Uniwersytet Wrocławski

# Scala in Practice

Uniwersytet Wrocławski

# Class structure

```
class SomeClassName(. . .) {
  . . .
}

new SomeClassName(...)
```

Uniwersytet Wrocławski

# Rational numbers

```
scala > val oneHalf = new Rational (1, 2)
oneHalf: Rational = 1/2

scala > val twoThirds = new Rational (2, 3)
oneHalf: Rational = 2/3

scala > val newRational = ( oneHalf / 7) + (1 - twoThirds )
newRational: Rational = 17/42
```

Uniwersytet Wrocławski

# Class definition

```
scala > class Rational ( n: Int, d: Int )
defined class Rational

scala > val oneHalf = new Rational (1 , 2)
oneHalf : Rational = Rational@58c1a471
```

Uniwersytet Wrocławski

# Default constructor

```
class Rational ( n: Int, d: Int ) {
  println ( " Created " + n + " / " + d )
}
```

Uniwersytet Wrocławski

# Adding methods

```
class Rational ( n : Int , d : Int ) {

  def +( other : Rational ): Rational = {
    val newN = n * other.d + other.n * d
    val newD = d * other.d
    new Rational ( newN, newD )
  }
}
```

```
scala >...
error: value d is not a member of this.Rational val newN = n * other.d + other.n * d
```

Uniwersytet Wrocławski

# Val arguments

```
class Rational ( val n : Int , val d : Int ) {

  def +( other : Rational ): Rational = {
    val newN = n * other.d + other.n * d
    val newD = d * other.d
    new Rational ( newN, newD )
  }
}

scala > new Rational (1, 2) + new Rational (3, 4)
res4 : Rational = Rational@2a491adf
```

Uniwersytet Wrocławski

# Default & named values

```
class Rational ( val n : Int , val d : Int = 1 ) {
   . . .
}

scala > new Rational (5)

scala > class Rational (n = 5)
```

Uniwersytet Wrocławski

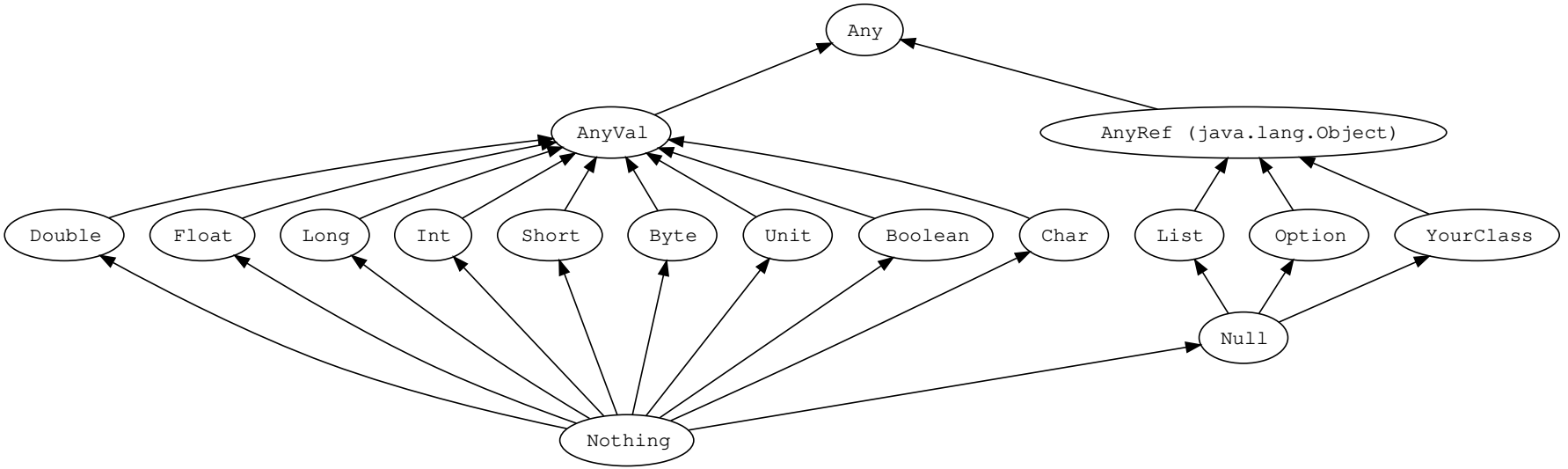
# Auxiliary constructors

```
class Rational ( val n : Int , val d : Int ) {
  def this ( n : Int ) = this (n , 1)
  . . .
}

scala > new Rational (5)
res0: Rational = Rational@528f8f8b
```

