Programming in Scala

Lecture One

Angelo Corsaro, PhD

8 October 2019

Chief Technology Officer, ADLINK Technology Inc.

Table of contents

Course Organisation

Goals

This course has two main goals:

- Ensure that all of you develop a good level of proficiency with the Scala programming language.
- Introduce you to the fundamentals of functional programming.

Organisation

The course is divided into:

- A series of lectures that will introduce the key Scala features along with the core concepts of functional programming.

- An individual project whose goal is to develop a non-trivial Scala application – the best way to learn a programming language is to use it!

Lectures Schedule

- **Tue 8/10 13h-17h Lecture (3407V)**
- **Tue 15/10 13h-17h Lecture (2401V)**
- **Tue 22/10 13h-17h Lecture (1109V)**
- **Tue 24/10 08h-12h Lecture (TBD)**

Project Guidelines

- The reason for this class to have individual projects is to ensure that everyone does a decent amount of programming.
- You will be expected to demo and walk through the code implementing the project and ask questions concerning specific design and implementation decision you took.
- Beware that if we detect any form of plagiarism you will probably fail the exam. Be clear on the libraries and third parties code you used. Beware that there are online tools to detect code plagiarism.

Programming Paradigms

What is a Programming Paradigm?

Definition
In Computer Science, by **Programming Paradigm** we intend the set of abstractions and conceptual mechanisms provided by a programming language in order to express programs in the given language.

Different paradigms differ in the **concepts and abstractions** provided to represent the elements of a program as well as the mechanism and **guarantees** provided for the **execution** of the program itself.

Programming Paradigms: Examples

Through the history of programming languages, several programming paradigms have been proposed. The most significant today are:

- Functional Programming
- Object Oriented Programming
- Logic Programming
- Procedural Programming
- Generic Programming

Object Oriented Programming

The Object Oriented Programming is a Programming Paradigm that promotes the decomposition of a problem into a set of objects and relationships between objects each characterized by a set of properties and operations.

An **object** is independent entity which can be treated in isolation from any other object. Objects are *first-class* constructs and have an identity – that means that two objects can be distinct even if their *value* is the same.

Object Properties

In an Object Oriented Programming Language, objects exhibit some general properties:

- **Encapsulation**
- **Inheritance**
- **Polymorphism**
- **Dynamic Binding**

Encapsulation

Encapsulation is the property of information hiding, in other terms, the hiding of data and implementation details of an object.

When an external viewer looks at an object, only its *exterior* interface is visible, the internal details are invisible, irrelevant and cannot be accessed.

Example

```
class Complex {
  private float real;
  private float img;
  public float getReal()
  public float getImg()
  // ..
}
```

```
class Complex {
  private float[] repr;
  public float getReal()
  public float getImg()
  // ..
}
```

Inheritance

Inheritance is a mechanism provided by Object Oriented Programming languages to express refinements over a given type.

Given a type T, inheritance allows to create a type S, such that S *inherits* the properties of T and $S <: T$.

The consequences of inheritance is that the type S can be used wherever a type T is expected.

Liskov Substitution Principle

Definition
Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where $S <: T$.

Liskov's notion of a behavioral subtype defines a notion of substitutability for objects.

If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program, such as correctness.

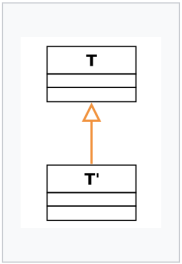
Variance

Within the type system of a programming language, a typing rule or a type constructor is:

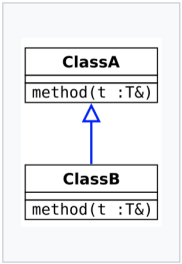
- **covariant** if it preserves the ordering of types ($\leq$), which orders types from more specific to more generic;
- **contravariant** if it reverses this ordering;
- **bivariant** if both of these apply
- **invariant** or nonvariant if neither of these applies.

