Exercises — Scala Day One (Part I)

Introduction to the language

Spring term 2016

You need to have installed the Scala distribution before commencing these exercises. Don't forget the documentation for the distribution as that will come in very useful.

Introductory

1. Verify your Java version by typing `java version` in a shell (command window). The version must be 1.6 or greater for Scala to work.
2. Verify your Scala version by typing `scala` in a shell (this starts the REPL). The version must be 2.10 or greater.
3. Quit the REPL by typing `:quit`.

The REPL

1. Create a *value* identifier and store (and print) the value `17`.
2. Using the value you just stored (`17`), try to change it to `20`. What happened?
3. Store (and print) the value `ABC1234`.
4. Using the value you just stored (`ABC1234`), try to change it to `DEF1234`. What happened?
5. Store the value `15.56`. Print it.

Expressions

1. Write an expression that evaluates to `true` if the sky is `sunny` and the temperature is more than 80 degrees.
2. Write an expression that evaluates to `true` if the sky is either `sunny` or `partly cloudy` and the temperature is more than 80 degrees.
3. Write an expression that evaluates to `true` if the sky is either `sunny` or `partly cloudy` and the temperature is either more than 80 degrees or less than 20 degrees.

4. Convert Fahrenheit to Celsius.
   Hint: first subtract 32, then multiply by 5/9. If you get 0, check to make sure you didnt do integer maths.

5. Convert Celsius to Fahrenheit.
   Hint: first multiply by 9/5, then add 32. Use this to check your solution for the previous exercise.

Methods

1. Create a method `getSquare` that takes an `Int` argument and returns its square. Print your answer. Test using the following code.

```
val a = getSquare(3)
assert(/* fill this in */)

val b = getSquare(6)
assert(/* fill this in */)

val c = getSquare(5)
assert(/* fill this in */)
```

2. Create a method `isArg1GreaterThanArg2` that takes two `Double` arguments. Return `true` if the first argument is greater than the second. Return `false` otherwise. Print your answer. Satisfy the following:

```
val t1 = isArg1GreaterThanArg2(4.1, 4.12)
assert(/* fill this in */)

val t2 = isArg1GreaterThanArg2(2.1, 1.2)
assert(/* fill this in */)
```

3. Create a method `manyTimesString` that takes a `String` and an `Int` as arguments and returns the `String` duplicated that many times. Print your answer. Satisfy the following:

```
val m1 = manyTimesString("abc", 3)
assert("abcabcabc" == m1, "Your message here")

val m2 = manyTimesString("123", 2)
assert("123123" == m2, "Your message here")
```

Classes & Objects

1. Create a `Range` object and print the `step` value. Create a second `Range` object with a `step` value of `2` and then print the `step` value. What's different?

2. Create a `String` object `s1` (as a `var`) initialised to `"Sally"`. Create a second `String` object `s2` (as a `var`) initialised to `"Sally"`. Use `s1.equals(s2)` to determine if the two `String`s are equivalent. If they are, print `s1 and s2 are equal`, otherwise print `s1 and s2 are not equal`.

Creating Classes

1. Create classes for `Hippo`, `Lion`, `Tiger`, `Monkey`, and `Giraffe`, then create an instance of each one of those classes. Display the objects. Do you see five different ugly-looking (but unique) strings? Count and inspect them.

2. Create a second instance of `Lion` and two more `Giraffe`s. Print those objects. How do they differ from the original objects that you created?

Methods inside Classes

1. Create a `Sailboat` class with methods to raise and lower the sails, printing `Sails raised`, and `Sails lowered`, respectively.

   Create a `Motorboat` class with methods to start and stop the motor, returning `Motor on`, and `Motor off`, respectively. Create an object (instance) of the `Sailboat` class. Use `assert` for verification:

```
val sailboat = new Sailboat
val r1 = sailboat.raise()
assert(r1 == "Sails raised", "Expected Sails raised, Got " + r1)

val r2 = sailboat.lower()
assert(r2 == "Sails lowered", "Expected Sails lowered, Got " + r2)

val motorboat = new Motorboat
val s1 = motorboat.on()
assert(s1 == "Motor on", "Expected Motor on, Got " + s1)

val s2 = motorboat.off()
assert(s2 == "Motor off", "Expected Motor off, Got " + s2)
```

2. Create a new class `Flare`. Define a `light` method in the `Flare` class. Satisfy the following:

```
val flare = new Flare
val f1 = flare.light
assert(f1 == "Flare used!", "Expected Flare used!, Got " + f1)
```

3. In each of the `Sailboat` and `Motorboat` classes, add a method `signal` that creates a `Flare` object and calls the `light` method on the `Flare`. Satisfy the following:

```
val sailboat2 = new Sailboat2
val signal = sailboat2.signal()
assert(signal == "Flare used!", "Expected Flare used! Got " + signal)
```

```
val motorboat2 = new Motorboat2
val flare2 = motorboat2.signal()
assert(flare2 == "Flare used!", "Expected Flare used!, Got " + flare2)
```

Fields in Classes

Given the following code:

```
class Cup {
var percentFull = 0
val max = 100
def add(increase:Int):Int = {
    percentFull += increase
    if(percentFull > max) {
      percentFull = max
    }
    percentFull // Return this value
  }
}
```

1. What happens in `Cups` `add` method if `increase` is a negative value? Is any additional code necessary to satisfy the following tests:

   ```
   val cup = new Cup
   cup.add(45) is 45
   cup.add(-15) is 30
   cup.add(-50) is -20
   ```

   Remember to include the `AtomicTest` class which you will find it under the `atomicscala` folder in the repo. You will need to import the methods from the class using the following statement:

   ```
   import com.atomicscala.AtomicTest._
   ```

2. Add code to handle negative values to ensure that the total never goes below 0. Satisfy the following tests:

   ```
   val cup = new Cup
   cup.add(45) is 45
   cup.add(-55) is 0
   cup.add(10) is 10
   cup.add(-9) is 1
   cup.add(-2) is 0
   ```

3. Can you set `percentFull` from outside the class? Try it, like this:

   ```
   cup.percentFull = 56
   cup.percentFull is 56
   ```

4. Write methods that allow you to both set and get the value of `percentFull`. Satisfy the following:

```
val cup = new Cup
cup.set(56)
cup.get() is 56
```

Vectors

1. Use the REPL to create several `Vector`s, each populated by a different type of data.
2. Use the REPL to see if you can make a `Vector` containing other `Vector`s.
3. Create a `Vector` and populate it with words (which are `String`s). Add a `for` loop that prints each element in the `Vector`. Append to a variable of type `String` to create a `sentence`. Satisfy the following test:

```
sentence.toString() is "The dog visited the fire station "
```

4. Create and initialise two `Vector`s, one containing `Int`s and one containing `Double`s. Call the `sum`, `min`, and `max` operations on each one.
5. Create two `Vector`s of `Int` named `myVector1` and `myVector2`, each initialised to `1, 2, 3, 4, 5, 6`. Use `AtomicTest` to show whether the two are equivalent.