Introduction to the Scala language

Dr. Michael Eichberg

First example

-
  ```
  object HelloWorld {
    def main(args: Array[String]) {
      println("Hello, world!")
    }
  }
  ```

- **`object`** denotes a singleton object, a class with only one instance
- **`def`** `main(args: Array[String])` is a procedure method (does not return a value)

Java Interoperabilty

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

  object FrenchDate {
    def main(args: Array[String]) {
      val now = new Date
      val df = getDateInstance(LONG, Locale.FRANCE)
      println(df format now)
    }
  }
  ```
- More powerful import statement
  - Import both Date and Locale from util
  - Wildcard import using _ (all members are imported)

Infix syntax

- Methods taking exactly one argument can be used with infix syntax
  - `df format now`
  - `df.format(now)`
- Both ways are equivalent

Everything is an object!

- Numbers are objects
  - `1 + 2 * 3 / x`
  - `(1).+(((2).*(3))./(x))`
- equivalent!
- +, * are valid identifiers in Scala

Everything is an object!

- Functions are objects

-
  ```
  object Timer {
    def oncePerSecond(callback: () => Unit) {
      while (true) { callback(); Thread sleep 1000 }
    }

    def timeFlies() {
      println("time flies like an arrow...")
    }

    def main(args: Array[String]) {
      oncePerSecond(timeFlies)
    }
  }
  ```

Everything is an object!

- Functions are objects

-
  ```
  object Timer {
    def oncePerSecond(callback: () => Unit) {
      while (true) { callback(); Thread sleep 1000 }
    }

    def timeFlies() {
      println("time flies like an arrow...")
    }

    def main(args: Array[String]) {
      oncePerSecond(timeFlies)
    }
  }
  ```

Everything is an object!

- Functions are objects

-
  ```
  object Timer {
    def oncePerSecond(callback: () => Unit) {
      while (true) { callback(); Thread sleep 1000 }
    }

    def timeFlies() {
      println("time flies like an arrow...")
    }

    def main(args: Array[String]) {
      oncePerSecond(timeFlies)
    }
  }
  ```

Everything is an object!

- Anonymous functions:

-
  ```
  object Timer {
    def oncePerSecond(callback: () => Unit) {
      while (true) { callback(); Thread sleep 1000 }
    }







    def main(args: Array[String]) {
      oncePerSecond(() => println("time flies like an arrow..."))
    }
  }
  ```

Everything is an object!

- Anonymous functions:

-
```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }







  def main(args: Array[String]) {
    oncePerSecond(() => println("time flies like an arrow..."))
  }
}
```

Classes

-
  ```
  class Complex(real: Double, imaginary: Double) {
    def re() = real
    def im() = imaginary
  }
  ```

-
  ```
  val c = new Complex(1.5, 2.3)
  println(c.re())
  ```

- Type Inference:
  - `def re() = real`
  - `def re(): Double = real`

Methods without arguments

-
  ```scala
  class Complex(real: Double, imaginary: Double) {
    def re = real
    def im = imaginary
  }
  ```

-
  ```scala
  val c = new Complex(1.5, 2.3)
  println(c.re)
  ```

Inheritance and Overriding

-
  ```
  class Complex(real: Double, imaginary: Double) {
    def re = real
    def im = imaginary
    override def toString() =
      "" + re + (if (im < 0) "" else "+") + im + "i"
  }
  ```

- Every class has a super-class.
- Defaults to `scala.AnyRef`

val vs. var vs. def

- `val`:
  - immutable variable
- `var`:
  - mutable variable
- `def`:
  - method

val vs. var vs. def

-
  ```
  object Test extends App {
    val val1 = {println("val: 1"); 1}
    def def1 = {println("def: 1"); 1}
    var var1 = {println("var: 1"); 1}

    //val1 = 5 // compile error: immutable
    println(val1 + val1)
    println(def1 + def1)

    var1 = 5
    println(var1 + var1)
  }
  ```

val vs. var vs. def

-
  ```
  object Test extends App {
    val val1 = {println("val: 1"); 1}
    def def1 = {println("def: 1"); 1}
    var var1 = {println("var: 1"); 1}

    //val1 = 5 // compile error: immutable
    println(val1 + val1)
    println(def1 + def1)

    var1 = 5
    println(var1 + var1)
  }
  ```
  - Output:

val vs. var vs. def

-
  ```
  object Test extends App {
    val val1 = {println("val: 1"); 1}
    def def1 = {println("def: 1"); 1}
    var var1 = {println("var: 1"); 1}