Variance

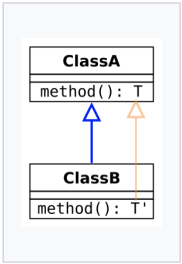
Variance and method overriding: overview



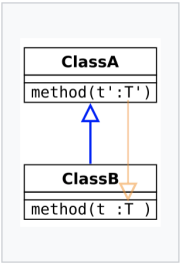
Subtyping of the argument/return type of the method.



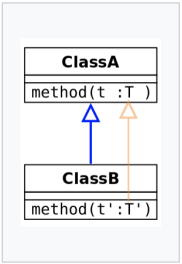
Invariance. The signature of the overriding method is unchanged.



Covariant return type. The subtyping relation is in the same direction as the relation between ClassA and ClassB.



Contravariant argument type. The subtyping relation is in the opposite direction to the relation between ClassA and ClassB.



Covariant argument type. Not type safe.

Variance

Example
Given $S <: T$, we have that:

- if a list type $List[A]$ is defined to be covariant in the type of its elements, then $List[S] <: List[T]$
- a function is contravariant in its argument types, consequently given $f : T \to Q$ and $g : S \to Q$ then $f <: g$. More in general function are contravariant in the argument types and covariant in the return types.

Polymorphism

Given a function f with two return types ρ_1 and ρ_2, f is polymorphic if and only if f can be called with an argument of type τ_1, returning a value of type ρ_1:

$$f : \tau_1 \to \rho_1$$

and can also be called with an argument of type τ_2, returning a value of type ρ_2:

$$f : \tau_2 \to \rho_1$$

Example
The binary operator $+$ is an example of polymorphic function you should be familiar with.

Dynamic Binding

Dynamic bindings means that the operation invoked on an object depends on its identity as opposed to the type used to refer to it.

Notice that dynamic binding works in close relationship with inheritance and in programming languages supporting it, ensures that the operation called is the one defined by the target object as opposed than that defined on a parent type.

Dynamic Binding

Example

```
class T {
  public String toString()
}
```

```
class S extends T {
  public String toString()
}
```

```
T t = new T();
t.toString(); // calls T.toString();
t = new S();
t.toString(); // calls S.toString();
```

Functional Programming

Functional Programming is a method of program construction that:

- Emphasises functions and their applications as opposed to commands and their execution.
- Uses simple mathematical notation that allows problems to be described clearly and concisely.
- Has a simple mathematical bases that supports equational reasoning about the properties of a program.

A functional programming languages is guided by two main ideas, **functions as first-class values** and **no side-effects**.

Functions as First-Class Values

In a functional programming language:

- A function is a value as an `Integer` or a `String`.
- Functions can be passed as arguments, returned as values as well as stored into variables.
- Functions can be named or anonymous and can be defined inside other functions.

No Side-Effects

- In a functional programming language, the result of applying some arguments to a functions should only depend on the input. In other terms, applying the same input to a given function always gives the same output.
- Functions that satisfy this property are know to be *referentially transparent*.
- Functional programming languages ecnourage the use of *immutable* data structures and *referentually transparent* functions.

Note: it is worth comparing the functional approach with the imperative style programming were everything is based on mutable data and side-effects.

Scala Overview

Scala Programming Paradigms

Scala, which stands for scalable language, is a multi-paradigm **statically-typed** programming language that provide advanced support for **Object-Oriented** and **Functional** and **Generic** Programming.

Scala blends these multiple paradigms in a coherent language that provides very good support to scale from small prototypes to large scale industrial projects.

Why Scala?

Beside being a very popular programming language used in distributed computing and analytics framework , **Scala** is an appealing programming language because of the following reasons:

Why Scala?

Beside being a very popular programming language used in distributed computing and analytics framework , **Scala** is an appealing programming language because of the following reasons:

- Scala targets the Java Virtual Machine (JVM) and seamlessly interoperates with Java, *i.e.*, Java libraries can be called from Scala and vice-versa.

Why Scala?

