A Crash Course on Scala

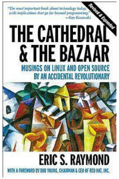
Amir H. Payberah & Lars Kroll
payberah@kth.se & lkroll@kth.se
10/09/2018

Introduction

- Scala: scalable language
- A blend of object-oriented and functional programming.
- Runs on the Java Virtual Machine.
- Designed by Martin Odersky at EPFL.

Cathedral vs. Bazaar

- Two metaphors for software development (Eric S. Raymond)



Cathedral vs. Bazaar

- The cathedral
  - A near-perfect building that takes a long time to build.
  - Once built, it stays unchanged for a long time.
- The bazaar
  - Adapted and extended each day by the people working in it.
  - Open-source software development.

Cathedral vs. Bazaar

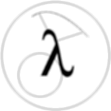
- The cathedral
  - A near-perfect building that takes a long time to build.
  - Once built, it stays unchanged for a long time.
- The bazaar
  - Adapted and extended each day by the people working in it.
  - Open-source software development.

Scala is much more like a bazaar than a cathedral!

Functional Programming (FP)

- In a restricted sense: programming without mutable variables, assignments, loops, and other imperative control structures.

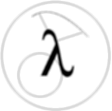
- In a wider sense: focusing on the functions.



Functional Programming (FP)

- In a restricted sense: programming without mutable variables, assignments, loops, and other imperative control structures.

- In a wider sense: focusing on the functions.



- Functions can be values that are produced, consumed, and composed.

FP Languages (1/2)

- In a restricted sense: a language that does not have mutable variables, assignments, or imperative control structures.
- In a wider sense: it enables the construction of programs that focus on functions.

FP Languages (1/2)

- In a restricted sense: a language that does not have mutable variables, assignments, or imperative control structures.
- In a wider sense: it enables the construction of programs that focus on functions.
- Functions are first-class citizens:
  - Defined anywhere (including inside other functions).
  - Passed as parameters to functions and returned as results.
  - Operators to compose functions.

FP Languages (2/2)

- In the restricted sense:
  - Pure Lisp, XSLT, XPath, XQuery, Erlang
- In the wider sense:
  - Lisp, Scheme, Racket, Clojure, SML, OCaml, Haskell, Scala, Smalltalk, Ruby

Scala

Clojure

ERLANG


HELLO
WORLD!

The “Hello, world!” Program

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

The “Hello, world!” Program

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

Or simply

```
object HelloWorld extends App {
    println("Hello, world!")
}
```

Run It Interactively!

```
> scala
This is a Scala shell.
Type in expressions to have them evaluated.
Type :help for more information.

scala> object HelloWorld {
     |   def main(args: Array[String]): Unit = {
     |     println("Hello, world!")
     |   }
     | }
defined module HelloWorld

scala> HelloWorld.main(null)
Hello, world!

scala>:q
>
```

Compile and Execute It!

```
// Compile it!
> scalac HelloWorld.scala
// Execute it!
> scala HelloWorld
```

Compile and Execute It!

```
// Compile it!
> scalac HelloWorld.scala
// Execute it!
> scala HelloWorld
```

It's always better to separate sources and build products.

```
// Compile it!
> mkdir classes
> scalac -d classes HelloWorld.scala
// Execute it!
> scala -cp classes HelloWorld
```

Script It!

```
# script.sh
#!/bin/bash
exec scala $0 $@
!#

object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello, world!")
  }
}

HelloWorld.main(null)

# Execute it!
> ./script.sh
```

Script It!

Or use Ammonite (http://ammonite.io/).

```
# script.sh
#!/usr/bin/env amm

@main
def main(): Unit = {
  println("Hello, world!")
}

# Execute it!
> ./script.sh
```

Outline

- Scala basics
- Functions
- Collections
- Classes and objects
- SBT

Outline

- Scala basics
- Functions
- Collections
- Classes and objects
- SBT

Scala Variables

- Values: immutable
- Variables: mutable

```scala
var myVar: Int = 0
val myVal: Int = 1

// Scala figures out the type of variables based on the assigned values
var myVar = 0
val myVal = 1

// If the initial values are not assigned, it cannot figure out the type
var myVar: Int
val myVal: Int
```

Always use immutable values by default, unless you know for certain they need to be mutable.

