Lecture 3
Functions in Scala
Instructor: Hossein Hojjat

September 1, 2017
Resources

- Scala courses on Coursera
- https://www.coursera.org/specializations/scala

Several books on Scala
## Getting Started in Scala

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scala:</td>
<td>- runs compiled Scala code (like java)</td>
</tr>
<tr>
<td></td>
<td>- if no arguments are supplied, starts the Scala interpreter</td>
</tr>
<tr>
<td>scalac:</td>
<td>- compiles Scala code (like javac)</td>
</tr>
<tr>
<td>sbt:</td>
<td>- build tool for larger project (incremental compilation, dependency and library management, ...)</td>
</tr>
</tbody>
</table>

There are plugins available for popular IDEs such as Eclipse.
For more information visit [http://www.scala-lang.org](http://www.scala-lang.org)
Two ways to introduce variables:

- **Immutable:** The “right” way (functional)
- **Mutable:** The “if-you-must” way (non-functional)

- With the `val` keyword:
  ```scala
  scala> val x = 5
  x: Int = 5
  ```

  Don’t try to change (re-assign) such a variable
  ```scala
  scala> x = x + 1
  <console>:12: error: reassignment to val
  x = x + 1
  ^
  ```

- If you absolutely must: mutable variables
  ```scala
  scala> var z = 1
  z: Int = 1
  scala> z += 1
  ```
• Use `def` to declare methods

```scala
def max(x: Int, y: Int): Int = {
  if (x < y) {
    return y;
  } else {
    return x;
  }
}
```

• or shorter:

```scala
def max(x: Int, y: Int) = if (x < y) y else x
```

• No explicit return, just a result expression
• Block evaluates to last expression

```scala
def adder(x: Int, y: Int): Int = {
  val z = x + y
  z
}
```

• Call a function in the usual way (parentheses required)

```scala
scala> adder(5, 7)
res0: Int = 12
```
Scala is Object-Oriented

- Uniform object model
- Every value is an object (including primitive values and functions)
- Every application of an operator is a method call
  - 1 + 2 is short for 1.+(2)
- No static class members (instead: singleton objects)
Singleton Objects

- Scala has singleton objects instead of static members
- A singleton object definition looks like a class definition, except that the keyword `class` is replaced by `object`

```scala
object Main {
  def main(args: Array[String]) {
    println("Hello world!")
  }
}
```

- or shorter:

```scala
object Main extends App {
  println("Hello world!");
}
```
Functions are Objects

- Functions are just objects with `apply` method
- When you write `f(args)` Scala reads `f.apply(args)`

```scala
scala> object inc { def apply(x: Int) = x + 1 }
```

- Call in the usual function call way

```scala
scala> inc(5)
res: Int = 6
```
def plus (x:Int, y:Int) = x + y

1. plus(1,3) → 1 + 3 → 4
2. plus(3 * 2, plus(1,3))
def plus (x:Int, y:Int) = x + y

1. plus(1,3) → 1 + 3 → 4

2. plus(3 * 2, plus(1,3))

→ plus(6, 1 + 3)
→ plus(6, 4)
→ 6 + 4
→ 10
Function Execution by Substitution

```
def plus (x:Int, y:Int) = x + y

1.  plus(1,3)  →  1 + 3  →  4

2.  plus(3*2, plus(1,3))
    →  plus(6, 1 + 3)
    →  plus(6, 4)
    →  6 + 4
    →  10
```

Final answer does not depend upon the order in which reductions are performed
def plus (x:Int, y:Int) = x + y

1. plus(1,3) → 1 + 3 → 4
2. plus(3 * 2, plus(1,3))

→ plus(6, 1 + 3)
→ plus(6, 4)
→ 6 + 4
→ 10

→ (3 * 2) + plus(1, 3)
→ 6 + (1 + 3)
→ 6 + 4
→ 10

Final answer does not depend upon the order in which reductions are performed
Confluence

- The order in which reductions are performed in a functional program does not affect the final outcome
- All functional programs (right or wrong) have repeatable behavior
- This is true for all functional programs regardless whether they are right or wrong

(A formal definition will be given later)
In programming language theory, an entity is first-class if
1. it can be passed as an argument,
2. returned as a result from a function,
3. stored in variables,
4. and can be constructed at run time

In functional programming languages functions are first-class citizens

First-class functions provide a flexible way to compose programs
Example: Calculating Sum of Integers

Compute the sum of integers smaller than or equal to \( n \)

```java
def sum (n: Int): Int =
    if (n == 0) 0 else (n + sum(n - 1))
```
Example: Calculating Sum of Integers

Compute the sum of integers smaller than or equal to \( n \)

```java
def sum (n: Int): Int =
  if (n == 0) 0 else (n + sum(n - 1))
```

Compute the sum of squares of integers smaller than or equal to \( n \)

```java
def sqr (n: Int) = n * n
def sum_sqr (n: Int): Int =
  if(n == 0) 0 else (sqr(n) + sum_sqr(n - 1))
```
Example: Calculating Sum of Integers

