Programming in Scala

based on the book
“Programming in Scala Third Edition”

Agenda

- Chapter IV - Classes and Objects
    - Classes, fields and methods
    - Semicolon inference
    - Singleton objects
- Chapter V - Basic types and operations
    - Basic types
    - Literals
    - String interpolations
    - Operators and methods
    - Objects equality
    - Operator precedence

Chapter IV - Classes and Objects

Class

- A class is a blueprint for objects
- A class can contain members
  - Fields (instance variables): `val, var`
  - Methods: `def`
- Objects instantiation happen with the `new` keyword

```
class ChecksumAccumulator {
    var sum = 0
}

val acc = new ChecksumAccumulator
```

Fields

- Inside the class definition you can place fields
- Fields are stores the data of an object
- Every instance of the objects get its own set of variables in the memory
- Access modifiers: `private, public`
    - By default (unlike JAVA) it is `public!`

```
class ChecksumAccumulator {
    private var sum = 0
}
```

Methods

- Can appear within a class, its definition starts with `def` keyword
- In Scala there are nested function, which are functions defined in the body of another function
- `return` is optional, if you skip it Scala return the last computed value in the block
- You can leave the curly braces `{}` around the block if the method only computes one single expression
- You can leave the return type, the compiler can infer it as well
- The “void” return type is `Unit` - **Side effects!**

```
def testMethod(Int a) = a + 1
```

Semicolon inference

- Semicolon is usually optional at the end of the statements
- It is required when you write multiple statements in a line

```
val s = "hello"; println(s)
```

- However, when + appears at the end of the line scala parses the next line with the previous

```
1  +            vs          1
2                           + 2
```

Singleton objects

- Scala doesn't know Java static members! Instead Scala has singleton objects
- Its definition starts with `object` instead of ~~`class`~~
- Normal classes can have attached singleton objects with the same name, the singleton classes called *companion classes* (objects)
- It can't be instantiated, it is initialized when first accessed
- Singleton objects without companion objects called *standalone objects*
- You can invoke the public methods of a singleton object like

```
SingletonClassName.methodName(...)
```

Scala application

- The minimum length Scala app is a **singleton object with a main method** that takes **one parameter**, an Array[String], and has a **result type of Unit**
- Class file name doesn't need to match the class/object name it defines!
    - `class B` can be defined in a file called A.scala
- To compile a scala application: **`scalac`** `A.scala B.scala ...`
- Or use **`fsc`** is a compiler daemon, for the first run it creates a local server runs in a port and it will connect to this server during the future executions
    - Stop daemon: `fsc -shutdown`
- Compile produces standard java class files contain bytecode
- To run a compiled class: `scala classFileName`

```
object Main {
    def main(args: Array[String]) = {
        // your logic comes here
    }
}
```

The App trait

- There is a build-in trait called App
- You can save some line of code when using it

```
object Main extends App {
    // your code comes here
}
```

Chapter V - Basic Types and Operations

Basic types of the `scala` package

Basic type	Range
`Byte`	8-bit signed two's complement integer (-2^7 to $2^7 - 1$, inclusive)
`Short`	16-bit signed two's complement integer (-2^{15} to $2^{15} - 1$, inclusive)
`Int`	32-bit signed two's complement integer (-2^{31} to $2^{31} - 1$, inclusive)
`Long`	64-bit signed two's complement integer (-2^{63} to $2^{63} - 1$, inclusive)
`Char`	16-bit unsigned Unicode character (0 to $2^{16} - 1$, inclusive)
`String`	a sequence of `Char`s
`Float`	32-bit IEEE 754 single-precision float
`Double`	64-bit IEEE 754 double-precision float
`Boolean`	`true` or `false`

Literals

- All the basic types can be written with literals
- `Integer` literals: `255, 0x5, 0x00FF`
- `Long` literals: like integer one end with an `l or L`
- `Double` literals: `1.2345, 1.2345e1, 1.2345E5 (1.2345 x 10`^{5}`)`
- `Float` literals: like `Doubles` but ends with an `f or F`
- `Char` literals: `'A', '\u0012'` (unicode literals can appear in the Scala source code anywhere `val B\u0041\u0044 = 1`)
- `String` literals: `"hello"`, `"\\\""`
    - or *raw String* literals -> `""" sdsdsd'\ """`
- `Boolean` literals: `true, false`

Symbol literals

- Syntax: `'identifier`
- Such literals are mapped to instances of the predefined class `scala.Symbol`

```
'hello → Symbol("hello")
```

- symbols are *interned*
  - If you write the same symbol literal twice, both expressions will refer to the exact same Symbol object.
- There is not much you can do with a symbol, except find out its name

```
val s = 'aSymbol
val nm = s.name
```

String interpolation

- It allows you to embed expressions within String literals

```
val name = "dani"
println(s"hello $name!")
```

- `s` is a built-in Scala string interpolator
    - it result is like: `"hello" + name + "!"`
- You can place any expression after the `$` like `${6 * 7}`
- Other interpolators: `raw, f`
- These are implemented with compile time code rewrite
- You can define your own one

Operators

- Operator are just syntax sugars, in reality all those are methods calls on objects: `1 + 2 → (1).+(2)`
- You can use any methods in operator notation:

  `s.indexOf('o') ➜ s indexOf 'o'` or `s indexOf ('o', 6)`

- Prefix and postfix operators are *unary* operators
- These are just shorthands for a method calls like: `**unary_**MethodName`

