

Scala Basics

what is this all about?



Semicolons ;

- Look ma, no semicolon! (in most cases optional)
 - Use if you want several definitions in one line
- How about long expressions?
 - use parantheses ()
 - write operator as last element of first line



Expressions

- Expressions are computable statements
- Primitive: 1, true, "Hello"
- Compound: 1+2
- You can output results of expressions using **println**
- Evaluation: left -> right

```
println(1 + 1) // 2  
println("Hello!") // Hello!
```

Values

- You can name results of expressions with the **val** keyword
- Named results, such as x here, are called values. Referencing a value does not re-compute it.
- Values cannot be re-assigned.
- Types of values can be inferred, but you can also explicitly state the type

```
val x = 1 + 1
println(x) // 2
val y: Int = 1 + 1
```

Variables



- Variables are like values, except you can re-assign them
- You can define a variable with the **var** keyword
- Type is inferred, or can be explicit
- Everytime you use **var** in a functional language, a unicorn dies

```
var x = 1 + 1  
x = 3  
println(x * x)
```

Functions

- Functions are expressions that take parameters.
- Functions can be named, or anonymous
- You can define an anonymous function (i.e. no name) that returns a given integer plus one

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

- On the left of `=>` is a list of parameters. On the right is an expression involving the parameters.
- You can also name functions.

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

- Functions may take multiple parameters
- Or, no parameters at all

```
val getTheAnswer = () => 42
```

The diagram shows a function definition with arrows pointing to its components and labels:

```
def max(x: Int, y: Int): Int = {  
  if (x > y)  
    x  
  else  
    y  
}
```

Annotations:

- "def" starts a function definition (points to `def`)
- function name (points to `max`)
- parameter list in parentheses (points to `(x: Int, y: Int)`)
- function's result type (points to `: Int`)
- equals sign (points to `=`)
- function body in curly braces (points to the block `{ if (x > y) x else y }`)

Methods

- Methods look and behave very similar to functions, but there are a few key differences between them.
- Methods are defined with the `def` keyword
- `def` is followed by a name, parameter lists, a return type, and a body.

```
def add(x: Int, y: Int): Int = x + y  
println(add(1, 2)) // 3
```

Parametrized Definition

- So, a method is a parametrized definition
- When you use the keyword **def**, what is on the right hand side of = is evaluated whenever the method is called
- A method can be parametrized
- The last expression in the body is the method's return value
- Contrast to val - evaluated when created

```
scala> def square(d:Double) = d*d  
square: (d: Double)Double
```

```
scala> square(4)  
res0: Double = 16.0
```

```
scala> def sumOfSquares(d:Double, e:Double) = square(d) + square(e)  
sumOfSquares: (d: Double, e: Double)Double
```

```
scala> sumOfSquares(3,4)  
res1: Double = 25.0
```


Value definitions

- `def` - by name - evaluated when used
- `val` - by value - evaluated at the point of definition, the name refers to the value

Function Calls & Lambda Calculus

- Evaluate from left to right
- Replace name by right hand side value
- Replace formal parameters by actual arguments
- Substitution model - reduce expression to value
- lambda calculus
- equivalent to Turing machine (Alonzo Church)
- only substitutions without side effects (C++)
- termination: does every expression reduce to a value (finite no of steps)? no

Main Method

- Like in Java, the main method is an entry point of a program
- The Java Virtual Machine requires a main method to be named main and take one argument, an array of strings

```
object Main {  
  def main(args: Array[String]): Unit =  
    println("Hello, Scala developer!")  
}
```

Value Types

- There are nine predefined value types and they are non-nullable
 - Double, Float, Long, Int, Short, Byte, Char, Unit, and Boolean
 - Unit is a value type which carries no meaningful information. There is exactly one instance of Unit which can be declared literally like so: ()

```
val list: List[Any] = List(  
  "a string",  
  732,    // an integer  
  'c',    // a character  
  true,   // a boolean value  
  () => "an anonymous function returning a string"  
)  
  
list.foreach(element => println(element))
```

Decisions

- Use the if-else expression
- Scala is an expression language (not statements like in java)
 - everything is an expression - also the if-else

```
//what is the value ?  
val exp = if( 2 == 3) false else true
```

Iterations

- To iterate over a collection, you call the foreach method and pass in a function
- In this case, you're passing in a function literal that takes one parameter named arg.
- The body of the function is println(arg)
- If a function literal consists of one statement that takes a single argument, you need not explicitly name and specify the argument.

```
args.foreach(arg => println(arg))  
//or  
args.foreach(println)
```

You can also loop



... but beware of dead unicorns, this is functional style

```
var i = 0
while (i < args.length) {
  if (i != 0)
    print(" ")
  print(args(i))
  i += 1
}
```

Recursion

- Always state explicit return type (not required for non-recursive functions)
- Make sure it returns at some point!
- Note that the recursive call is the last thing that happens in the evaluation of function approximate's body.
- Functions like approximate, which call themselves as their last action, are called **tail recursive**.
- The Scala compiler detects tail recursion and replaces it with a jump back to the beginning of the function, after updating the function parameters with the new values
- Tail recursion is good - it is reusing stack frames - you don't run out of stack space
- Often, a recursive solution is more elegant and concise than a loop-based one. If the solution is tail recursive, there won't be any runtime overhead to be paid.

```
def approximate(guess: Double): Double =  
  if (isGoodEnough(guess)) guess  
  else approximate(improve(guess))
```


Blocks

- Blocks define a lexical scope for
 - nesting functions
 - definitions and expressions
- definitions in block only visible from block
- definitions in block shadow definitions outside the block
- Blocks have a value!
- **Last expression** is the “value” of the block

```
println({  
  val x = 1 + 1  
  x + 1  
}) // 3
```

REPL basics

- Read - Evaluate - Print - Loop
- The Scala REPL is a tool for evaluating expressions in Scala
- Useful commands
 - :help
 - :t, :type - finds the type
 - :t function_name _ - finds type of function
 - :q, :quit

```
Agatas-MacBook-Pro:~ agatanoair$ scala
Welcome to Scala 2.12.4 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_74).
Type in expressions for evaluation. Or try :help.

scala>
```

Lab Time !

language_basics_02

