Programming in Scala

based on the book
“Programming in Scala Third Edition”

Agenda

- Chapter VI - Classes and Objects
  - Constructing classes
  - Override methods
  - Check parameters
  - Self references
  - Method overloading
- Chapter VII - Built-in Control Structures
  - If expression
  - While loops
  - For expression
  - Exception handling
  - Match expression
  - Variable scopes

Chapter VI - Classes and Objects

Primary constructor

- Every class must have a primary constructor

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

- The attributes are called class parameters
- Always try to use immutable objects!
- Compiler will insert every line of code placed into the classed body to the primary constructor

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

Auxiliary constructors

- Constructors other than the primary one are called auxiliary constructors

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

- Every auxiliary constructor must invoke another constructor of the same class as its first action
- The primary constructor is thus the single point of entry of a class.
- In Scala only the primary constructor can invoke a superclass constructor.

Constructor preconditions

- You can check certain preconditions before calling the primary constructor
- A precondition is the constraint applied to the constructor parameters
- Examples: `assert, assume, require, ensuring`
- All of these throws exceptions in case the parameter doesn't meet the requirements
- All those are defined in the `Predef` automatically imported object from the Scala standard library

```
class Rational(n: Int, d: Int) {
    require(d != 0)
}
```

Overriding methods

- It is mandatory to use the override when you override a concrete(non-abstract!) method from the superclass

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

Fields

- You can access to class parameters only inside the defining class
- You need to use fields to be able to re-use the parameter values from other instances

```
class Rational(n: Int, d: Int) {
    val numer: Int = n
    val denom: Int = d
}
```

Private fields and methods

- Methods and fields can get `private` modifier

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

Identifiers in Scala

- 4 valid types of identifiers
  - Alphanumeric: starts with a letter or underscore, which can be followed by further letters, digits, or underscores
    - Convention to name constants: the first character should be uppercase, the next is camelcase e.g.: XOffset
  - Operator: an operator character belongs to the Unicode set of mathematical symbols(Sm) or other symbols(So), or to the 7-bit ASCII characters that are not letters, digits,
  - Mixed identifiers: alphanumeric + _ + operator e.g.: `unary_+`
  - Literal identifier: is an arbitrary string enclosed in backticks (` ...`).
    - It is good because you can use reserved words as identifiers this way, like `` `yield`() ``

Method overloading

- Scala's process of overloaded method resolution is very similar to Java's.
- In every case, the chosen overloaded version is the one that best matches the static types of the arguments.
- Sometimes there is no unique best matching version; in that case the compiler will give you an "`ambiguous reference`" error.

Implicit conversions

- The implicit modifier in front of the method tells the compiler to apply it automatically in a number of situations.
- For an implicit conversion to work, it needs to be in scope.
- Because they are so powerful, they can also be easily misused!!

```
implicit def intToRational(x: Int) = new Rational(x)
```

Chapter VII - Built-in Control Structures

If expression

- Unlike in Java, Scala's if is an expression that results in a value.

```
val filename = if (!args.isEmpty) args(0)

else "default.txt"
```

- Using a val is the functional style, and as a second advantage to using a val instead of a var is that it better supports equational reasoning. (referential transparency!)

While, do-while loops

- Both exists in Scala
- Don’t return any interesting value, but `Unit`
  - The return value is `()` (unit value), which is the only value of `Unit`
  - In Scala assignment always results in the unit value, `()`.
- Because the while loop results in no value, it is often left out of pure functional languages.
- Prefer recursion over while loops!

```
while(a <= 2) {
    a = a + 1
}
```

For expression

- Scala's `for` expression is a Swiss army knife of iteration.
- This is an expression like `if`, so it can return a value, but it is not necessary!

```
for (i <- 1 to 5)
    println(i)
```

- `i <- 1 to 5` is called the generator
- To the right of the <- symbol in a for expression can be any type that has certain methods (in this case foreach) with appropriate signatures.
  - `Range` type: `1 to 5` or `1 until 5`