Beside being a very popular programming language used in distributed computing and analytics framework , **Scala** is an appealing programming language because of the following reasons:

- Scala targets the Java Virtual Machine (JVM) and seamlessly interoperates with Java, *i.e.*, Java libraries can be called from Scala and vice-versa.

- Scala syntax is extremely concise and even if somebody is not interested functional or higher order programming it can be used as a better Java.

Why Scala? / Cont.

- Scala makes it easy to create high level abstractions thus helping developer to master the complexity of the task at hand.

Why Scala? / Cont.

- Scala makes it easy to create high level abstractions thus helping developer to master the complexity of the task at hand.

- Scala is statically typed, thus helps detecting most of the errors at compile time. At the same time, the type system uses automatic type inference to minimise explicit annotations and give a very compact syntax.

Your First Scala Application

```scala
package edu.esiee.scala17

object Welcome {
  def main(args: Array[String]): Unit = {
    val name = if (args.length > 0) args(0) else "student"
    println(s"Dear $name, welcome to the Scala class!")
  }
}
```

The the previous deserves a few observations and raises a few question:

- The **object** keyword, what does it mean?
- The **Unit** return type seems suspicious...
- The **if** construct is an expression.
- The **println** text is an interpolated **string**.

Scala Primitive and Fundamental Types

Primitive Types

Scala is a very regular language in the sense that everything is an object. The number 42 is an object, a function is an object, etc.

Scala provides the following built-in types:

Integral types:

- `Byte`
- `Char`
- `Short`
- `Int`
- `Long`

Numeric types:

- `Float`
- `Double`

Logical types:

- `Boolean`

Sequence types:

- `String`

Fundamental Types

There are a few types provided as part of the standard library that are ubiquitously used in Scala programming, these are:

- **Tuple**: A fixed length, immutable, collection of heterogeneous types.
- **Array**: A fixed length, mutable, sequence of homogeneous types.
- **List**: An immutable sequence of homogeneous types.
- **Map**: An immutable dictionary mapping values of a type **K** to values of a type **T**.

```
1  // this is a tuple of type (Int, String, (Double, Double))
2  val t = (1, "alpha", (0.3, 0.1))
3  // this is an Array[Int]
4  val as = Array(1,2,3)
5  // this is a List[Int]
6  val xs = List(1,2,3,4,5)
7  // this is a Map[Int, String]
8  val ms = Map(1 -> "Uno", 2 -> "Due", 3 -> "Tre", 4 -> "Quattro", 5 -> "Cinque")
```

Braces Usage

You should be aware that in Scala:

- *square brackets* [] indicate a type parameter, *e.g.* `List[Int]`, `Array[String]`
- *parenthesis* () indicate the application of a function, that said array is a form function application (more later...), *e.g.*, `pow(2,1024)`, `as(10)`

Scala Object Oriented Constructs

Classes and Singleton Objects

Differently from most of the mainstream object oriented programming languages, Scala supports as a first-class concept both **Classes** and **Singleton Objects**.

- A **class** is a blueprint for objects. Objects of a given class can be instantiated by using the **new** keyword. Differently from Java, a Scala class cannot have static members.
- A **class** can be declared abstract by using the **abstract** keyword. Notice that for an abstract class there cannot be object instances.
- A **singleton object** declaration is similar to that of a class, with the difference that the **object** keyword is used. As the name implies, a singleton object defines a type for which which only one instance exit. This somehow blurs the line between the time and the instance.

Companion Objects

A **Singleton Object** declared in the same file of a class **C** and named after the class is named **Companion Object**. A class and its companion object can access each other private members.

Example

```scala
package edu.esiee.scala17

object Complex {
  def apply(re: Double, im: Double) = new Complex(re, im)
}

class Complex(val re: Double, val im: Double) {
  override def toString = s"$re+i$im"
}
```

Class Constructor

A class always has a primary constructor and may have zero or more auxiliary constructors.

