Scala Workshop

Konrad Malawski
Jerzy Müller
JDD 2013

Hello!

- About me
- About you
- Known languages?
- Whish List

ebay

polish java user group

GeeCON

Konrad `@ktosopl` Malawski

[sckrk]

Krakow Scala

Jerzy Muller

KRAKOW SCALA

Scala **Syntax** Basics 100	**Hello 世界！** 100	**SBT** Basics 100	Scala **REPL** (basics & :commands) 150
Pattern **Matching** 200	Functional **Set** impl 200	**Option** impl 200	**Functions** PartialFunctions 200
"Like a **case class**" **manual impl** 300	**override** matters (linearization) 300	**List** impl 300	**Tree** impl 300
ifAndOnlyIf (call-by-name) 300	**implicit case** **value class** 500	**JSON** Serializer impl (type class) 500	**SBT** Build Patterns 500
Modules impl (cake pattern) 700	**Config** pattern (traits traits traits!) 700	Custom String **Interpolation** 700	Super Type Safeness (**Phantom Types**) 700
ScalaTest impl **1000**	**Akka impl** **1000**	**Rogue impl** **1000**	

Core Concepts

Core Concepts

- Static Typing
- Object Oriented
- Functional & Immutable
- Small interfaces, richly combined
- JVM eco-system

Scala REPL

Scala REPL

Scala REPL

- $ scala

```
ktoso@moon /Users/ktoso
$ scala
Welcome to Scala version 2.10.0-RC5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_37).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Your Best Friend
in
Scala Experimentation

Console - res##

```
scala> "something"
res1: String = something
```

Console - val's

```
scala> val name = "a name"
name: String = a name
```

Console - val's

```
scala> val name = "a name"
name: String = a name
```

```
scala> val name = "other"
name: String = other
```

You can redefine vals in the REPL.

:paste mode

```
scala> :paste
// Entering paste mode (ctrl-D to finish)
```

Must be defined "together" with class Person!

:paste mode

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

case class Person(name: String)

object Person {
  def from(s: String) = new Person(name)
}
```

Must be defined "together" with class Person!

:paste mode

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

case class Person(name: String)

object Person {
  def from(s: String) = new Person(name)
}


[CTRL-D]
// Exiting paste mode, now interpreting.

defined class Person
defined module Person
```

Must be defined "together" with class Person!

Power Tip: :javap

```
scala> :javap Person

Compiled from "<console>"
public class Person extends java.lang.Object implements
scala.Product, scala.Serializable {
    public java.lang.String name();
    public Person copy(java.lang.String);
    public java.lang.String copy$default$1();
    public java.lang.String productPrefix();
    public int productArity();
    public java.lang.Object productElement(int);
    public scala.collection.Iterator productIterator();
    public boolean canEqual(java.lang.Object);
    public int hashCode();
    public java.lang.String toString();
    public boolean equals(java.lang.Object);
    public Person(java.lang.String);
}
```

REPL - tricks

- **TAB completion** works!

- "**sbt console**"
  - Prepares **classpath** for *your project's classes*

- **:paste** - "paste mode", try examples from slides
- **:history** - typed in code
- **:javap** - to see generated Java interface
  - **:javap -v** - to see generated bytecode

- **:help**

REPL - tricks

Experiment in it!

Basics

+

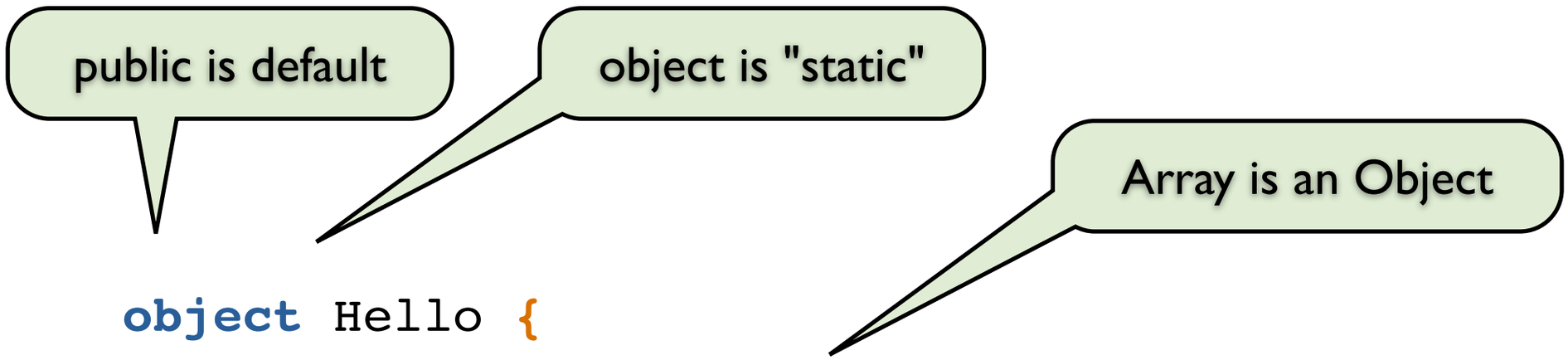
Hello World

Hello World

Hello World