Scala Data Types

- Boolean: `true` or `false` literals
- Byte: 8 bit signed value
- Short: 16 bit signed value
- Char: 16 bit unsigned Unicode character
- Int: 32 bit signed value
- Long: 64 bit signed value
- Float: 32 bit IEEE 754 single-precision float
- Double: 64 bit IEEE 754 double-precision float
- String: A sequence of characters
- Unit: A unique singleton value, it's literal is written ()

```
var myInt: Int
var myString: String
```

If ... Else

```
var x = 30;

if (x == 10) {
  println("Value of X is 10");
} else if (x == 20) {
  println("Value of X is 20");
} else {
  println("This is else statement");
}
```

Note that in Scala if-else blocks are expressions and the compiler will infer a return type for you.

Loops (1/3)

```
var a = 10

// do-while
do {
  println(s"Value of a: $a") // fancy string interpolations
  a = a + 1
} while(a < 20)

// while loop execution
while(a < 20) {
  println(s"Value of a: $a")
  a = a + 1
}
```

Loops (2/3)

```
var a = 0
var b = 0

for (a <- 1 to 3; b <- 1 until 3) {
  println(s"Value of a: $a, b: $b")
}

/* Output
Value of a: 1, b: 1
Value of a: 1, b: 2
Value of a: 2, b: 1
Value of a: 2, b: 2
Value of a: 3, b: 1
Value of a: 3, b: 2
*/
```

Loops (3/3)

```
// loop with collections
val numList = List(1, 2, 3, 4, 5, 6)
for (a <- numList) {
  println(s"Value of a: $a")
}

// for loop with multiple filters
for (a <- numList if a != 3; if a < 5) {
  println(s"Value of a: $a")
}

// for loop with a yield
// store return values from a for loop in a variable
var retVal = for(a <- numList if a != 3; if a < 6) yield a
println(retVal)
```

Exception Handling

```scala
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException

object Test {
  def main(args: Array[String]) {
    try {
      val f = new FileReader("input.txt")
    } catch {
      case ex: FileNotFoundException => println("Missing file exception")
      case ex: IOException           => println("IO Exception")
    } finally {
      println("Exiting finally...")
    }
  }
}
```

Exception Handling

You can also use the Try ADT to do functional exception handling à la Haskell.

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException
import scala.util.{Try, Success, Failure}

object Test {
  def main(args: Array[String]) {
    val result = Try {
      val f = new FileReader("input.txt")
    };
    result match {
      case Success(_)                         => () // yay
      case Failure(ex: FileNotFoundException) => println("Missing file exception")
      case Failure(ex: IOException)           => println("IO Exception")
    }
    println("Exiting finally...")
  }
}
```

Outline

- Scala basics
- Functions
- Collections
- Classes and objects
- SBT

Functions - Definition

```scala
// def [function name]([list of parameters]): [return type] = [expr]
// the expression may be a {}-block

def addInt(a: Int, b: Int): Int = a + b

println("Returned Value: " + addInt(5, 7))
// Returned Value: 12
```

Functions - Definition

```
// def [function name]([list of parameters]): [return type] = [expr]
// the expression may be a {}-block

def addInt(a: Int, b: Int): Int = a + b

println("Returned Value: " + addInt(5, 7))
// Returned Value: 12
```

You can also specify default values for all or some parameters.

```
def addInt(a: Int = 5, b: Int = 7): Int = a + b

// and then invoke with named parameters
println("Returned Value:" + addInt(a = 10))
// Returned Value: 17
```

Functions - Variable Arguments

```
def printStrings(args: String*) = {
  var i : Int = 0;
  for (arg <- args) {
    println(s"Arg value[$i] = $arg")
    i += 1;
  }
}

printStrings("SICS", "Scala", "BigData")
```

Functions - Nested Functions

```
def factorial(i: Int): Int = {
  def fact(i: Int, accumulator: Int): Int = {
    if (i <= 1)
      accumulator
    else
      fact(i - 1, i * accumulator)
  }

  fact(i, 1)
}

println(factorial(5))
```

Functions - Anonymous Functions

- Lightweight syntax for defining anonymous functions.