- Scala creates the **primary constructor** by defining a constructor that takes the **class parameters** – parameters defined right after the class name. Any attribute initialisation code defined as part of the class body is executed by the primary constructor.
- A **Auxiliary Constructor** can be defined explicitly by means of a special method defined as **def this(...)**

Class Constructor

Example

```
1  package edu.esiee.scala17
2
3  class Rational(a: Int, b: Int) {
4    assert(b != 0)
5    val num: Int = a / gcd(a, b)
6    val den: Int = b / gcd(a, b)
7
8    def this(n: Int) = this(n, 1)
9  }
```

- The default constructor will execute line 4-6
- Line 8 defines the auxiliary constructor

Attributes

Class attributes are declared by using the **var** or **val** keyword followed by the attribute name, columns and the attribute type.

```
1  class Foo {
2          val attrA: Int
3          var attrB: Int
4  }
```

- **val** is used to declare immutable attributes.
- **var** is used to declare mutable attributes.

Methods

Class and **Singleton Object methods** are defined through the **def** keyword and have the following structure:

```
def methodName(a: A, b: B, c: C, ...): T
```

- Declaring a method without providing a definition makes the class abstract.
- The return type is not strictly necessary and can be omitted.
- Method always have a return type, where the type **Unit** is used for functions that don't return anything.
- Parenthesis can be dropped on methods that don't take any parameters. This provides uniform access between attributes and methods.
- Overloading of parent methods should be annoted with the **override** keyword.

Methods

Example

```scala
package edu.esiee.scala17

object CRational {
  def inverse(i: Int): CRational = {
    assert(i != 0)
    new CRational(1, i)
  }
}
class CRational(a: Int, b: Int) {
  assert(b != 0)
  val num: Int = a / gcd(a, b)
  val den: Int = b / gcd(a, b)

  def this(n: Int) = this(n, 1)

  override def toString: String = s"$num/$den"
}
```

Methods

Methods can be invoked on an object by using the dot notation or by simply having the method name follow the object, as shown below:

```
1  val s = new S()
2  val t = new T()
3  s.someMethod(t)
4  s  someMethod t
```

Operators

Operators in Scala are regular methods. Likewise regular methods can be used as operators.

To this end, notice that:

```
1+2
```

Is actually a short form for:

```
1.+(2)
```

Unary operators can be defined by pre-pending the operator name with **unary_**, as in:

```
def unary_-()
```

Function Application

Scala allows also to define a **function application** operator for classes and singleton objects. This is simply done by declaring a method called **apply**.

Example

```
1  class Complex(re: Double, im: Double) {
2    // Any even number number gets the real part and odd the imaginary
3    def apply(i: Int) = if (i%2 == 0) re else im
4  }
5  val c = new Complex(3, 7)
6  val re = c(42)
```

Access Modifiers

Scala supports the following access modifiers:

- **public**
- **private**
- **protected**

By default declarations in a class are public.

Access scope can be qualified by providing a scope, such as in **private[X]**, **protected[packageName]**, **private[this]**.

Exercise

Define a **Complex** type with operations for sum, subtraction, multiplication, division, modulo and negation.

Built-in Control Structures

Scala provides the following built-in control structures:

- **if** expression
- **while** loop
- **for** expression
- **match** expression
- **try/catch/finally** constructs

if expression

In line with functional programming language, Scala's **if** conditional construct is an expression. This means that the **if** construct always returns a value – yes, this value may be **Unit**

The general form of the if construct is:

```
1 if (predicate) exprT else exprE
```

while loop

Scala's **while** loop, has no surprises, behaves as in other programming languages.

The general form of the **while** loop is:

```
while (predicate) {
        body
}
```

Where the body is executed while the predicate holds true.

Scala also has a **do-while** loop:

```
do {
        body
} while (predicate)
```

In this case the body is executed at least once and until the predicate remains true.

for expression

Scala's **for** construct is an expression and not simply a looping construct.

As we will discover later in the course, it is very close to Haskell's **do** construct used to express monadic computations. It is also in a way similar of the list comprehension supported by haskell.

As a word of warning, you should be aware that it does not translate to an efficient iterating construct. If this is not understood you could introduce serious performance issues in your code.

for expression

Scala's **for** construct is an expression and not simply a looping construct.

