Exercises — Week Six

Pattern Matching, Class Arguments, Named & Default Arguments,
Constructors, Auxiliary Constructors, Case Classes,
Parameterised Types, Functions as Objects, map & reduce

Spring term 2016

You need to have installed the current Scala distribution and the current Java distribution before commencing these exercises. Don't forget the documentation for both distributions as they will come in very useful.

For some of these exercises you will need the test code you used in a previous exercise sheet, namely the classes below the `atomicscala` folder on the repo. Any source code examples are available under the `week06` folder.

Pattern Matching

1. Create a method `forecast` that represents the percentage of cloudiness, and use it to produce a `"weather forecast"` string such as `"Sunny"` (100), `"Mostly Sunny"` (80), `"Partly Sunny"` (50), `"Mostly Cloudy"` (20), and `"Cloudy"` (0). For this exercise, only match for the legal values 100, 80, 50, 20, and 0. All other values should produce `"Unknown"`.

   Satisfy the following tests:

```
package forecast

import atomic.AtomicTest._

object MyForecastTest extends App {
  forecast(100) is "Sunny"
  forecast(80) is "Mostly Sunny"
  forecast(50) is "Partly Sunny"
  forecast(20) is "Mostly Sunny"
  forecast(0) is "Cloudy"
  forecast(15) is "Unknown"

  def forecast(temp: Int): String = ???
}
```

2. Create a `Vector` named `sunnyData` that holds the values (100, 80, 50, 20, 0, 15). Use a `for` loop to call `forecast` with the contents of `sunnyData`. Display the answers and ensure that they match the responses above.

Class Arguments

3. Create a new class `Family` that takes a variable argument list representing the names of family members. Satisfy the following tests:

```
package classargs

object TestArgs extends App {
  val family1 = new Family("Mum", "Dad", "Sally", "Dick")
  family1.familySize() is 4

  val family2 = new Family("Dad", "Mom", "Harry")
  family2.familySize() is 3
}
```

4. Adapt the `Family` class definition to include class arguments for a mother, father, and a variable number of children. What changes did you have to make? Satisfy the following tests:

```
val family3 = new FlexibleFamily("Mum", "Dad", "Sally", "Dick")
family3.familySize() is 4
val family4 = new FlexibleFamily("Dad", "Mom", "Harry")
family4.familySize() is 3
```

5. Write a method that squares a variable argument list of numbers and returns the sum. Satisfy the following tests:

```
squareThem(2) is 4
squareThem(2, 4) is 20
squareThem(1, 2, 4) is 21
```

Named & Default Arguments

6. Define a class `SimpleTime` that takes two arguments: an `Int` that represents hours, and an `Int` that represents minutes. Use named arguments to create a `SimpleTime` object. Satisfy the following tests:

```
val t = new SimpleTime(hours=5, minutes=30)
t.hours is 5
t.minutes is 30
```

7. Using the solution for `SimpleTime` above, default `minutes` to 0 so that you don't have to specify them. Satisfy the following tests:

```
val t2 = new SimpleTime2(hours=10)
t2.hours is 10
t2.minutes is 0
```

8. Create a class `Planet` that has, by default, a single moon. The `Planet` class should have a name (`String`) and description (`String`). Use named arguments to specify the name and description, and a default for the number of moons. Satisfy the following tests:

```
val p = new Planet(name = "Mercury",
description = "small and hot planet", moons = 0)
p.hasMoon is false
```

9. Modify your solution for the previous exercise by changing the order of the arguments that you use to create the `Planet`. Did you have to change any code? Satisfy the following tests:

```
val earth = new Planet(moons = 1, name = "Earth",
    description = "a hospitable planet")
earth.hasMoon is true
```

10. Demonstrate that named and default arguments can be used with methods. Create a class `Item` that takes two class arguments: A `String` for `name` and a `Double` for `price`.

    Add a method `cost` which has named arguments for `grocery (Boolean)`, `medication (Boolean)`, and `taxRate(Double)`. Default `grocery` and `medication` to `false`, `taxRate` to `0.10`.

    In this scenario, groceries and medications are not taxable. Return the total cost of the item by calculating the appropriate tax. Satisfy the following tests:

```
val flour = new Item(name="flour", 4)
flour.cost(grocery=true) is 4
val sunscreen = new Item(
name="sunscreen", 3)
sunscreen.cost() is 3.3
val tv = new Item(name="television", 500)
tv.cost(rate = 0.06) is 530
```

Constructors

11. Create a new class `Tea` that has two methods:

    `describe` — which includes information about whether the tea includes milk, sugar, is decaffeinated, and includes the name; and

    `calories` — which adds 100 calories for milk and 16 calories for sugar.