```
val inc = (x: Int) => x + 1
val x = inc(7) - 1

val mul = (x: Int, y: Int) => x * y
println(mul(3, 4))

val userDir = () => { System.getProperty("user.dir") }
println(userDir())
```

Functions - Higher-Order Functions

```
def apply(f: Int => String, v: Int): String = f(v)

def layout[A](x: A): String = s"[$x]"

println(apply(layout, 10))
// [10]
```

Functions - Call-by-Value

- Call-by-Value: the value of the parameter is determined before it is passed to the function.

```
def time() = {
  println("Getting time in nano seconds")
  System.nanoTime
}
def delayed(t: Long) {
  println("In delayed method")
  println(s"Param: $t")
}
delayed(time())
/* Output
Getting time in nano seconds
In delayed method
Param: 2532847321861830
*/
```

Functions - Call-by-Name

- Call-by-Name: the value of the parameter is not determined until it is called within the function.

```
def time() = {
  println("Getting time in nano seconds")
  System.nanoTime
}
def delayed2(t: => Long) {
  println("In delayed method")
  println(s"Param: $t")
}
delayed2(time())
/* Output
In delayed method
Getting time in nano seconds
Param: 2532875587194574
*/
```

Functions - Partial Application

- If you do not pass in arguments for all of the parameters.

```
def adder(m: Int, n: Int, p: Int) = m + n + p
// adder: (m: Int, n: Int, p: Int)Int
val add2 = adder(2, _: Int, _: Int)
// add2: (Int, Int) => Int
add2(3, 5)
// 10
```

Functions - Currying (1/2)

- Transforms a function with multiple arguments into a chain of functions, each accepting a single argument and returning another function.
- For example transforms `f(x, y, z)` (Int,Int,Int) $\Rightarrow$ Int to `g(x)(y)(z)` (Int) $\Rightarrow$ ((Int) $\Rightarrow$ ((Int) $\Rightarrow$ Int)), in which `g(x)` returns another function, `h(y)` that takes an argument and returns `k(z)`.
- Used to partially apply a function to some value while leaving other values undecided,

Functions - Currying (2/2)

```
def adder(m: Int)(n: Int)(p: Int) = m + n + p
// adder: (m: Int)(n: Int)(p: Int)Int

// The above definition does not return a curried function yet. To obtain a curried version
// we still need to transform the method into a function value.

val currAdder = adder _
// currAdder: Int => Int => Int => Int

// Alternatively with a "normal" method
def normalAdder(m: Int, n: Int, p: Int) = m + n + p
// normalAdder: (m: Int, n: Int, p: Int)Int
val currAdder = (normalAdder _).curried
// currAdder: Int => (Int => (Int => Int))

val add2 = currAdder(2)
val add5 = add2(3)
add5(5)
// 10
```

Outline

- Scala basics
- Functions
- Collections
- Classes and objects
- SBT

Collections

- Scala's standard library provides both mutable and immutable collections.
- Mutable collections can be updated or extended in place.
- Immutable collections never change: additions, removals, or updates operators return a new collection and leave the old collection unchanged.

Collections

- Arrays
- Lists
- Sets
- Maps

Collections - Arrays

- A fixed-size sequential collection of elements of the same type
- Mutable

```
// Array definition
val t: Array[String] = new Array[String](3)
val t = new Array[String](3)


// Assign values or get access to individual elements
t(0) = "zero"; t(1) = "one"; t(2) = "two"


// There is one more way of defining an array
val t = Array("zero", "one", "two")
```

Collections - Lists

- A sequential collection of elements of the same type
- Immutable (there are also a few mutable implementations)
- Lists represent a linked list

```scala
// List definition
val l1 = List(1, 2, 3)
val l1 = 1 :: 2 :: 3 :: Nil

// Adding an element to the head of a list
val l2 = 0 :: l1

// Adding an element to the tail of a list
val l3 = l1 :+ 4

// Concatenating lists
val t3 = List(4, 5)
val t4 = l1 ::: t3
```

Collections - Sets

- A collection of elements of the same type
- Immutable and mutable
- No duplicates and no order.

