name )
      case None                   => println( “not found“ )
}
```

Explicit Handling the absensce of a result

Forces 'Awareness'

... beside from that ... what's the deal ???

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option

```
val customerHit = customerDAO.findCustomer( 123 );
...

customerHit match {
        case Some( customer )        => println( customer.name )
        case None                    => println( “not found“ )
}
```

Explicit Handling the absensce of a result

Forces 'Awareness'

... beside from that ... what's the deal ???

'Combination' !!!

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option

```
val projects =     Map(  "Jan"  -> "IKT",
                         "Joe"  -> "TensE",
                         "Luca" -> "InTA" )

val customers = Map(  "IKT"  -> "Hanso GmbH",
                      "InTA" -> "RAIA Duo" )

val cities =       Map(  "Hanso GmbH" -> "Stuttgart",
                         "Mogno" -> "Mailand" )
```

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option

```
val projects =     Map(  "Jan"  -> "IKT",
                          "Joe"  -> "TensE",
                          "Luca" -> "InTA" )

val customers = Map(  "IKT"  -> "Hanso GmbH",
                         "InTA" -> "RAIA Duo" )

val cities =          Map(  "Hanso GmbH" -> "Stuttgart",
                         "Mogno" -> "Mailand" )
```

Where is Jan ?

Jan -> IKT -> Hanso GmbH -> Stuttgart

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option

```
val projects =     Map(  "Jan"  -> "IKT",
                         "Joe"  -> "TensE",
                         "Luca" -> "InTA" )

val customers = Map(  "IKT"  -> "Hanso GmbH",
                      "InTA" -> "RAIA Duo" )

val cities =        Map(  "Hanso GmbH" -> "Stuttgart",
                          "Mogno" -> "Mailand" )
```

Where is Luca ?

Luca -> InTA -> RAIA Duo -> ??? ('unknown')

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option

Java

```
public String whereIs( String name ){
   String project = projects.get( name );
   if( project != null ){
      String customer = customers.get( project );
      if( customer != null ){
         String city = cities.get( customer )
         if( city != null ) return city;
           else return “unknown“;
      }
      else return "unknown";
   }
   else return "unknown";
}
```

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option

Scala

```
def whereIs( name: String ) = {
  projects.get( name )
      .flatMap( project =>  customers get project )
      .flatMap( customer => cities get customer )
      .getOrElse( "Unknown!" )
}
```

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option

Scala

```
def whereIs( name: String ) = {
  projects.get( name )
      .flatMap( project =>  customers get project )
      .flatMap( customer => cities get customer )
      .getOrElse( "Unknown!" )
}
```

Results in Option[String]

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option

Scala

```
def whereIs( name: String ) = {
  projects.get( name )
      .flatMap( project =>  customers get project )
      .flatMap( customer => cities get customer )
      .getOrElse( "Unknown!" )
}
```

Results in Option[String]

Option[A].map((A) => B)   => Option[B]

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option

Scala

```
def whereIs( name: String ) = {
  projects.get( name )
     .flatMap( project =>  customers get project )
     .flatMap( customer => cities get customer )
     .getOrElse( "Unknown!" )
}
```

Results in Option[String]

Option[A].map((A) => B)   => Option[B]

B -> Option[B])   => Option[Option[B]

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option

Scala

```
def whereIs( name: String ) = {
  projects.get( name )
      .flatMap( project =>  customers get project )
      .flatMap( customer => cities get customer )
      .getOrElse( "Unknown!" )
}
```

Results in Option[String]

```
Option[A].map(  ( A ) => B )     => Option[B]
B -> Option[B] )            => Option[Option[B]
...flatmap( ( A ) => Option[B] )  => Option[B]
```

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option

Scala

```
def whereIs( name: String ) = {
  projects.get( name )
      .flatMap( project =>  customers get project )
      .flatMap( customer => cities get customer )
      .getOrElse( "Unknown!" )
}
```

Alternative (else) if None

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option

Scala

```
def whereIs( name: String ) = {
  projects.get( name )
      .flatMap( project =>  customers get project )
      .flatMap( customer => cities get customer )
      .getOrElse( "Unknown!" )
}
```

- No tests of absence during 'combination' of Maps *projects*, *customers* and *cities* necessary
- Option monad provides safe 'binding' of operations

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option

Scala

```
def whereIs( name: String ) = {
  projects.get( name )
      .flatMap( customers get )
      .flatMap( cities get )
      .getOrElse( "Unknown!" )
}
```

shortcut for (project => customers get project)

(Some) Features

- Composition
- Pattern Matching
- Modules
- **Monads**

A simple Monad: Option

Scala

```
def whereIs( name: String ) = {

    (  for(  project      <- projects get name;
             customer <- customers get project;
             city          <- cities get customer

        ) yield city

    ).getOrElse( "Unknown!" )
}
```

- Combination of Operations on Maps written as

for-comprehension

(Some) Features

- Composition
- Pattern Matching
- Modules
- Monads
- **Any many more ...**

(Some) Features

- Composition
- Pattern Matching
- Modules
- Monads
- **And many more ...**

Continuations (2.8)

Named Parameters (2.8)

View Bounds

Nested Methods

Extractor Objects

Implicit Parameters

(abstract) Type members

Combinator Parsing

Summary

Scala is ...

- Object Oriented
- Functional
- Pragmatic
- Scalable

Summary

Thank you !

Reference

M.Odersky, L.Spoon, B.Venners	'Programming in Scala' (Artima Inc)
Scala Home	http://www.scala-lang.org
Scala By Example	http://www.scala-lang.org/docu/files/ScalaByExample.pdf
Jonas Boner	'Pragmatic Real World Scala' http://www.infoq.com/presentations/Scala-Jonas-Boner
D.Wampler, A.Payne	'Programming Scala' (O'Reilly) http://programming-scala.labs.oreilly.com/index.html
James Strachan	Scala as the long term replacement for java/javac? http://macstrac.blogspot.com/2009/04/scala-as-long-term-replacement-for.html