Uniwersytet Wrocławski

# Basic types hierarchy



Uniwersytet Wrocławski

# Overriding methods

```
class Rational ( val n : Int , val d : Int ) {
  . . .
  override def toString = n + " / " + d
}

scala > new Rational (1 , 2) + new Rational (3, 4)
res4 : Rational = 10 / 8
```

Uniwersytet Wrocławski

# Class fields

```
class Rational ( n : Int , d : Int ) {
  def gcd ( a : Int , b : Int ) : Int = if ( b ==0) a else gcd (b , a % b )
  val g = gcd ( n.abs , d.abs )

  val numer = n / g
  val denom = d / g
  . . .
  override def toString = numer + " / " + denom
}

scala > new Rational (2, 4)
res4 : Rational = 1 / 2
```

Uniwersytet Wrocławski

# Private scope

```
class Rational ( n : Int , d : Int ) {
  private def gcd ( a : Int , b : Int ) : Int = if ( b ==0) a else gcd (b , a % b )
  private val g = gcd ( n. abs , d. abs )

  val numer = n / g
  val denom = d / g
  . . .
  override def toString = numer + " / " + denom
}

scala > new Rational (2, 4)
res4 : Rational = 1/ 2
```

Uniwersytet Wrocławski

# Explicit type

```
class Rational ( n : Int , d : Int ) {
  private def gcd ( a : Int , b : Int ) : Int = if ( b ==0) a else gcd (b , a % b )
  private val g = gcd ( n. abs , d. abs )

  val numer : Int = n / g
  val denom : Int = d / g
  . . .
  override def toString: String = numer + " / " + denom
}

scala > new Rational (2, 4)
res4 : Rational = 1/ 2
```

Uniwersytet Wrocławski

# Working version

```
scala > val oneHalf = new Rational(1, 2)
oneHalf : Rational = 1/2

scala > val twoThirds = new Rational(2, 3)
oneHalf : Rational = 2/3

scala > ( oneHalf / 7) + (1 - twoThirds )
res0 : Rational = 17/42
```

Uniwersytet Wrocławski

# Singleton objects

- There are no static methods in Scala
- Scala supports *singleton objects*
- Use cases:
  - Single point access to a common resource (large data structures…)
  - Repository for utility methods
  - Companion objects for classes (same name as Class) to define “static” methods and factories
  - Scala applications entry point

# Object structure

```
object SomeObjectName {
  . . .
}

scala > new SomeObjectName
<console>:12: error: not found: type SomeObjectName
       new SomeObjectName
           ^
```

Uniwersytet Wrocławski

# As Singleton example

```
object RationalSummer {
  var sum: Double = 0.0

  def add ( r: Rational ): Double = {
    sum += r.numer.toDouble
    sum
  }
}
```

Uniwersytet Wrocławski

# As Utility / Helper example

```
object Logger {
  def error(message: String): Unit = . . .

  def warn(message: String): Unit = . . .

  def info(message: String): Unit = . . .

  def debug(message: String): Unit = . . .
}
```

Uniwersytet Wrocławski

# As Factory example

```
object RationalFactory {
  val zero: Rational = new Rational(0, 1)

  val one: Rational = new Rational(1, 1)

  def double(number: Rational): Rational = number + number

  . . .
}
```

Uniwersytet Wrocławski

# As Companion object example

```
class Rational {...}

object Rational {
  def invertRational ( r: Rational ) = new Rational( r.d, r.n )
  def giveGcd ( r: Rational ) = r.g
  def apply ( n: Int , d: Int ) = new Rational ( n, d )
  def apply ( n: Int ) = new Rational ( n )
}

scala > Rational.invertRational ( r )
scala > Rational (1, 2) + Rational (3)
```

Uniwersytet Wrocławski

# As Application entry example

```
object MyApplication {
  def main ( args: Array [String]) {
    println ( Rational (1, 2) + Rational (2, 3))
  }
}
```

```
scala > scalac FractionApp.scala
scala > scala FractionApp
7 / 6
```

Uniwersytet Wrocławski

# As Application entry example

```
object MyApplication extends App {
    println(args)
    println ( Rational (1, 2) + Rational (2, 3))
}
```

