Scala in Practice

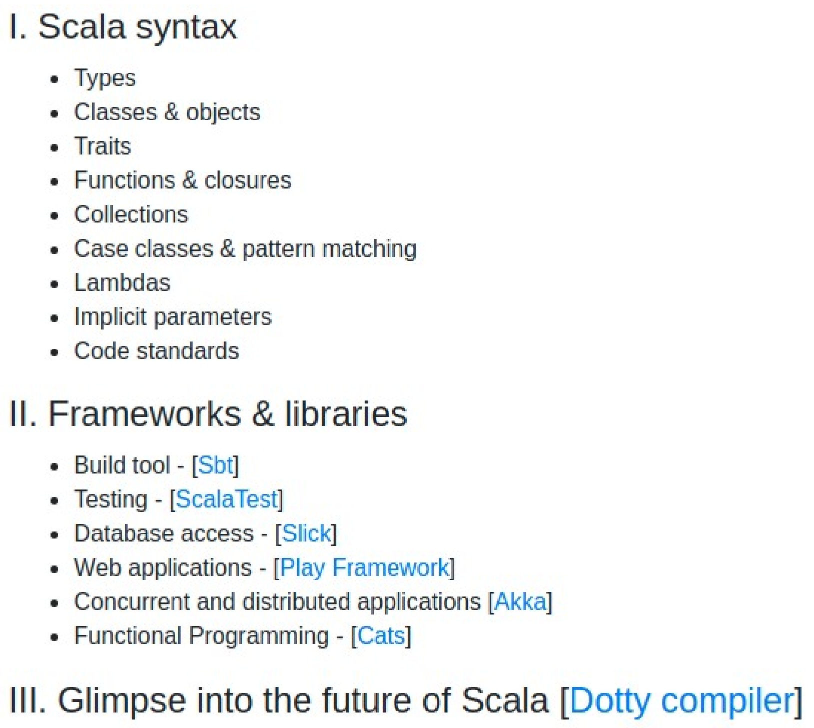
Lecturer

www.ii.uni.wroc.pl/~kowalczykiewicz

Course Goal

- Basic knowledge of Scala language
- Overview of frameworks & libraries
- Practical tips & tricks

Course syllabus



I. Scala syntax

- Types
- Classes & objects
- Traits
- Functions & closures
- Collections
- Case classes & pattern matching
- Lambdas
- Implicit parameters
- Code standards

II. Frameworks & libraries

- Build tool - [Sbt]
- Testing - [ScalaTest]
- Database access - [Slick]
- Web applications - [Play Framework]
- Concurrent and distributed applications [Akka]
- Functional Programming - [Cats]

III. Glimpse into the future of Scala [Dotty compiler]

Labs

- Grading rules
  - Remote labs (I & II): **code review.** The **only** way to gain points is to send the code to my email (***michal.kowalczykiewicz@cs.uni.wroc.pl***)
  - Stationary lab (III): **code walkthrough** during lab hours
  - Each list with exercises is worth in total:
    - **10** points when finished before deadline
    - **5** points when finished one week after deadline
    - later submissions will **not be graded**

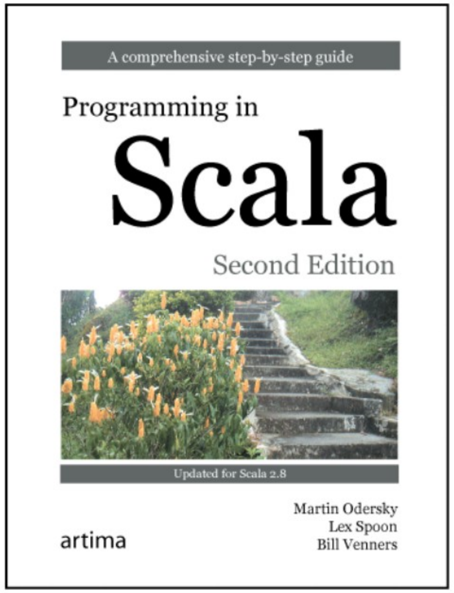
- Exercises will be here: ***~kowalczykiewicz/scala***