  `-2 ➜ (2).unary_-`
- The only identifiers can be used as prefix operators are: `+, -, !, ~`
- Postfix operators are methods that take no arguments, when they are invoked without a dot or parentheses: `s.toLowerCase() → s toLowerCase`

Built-in operators

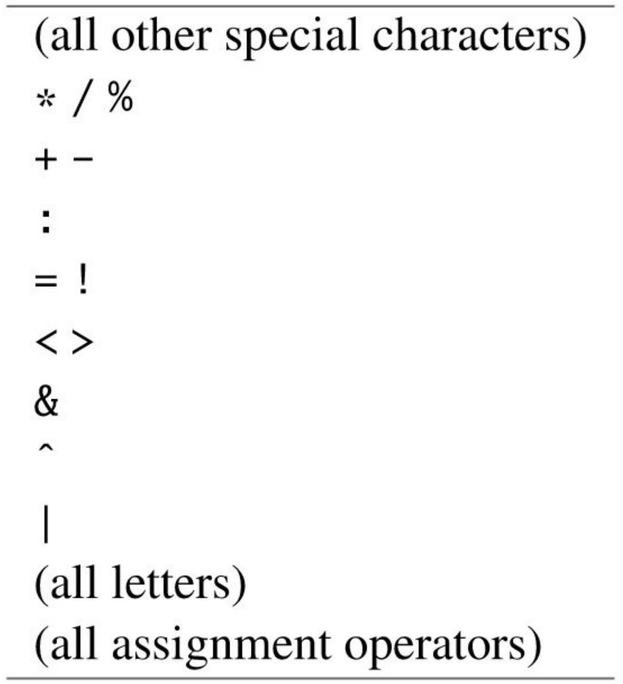
- Arithmetic operators: `+`, `-`, `*`, `/`, `%`
- Relational and logical operators: `>`, `<`, `<=`, `>=`, `!`, `&&`, `||` (short circuit), `&`, `|`
- Bitwise operators: `&`, `|`, `^` (xor), `~` (bitwise complement), `<<`, `>>`, `<<<`, `>>>` (unsigned shift)
- (Object) Equality: `==`, `!=`
  - !!Unlike JAVA (in case of objects) it is not a reference equality it calls a null check first for the left side operand, then it calls its equals method!!

Operator precedence

The higher a character is in this table, the higher the precedence of methods that start with that character.

The one exception to the precedence rule, If an operator ends in an equals character (e.g.: *=), and the operator is not one of the comparison operators, then the precedence of the operator is the same as that of simple assignment (=).

Table 5.3 · Operator precedence



(all other special characters)
* / %
+ -
:
= !
< >
&
^
|
(all letters)
(all assignment operators)

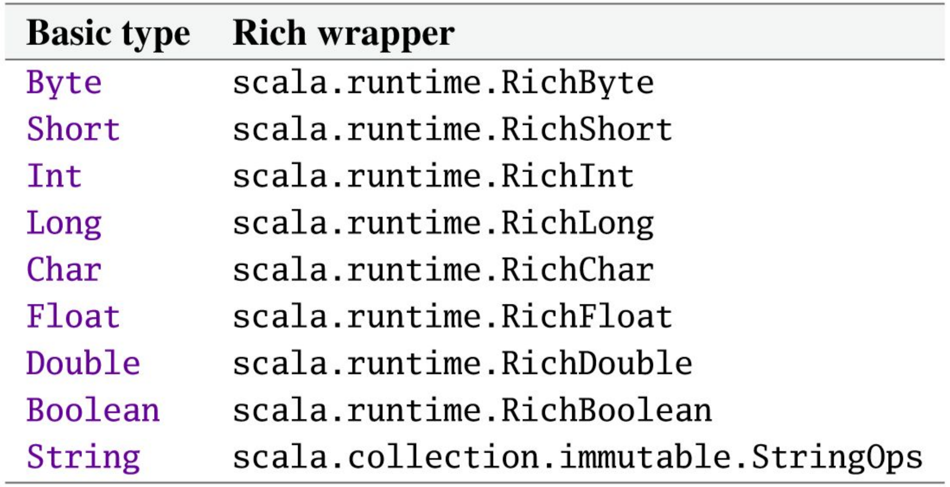
Operator associativity

- Multiple operators of the same precedence side by side ➔ associativity of the operators determines the way operators are grouped
- The associativity of an operator in Scala is determined by its last character
- Any method that ends in a ':' character is invoked on its right operand, passing in the left operand: `a * b → a.*(b)` vs `a ::: b b.:::(a)`
- No matter what associativity an operator has, however, its operands are always evaluated left to right:

`a ::: b ::: c` ➔ `a ::: (b ::: c)` **vs** `a * b * c` ➔ `(a * b) * c`

Rich wrappers

- Each basic type has a "rich wrapper" that provides several additional methods
- These methods are available via *implicit conversions*



Basic type	Rich wrapper
`Byte`	`scala.runtime.RichByte`
`Short`	`scala.runtime.RichShort`
`Int`	`scala.runtime.RichInt`
`Long`	`scala.runtime.RichLong`
`Char`	`scala.runtime.RichChar`
`Float`	`scala.runtime.RichFloat`
`Double`	`scala.runtime.RichDouble`
`Boolean`	`scala.runtime.RichBoolean`
`String`	`scala.collection.immutable.StringOps`

+1 The Homework :)

Scala notebook

- Create a Scala application that defines the `Notebook` and `Note` classes
- You can add a note with the `+` operator to the notebook
- You can remove a note with – operator.
- Prevent duplications
- Implement the merge (`:::`) operation for you `Notes` and `Notebooks` as well