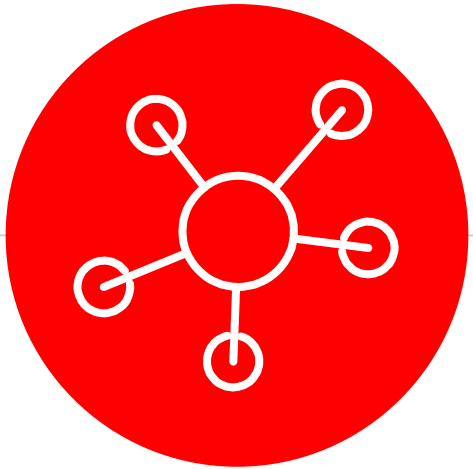
# An introduction to Scala

by Nikos Marousis


- Multi-paradigm programing
- Compiles to Java bytecode
- Runs on the JVM

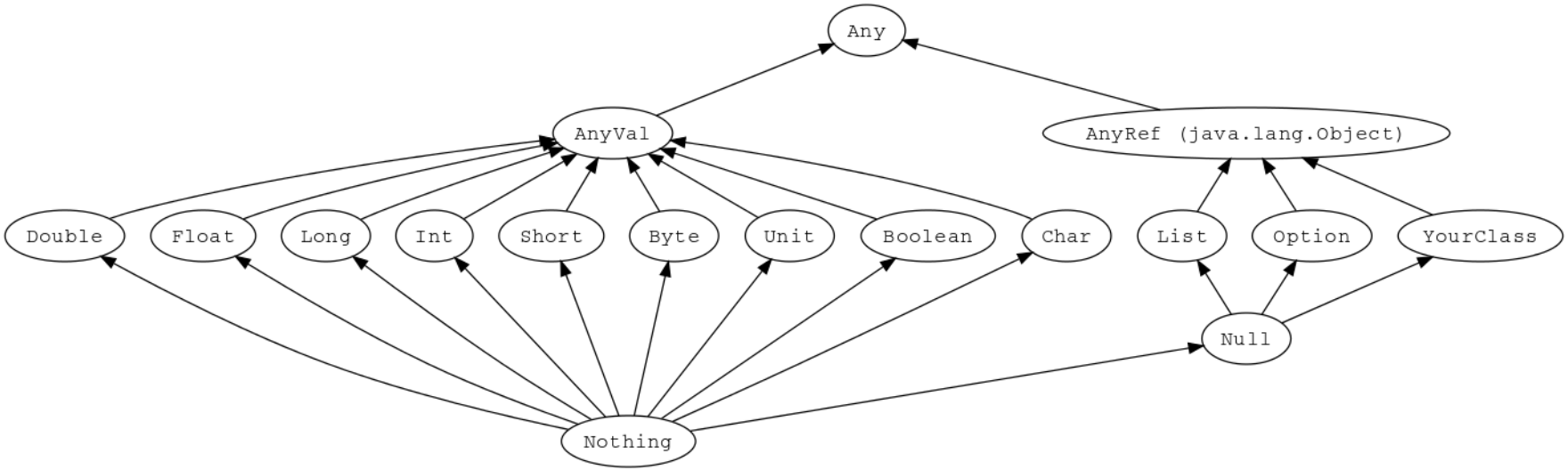
*The name Scala stands for "scalable language". The language is so named because it was designed to grow with the demands of its users. You can apply Scala to a wide range of programming tasks, from writing small scripts to building large systems.*



# Object-oriented

*Scala is a pure object-oriented language in the sense that everything is an object, including numbers or functions. It differs from Java in that respect, since Java distinguishes primitive types from reference types, and does not enable one to manipulate functions as values.*

# Object-oriented



# Object-oriented

**Mixins**

Multiple inheritance via Mixins, adding functionality to classes while omitting any common components.

**Traits**

Types that contain fields and methods, similar to Java interfaces, and cannot be instantiated.

**Generic classes**

Generic class support.

**Overloading**

Support for method overloading, based on both parameter type and parameter count.

**Anonymous classes**

Ability to create single-use instantiations of an anonymous class that extends another (abstract) class.

**Case Classes**

Immutable objects containing data. They facilitate easier comparison and copying.