Filtering in for expression

- Sometimes you don't want to iterate through a collection in its entirety; you want to filter it down to some subset.
- You can do this with a for expression by adding a filter, an if clause inside the for's parentheses
- You can have multiple filterings within a single for

```
for (i <- 1 to 5 if i % 2 == 0)
   println(i)
```

vs

```
for(i <- 1 to 5)
   if (i % 2 == 0)
      println(i)
```

Nested iteration in a for expressions

- If you add multiple <- clauses, you will get nested “loops.”

```
for (
    file <- filesHere
    if file.getName.endsWith(".scala");
    line <- fileLines(file)
    if line.trim.matches(pattern)
) println(file + ": " + line.trim)
```

- If you prefer, you can use curly braces instead of parentheses. One advantage to using curly braces is that you can leave off some of the semicolons, the Scala compiler will not infer semicolons while inside parentheses.

Mid-stream variable bindings

- You can introduce variables inside the head of the for loop
- You can use the variable only after you declared it and only in the scope of the `for` loop

```
for {
    file <- filesHere
    if file.getName.endsWith(".scala")
    line <- fileLines(file)
    trimmed = line.trim
    if trimmed.matches(pattern)
} println(file + ": " + trimmed)
```

Producing “return” value in a for loop

- You can also generate a value to remember for each iteration.
- You have to prefix the **whole** body of the for expression by the keyword `yield`
- Be careful! The yield goes before the entire body, if it is a code block, then before the curly brace!

```
val files =
    for {
        file <- filesHere
        if file.getName.endsWith(".scala")
    } yield file
```

Exception handling

- It works like Java's exception handling
- You can throw an exception: `throw new IllegalArgumentException`
  - It is an expression, so it has a return type, but it will never evaluate to anything: `Nothing`
- Catching exceptions uses pattern matching
- Scala does not require you to catch checked exceptions or declare them in a throws clause!

```
try {
    // open a file, operate on thatű
}
catch {
    case ex: FileNotFoundException => // Handle missing file
    case ex: IOException => // Handle other I/O error
}
```

The Finally clause

- You can wrap an expression with a finally clause if you want to cause some code to execute no matter how the expression terminates.
- `try` and `finally` do not require parentheses if they contain only one expression, `catch` requires!
- `finally` can't return any value, if you calculates a value here, it will be ignored! It is there to do some operations like closing a file
- `try` and `catch` can return a value

```
def g(): Int = try 1 finally 2 // g = 1
```

Match expression

- Scala's `match` expression lets you select from a number of alternatives (like `switch` in Java)
- It uses patterns to differentiate values
- Default case is: _ (underscore)
- Any type of constants can be used not just integers, Strings and enums
- It is also an expression so it results is a value!
- There are no breaks at the end of all cases, it is implicit! It always stops the evaluation

```
val friend =
    firstArg match {
        case "salt" => "pepper"
        case "chips" => "salsa"
        case "eggs" => "bacon"
        case _ => "huh?"
    }
```

Living without break and continue

- Those are not really fit into functional programming so Scala doesn't have them
- The simplest approach is to replace every `continue` by an `if` and every `break` by a `boolean` variable
- Or you can rewrite the whole loop to a recursion
- As a last chance there is a `break` method in `scala.util.control.Breaks` to exit from a block marked with `breakable`

```
breakable {
    while (true) {
        println("? ")
        if (in.readLine() == "")
            break
    }
 }
```

Variable scope

“If you’re a Java programmer, you’ll find that Scala’s scoping rules are almost identical to Java’s. ***One difference*** *between Java and Scala is that* ***Scala allows you to define variables of the same name in nested scopes.*** *So if you’re a Java programmer, you may wish to at least skip this section.”*

Conclusion

- Try to avoid side effects and have method with real return types
- Try to avoid `while` loops, use `for` loops or recursion instead
- Scala has minimal set of built-in control structures
- They tend to result in a value

+1 The Homework :)

Try to create the `Complex` class

1. Implement those operations as a minimum
   a. Addition
   b. Subtraction
   c. Multiplication
   d. Division
   e. Modulus (with operator **!**)
2. Try to use implicit conversion to be able to write expressions like this:

   E.g: `2 + new Complex(1,2)`

3. Try to use real operators as identifiers for method names (+, -, /, *)
4. Make it **immutable**!
5. Override the toString implementation