    //val1 = 5 // compile error: immutable
    println(val1 + val1)
    println(def1 + def1)

    var1 = 5
    println(var1 + var1)
  }
  ```
  - Output:

    -
      ```
      val: 1
      ```

val vs. var vs. def

-
  ```
  object Test extends App {
    val val1 = {println("val: 1"); 1}
    def def1 = {println("def: 1"); 1}
    var var1 = {println("var: 1"); 1}

    //val1 = 5 // compile error: immutable
    println(val1 + val1)
    println(def1 + def1)

    var1 = 5
    println(var1 + var1)
  }
  ```

  - Output:
    - val: 1
    - var: 1

val vs. var vs. def

```
object Test extends App {
  val val1 = {println("val: 1"); 1}
  def def1 = {println("def: 1"); 1}
  var var1 = {println("var: 1"); 1}

  //val1 = 5 // compile error: immutable
  println(val1 + val1)
  println(def1 + def1)

  var1 = 5
  println(var1 + var1)
}
```

- Output:
  - val: 1
  - var: 1
  - 2

val vs. var vs. def

-
  ```
  object Test extends App {
    val val1 = {println("val: 1"); 1}
    def def1 = {println("def: 1"); 1}
    var var1 = {println("var: 1"); 1}

    //val1 = 5 // compile error: immutable
    println(val1 + val1)
    println(def1 + def1)

    var1 = 5
    println(var1 + var1)
  }
  ```

  - Output:
    - val: 1
    - var: 1
    - 2
    - def: 1

val vs. var vs. def

-
  ```
  object Test extends App {
    val val1 = {println("val: 1"); 1}
    def def1 = {println("def: 1"); 1}
    var var1 = {println("var: 1"); 1}

    //val1 = 5 // compile error: immutable
    println(val1 + val1)
    println(def1 + def1)

    var1 = 5
    println(var1 + var1)
  }
  ```

  - Output:
    - val: 1
    - var: 1
    - 2
    - def: 1
    - def: 1

val vs. var vs. def

```
object Test extends App {
  val val1 = {println("val: 1"); 1}
  def def1 = {println("def: 1"); 1}
  var var1 = {println("var: 1"); 1}

  //val1 = 5 // compile error: immutable
  println(val1 + val1)
  println(def1 + def1)

  var1 = 5
  println(var1 + var1)
}
```

- Output:
  - val: 1
  - var: 1
  - 2
  - def: 1
  - def: 1
  - 2

val vs. var vs. def

```scala
object Test extends App {
  val val1 = {println("val: 1"); 1}
  def def1 = {println("def: 1"); 1}
  var var1 = {println("var: 1"); 1}

  //val1 = 5 // compile error: immutable
  println(val1 + val1)
  println(def1 + def1)

  var1 = 5
  println(var1 + var1)
}
```

- Output:

```
val: 1
var: 1
2
def: 1
def: 1
2
10
```

Case Classes

-
  ```
  abstract class Tree
  case class Sum(l: Tree, r: Tree) extends Tree
  case class Var(n: String) extends Tree
  case class Const(v: Int) extends Tree
  ```
- Mix between concrete classes in the OO world and algebraic data types in Functional Programming
- Differences to normal classes:
  - no **`new`** keyword: e.g.: `val c = Const(1)`
  - getters automatically defined: e.g.: `c.v`
  - `equals` and `hashCode` work on the structure instead of identity
    - `Const(1) == Const(1)` => true
  - Default toString() implementation:
    - `Sum(Const(1), Const(2))` prints to Sum(Const(1), Const(2))
  - Pattern Matching can be used

Pattern Matching

-
  ```
  def eval(t: Tree, env: Environment): Int = t match {
    case Sum(l, r) => eval(l, env) + eval(r, env)
    case Var(n) => env(n)
    case Const(v) => v
  }
  ```

-
  ```
  def derive(t: Tree, v: String): Tree = t match {
    case Sum(l, r) => Sum(derive(l, v), derive(r, v))
    case Var(n) if (v == n) => Const(1)
    case _ => Const(0)
  }
  ```
- Wildcards using `_`
- guarded cases using `if`

Pattern Matching

-
  ```
  def eval(t: Tree, env: Environment): Int = t match {
    case Sum(l, r) => eval(l, env) + eval(r, env)
    case Var(n) => env(n)
    case Const(v) => v
  }
  ```

