Topic 13: Packages and Imports

Exercise

Jesper Pedersen Notander

Department of Computer Science
Lund University Faculty of Engineering

Learning Scala
Seminar 3

Instructions

- Study the package structure of the code on the next slide.
- Revwrite the code using nested packages and imports so that the classes can be referenced with a simple name. The package structure should be kept intact.

Code

```
package manufacturers {
  class Property(val name: String)
  abstract class Car(val properties: List[Property]) {
    def compareProperties(other: Car): List[Property] = List[Property]()
  }
}

package manufacturers.framking {
  class EcoCar(properties: List[manufacturers.Property]) extends
        manufacturers.Car(properties)
}

package manufacturers.allvelo {
  class EcoCar(properties: List[manufacturers.Property]) extends
        manufacturers.Car(properties)
}

class Retailer {
  def printComparison(car1: manufacturers.Car, car2: manufacturers.Car) {
    car1.compareProperties(car2).foreach((x) => println("" + x))
  }
}

object RetailerTest extends App {
  val car1 = new manufacturers.framking.EcoCar(Nil);
  val car2 = new manufacturers.allvelo.EcoCar(Nil);
  (new Retailer()).printComparison(car1, car2);
}
```

Solution

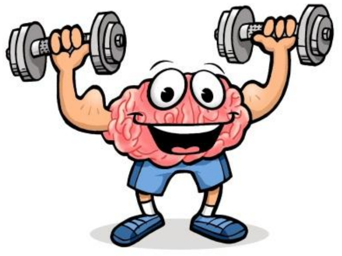
```
package manufacturers {
  case class Property(name: String)
  abstract class Car(val properties: List[Property]) {
    def compareProperties(other: Car): List[Property] = List[Property]()
  }
  package framking {
    class EcoCar(properties: List[Property]) extends Car(properties)
  }
  package allvelo {
    class EcoCar(properties: List[Property]) extends Car(properties)
  }
}

class Retailer {
  import manufacturers.Car
  def printComparison(car1: Car, car2: Car) {
    car1.compareProperties(car2).foreach((x) => println("" + x))
  }
}

object RetailerTest extends App {
  import manufacturers.framking.{EcoCar=>FramKingCar}
  import manufacturers.allvelo.{EcoCar=>AllveloCar}

  val car1     = new FramKingCar(Nil);
  val car2     = new AllveloCar(Nil);
  (new Retailer()).printComparison(car1, car2);
}
```

Exercise: Assertions



The “Triangle Program” is a classic testing exercise. It was first published in Glenford Meyer’s *The Art of Software Testing* in 1979, intended for punchcards rather than Scala.

The program should accept input as three sides of a triangle and give output on what type of triangle it is, i.e. “Scalene” (no sides are same), “Isosceles” (any two sides are same) or “Equilateral” (All the three sides are same).

Implement the Triangle Program in Scala. Add assertions to make sure no input results in invalid triangles. Enable assertions and see what happens for:

(i) negative values

(ii) impossible combinations of side lengths

Topic 15 - Case Classes and Pattern Matching

Alfred Theorin

2012-08-26

Expression Evaluation

Write a function that evaluates expressions using pattern matching, given the following class hierarchy. It should at least support the binary operations addition, subtraction, and multiplication and unary minus.

```
sealed abstract class Expr
case class Var(name: String) extends Expr
case class Num(num: Double) extends Expr
case class UnOp(op: String, expr: Expr) extends Expr
case class BinOp(op: String, l: Expr, r:Expr) extends Expr
```

The signature of the function could be:

```
def evaluate(expr: Expr, vars: Map[String, Double]): Double
```

For example `evaluate(BinOp("+", Num(1), Var("x")), Map("x"->2))` should return 3.

Hint: Use a nested `match` for the variable lookup.

Exercise

Create an integer list containing 10 elements and sort it in ascending order without using sort method.