Uniwersytet Wrocławski

# Inheritance

```
class Rectangle ( w : Double , h : Double ) {
  def area = w * h
  val description = " Rectangle "
}

class Square ( w: Double ) extends Rectangle ( w, w ) {
  override val description = " Square "
}
```

Uniwersytet Wrocławski

# Abstract classes

```
abstract class Shape {
  def area : Double
  val description : String
  override def toString = description + " , size : " + area
}

class Rectangle ( w : Double , h : Double ) extends Shape {
  def area = w * h
  val description = " Rectangle "
}

class Square ( w : Double ) extends Rectangle (w , w ) {
  override val description = " Square "
}

scala > val x = new Square (3)
x : Square = Square , size : 9.0
```

Uniwersytet Wrocławski

# Final members

```
abstract class Shape {
  def area : Double
  final val description : String = " Shape "
  override def toString = description + " , size : " + area
}

class Blob extends Shape {
  val area : Double = 12
  val description = " Blob "
}
```

```
< console >:10: error : overriding value description in class Shape of type String ; value description
cannot override final
```

Uniwersytet Wrocławski

# Polymorphism

```
scala > val x = new Rectangle (2, 3)
x : Rectangle = Rectangle , size : 6.0

scala > val y = new Square (5)
y : Square = Square , size : 25.0

scala > val z = new Blob
z : Blob = Blob , size : 12.0

scala > val l : List  = List (x, y, z)
l : List [ Shape ] = List ( Rectangle , size : 6.0 , Square , size : 25.0 , Blob , size : 12.0)

scala > for (x < - l ) println ( x . description + " " + x . area )
Rectangle 6.0
Square 25.0
Blob 12.0
```

Uniwersytet Wrocławski

# Generics

```
class Stack[ T ] {
  var elems : List [ T ] = Nil
  def push ( x : T ) { elems = x :: elems }
  def top : T = elems.head
  def pop () { elems = elems.tail }
}

scala > val stackInt = new Stack[Int]
stackInt = Stack@38c9e0d6
```

Uniwersytet Wrocławski

# Packages

- Modularize programs, so that parts of it can be re-used
- Package are special objects that define a set of member classes, objects and other packages

Uniwersytet Wrocławski

# Package structure

*package someProjectName.someModuleName....*

*myProject.calculus*

Uniwersytet Wrocławski

# File beginning

```
package myProject.calculus

class Rational {...}
. . .
```

Uniwersytet Wrocławski

# Multiple packages in one file

```
package  myProject.calculus {
  class Rational {...}
}

package ... {
  ...
}
```

Uniwersytet Wrocławski

# Nested packages

```
package myProject {
    package calculus {
      class Rational
    }

    package tests {
      class RationalSuite
    }
}
```

Uniwersytet Wrocławski

# Import

*val rationalNum = new myProject.calculus.Rational(...)*

*import myProject.calculus.Rational*
*val rationalNum = new Rational(..)*

*import myProject.calculus._*
*val rationalNum = new Rational(...)*

*import myProject._*
*val rationalNum = new calculus.Rational(...)*

Uniwersytet Wrocławski

# Import tricks

```
import myProject.calculus.{ Rational, SomeDifferentClass,... }
val rationalNum = new Rational(..)

import myProject.calculus.{ Rational => RationalNumber }
val rationalNum = new RationalNumber(...)

import myProject.calculus.RationalSummer.add
val navigator = add(...)
```

Uniwersytet Wrocławski

# Import anywhere

```
def printAndAddFraction ( r : Rational ) = {
  println (r)
  import myProject.calculus.RationalSummer.add
  val total = add (r)
}
```

Uniwersytet Wrocławski

# Import object's fields

```
def printRational ( r : Rational ) = {
  import r._
  println ( numer + ", " + denom)
}
```

Uniwersytet Wrocławski

# Scoping

*public ...*

*protected ...*

*private ...*

Uniwersytet Wrocławski

# Object-private scope

```
class Rational(...) {
    ...
    private[this] def someFunction(...) = {...}
    def doX(other: Rational) {
        other.someFunction(...)  // this line won't compile
    }
}
```

Uniwersytet Wrocławski

# Package scope

```
package myProject.calculus {
    class Rational(...) {
        . . .
        private[calculus] def someFunction(...) {...}
    }
    class SomeDifferentClassInTheSamePackage {
        val r = new Rational(...)
        r.someFunction(...)  // compiles
    }
}
```