-
  ```
  def derive(t: Tree, v: String): Tree = t match {
    case Sum(l, r) => Sum(derive(l, v), derive(r, v))
    case Var(n) if (v == n) => Const(1)
    case _ => Const(0)
  }
  ```
- Wildcards using **_**
- guarded cases using `if`

Pattern Matching

-
  ```
  def eval(t: Tree, env: Environment): Int = t match {
    case Sum(l, r) => eval(l, env) + eval(r, env)
    case Var(n) => env(n)
    case Const(v) => v
  }
  ```

-
  ```
  def derive(t: Tree, v: String): Tree = t match {
    case Sum(l, r) => Sum(derive(l, v), derive(r, v))
    case Var(n) if (v == n) => Const(1)
    case _ => Const(0)
  }
  ```
- Wildcards using _
- guarded cases using `if`

Pattern Matching

-
  ```
  def eval(t: Tree, env: Environment): Int = t match {
    case Sum(l, r) => eval(l, env) + eval(r, env)
    case Var(n) => env(n)
    case Const(v) => v
  }
  ```

-
  ```
  def derive(t: Tree, v: String): Tree = t match {
    case Sum(l, r) => Sum(derive(l, v), derive(r, v))
    case Var(n) if (v == n) => Const(1)
    case _ => Const(0)
  }
  ```

- Wildcards using _
- guarded cases using `if`

Traits

- For the Java Programmer: Interfaces that can also contain code

-
  ```
  trait Ord {
    def < (that: Any): Boolean
    def <=(that: Any): Boolean = (this < that) || (this == that)
    def > (that: Any): Boolean = !(this <= that)
    def >=(that: Any): Boolean = !(this < that)
  }
  ```

Traits

-
  ```
  class Date(y: Int, m: Int, d: Int) extends Ord {
    def year = y
    def month = m
    def day = d

    override def equals(that: Any): Boolean =
      that.isInstanceOf[Date] && {
      val o = that.asInstanceOf[Date]
      o.day == day && o.month == month && o.year == year }

    def <(that: Any): Boolean = {
      if (!that.isInstanceOf[Date])
        error("cannot compare " + that + " and a Date")
      val o = that.asInstanceOf[Date]
      (year < o.year) ||
      (year == o.year && (month < o.month ||
                         (month == o.month && day < o.day)))
  }
  ```

Mixins

-
  ```
  trait Singer {
    def sing { println("singing…") }
  }
  trait Flyer {
    def fly { println("flying…") }
  }
  trait Carnivore {
    def eat(a: Animal) { println("eating: " + a) }
  }
  class Animal { … }
  class Bird extends Animal with Singer with Flyer { … }
  class Hawk extends Bird with Carnivore { … }

  val bird = new Bird()
  bird.sing; bird.fly

  val hawk = new Hawk()
  hawk.sing; hawk.fly
  hawk eat bird
  ```

Genericity

-
  ```
  class Reference[T] {
    private var contents: T = _
    def set(value: T) { contents = value }
    def get: T = contents
  }
  ```

-
  ```
  object IntegerReference {
    def main(args: Array[String]) {
      val cell = new Reference[Int]
      cell.set(13)
      println("Reference contains the half of " + (cell.get * 2))
    }
  }
  ```

REPL

- Read-Eval-Print-Loop => Scala Interpreter

- run `scala` in your console
- every expression is directly evaluated

```
> scala
This is a Scala shell.
Type in expressions to have them evaluated.
Type :help for more information.

scala> object HelloWorld {
     |   def main(args: Array[String]) {
     |     println("Hello, world!")
     |   }
     | }
defined module HelloWorld

scala> HelloWorld.main(null)
Hello, world!

scala>:quit
>
```

SBT

- Download at http://www.scala-sbt.org
- Build tool for Scala and Java
- Default directory structure:

- src
  - ↪ main
    - ↪ scala
    - ↪ java
  - ↪ test
    - ↪ scala
    - ↪ java
- (project)
  - ↪ (Build.scala)

- `sbt run`
  - runs program
  - searches for main methods, prompts if multiple found
- `sbt test`
  - runs all tests in test folder
  - additional configuration for JUnit or ScalaTest may be needed (→ Build.scala)

Credits

- http://www.scala-lang.org/docu/files/ScalaTutorial.pdf

- Same examples, more detail
  - look at the pdf if the presentation was to fast or something was unclear