```
// Set definition
val s = Set(1, 2, 3)

// Add a new element to the set
val s2 = s + 0

// Remove an element from the set
val s3 = s2 - 2

// Test the membership
s.contains(2)
```

Collections - Maps

- A collection of key/value pairs
- Immutable and mutable

```scala
// Map definition
var m1 = Map.empty[Int, String]
val m2 = Map(1 -> "Carbon", 2 -> "Hydrogen")

// Finding the element associated to a key in a map
m2(1)

// Adding an association in a map
m2 += (3 -> "Oxygen")

// Returns an iterable containing each key (or values) in the map
m2.keys
m2.values
```

Common Data Types

- Tuples
- Option
- Either

Common Data Types - Tuples (1/2)

- Tuples are an implementation of Product Types
- A fixed number of items of different types together
- Immutable

```
// Tuple definition
val t2 = (1 -> "hello") // special pair constructor (an implicit conversion, really)
val t3 = (1, "hello", 20)
val t3 = Tuple3(1, "hello", 20)

// Tuple getters
t._1 // 1
t._2 // hello
t._3 // 20
```

Common Data Types - Tuples (2/2)

- Tuples can also be used as function arguments

```
val fun: (Int, String) => String = (a, b) => s"$a + $b"
// fun: (Int, String) => String
val funTup = fun.tupled
// funTup: ((Int, String)) => String
funTup  (1 -> "hello")
// 1 + hello
```

Common Data Types - Option (1/2)

- Sometimes you might or might not have a value.
- Java typically returns the value null to indicate nothing found.
  - You may get a NullPointerException, if you don't check it.
- Scala has a null value in order to communicate with Java.
  - You should use it only for this purpose.
- Otherwise, you should use Option.

Common Data Types - Option (2/2)

```scala
val numbers = Map(1 -> "one", 2 -> "two")
// numbers: scala.collection.immutable.Map[Int, String] = Map((1, one), (2, two))

numbers.get(2)
// res0: Option[String] = Some(two)

numbers.get(3)
// res1: Option[String] = None

// Check if an Option value is defined (isDefined and isEmpty).
numbers.get(3).isDefined
//  false

// Extract the value of an Option or get a default value.
numbers.get(3).getOrElse("zero")
// zero
```

Common Data Types - Either

- Sometimes you might definitely have a value, but it can be one of two different types
- Scala provides the Either type for these cases

```
def getNum(s: String): Either[Int, String] = try {
  Left(s.toInt)
} catch {
  case _ => Right(s)
}
getNum("5")
// Left(5)
```

Note that, if you are using the Either type to do error handling (like above).

- It is probably better to use the Try type instead, unless your error handling does not involve any exceptions.

Functional Combinators

- map
- foreach
- filter
- flatten
- flatMap
- foldLeft and foldRight

Functional Combinators - map

- Evaluates a function over each element in a collection, returning a collection with the same number of elements

```
scala> val numbers = List(1, 2, 3, 4)
numbers: List[Int] = List(1, 2, 3, 4)


scala> numbers.map((i: Int) => i * 2)
res0: List[Int] = List(2, 4, 6, 8)


scala> def timesTwo(i: Int): Int = i * 2
timesTwo: (i: Int)Int


scala> numbers.map(timesTwo _)
or
scala> numbers.map(timesTwo)
res1: List[Int] = List(2, 4, 6, 8)
```

Functional Combinators - foreach

- It is like map but returns a Unit value
- This is usually a nicer substitute for for-loops over collections

```
scala> val numbers = List(1, 2, 3, 4)
numbers: List[Int] = List(1, 2, 3, 4)


scala> val doubled = numbers.foreach((i: Int) => i * 2)
doubled: Unit = ()


scala> numbers.foreach(print)
1234
```

Functional Combinators - filter

- Removes any elements where the function you pass in evaluates to false

```
scala> val numbers = List(1, 2, 3, 4)
numbers: List[Int] = List(1, 2, 3, 4)


scala> numbers.filter((i: Int) => i % 2 == 0)
res0: List[Int] = List(2, 4)


scala> def isEven(i: Int): Boolean = i % 2 == 0
isEven: (i: Int)Boolean


scala> numbers.filter(isEven)
res2: List[Int] = List(2, 4)
```

Functional Combinators - foldLeft

- It goes through the whole collection and passes each value to `f`.
- For the first list item, that first parameter, `z`, is used as the first parameter to `f`.
- For the second list item, the result of the first call to `f` is used as the `B` type parameter.