Exercise- Collections

Create a simple (sorted!) phone book with a few of your friends and their numbers.

Create a function `drunkenDial` that returns a random friend and his or her phone number.

Exercises for chapter 18
Stateful Objects

Björn Regnell

Exercise 18.1: Age as property

- Make a class “Person” that has a private field for keeping track of a person’s age, providing a setter and getter for the property “age” that never can be less than 0
- What happens if you only provide a setter but no getter?

Exercise 18.2: Caching factorial values

- Make an object that is purely functional on the outside called SmartFactorial that keeps track of already computed factorial in an internal state variable “cache” of type collection.immutable.Map[Int, BigInt]

Exercise 18.3: Stateful Scheduler

Explain what the CanSchedule trait does.

How can this trait be used on the client side?

How is the state of instances of this trait changed over time?

Assume that loop{block} executes block as an eternal loop in another thread.

```
trait CanSchedule[T] {
     import scala.collection.immutable.Queue
     def workItem: T
     type Work = T => Unit
     var agenda = Queue.empty[Work]
     def schedule(w:Work) = agenda :+= w
     loop {
        if (!agenda.isEmpty) {
           val (firstWork, agendaMinusFirst) = agenda.dequeue
           agenda = agendaMinusFirst
           firstWork(workItem)
        }
     }
  }
```

The following code will not compile, because A is covariant and is used in a wrong way.

```
class Box[+A](private[this] var value: A) {
  def get: A = value
  def set(x: A) {
    value = x
  }
}
```

a) Compile the code and read the error message.
b) Construct an example where the use of this could go wrong.

Exercise: Abstract members

Niklas Fors

August 27, 2012

Exercise

Rewrite the following code and use an abstract type instead of a type parameter. You can run the code as a script (`$ scala file.scala`).

```
abstract class AbsItr[T] {
    def hasNext: Boolean
    def next: T
}
trait RichItr[T] extends AbsItr[T] {
    def foreach(f: T => Unit) { while (hasNext) f(next) }
}
class StringItr(s: String) extends AbsItr[Char] {
    private var i = 0
    def hasNext = i < s.length()
    def next = { val ch = s charAt i; i += 1; ch }
}

class Itr extends StringItr("hello world") with RichItr[Char]
val itr = new Itr
itr foreach println
```

(Solution: http://docs.scala-lang.org/tutorials/tour/mixin-class-composition.html)

- Assume that you had a gigantic project for handling personnel data so you divided it to two parts and assigned it to two programmers. When you merged the two parts, the result was this:

```
final class Personnel(n: String, i: Int){
    val name: String = n
    val id: Int = i

}

def summarize(p: Personnel) = p.name concat "_" concat p.id

var p = new Personnel("Some Guy", 1)

print(summarize(p))
```

- Since it would hurt the feelings of the programmers if you would change their source code, fix this code without changing any existing code. (Lucky you, one line is enough)

`for` expressions revisited

The class `Country` is defined as follows:

```
case class Country(name: String, cities: String*)
```

The `val countries` is of type `List[Country]`, and is initialized with multiple instances of `Country`.

a) Write a `for` expression that finds all countries that start with the letter ‘S’.

b) Write a for expression that find all cities that have no more than 5 letters in their name (duplicates OK).

c) Write a for expression that finds all cities that exist in 2 or more countries (duplicates OK).

```
a) for (c <- countries; if c.name startsWith "S") yield c
b) for (c <- countries; z <- c.cities; if z.length <= 5) yield z
c) for (c1 <- countries; c2 <- countries; if c1 != c2; z1 <-
     c1.cities; z2 <- c2.cities; if z1 == z2) yield z1
```

Problem

Is this legal code, and if so, what will it print:

```
val hamlet =
  ”to be or not to be”.split(” ”)
val independence =
  ”we hold these truths to be self-evident”.split(” ”)
(0 until (hamlet.length max independence.length)).map {
  hamlet orElse independence}.foreach(println)
```