
APACHE
Spark™
Session 1: Scala Primer
• Introduction
• Collections
• Functions/Methods
• Class/Object/Trait

Session Objectives

- Learn what Scala is, and why it's useful
- Understand Scala basics and syntax
- Be able to write and understand Scala code
- Be comfortable with functional programming in Scala
  - One of the core features used with Spark
- This lesson assumes you know basic programming and OO principles
  - It does not cover any of these basics — just Scala basics


APACHE
Spark™
Introduction
Collections
Functions/Methods
Class/Object/Trait

What is Scala?

- An **object-oriented** programming language:
  - Pure OO language — everything is an object
  - Including numbers and functions
- A **functional** language
  - Functions are full class objects
  - They can be passed as arguments
- Interoperates with **Java**
  - Runs on Java Virtual Machine (JVM)
  - Scala programs can use Java libraries
- Very concise / dense language (Java is quite verbose)
  - Java : `System.out.println ("hello world");`
  - Scala : `println ("hello world")`

Session 1: Scala Primer

The name Scala comes from "Scalable Language".

A language is scalable if it can be used for very small as well as very large systems.

Why Scala and Spark?

- Spark designers had two core goals
- Work within **the Hadoop ecosystem**
  - JVM-based, so Spark had to run on the JVM
- Wanted **concise** programming interface
  - With strong support for **functional programming**
- **Scala** was the only language that provided this
  - Including ability to capture functions and ship them across the network (1)
  - Java 8 adds better support for functional programming — the Spark API supports this (2)

See the comments by Matei Zaharia in the thread below:
- http://apache-spark-user-list.1001560.n3.nabble.com/Why-Scala-td6536.html

(1) We'll see when working with Spark, that functional programming is core to the API.
- Additionally, in the Spark architecture, functions are routinely shipped to nodes to perform parallel calculations on data.
- Scala's ability to do this easily was important to the Spark implementation.

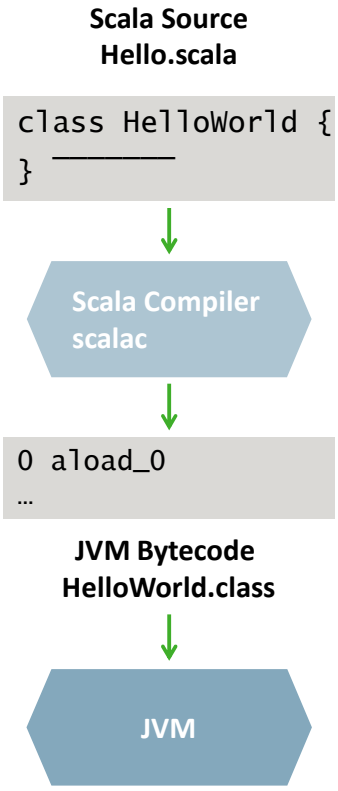
Some people feel that Scala has enough issues to make it problematic as a programming language.
- See: http://overwatering.org/blog/2013/12/scala-1-star-would-not-program-again/

On the other hand, some people, like Twitter, love it.
- But do think it needs some guidelines: http://twitter.github.io/effectivescala

(2) See: http://databricks.com/blog/2014/04/14/Spark-with-Java-8.html

Working with Scala

- Scala code/programs written/run similarly to Java programs
  - Compiled to bytecode, and run in a **JVM**
    - Java Virtual Machine
- Scala also has an **interpreter**
  - For learning and interactive coding
  - Also called the REPL (1)
  - Good for ad-hoc querying in Spark
  - Used a lot in the course
- We'll show isolated code samples to start
  - And run things in the REPL (2)
  - We'll write standalone programs later



Session 1: Scala Primer

(1) The Scala interpreter is actually just a thin layer over the compiler.
- The interpreter is a program that runs a **R**ead-**E**val-**P**rint **L**oop when you type at it.
  - Hence the name REPL
- It compiles what you type into bytecode, then evaluates it.
- It is still using the compiler, so is not a true interpreter.

(2) We'll show isolated code samples in our examples.
- We'll also try them out in the interpreter.
- This is just to ramp up on Scala basics.
- We'll work with complete programs as needed later on.