```
// def foldLeft[B](z: B)(f: (B, A) => B): B
val numbers = List(1, 2, 3, 4, 5)
numbers.foldLeft(0) { (i, acc) =>
  println("i: " + i + " acc: " + acc);
  i + acc
}
/* Output
i: 0 acc: 1
i: 1 acc: 2
i: 3 acc: 3
i: 6 acc: 4
i: 10 acc: 5
15 */
```

Functional Combinators - foldRight

- It is the same as foldLeft except it runs in the opposite direction
- For collections without clear ordering (like HashMaps and -Sets) there is no real difference

```
// def foldRight[B](z: B)(f: (A, B) => B): B
val numbers = List(1, 2, 3, 4, 5)
numbers.foldRight(0) { (i, acc) =>
  println("i: " + i + " acc: " + acc);
  i + acc
}
/* Output
i: 5 acc: 0
i: 4 acc: 5
i: 3 acc: 9
i: 2 acc: 12
i: 1 acc: 14
15 */
```

Functional Combinators - flatten

- It collapses one level of nested structure
- Also works across container types (mostly)

```
scala> List(List(1, 2), List(3, 4)).flatten
res0: List[Int] = List(1, 2, 3, 4)

scala> List(Some(1), None, Some(3)).flatten
res0: List[Int] = List(1, 3)
```

Functional Combinators - flatMap (1/2)

- It takes a function that works on the collection's element type and produces a container type
- and finally flattens the result again to the element type of the result container
- For outer type O[T] and inner type I[U] the signature is something like `flatMap[U](f: T => I[U]): O[U]` (note that for general Monads O=I)

```
scala> val numbers = List(1, 2, 3, 4, 5)
scala> numbers.flatMap(i => if (i % 2 == 0) Some(i) else None)
res11: List[Int] = List(2, 4)
// Think of it as short-hand for mapping and then flattening:
scala> numbers.map(i => if (i % 2 == 0) Some(i) else None).flatten
res1: List[Int] = List(2, 4)
// Or conversely of map as flatMap with a container constructor
// and flatten as flatMap with the identity function
numbers.flatMap(i => Some(if (i % 2 == 0) Some(i) else None)).flatMap(x => x)
res1: List[Int] = List(2, 4)
```

Functional Combinators - flatMap (2/2)

- flatMap is also the underlying function that enables Scala's for-comprehensions
- This is essentially an imperative looking way of writing purely functional code

```
scala> for {
         a <- Try("5".toInt);
         b <- Try("-10".toInt)
       } yield a + b
res26: scala.util.Try[Int] = Success(-5)

// Equivalent desugared
scala> Try("5".toInt).flatMap(a =>
         Try("-10".toInt).flatMap(b =>
           Try(a + b)
         )
       )
res26: scala.util.Try[Int] = Success(-5)
```

Outline

- Scala basics
- Functions
- Collections
- Classes and objects
- SBT

Everything is an Object

- Scala is a pure object-oriented language.

- Everything is an object, including numbers.

```
1 + 2 * 3 / x
(1).+(((2).*(3))./(x))
```

- Functions are also objects, so it is possible to pass functions as arguments, to store them in variables, and to return them from other functions.

Classes and Objects

```scala
// constructor parameters can be declared as fields and can have default values
class Calculator(val brand = "HP") {
  // an instance method
  def add(m: Int, n: Int): Int = m + n
}

val calc = new Calculator
calc.add(1, 2)
println(calc.brand)
// HP
```

Inheritance and Overloading Methods

- Scala allows the inheritance from just one class only.

```scala
// avoid shadowing fields with subclass constructor parameters
class SciCalculator(_brand: String) extends Calculator(_brand) {
  def log(m: Double, base: Double) = math.log(m) / math.log(base)
}

class MoreSciCalculator(_brand: String) extends SciCalculator(_brand) {
  def log(m: Int): Double = log(m, math.exp(1))
}
```

Singleton Objects

- A singleton is a class that can have only one instance.

```
class Point(val x: Int, val y: Int) {
  def printPoint {
    println (s"Point x location: $x");
    println (s"Point y location: $y");
  }
}

object SpecialPoint extends Point(10, 20)

SpecialPoint.printPoint
/* Output
Point x location: 10
Point y location: 20
*/
```

Companion Objects

- Scala has no static keyword like Java
- If you define an object with the same name as a class it's called a companion object
- Putting methods or fields into the companion object is equivalent to Java's static methods and fields

```
class Point(val x: Int, val y: Int) {
  Point.instanceCount += 1;
}

object Point {
 var instanceCount = 0;
}

val p1 = new Point(10, 20)
val p2 = new Point(20, 40)
println(Point.instanceCount)
// 2
```

Abstract Classes

```scala
abstract class Shape {
  // subclass should define this
  def getArea(): Int
}

class Circle(r: Int) extends Shape {
  // use the override annotation to make the compiler check that
  // you didn't misspell it and it actually overrides something
  override def getArea(): Int = { r * r * 3 }
}

val s = new Shape // error: class Shape is abstract
val c = new Circle(2)
c.getArea
// 12
```

Traits

- A class can mix in any number of traits.