As we will discover later in the course, it is very close to Haskell's **do** construct used to express monadic computations. It is also in a way similar of the list comprehension supported by haskell.

As a word of warning, you should be aware that it does not translate to an efficient iterating construct. If this is not understood you could introduce serious performance issues in your code.

The best way of appreciating the power of Scala's **for** construct is to see some examples of its use.

for expression in action

```scala
for (i <- 1 to 10)
  println(i)

val xs = for (i <- 1 to 10; if i % 2 == 0) yield i

for (x <- xs)
  println(x)

val ys = for {
  i <- 1 to 5
  j <- 1 to i
} yield (i, j)

for ((a, b) <- ys)
  println(s"($a, $b)")
```

match expression

As a very first and extremely inaccurate approximation, you can consider Scala's **match** expression as something that recalls the **switch** construct found in Java and other programming languages.

That said, Scala's **match** is first and foremost an expression and is extremely powerful as it plays a key role, as we will see on the manipulation of **algebraic data types**.

Scala's **match** expression allow you to match a series of alternatives. These alternatives however can be expresses through patterns (more later).

match expression / example

Below are some examples of what the match expression can do for us.

```
def digit2String(i: Int) =
        i match {
                case 0 => "Zero"
                case 1 => "One"
                //...
                case 9 => "None"
                default => "Invalid Digit"
        }
}
```

match expression / example

```
1  def string2Digit(i: Int) =
2    i match {
3      case 0 => "Zero"
4      case 1 => "One"
5      //...
6     default => "Invalid Digit"
7    }
8  }
```

match expression / example

```
1  @tailrec
2  def gcd(t: (Int, Int)):Int = t match {
3    case (a, 0) => a
4    case (_, 1) => 1
5    case (a, b) if a > b => gcd(b, a % b)
6    case (a, b) if a < b => gcd(b, a)
7  }
```

Exceptions

Scala provides support for exception handling that is similar to that of Java, but as you can expect by now there are a few catches.

The first difference you should be aware of is that in Scala you are not forced to catch checked exceptions nor are asked to declare them as part of the function or method signature.

As we will see shortly, exceptions leverage Scala's pattern matching (more on the subject next lecure) to facilitate the handling.

Throwing Exceptions

Exceptions can be thrown by using the **throw** keyword.

Example
Let's improve our `string2Digit` function to actually throw an exception if the argument is not a single digit.

```
1  def string2Digit(i: Int) =
2    i match {
3      case 0 => "Zero"
4      case 1 => "One"
5      //...
6      case 9 => "None"
7      default => throw new RuntimeException("The digit should be between 0 and 9")
8    }
9  }
```

Catching Exceptions

Exceptions are caught by using a **try/catch** construct, where the **try** encloses the block of code that may raises an exception and the **catch** pattern matches expressions.

Example

```
try {
  val a = string2Digit(        3)
  val b = string2Digit(18)
} catch {
   case ex: RuntimeException => // Handle exception
   case _: => // catch any other kind of exception and handle it
}
```

Catching Exceptions / Example

Example

```
1  import java.io.FileReader
2  import java.io.FileNotFoundException
3  import java.io.IOException
4
5  try {
6    val f = new FileReader("input.txt")
7    // Use and close file
8  } catch {
9    case ex: FileNotFoundException => // Handle missing file
10   case ex: IOException => // Handle other I/O error
11 }
```

finally clause

In some cases we want to executed some code no matter how an expression terminates.

In this case the **finally** clause can be used to ensure that the code associated with it will **always** be executed.

Example

```
1  import java.io.FileReader
2
3  val file = new FileReader("input.txt")
4  try {
5     /* Use the file */
6  } finally {
7     file.close()  /* Be sure to close the file */
8  }
```

Homeworks

Reading

From the **Programming in Scala** book you should read:

- Chapter 1
- Chapter 2
- Chapter 3
- Chapter 4
- Chapter 5
- Chapter 6
- Chapter 7