# Traits

```
trait Iterator[A] {
  def hasNext: Boolean
  def next(): A
}

class IntIterator(to: Int) extends Iterator[Int] {
  private var current = 0
  override def hasNext: Boolean = current < to
  override def next(): Int =  {
    if (hasNext) {
      val t = current
      current += 1
      t
    } else 0
  }
}
```

# Mixins

```
abstract class A {
  val message: String
}

class B extends A {
  val message = "Class B"
}

trait C extends A {
  def loud = message.toUpperCase()
}

class D extends B with C
```

```
val d = new D

println(d.message)
// Class B

println(d.loud)
// CLASS B
```

# Generic classes

```
class Stack[A] {
  private var elements:List[A] = Nil

  def push(x: A) {
    elements = x :: elements
  }

  def peek: A = elements.head

  def pop(): A = {
    val currentTop = peek
    elements = elements.tail
    currentTop
  }
}
```

```
class Fruit
class Apple extends Fruit
class Banana extends Fruit

val stack = new Stack[Fruit]
val apple = new Apple
val banana = new Banana

stack.push(apple)
stack.push(banana)
```

# Case classes

```
case class Message(
  sender: String,
  recipient: String,
  body: String
)
val message2 = Message(
  "jorge@catalonia.es",
  "guillaume@quebec.ca",
  "Com va?"
)
val message3 = Message(
  "jorge@catalonia.es",
  "guillaume@quebec.ca",
  "Com va?"
)
val s = message2 == message3 // true
```

```
val message4 = Message(
  "julien@bretagne.fr",
  "travis@washington.us",
  "Me zo o komz gant ma amezeg"
)
val message5 = message4.copy(
  sender = message4.recipient,
  recipient = "claire@bourgogne.fr"
)

println(message5.sender)
// travis@washington.us
println(message5.recipient)
// claire@bourgogne.fr
println(message5.body)
// Me zo o komz gant ma amezeg
```

# Anonymous classes

```
abstract class Listener { def trigger }

class Listening {
  var listener: Listener = null
  def register(l: Listener) { listener = 1 }
  def sendNotification() { listener.trigger }
}

val notification = new Listening()

notification.register(
  new Listener { def trigger { println(s"Trigger at ${new java.util.Date}") } }
)

notification.sendNotification
// Trigger at Fri Jan 24 13:15:32 PDT 2014
```

# Operator overloading

```
class Complex(val real : Double, val imag : Double) {
  def +(that: Complex) = new Complex(this.real+that.real, this.imag+that.imag)
  def -(that: Complex) = new Complex(this.real-that.real, this.imag-that.imag)
  override def toString = real + " + " + imag + "i"
}

var a = new Complex(4.0,5.0)
var b = new Complex(2.0,3.0)

println(a + b)
// 6.0 + 8.0i

println(a - b)
// 2.0 + 2.0i
```

# Functional

*Scala has full support for first-class functions, higher-order functions, and the use of declarative programming. As with other types of data such as String or Int, functions have types based on the types of their input arguments and their return value.*

# Functional

**Higher order functions**

Functions can be both parameters and return results of other functions.

**Nested methods**

Support for nested method creation.

**Currying**

Support for currying with a concise syntax for curried functions.

**Lazy evaluation**

Optional lazy evaluation of fields (values or function) with beneficial performance impact.

**Function literals**

Support for anonymous functions via the definition of function literals.

**Pattern matching**

Pattern matching for the application of function or operators, with additional support for case classes.

# Higher order functions

```
def urlBuilder(ssl: Boolean, domainName: String): (String, String) => String = {

  val schema = if (ssl) "https://" else "http://"

  (endpoint: String, query: String) => s"$schema$domainName/$endpoint?$query"

}

val domainName = "www.example.com"
def getURL = urlBuilder(ssl=true, domainName)

val endpoint = "users"
val query = "id=1"

val url = getURL(endpoint, query)
// "https://www.example.com/users?id=1"
```

# Pattern Matching

```
abstract class Notification
case class Email(sender: String, title:String, body: String) extends Notification
case class SMS(caller: String, message: String) extends Notification

def showNotification(notification: Notification): String = {
  notification match {
    case Email(email, title, _) =>
      s"You got an email from $email with title: $title"
    case SMS(number, message) =>
      s"You got an SMS from $number! Message: $message"
  }
}

val someSms = SMS("12345", "Are you there?")
println(showNotification(someSms))
// You got an SMS from 12345! Message: Are you there?
```