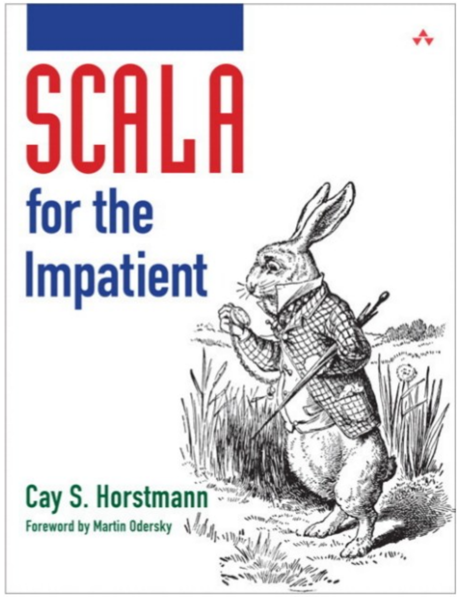
- *Ad hoc* consultations

Percentage	Grade
50%	3
60%	3.5
70%	4
80%	4.5
90%	5

Uniwersytet Wrocławski

Recommended books





Online resources

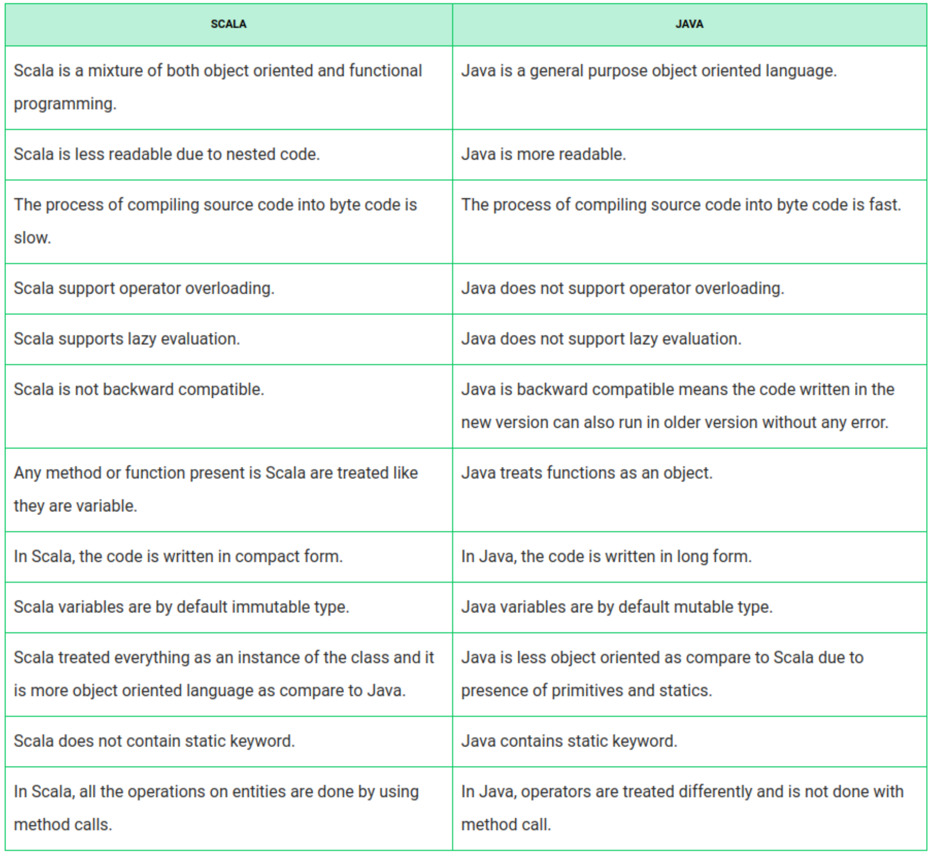
docs.scala-lang.org/api

Scala Summary

- Created by Martin Odersky (2001)
- General purpose, high-level language
- Running on JVM
- Functional & object oriented
- Static typing