Scala Variables

- `var`: Defines a variable
- `val`: Defines a constant (immutable/read-only)
- Type is optional — Scala infers it if not supplied
  - Compile time checking—generally works well [1]

```
var x = 5          // Variable (type inferred)
x = 7;             // Change the value (With optional semicolon)

val y = 10         // Constant
y=11               // error: reassignment to val

var d: Double = 5 // Explicit type declaration of Double

// Error-type mismatch. 1.1 is a double, not assignable to Int
val n : Int = 1.1

// Chained assignment doesn't work in Scala
var z = 1
z = x = 2  // This will give an error
```


[1] Scala supports type inference:
Scala has a built-in type inference mechanism which allows the programmer to omit certain type annotations. It is, for instance, often not necessary in Scala to specify the type of a variable, since the compiler can deduce the type from the initialization expression of the variable. Also return types of methods can often be omitted since they corresponds to the type of the body, which gets inferred by the compiler.
[Scala Tutorial: http://docs.scala-lang.org/tutorials/tour/local-type-inference.html]

The Scala Interpreter

- Start at command line via `scala` or `bin/spark-shell` for Spark [1]
  - Prints welcome text then a prompt
  - Type **`:help`** to get help, **`:history [n]`** to list recent history
  - Use arrow keys to scroll through command history
  - Emacs search supported (e.g. Ctrl-R to reverse search history)
  - **Tab completion** is well supported (try it!)

```
> :help
All commands can be abbreviated, e.g. :he instead of :help.
Those marked with a * have more detailed help, e.g. :help imports

:cp <path>               add a jar or directory to the classpath
:help [command]          print this summary or command-specific help
… Remaining detail omitted

// We will use > as the shell prompt to save space (not scala>)
> :history 2 // Get last two commands
339  :help
340  :history 2 // Get last two commands
```


[1] Spark ships with the standard Scala interpreter which also provides access to the Spark libraries, and a pre-instantiated Spark context to write Spark code with.

- This is packaged up as the spark-shell program.
- The Spark-specific capabilities are not relevant to this Scala ramp-up.
- If using Spark, you can use the shell's standard Scala capabilities to work with Scala — it's a standard Scala compiler/interpreter, so will work fine for our needs here.

Using the Interpreter

- Anything typed at the `scala>` prompt is read and evaluated
  - Response is the result of the evaluation given in form:
    `name : Type = Value`
  - It then waits for more input (1)

```
> val y = 10
y: Int = 10

> y = 5
<console>:12: error: reassignment to val

> var z: Double = 5
z: Double = 5.0

> 1+2
res0: Int = 3

> var n:Int = 1.1
<console>:10: error: type mismatch;
```


(1) That's why the Scala interpreter is called the REPL.
- **Read | Evaluate | Print | Loop**
- It reads your input, evaluates it, prints the result, then loops to read input again.

Numeric / Boolean Types

- Scala Numeric Types: `Byte, Char, Short, Int, Long, Float, Double, Boolean`
  - Numeric values are objects — can call methods on them
- Helper classes add functionality (`StringOps, RichInt …`)
  - This functionality can be used as if added to the class itself (1)
  - `StringOps` adds over 100 operations on strings
    - e.g. `"Hello".intersect("loop")` // Yields lo
    - Can also be written `"Hello" intersect "loop"`
  - `scala.Predef` provides easily accessed common definitions
  - e.g. `println` is alias to `Scala.console.println`
- Some special types
  - `Any`/AnyVal: Root of all types / Root of numerical types
  - `Unit`: Equivalent to Java void — no usable value

`BigInt` and `BigDecimal` support arbitrary precision decimals.
- They provide familiar operators for ease of use (e.g. +).

(1) The additional operations provided by classes like `StringOps` can generally be used as if they were defined on the underlying class (e.g. `String`).
- This is due to some automatic Scala conversion magic, which is beyond the scope of this overview.
- The bottom line is that they're very easy to use — you can generally act as if they are defined on the underlying class itself.

Operator Overloading

- Arithmetic operators look pretty normal, e.g. adding two `Ints: a, b`
  `a + b`
  - But this actually calls the method `a.+(b)` (1)
  - This is a special case of calling `a.meth(b)` using a `meth b` (2)
- Overloading's available for other types, and used in the Scala libs
  - e.g. for `BigInt` shown below (3)
- Easy to add nifty new methods/operators, e.g.
  - `1 to 10 // Really 1.to(10)—numbers ranging from 1 to 10`

```
> val bi : BigInt = 1
bi: BigInt = 1

> bi + 2  // Same as bi.+(2)
res14: scala.math.BigInt = 3

> 1 to 10  // We'll work with collections and ranges soon
res15: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5, 6,
7, 8, 9, 10)
```


(1) Since everything is an object in Scala, numbers actually have methods defined on them.
- e.g., the method named "+"
- To make life easy though, Scala allows you to call a method via the syntax below:
  `variable methName argument`
- This allows you to say `5 + 3` instead of `(5).+(3)`.