```
object Hello extends App {

  println ("Hello World!")

}
```

The Scala/Java Way




```
object Hello {

  def main(args: Array[String]) {
    println ("Hello World!")
  }

}
```

Hello World

public is default

no ;

```
object Hello extends App {

  println ("Hello World!")

}
```

Hello World

Type Inference

"final String"

```
object Hello extends App {

  val text: String = "Hello World!"

  println (text)

}
```

Hello World

```
object Hello extends App {

  val text = "Hello %s!".format("世界!")

  println (text)

}
```

java.lang.String

That's not from String!

Hello Scala World

Some core concepts:

- **No static**, just **object**
- **Embrace val**,
  - try not using **var**

Basics

Val = Value

```
val name = "Konrad"
```

Val = Value

Type Inference

```
val name = "Konrad"
```

```
val name2: String = "Konrad"
```

"Type Annotation"

Val = Value

```
val name = "Konrad"
```

```
name = "Zbigniew" // compile error
```

"reassignment to val"

Var = Variable

```
var name = "Konrad"
```

Var = Variable

```
var name = "Konrad"
name = "Zbigniew" // OK!
```

Var = Variable

```
var name = "Konrad"
name = "Zbigniew" // OK!
```

Scala Style: Avoid var!

def = define

```
def name()
```

def = define

```
// Ugly
def name(): String = {
   return "Bob"
}
```

```
// "Scala Style"
def name() = "Bob"
```

{} are optional for "one-liners"

type is inferred

avoid return - it's optional

def = define

Type Inference

```
def name() = "Bob"
```

method, always returning "Bob"

def = define

Without Type Inference

```
def name(): String = "Bob"
```

def = define

```
def name() = "Bob"

val got = name()
```

simple method call

def = define

```
def name() = "Bob"

val got  = name()
val got2 = name
```

() is optional here!

def = define

```
def name = "Bob"
```

Style: *"this method has no side effects"*

```
val got  = name
val got2 = name()
```

won't compile.

def = define

required type annotation

```
def greet(name: String) =
  "Hello, " + name + "!"
```

```
greet("Bob") should equal ("Hello, Bob!")
```

def = define

```
def concat(a: String, b: String) =
  a + b
```

abstract def

```
abstract class Example {
  def concat(a: String, b: String): String
}
```

has no implementation = **abstract**

has abstract def => must be **abstract**

abstract def

```
abstract class Example {
  def concat(a: String, b: String): String
}
```

there is not "**abstract def**", it's inferred

functions and void

In Scala, methods are Functions.
Functions always return something.

What about **void**?

Scala has **no void return type.**

Unit, the void of Scala

value of this function = result of block

```
def doNothing() = {
  // return type = anything that's last here
}
```

value of this function is **meaningless**, it's Unit