Uniwersytet Wrocławski

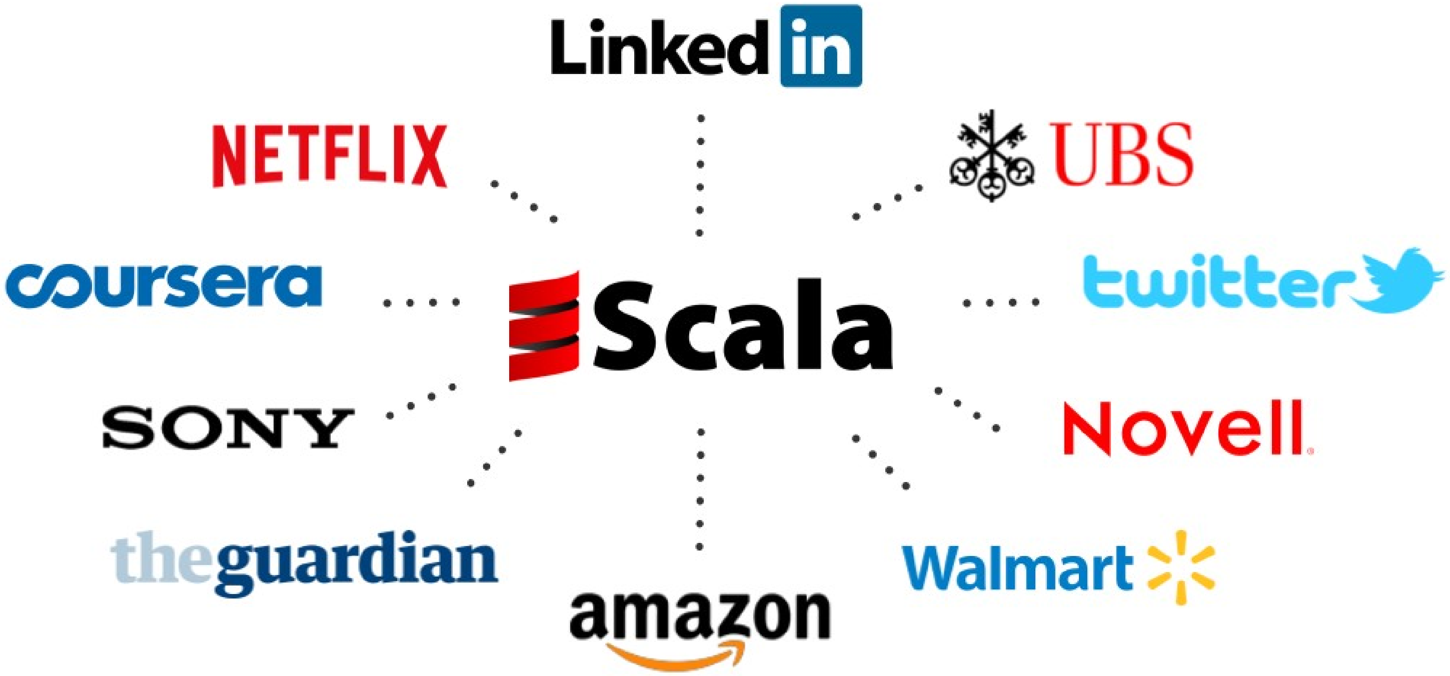
Java vs. Scala



SCALA	JAVA
Scala is a mixture of both object oriented and functional programming.	Java is a general purpose object oriented language.
Scala is less readable due to nested code.	Java is more readable.
The process of compiling source code into byte code is slow.	The process of compiling source code into byte code is fast.
Scala support operator overloading.	Java does not support operator overloading.
Scala supports lazy evaluation.	Java does not support lazy evaluation.
Scala is not backward compatible.	Java is backward compatible means the code written in the new version can also run in older version without any error.
Any method or function present is Scala are treated like they are variable.	Java treats functions as an object.
In Scala, the code is written in compact form.	In Java, the code is written in long form.
Scala variables are by default immutable type.	Java variables are by default mutable type.
Scala treated everything as an instance of the class and it is more object oriented language as compare to Java.	Java is less object oriented as compare to Scala due to presence of primitives and statics.
Scala does not contain static keyword.	Java contains static keyword.
In Scala, all the operations on entities are done by using method calls.	In Java, operators are treated differently and is not done with method call.

https://www.geeksforgeeks.org/scala-vs-java

Industry


LinkedIn
NETFLIX
UBS
coursera
Scala
twitter
SONY
Novell
theguardian
amazon
Walmart

REPL

```
$ scala
Welcome to Scala 2.12.1 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_191).
Type in expressions for evaluation. Or try :help.

scala > 40 + 2
res0: Int = 42
scala > res0 * 2
res1: Int = 84
```