# Currying

```
def filter(xs: List[Int], p: Int => Boolean): List[Int] =
  if (xs.isEmpty) xs
  else if (p(xs.head)) xs.head :: filter(xs.tail, p)
  else filter(xs.tail, p)

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

val nums = List(1, 2, 3, 4, 5, 6, 7, 8)

println(filter(nums, modN(2)))
// List(2,4,6,8)

println(filter(nums, modN(3)))
// List(3,6)
```

# Function literals

```
def safeStringOp(s: String, f: String => String) = {
  if (s != null) f(s) else s
}

val uuid = java.util.UUID.randomUUID.toString
println(uuid)
// bfe1ddda-92f6-4c7a-8bfc-f946bdac7bc9

val timedUUID = safeStringOp(uuid, { s =>
    val now = System.currentTimeMillis
    val timed = s.take(24) + now
    timed.toUpperCase
  }
)
println(timedUUID )
// BFE1DDDA-92F6-4C7A-8BFC-1394546043987
```

# Nested functions

```
def square(x: Double) = x * x

def sqrt(x: Double) = {
  def sqrtIter(guess: Double): Double =
    if (isGoodEnough(guess)) guess
    else sqrtIter(improve(guess))

  def improve(guess: Double) =
    (guess + x / guess) / 2

  def isGoodEnough(guess: Double) =
    scala.math.abs(square(guess) - x) < 0.001

  sqrtIter(1.0)
}
```

# Lazy evaluation

```
def isPrime(n: Int) = !Range(2,sqrt(n).toInt+1).exists(n % _ == 0)

// Streams are similar to lists, but their tail is evaluated only on demand.
def streamRange(lo: Int, hi: Int): Stream[Int] =
  if (lo >= hi) Stream.empty
  else Stream.cons(lo, streamRange(lo + 1, hi))

println( (streamRange(100000, 1000000) filter isPrime)(1) )
// 100019
// Elapsed time: 17 ms

println( ((100000 to 1000000) filter isPrime)(1) )
// 100019
// Elapsed time: 3654 ms
```

*The two programming styles have complementary strengths when it comes to scalability. Scala's functional programming constructs make it easy to build interesting things quickly from simple parts. Its object-oriented constructs make it easy to structure larger systems and to adapt them to new demands*

# Why Scala?

# Why Scala?

**Java compatible**

Access to the wide range of Java libraries, without any major compromise.

Portability of Scala classes to Java programs.

**Concise syntax**

Strong operators and syntactic schemes facilitate code that shorter and easier to read.

**High-level**

Greater freedom of implementation, via features such as multiple inheritance and currying, facilitating greater abstraction in the implementation of software artifacts.

**Statically typed**

Earlier error recognition during software implementation, more robust software specifications and easier refactoring.

Support for type inference.

# Java compatible

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._

val now = new Date

val df = getDateInstance(LONG, Locale.FRANCE)

println(df format now)
// 26 février 2018
```

# Concise syntax

```java
// this is Java
class MyClass {

  private int index;
  private String name;

  public MyClass(
    int index,
    String name
  ) {
    this.index = index;
    this.name = name;
  }
}
```

```scala
// This is ~~Sparta~~ Scala
class MyClass(
  index: Int,
  name: String
)
```

# High-level

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

// This is Scala
val nameHasUpperCase = name.exists(_.isUpperCase)
```

# Statically typed

```
case class MyPair[A, B](x: A, y: B);

val p = MyPair(1, "scala")
// type: MyPair[Int, String]

def id[T](x: T) = x
val q = id(1)
// type: Int

Seq(1, 3, 4).map(x => x * 2)
// List(2, 6, 8)
```

# Sources and references

- **Tour of Scala & Scala documentation**
- **Learning Scala** – Jason Swartz
- **Programming in Scala First edition** – Martin Odersky, Lex Spoon, Bill Benners
- **Operator Overloading** – Tom's Programming Blog
- **Presentation template** – SlidesCarnival

# Food for code

- Scala Learning Resources
- Scala Exercises
- Scastie – Scala online compiler

# Thanks!

Are there any questions?