Stream programming Summer 2019

Lecture Hierarchy, collections, functions — 13.03, 2019

dr inż. Wojciech Macyna *Scribe: Kamila Kaszuba*

1 Overview

In this lecture after repetition from defining classes and their hierarchy, we get acquaintanted with collections and functions, also function closures.

2 Repetition

We can not define an object of abstract class, but we can use it to create more specified classes extending this one.

```
abstract class Animal{
 var animalType = "animal"}
class Bird extends Animal
```

We can use more than 1 trait to extend a class. We couldn't do that using class instead of trait, because extension of more than 1 class is forbidden.

```
trait Wing{
 def setnumberofWings(x:Int)}
trait Leg{
 def setnumberofLegs(x:Int)}
class Bird extends Animal with Wing and Leg
```

We can make it even more complexed to have more specified class:

```
abstract class Bird extends Animal with Wing and Leg
class Sparrow extends Bird
```

Then we can define objects like

```
val s = new Sparrow
val s1 = new Sparrow
val s2 = new Sparrow
```

```
var r:Array[Sparrow] = newArray[Sparrow](2)
r(0) = s1
r(1) = s

var b:Array[Bird] = newArray[Bird](2)
b(0) = s
b(1) = s1
```

That is legal, because each Sparrow is a Bird. This is also legal, because s,s1 are animals with wings:

```
var w:Array[Wing] = newArray[Wing](2)
w(0) = s
w(1) = s1
```

3 Collections

3.1 Set

We can define a set like this:

```
var s:Set[Int] = Set(1,3,5,7,9)
var s1:Set[Int] = Set(1,3,5,7,9)
```

We can iterate objects in set using commend foreach:

```
s.foreach((i)->println(i))
```

We can get a set's head just like that:

```
s.head
```

There is also special operation on collections called concatonation:

```
var s2 = s ++ s1
```

3.2 List

We can define a list like this:

```
val fruit:List[String] = List('orange','apple')
```

but then this is illegal, because val object is not changeable:

```
fruit(0) = 'banana'
```

Here are some operations on lists:

```
val SomeNumber = List(-11,-10,-5,0,5,10)
SomeNumber.foreach((x:Int) => println(x))
SomeNumber.filter((x:Int) => x > 0)
SomeNumber.filter(x => x > 0)
```

Last one is also legal, beacause we previosly definied SomeNumber and we know that its elements are integers. We can do the same even like that:

```
SomeNumber.filter( _ > 0 )
```

3.3 Map

Map is a collection with keys and values. To define a map we need to type:

```
val myMap Map[Char,Int] = Map()
```

Adding element to the map:

```
myMap = myMap + ('I'->1)
myMap = myMap + ('K'->9)
```

We can also iterate elements of the map, for example:

```
map.key.foreach{i->print('key')
   println('Value')}
```

4 Function programming

Short definition of function literal:

```
(x:Int) => x + 1
```

Function increase takes 1 argument of type integer and increases it by 1:

```
var increase = (x:Int) => x+1
```

Example of more complexed function:

```
var increase = (x:Int) => {print("This")
       print("is")
       x+1}
```

More examples:

1. a sum of 2 integers:

   ```
   var f = (_:Int) + (_:Int)
   f(5,10) // 15
   ```

2. illegal, type of arguments is unknown:

   ```
   val f1 = _ + _
   ```

3. ```
   def sum(a:Int,b:Int,c:Int) = a + b + c
   sum(1,3,5)  // 9
   ```

4. ```
   val a = sum
   a(1,3,5)  // 9
   val b = sum(1,_:Int,3)
   b(2)  // 6 (=1+2+3)
   ```

4.1 Function closure

This kind of function takes other function as one of arguments, so we have to pass function name to it as an argument. Example:

```
class ClosureExample{

def exec(f:(String) => Unit, name:String)
{f(name)}

def main(args:Array[String]){
var hello="Hello"

def sayHello(name:String){
println(hello,name)
}
}
}

val v = new ClosureExample
v.exec(sayHello,"Something")  // Hello Something
```