Scala

Session 3

Muhammad Asif Fayyaz

Course Contents

1. The Basics
2. **Getting up to speed**
3. Collections
4. Classes and Objects
5. Inheritance and Traits
6. Functional Programming

Today's Session (Getting Up to Speed)

- Exceptions
- Functions
- Higher Order Functions

Exceptions

- `$ throw new IllegalArgumentException("x should not be negative")`
- Exception Example

```
$ if (x >= 0) {
$   sqrt(x)
$ } else {
$   throw new IllegalArgumentException("x should not be negative")
$ }
```

Functions

Parameters and Return Types

```
scala> def square(x: Double) = x * x
scala> square(square(4))
scala> def sumOfSquares(x: Double, y: Double) = square(x) + square(y)
```

val vs def

- `scala> def x = square(2)`
- `scala> val y = square(2)`
- x refers to square(2)
- y refers to 4 and not square(2)
- def: right hand side is evaluated on its use
- val: right hand side is evaluated at the point of its definition. Afterwards, the name refers to the value

- Another example:
  - scala> def loop: Boolean = loop
  - scala> def x = loop
  - scala> val y = loop

Task

Lets write a program to calculate the square root.

> `def sqrt(x: Double): Double=...`

To compute sqrt(x) :

- Start with an initial estimate y (let's pick y = 1).
- Repeatedly improve the estimate by taking the mean of y and x/y .

Example: x = 2

Estimation	Quotient	Mean
1	2 / 1 = 2	1.5
1.5	2 / 1.5 = 1.333	1.4167
1.4167	2/1.4167 = 1.4118	1.4142
1.4142	……..	……..

Task (Cont…)

- First, define a function which computes one iteration step

```
> def sqrtIter(guess: Double, x: Double): Double =
    if (isGoodEnough(guess, x)) guess
    else sqrtIter(improve(guess, x), x)
```

- Note that sqrtIter is recursive, its right-hand side calls itself.
- Recursive functions need an explicit return type in Scala.
- For non-recursive functions, the return type is optional

Task (Cont…)

- Now Define a function improve to improve an estimate and a test to check for termination:

```
> def improve(guess: Double, x: Double) = (guess + x / guess) / 2
> def isGoodEnough(guess: Double, x: Double) = abs((guess * guess) - x) < 0.001
```

- Third, define the sqrt function:

```
> def sqrt(x: Double) = sqrtIter(1.0, x)
```

How can we improve this code?

- It is good to split up the functionality into multiple functions.
- But, functions like sqrtIter, improve, isGoodEnough matter only to the implementation of sqrt.
- They should not be exposed.
- We can achieve this and at the same time avoid name-space pollution by moving them inside the main function.

Lets Improve sqrt

Blocks in Scala

- A block is delimited by braces { ... } .

```
{ val x = f(3)
  x * x
}
```

- It contains a sequence of definitions or expressions.
- The last element of a block is an expression that defines its value.
- This return expression can be preceded by auxiliary definitions.
- Blocks are themselves expressions; a block may appear everywhere an expression can.

Blocks and Visibility

```
val x = 0
def f(y: Int) = y + 1
val result = {
     val x = f(3)
     x * x
}
```

- The definitions inside a block are only visible from within the block.
- The definitions inside a block shadow definitions of the same names outside the block.

Exercise: Scope Rules

- Question: What is the value of result in the following program?

```
val x = 0
def f(y: Int) = y + 1
val result = {
     val x = f(3)
     x * x
} + x
```

- Possible answers:
  - 0
  - 16
  - 32
  - reduction does not terminate

Lexical Scoping

- Definitions of outer blocks are visible inside a block unless they are shadowed.
- Therefore, we can simplify sqrt by eliminating redundant occurrences of the x parameter, which means everywhere the same thing:

The sqrt Function, Take 3

Semicolons

In Scala, semicolons at the end of lines are in most cases optional

You could write

```
val x = 1;
```

but most people would omit the semicolon.

On the other hand, if there are more than one statements on a line,

they need to be separated by semicolons:

```
val y = x + 1; y * y
```

Semicolons and infix operators

One issue with Scala's semicolon convention is how to write expressions that span several lines. For instance

```
someLongExpression
+ someOtherExpression
```

would be interpreted as two expressions:

```
someLongExpression;
+ someOtherExpression
```

Semicolons and infix operators

There are two ways to overcome this problem.

You could write the multi-line expression in parentheses, because semicolons are never inserted inside (...) :

```
(someLongExpression
+ someOtherExpression)
```

Or you could write the operator on the first line, because this tells the Scala compiler that the expression is not yet finished:

```
someLongExpression +
someOtherExpression
```

Functions

- return type is not needed except in recursive function.
- Default arguments
- Named arguments
- Variable number of arguments
- Procedures
- Lazy Values

Higher Order Functions

- Scala treat functions as first-class values.
- This means that like any other value, a function can be passed as a parameter and returned as a result.
- This provides a flexible way to compose programs.
- Functions that take other functions as parameters or that return functions are called higher order functions

Example

Take the sum of the integers between a and b:

```
def sumInts(a: Int, b: Int): Int =
     if (a > b) 0 else a + sumInts(a + 1, b)
```

Take the sum of the cubes of all the integers between a and b :

```
def cube(x: Int): Int = x * x * x
def sumCubes(a: Int, b: Int): Int =
    if (a > b) 0 else cube(a) + sumCubes(a + 1, b)
```

Take the sum of the factorials of all the integers between a and b :

```
def sumFactorials(a: Int, b: Int): Int =
     if (a > b) 0 else fact(a) + sumFactorials(a + 1, b)
```

The code above looks very similar. May be we can improve it

Summing with Higher-Order Function

Let's define:

```
def sum(f: Int => Int, a: Int, b: Int): Int =
   if (a > b) 0
   else f(a) + sum(f, a + 1, b)
```

We can then write:

```
def sumInts(a: Int, b: Int) = sum(id, a, b)
def sumCubes(a: Int, b: Int) = sum(cube, a, b)
def sumFactorials(a: Int, b: Int) = sum(fact, a, b)
```

where

```
def id(x: Int): Int = x
def cube(x: Int): Int = x * x * x
def fact(x: Int): Int = if (x == 0) 1 else fact(x - 1)
```