    Satisfy the following tests:

```
package tea

import atomic.AtomicTest._

object TeaScript extends App {
  val tea = new Tea
  tea.describe is "Earl Grey"
  tea.calories is 0

  val lemonZinger = new Tea(decaf = true, name="Lemon Zinger")
  lemonZinger.describe is "Lemon Zinger decaf"
  lemonZinger.calories is 0

  val sweetGreen = new Tea( name="Jasmine Green", sugar=true)
  sweetGreen.describe is "Jasmine Green + sugar"
  sweetGreen.calories is 16

  val teaLatte = new Tea(sugar=true, milk=true)
  teaLatte.describe is "Earl Grey + milk + sugar"
  teaLatte.calories is 116
}
```

Auxiliary Constructors

12. Create a class called `ClothesWasher` with a default constructor and two auxiliary constructors, one that specifies `modelName` (as a `String`) and one that specifies `capacity` (as a `Double`).

13. Create a class `ClothesWasher2` that looks just like your solution above, but use named and default arguments instead so that you can produce the same results with just a default constructor.

14. Show that the first line of an auxiliary constructor must be a call to the primary constructor.

Case Classes

15. Create a *case class* to represent a `Person` in an address book, complete with `String`s for the name and a `String` for contact information. Satisfy the following tests:

```
val p = Person("Jane", "Smile", "jane@smile.com")
p.first is "Jane"
p.last is "Smile"
p.email is "jane@smile.com"
```

16. Create some `Person` objects. Put the `Person` objects in a `Vector`. Satisfy the following tests:

```
val people = Vector(
    Person("Jane","Smile","jane@smile.com"),
    Person("Ron","House","ron@house.com"),
    Person("Sally","Dove","sally@dove.com"))

people(0) is "Person(Jane,Smile,jane@smile.com)"
people(1) is "Person(Ron,House,ron@house.com)"
people(2) is "Person(Sally,Dove,sally@dove.com)"
```

Parameterised Types

17. Modify the method `explicit` in the following code:

```
import atomic.AtomicTest._

// Return type is inferred:
def inferred(c1: Char, c2: Char, c3: Char) = {
  Vector(c1, c2, c3)
}

// Explicit return type:
def explicit(c1: Char, c2: Char, c3: Char):
Vector[Char] = {
  Vector(c1, c2, c3)
}

inferred('a', 'b', 'c') is "Vector(a, b, c)"
explicit('a', 'b', 'c') is "Vector(a, b, c)"
```

so it creates and returns a `Vector` of `Double`. Satisfy the following tests:

```
explicitDouble(1.0, 2.0, 3.0) is Vector(1.0, 2.0, 3.0)
```

18. Building on the previous exercise, alter `explicit` to take a `Vector` and create and return a `List`. Refer to the `ScalaDoc` for `List`, if necessary.

Satisfy the following tests:

```
explicitList(Vector(10.0, 20.0)) is List(10.0, 20.0)

explicitList(Vector(1, 2, 3)) is List(1.0, 2.0, 3.0)
```

19. Building on the previous exercise, change `explicit` to return a `Set`.

Satisfy the following tests:

```
explicitSet(Vector(10.0, 20.0, 10.0)) is Set(10.0, 20.0)
explicitSet(Vector(1, 2, 3, 2, 3, 4)) is Set(1.0, 2.0, 3.0, 4.0)
```

Functions as Objects

20. Working from your solution to the exercise above, add a comma between each number. Satisfy the following test:

    ```
    str is "1,2,3,4,"
    ```

21. Create an anonymous function that calculates age in *dog years* (by multiplying years by 7). Assign it to a `val` and then call your function.

    Satisfy the following test:

    ```
    val dogYears = // Your function here
    dogYears(10) is 70
    ```

22. Create a `Vector` and append the result of `dogYears` to a `String` for each value in the `Vector`.

    Satisfy the following test:

    ```
    var s = ""
    val v = Vector(1, 5, 7, 8)
    v.foreach(/* Fill this in */)
    s is "7 35 49 56 "
    ```

23. Create an anonymous function to square a list of numbers. Call the function for every element in a `Vector`, using `foreach`.

    Satisfy the following test:

    ```
    var s = ""
    val numbers = Vector(1, 2, 5, 3, 7)
    numbers.foreach(/* Fill this in */)
    s is "1 4 25 9 49 "
    ```