Scripts

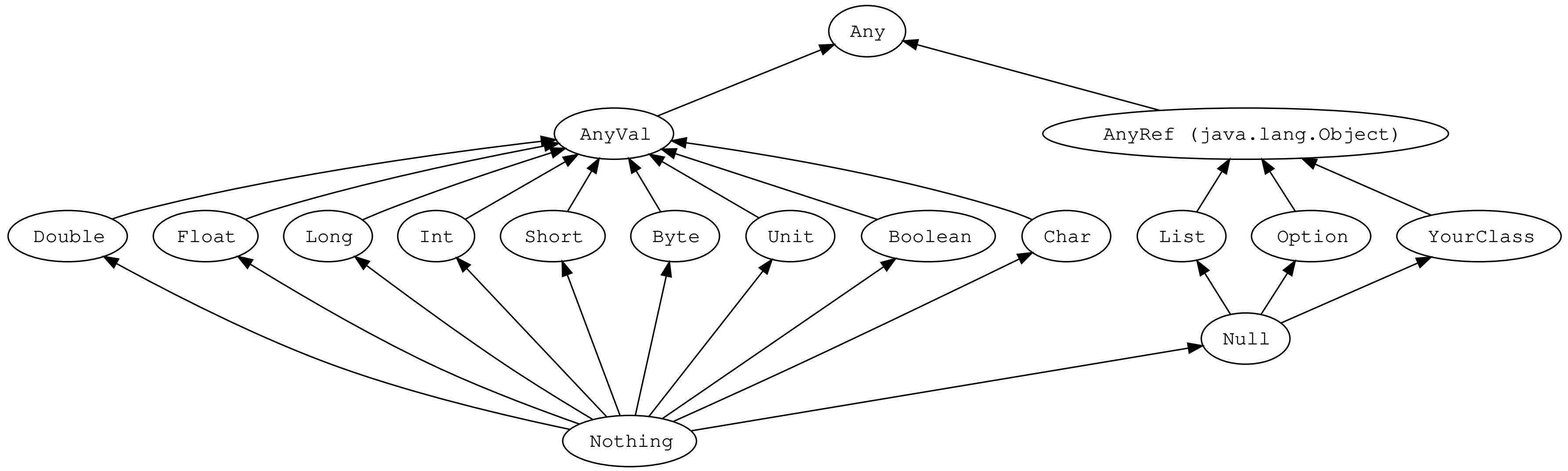
```
bash$ echo ‘println("Hello World")’ > file.scala

$ scala file.scala
Hello World
```

Compile/execute

SomeClass.scala → scalac → SomeClass.class → scala

Basic types hierarchy



Variables

```
scala > var msg: String = "Hello "
msg: String = "Hello "

scala > var msg2 = " World"
msg2: String = " World"
```

Vars can be reassigned

```
scala > var number = 3
number: Int = 3

scala > var number = number + 3
number: Int = 6

scala > number = "Hello"
<console>:12: error: type mismatch;
 found   : String("Hi")
 required: Int
       number = "Hi"
                ^
```

Vals can’t be reassigned

```
scala > val number = 42
number : Int = 42

scala > number = 23
< console >:12: error : reassignment to val
number = 23
       ^
```

Uniwersytet Wrocławski

Integer types

Byte	8-bit signed integer
Short	16-bit signed integer
Int	32-bit signed integer
Long	64-bit signed integer
Char	16bit Unicode character (unsigned)

```
scala > val x = ’s ’; val y = 0 x10
x : Char = s
y : Int = 16

scala > -x + y
res2 : Int = -99
```

Uniwersytet Wrocławski

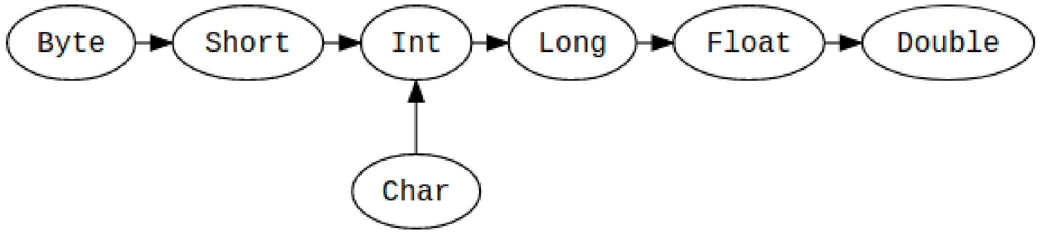
Floating types

`Float`	32-bit single precision float
`Double`	64-bit single precision float

```
scala > val y = 42E-4; val x = 32.0
y : Double = 0.0042
x : Double = 32.0

scala > x * y
res8 : Double = 0.134
```

Type casting



```
val x: Long = 987654321
val y: Float = x  // 9.8765434E8
val number: Int = '☺'  // 9786
val z: Long = y
<console>:12: error: type mismatch;
 found   : Float
 required: Long
       val z: Long = y
```

String types

```
scala > val hello = "Hello"
hello : String = Hello
```

```
scala > val msg: java.lang.String = "Hello"
msg : java.lang.String = Hello
```

Booleans

```
scala > x < 3+2
res3 : Boolean = false

scala > val y = "Hello"
y : String = Hello

scala > y == "Hello"
res4 : Boolean = true
```