```
def doNothing() {
  // return type = Unit
}
```

```
def doNothing(): Unit = {
  () // "Unit"
}
```

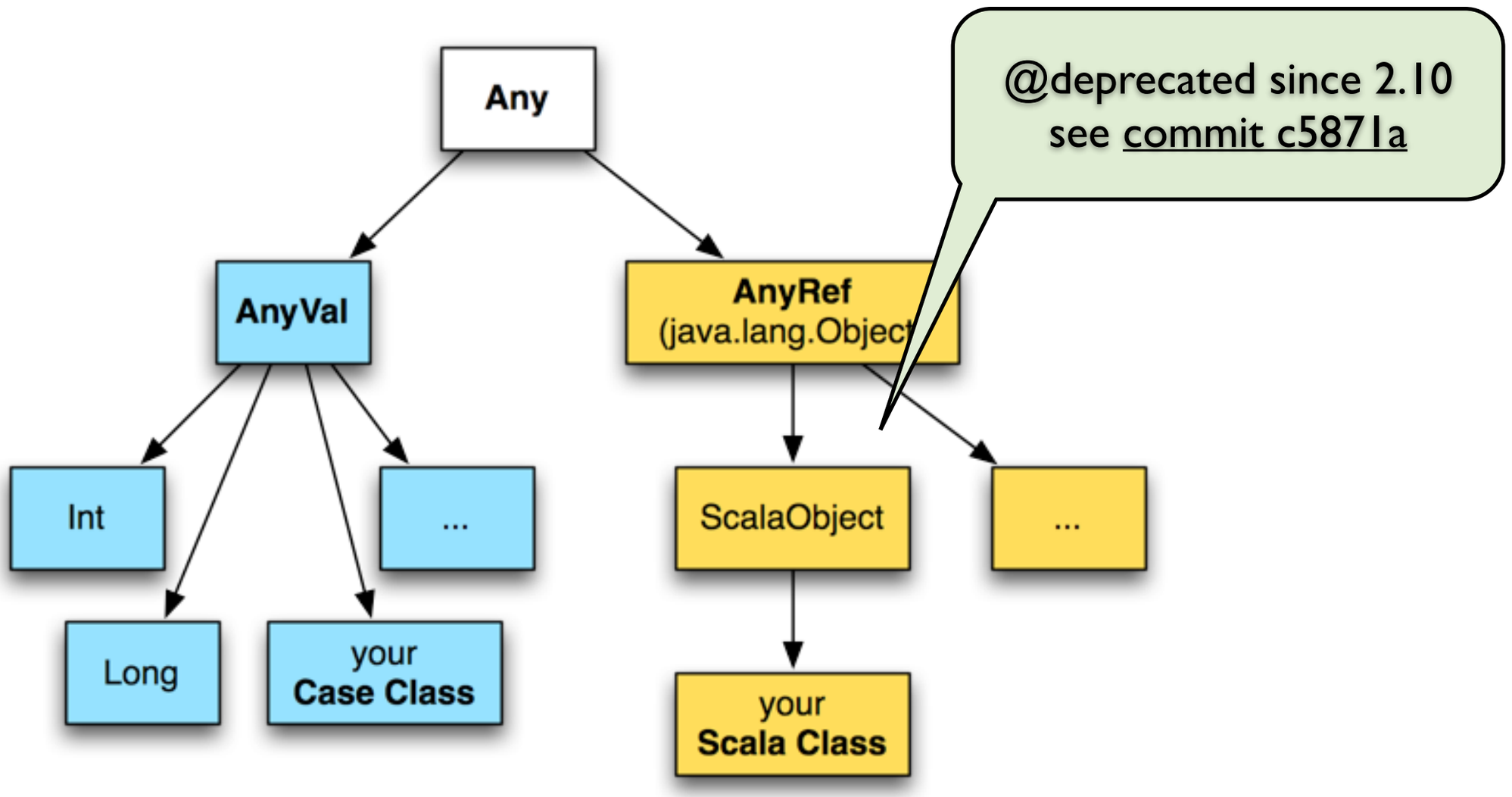
you can explicitly **return Unit**!

everything is an object

```
val s: String = 34.toString
```

```
s should equal ("34")
```

everything is an object



everything is an object

```
val s: String = (34: Int).toString
```

```
s should equal ("34")
```

everything is an object

```
12 + 13
```

```
12.+(13)
```

def +(b: Int)

everything is an object

```
final abstract class Int extends /*...*/ {

  // ...

  def +(x: Int): Int
}
```

everything is an object

Does Scala have *operator overloading*?
No.

Does Scala have *funny method names*?
Yes.

Int in runtime is "int"

Compiler tricks with numbers:

- **scala.Int** is "**JVM int**", not Integer
- **scala.Long** is "**JVM long**", not Long

But:

- the compiler adds sugar, to make it act like "scala.Int" *the object*

- **Win:** int is small / cheap!

Basics

Building blocks, so far:

- **val / var** - value / variable
- **def** - **def**ine a method

Option

Option

Option[A]

- Our first **monad**!
- Never again *NullPointerException*
- Just like:
- Java: Guava's Optional<T>
- Haskell: **Maybe**Monad

Option = Some / None