Compute the sum of integers smaller than or equal to \( n \)

```python
def sum(n: Int): Int =
    if (n == 0) 0 else (n + sum(n - 1))
```

Compute the sum of squares of integers smaller than or equal to \( n \)

```python
def sqr(n: Int) = n * n
def sum_sqr(n: Int): Int =
    if(n == 0) 0 else (sqr(n) + sum_sqr(n - 1))
```

Compute the sum of cubes of integers smaller than or equal to \( n \)

```python
def cube(n :Int) = n * n * n
def sum_cube(n: Int): Int =
    if(n == 0) 0 else (cube(n) + sum_cube(n - 1))
```
Example: Calculating Sum of Integers

Compute the sum of integers smaller than or equal to $n$

```java
def sum(n: Int): Int =
    if (n == 0) 0 else (n + sum(n - 1))
```

Compute the sum of squares of integers smaller than or equal to $n$

```java
def sqr(n: Int) = n * n
def sum_sqr(n: Int): Int =
    if(n == 0) 0 else (sqr(n) + sum_sqr(n - 1))
```

Compute the sum of cubes of integers smaller than or equal to $n$

```java
def cube(n: Int) = n * n * n
def sum_cube(n: Int): Int =
    if(n == 0) 0 else (cube(n) + sum_cube(n - 1))
```
Example: Calculating Sum of $f(n)$

Compute the sum of $f(i)$ where $i$ is smaller than or equal to $n$:

$$\sum_{i=0}^{n} f(i)$$
Example: Calculating Sum of $f(n)$

Compute the sum of $f(i)$
where $i$ is smaller than or equal to $n$:

\[ \sum_{i=0}^{n} f(i) \]

- First-class functions give us the power to write a general `sum_f`:

```scala
def sum_f(n: Int, f: Int => Int): Int = 
  if (n == 0) 0 else f(n) + sum_f(n - 1, f)
```

We can use the general `sum_f` function to define:

```scala
def sum_sqr(n: Int) = sum_f(n, sqr)
def sum_cube(n: Int) = sum_f(n, cube)
```

- But we still need to define `sqr` and `cube`:

```scala
def sqr(n: Int) = n * n
def cube(n: Int) = n * n * n
```
Example: Calculating Sum of $f(n)$

Compute the sum of $f(i)$
where $i$ is smaller than or equal to $n$:

\[
\sum_{i=0}^{n} f(i)
\]

- First-class functions give us the power to write a general `sum_f`:

```scala
def sum_f(n: Int, f: Int => Int): Int = 
  if (n == 0) 0 else f(n) + sum_f(n - 1, f)
```

- We can use the general `sum_f` function to define:

```scala
def sum_sqr(n: Int) = sum_f(n, sqr)
def sum_cube(n: Int) = sum_f(n, cube)
```

- But we still need to define `sqr` and `cube`:

```scala
def sqr(n: Int) = n * n
def cube(n: Int) = n * n * n
```
Example: Calculating Sum of $f(n)$

```scala
def sqr (n: Int) = n * n
def sum_sqr (n: Int): Int = sum_f(n, sqr)
```
Example: Calculating Sum of $f(n)$

```scala
def sqr (n: Int) = n * n

def sum_sqr (n: Int): Int = sum_f (n, sqr)
```

- Compare to integers: we do not need to first define and name an integer and then use it

```scala
val z = 0
def is_zero (n: Int) = (n == z)
```

```scala
def is_zero (n: Int) = (n == 0)
```
Example: Calculating Sum of $f(n)$

\[
\text{def } \text{sqr} \ (n: \text{Int}) = n \times n
\]
\[
\text{def } \text{sum_sqr} \ (n: \text{Int}): \text{Int} = \text{sum_f}(n, \text{sqr})
\]

- Compare to integers: we do not need to first define and name an integer and then use it

\[
\text{val } z = 0
\]
\[
\text{def } \text{is_zero} \ (n: \text{Int})= (n==z)
\]

- Analogous to integer literals we want to have function literals to let us write a function without giving it a name

- Such literals are called “anonymous functions”
  - (function without a name)
Anonymous Functions

- Function that maps \( x \) to “\( x \) times \( x \)” : \( x \mapsto (x \times x) \)
  - Mathematics: \( \lambda x. x \times x \)
  - Scala: \((x : \text{Int}) \Rightarrow x * x\)

- \( x \) is the parameter and \((x \times x)\) is the body of the function
def sum_f(n: Int, f: Int => Int): Int =
  if(n == 0) 0 else f(n) + sum_f(n - 1, f)

def sum_sqr(n: Int): Int = sum_f(n, (x: Int) => x * x)

def sum_cube(n: Int): Int = sum_f(n, (x: Int) => x * x * x)
• Shortened form of anonymous functions: replace named parameter with wildcard operator (_)
• You can use _ instead of a variable name when the parameter appears only once in your function

```scala
scala> val double: Int => Int = (_, _ * 2)
double: Int => Int = <function1>

scala> double(5)
res0: Int = 10
```