Uniwersytet Wrocławski

if-conditions

if (*condition*) *expression* [else if (*condition*) *expression*] [else *expression*]

```
if (a % 2 == 0) {
  println ("a is even")
} else if(a % 3 == 0) {
    println("a is a multiple of 3")
} else println("none of the above")
```

Expressions

- There are no statements in Scala
- Every block of code returns a value. This value can be Unit
- Compound expressions (with {...} return result of last expression)
- Type of expression is automatically inferred (or can make it explicit)

```
scala > val z : Int = { val x = 4; val y = 2; x / y }
z : Int = 2

scala > val a = { val b = 4; }
a : Unit = ()
```

Expressions

```
scala > val z : Boolean = if (someBoolCondition) true else ()
< console >:7: error : type mismatch ;
found: Unit
required : Boolean
val z : Boolean = if (someBoolCondition) true
                               ^

scala > val z = if (someBoolCondition) true
z : AnyVal = true
```

Loops – while & do

```
scala > var x = 1
scala > while ( x <= 3) { println ( x ); x += 1 }
1
2
3

scala > x
res1 : Int = 4
scala > do { x +=1; println ( x ) } while (x <= 5)
5
6
```

Methods

$$\boxed{\text{def } \mathit{name}(\mathit{param}_1\text{: } \mathit{type}_1, \cdots \mathit{param}_n\text{: } \mathit{type}_n) : \mathit{return\_type} = \mathit{body}}$$

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

Methods

$$\boxed{\text{def } \mathit{name}(\mathit{param}_1\text{: } \mathit{type}_1, \cdots \mathit{param}_n\text{: } \mathit{type}_n) : \mathit{return\_type} = \mathit{body}}$$

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

max : ( x: Int, y: Int )Int
```

Methods

```
scala > max(2, 3)
res1 : Int = 3

scala > def greet() = println( "Hello")
greet : () Unit

scala > greet
Hello
```

All types are classes

```
scala > 31.toString
res0 : String = 31
```

All operators are methods

```
scala > 3 .+(2)
res1 : Int = 5

scala > 3 .==(5)
res2 : Boolean = false

scala > "fortunate". contains ( "tuna")
res3 : Boolean = true

scala > "fortunate" contains "tuna"
res4 : Boolean = true
```

Lists

```
scala > val numbers = Nil
numbers : scala.collection.immutable.Nil.type = List()

scala > val numbers = 1 :: 2 :: Nil
numbers : List [ Int ] = List (1 , 2)

scala > val numbers2 = List (3, 4, 5)
numbers2 : List [ Int ] = List (3, 4, 5)

scala > numbers ::: numbers2
res2 : List [ Int ] = List (1 , 2, 3, 4, 5)
```

Lists

```
scala > 1 :: List(2, 3)
res18 : List [ Int ] = List (1, 2, 3)

scala > List (2, 3).::(1)
List [ Int ] = List (1, 2, 3)
```

Immutable objects

```
scala > val numbers = List (1, 2, 3, 4)
numbers : List [ Int ] = List (1, 2, 3, 4)

scala > numbers(3)
res1 : Int = 4

scala > numbers(3) = 100
< console >:9: error : value update is not a member of List [ Int ]
numbers(3) = 100
```

Mutable objects

```
scala > val a = Array(1, 2, 3, 4)
a : Array [ Int ] = Array(1, 2, 3, 4)

scala > a(3) = 100


scala > a
res2 : Array [ Int ] = Array(1, 2, 3, 100)
```

For loops

```
scala > for (y <- List (1, 2, 3)) { println ( y ) }
1
2
3
```

Range objects

```
scala > 1 to 10
res1 : scala.collection.immutable.Range.Inclusive =
Range (1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10)

scala > 10 to (0, -2)
res2 : scala. collection.immutable.Range.Inclusive =
Range (10 , 8 , 6 , 4 , 2 , 0)

scala > for ( i <- 1 to 3) println ( i * 2)
2
4
6
```

For as an expression

```
for (seq) yield expression
```

```
scala > for ( x <- List (1, 2, 3) ) yield x * 2
res8 : List [ Int ] = List (2, 4, 6)

scala > for { x <- 1 to 7
               y = x % 2
               if ( y == 0 )
             } yield {
                  println ( x )
                  x
               }
2
4
6
res1 : scala.collection.immutable.IndexedSeq[ Int ] =
Vector (2 , 4 , 6)
```

For as an expression

```
scala > for { x <- List (1, 2, 3)
              y <- List (4, 5)
            } yield x * y
res10 : List [ Int ] = List (4, 5, 8, 10, 12, 15)
```