```
// simplified
sealed abstract class Option[T]

class Some[T](value: T) extends Option[T]

class None[T]           extends Option[T]
```

Option[A]

```
val maybeName = Option("caplin")
```

Option[A]

```
val maybeName = Option("caplin")

val maybeName2 = Some("caplin")
```

Option[A]

```
val maybeName = Option("caplin")

val maybeName2: Option[String] = Some("caplin")
```

getOrElse

```
val maybeName = Option("caplin")
```

```
val name = maybeName.getOrElse("no name")
```

getOrElse

```
val maybeName = Option("caplin")
```

```
val name = maybeName.getOrElse("no name")
```

String

Option[String

String

getOrElse

```scala
val maybeName = Option("caplin")

val name = maybeName.getOrElse("no name")

// scala style tip:
val nnnn = maybeName getOrElse "no name"
```

"." is optional

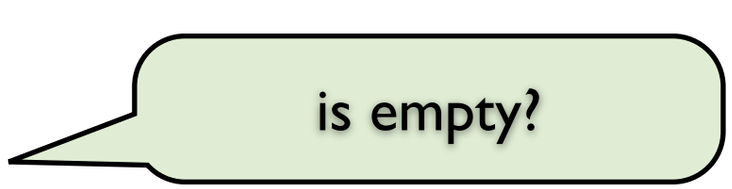
"()" is optional

Option[A]

```
// Java - Style... (Don't do this)

val maybeName = Option("caplin")

val name =
  if(maybeName.isDefined) {
    maybeName.get
  } else {
    "No Name"
  }
```

Option[A]

```
// Java - Style... (Don't do this)

val maybeName = Option("caplin")

val name =
  if(maybeName.isEmpty) {
    maybeName.get
  } else {
    "No Name"
  }
```



Option[A]

```
// Java - Style... (Don't do this)

val maybeName = Option("caplin")

val name =
  if(maybeName.isEmpty) {
    maybeName.get
  } else {
    "No Name"
  }
```

NoSuchElementException

```
java.util.NoSuchElementException: None.get
        at scala.None$.get(Option.scala:313)
```

option foreach {}

```
val maybeName = Some("caplin")

maybeName foreach { name =>
  println(name)
}
```

Option foreach {}

```scala
val maybeName = Some("caplin")

maybeName foreach { name =>
  println(name)
}
```

```scala
maybeName foreach println
```

option map {}

```scala
val maybeName = Some("caplin")

val m: Option[String] = maybeName map { name => name }
```

option map {}

```
val maybeName = Some("caplin")

val m: Option[String] = maybeName map { _.toUpperCase }

m.get should equal ("CAPLIN")
```

option map {}

```
val maybeName = Some("caplin")

val m: Option[String] = maybeName map { _.toUpperCase }

m.get should equal ("CAPLIN")
```

option match {}

```scala
val maybeName = Some("caplin")

maybeName match {
  case Some(name) => println(name)
  case None       => // ...
}
```

option match {}

```scala
val maybeName = Some("caplin")

maybeName match {
  case Some(name) => println(name)
  case _          => // ...
}
```

"Option Wall"

```
val maybeName = Some("caplin")
val maybeSurname = None

for {
  name    <- maybeName
  surname <- maybeSurname
} {
  println(s"$name $surname")
}
```

"Option Wall"

```
val maybeName = Some("caplin")
val maybeSurname = None

for {
  name    <- maybeName
  surname <- maybeSurname
  if name.length > 0
  if surname.length > 0
} {
  println(s"$name $surname")
}
```

"Option Wall"

```scala
val maybeName = Some("caplin")
val maybeSurname = None

val fullName: Option[String] = for {
  name    <- maybeName
  surname <- maybeSurname
  if name.length > 0
  if surname.length > 0
} yield s"$name $surname"
```

option flatMap {}

```
val option = Option(0)
```

like **map** but you return **Option**

```
Option[Int]: flatMap( Int => Option[Int] )
```

```
val mapped: Option[Int] = option flatMap {
  num: Int => Some(num)
}
```

```
val mapped = option flatMap {
  num => /*doTo(num)*/ Some(num)
}
```

we'll re-examine flatMap with Lists

Option[A]

Style guide:

- **match** - you care about both **Some** and **None**
  - ommit **case None**, use **case** _ instead

- **foreach** - you **do something when Some**

- **map** - you **return Some**(mapped) **or None**