```
trait Car {
  val brand: String
}

trait Shiny {
  val shineRefraction: Int
}

class BMW extends Car with Shiny {
  val brand = "BMW"
  val shineRefraction = 12
}
```

Generic Types (1/2)

- Generics are a bit more powerful (and stricter) than in Java

```
// a generic trait
trait Cache[K, V] {
  def get(key: K): V
  def put(key: K, value: V)
  def delete(key: K)
}

// a generic function
def remove[K](key: K)
```

Generic Types (2/2)

- Generics are a bit more powerful (and stricter) than in Java
- Scala allows variance annotations (invariant, covariant, contravariant)

```scala
trait A {
  def callMe(): String
}
class B(s: String) extends A {
  def callMe(): String = s"Called $s!"
}

// trait Option[+T]
def callMeMaybe(nOpt: Option[A]): Option[String] = nOpt.map(_.callMe())

// the +T generic covariant tells Scala that Option[B] <: Option[A] since B <: A
callMeMaybe(Some(new B("the B")))
// res2: Option[String] = Some(Called the B!)

// The contravariant Option[-T] would imply that Option[A] <: Option[B] if B <: A
```

Case Classes and Pattern Matching

- Case classes and Case objects are meant as data types
- They are designed to be used with pattern matching.
- You can construct case classes without using new and they automatically generate copy constructors and well as hash, equals, and toString methods.

```
scala> case class Calculator(brand: String, model: String)
scala> val hp20b = Calculator("hp", "20B")

def calcType(calc: Calculator) = calc match {
  case Calculator(s, "20B") => s"financial from $s"
  case Calculator(s, "48G") => s"scientific from $s"
  case Calculator(s, "30B") => s"business from $s"
  case Calculator(_, _) => "Calculator of unknown type"
}

scala> calcType(hp20b)
```

Outline

- Scala basics
- Functions
- Collections
- Classes and objects
- SBT

Simple Build Tool (SBT)

- An open source build tool for Scala and Java projects.
- Similar to Java's Maven or Ant.
- It is written in Scala.

SBT - Hello World!

```
$ mkdir hello
$ cd hello
$ cp <path>/HelloWorld.scala .
$ sbt
...
> run
```

Running SBT

- Interactive mode

```
$ sbt
> compile
> run
```

- Batch mode

```
$ sbt clean run
```

- Continuous build and test: automatically recompile or run tests whenever you save a source file.

```
$ sbt
> ~ compile
```

Common Commands

- clean: deletes all generated files (in target).
- compile: compiles the main sources (in src/main/scala).
- test: compiles and runs all tests.
- console: starts the Scala interpreter.
- run <argument>*: run the main class.
- package: creates a jar file containing the files in src/main/resources and the classes compiled from src/main/scala.
- help <command>: displays detailed help for the specified command.
- reload: reloads the build definition (build.sbt, project/*.scala, project/*.sbt files).

Create a Simple Project

- Create `project` directory.
- Create `src/main/scala` directory.
- Create `build.sbt` in the project root.

build.sbt

- A list of Scala expressions, separated by blank lines.
- Located in the project's base directory.

```
$ cat build.sbt
name := "hello"

version := "1.0"

scalaVersion := "2.11.5"
```

Add Dependencies

- Add in `build.sbt`.
- Module ID format:
  `"groupID" %% "artifact" % "version" % "configuration"`

```
libraryDependencies += "org.apache.spark" %% "spark-core" % "2.2.1"

// multiple dependencies
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % "2.2.1",
  "org.apache.spark" % "spark-streaming_2.11" % "2.2.1",
  "org.apache.spark" % "spark-streaming-kafka-0-8_2.11" % "2.2.1"
)
```

Summary

Summary

- Scala basics
- Functions
- Collections
- Classes and objects
- SBT

References

- M. Odersky, Scala by example, 2011.

Questions?