(2) Scala allows you to call any method with one explicit parameter using a shorthand notation.
- e.g. given a variable "`a`" of a type that supports a method "`meth1`" that takes a single parameter `b`, you would normally call it as
  `a.meth1(b)`
- However, with the shorthand notation, you can call it in the same way that operators are used:
  `a meth1 b`
- This gives a LOT of power to create new types with their own operators.
  - Very useful — but take care not to make an unintelligible mess — have pity on your future coders and maintainers !

(3) `scala.math.BigInt` is part of the standard Scala library, and is designed to hold

large integer values.

Looping

- Scala has fairly standard `for` and `while` loops (1)
  - We illustrate the syntax below (using the values in `1 to 2`)
  - We use the built in `println()` function for output

```
> val b = 1 to 2
b: scala.collection.immutable.Range.Inclusive = Range(1, 2)

// We'll see other ways to iterate over a collection later
> for (i <- 0 to b.length-1) println(b(i))
1
2

> for (cur <- b) println(cur)  // Another way to iterate
1
2

> var i = b.length-1   // Result of assignment not shown here
> while (i >= 0) {
        println(b(i))
        i -= 1 }
2
1
```

Session 1: Scala Primer

`for` loops have a lot of additional capability, including:

Multiple generators of the form variable <- expression

```
for (i <- 0 to 2; j <- 1 to 3) { println(i,j) }
```

Conditionals

- Fairly standard `if/else` statements
  - Supports same syntax as Java/C, as shown below
- An if/else has a value
  - The value of the if or else (depending on the condition result)
  - Can use the value in an assignment as shown at bottom (1)

```
> val i = 1
> val j = 2   // Results not shown

> if (i>j) {
     | println ("i greater")
     | } else {
     | println("j greater")
     | }
j greater

> val k = if (i>j) i else j   // Use value of if / else
k: Int = 2
```


(1) The value of an `if/else` is the value of the expression that is evaluated during the `if/else`.

– In the slide example, since j > i, the condition is false, and the if/else evaluates to j (which has the value of 2).

Blocks also have a value — the value of the last expression evaluated.

match Expression

- `match` lets you select from a number of alternatives of form
  - `case matchValue => expression-to-eval`
  - The match can return a value (second example below)

```
> val monthNum = 2

> monthNum match {
     |   case 1 => println("January")
     |   case 2 => println("February")
     |   case 3 => println("March")
     |   case _ => println("Bad month") // _ = Default case
     | }
February

> val monthName = monthNum match {
     |    case 1 => "January"
     |    case 2 => "February"
     |    case 3 => "March"
     |    case _ => "Bad month"
     | }
monthName: String = February
```

Lab 1.1: Start Scala Interpreter

There are several mini-labs in this session that require the Scala interpreter. Start the interpreter as per the separate instructions for your environment, then come back to the mini-lab

LAB

Play with Scala

Mini-Lab — 5 minutes

- Start up the Scala interpreter as in the previous lab
  - Type `:help` to get help
  - Review the `:paste` command (useful for pasting in code)
- See Notes for some samples of what we'll try here
- Declare some variables (including immutable ones)
  - Let Scala infer the type on some
  - Specify the type on some
- Use some looping constructs and conditionals
- Look (briefly) at the API docs (http://www.scala-lang.org/api)
  - To filter, type the name of a type in the search box in the upper left
  - Try looking at `Predef, StringOps, RichInt`

Session 1: Scala Primer

Try these at the scala prompt:

```
val x = 10
val y : Double = 1.3
y = 3.1 // error
var greeting = "hello world"
greeting = "good bye world"  // ok

val r = 1 to 10
r(0)
```


APACHE
Spark™
Introduction
Collections
Functions/Methods
Class/Object/Trait

Collection Classes

- Rich set of collection classes (Containers for things)
  - **Seq**: Sequences — ordered collection (**List, Vector**)
  - **Set**: Unordered collection with no duplicates
  - **Map**: Collection of key/value pairs
  - Won't explore details here — fairly standard
- Collections are parameterized by a type using [type]
  - And can hold instances of that type
  - The type is often inferred by contents, and not explicitly set (1)

```
// Create List and explicitly specify type
> val colors = List[String]("red", "green", "blue")
colors: List[String] = List(red, green, blue)

// Create List, let type be inferred
> val colors = List("red", "green", "blue")
colors: List[String] = List(red, green, blue)
```

Session 1: Scala Primer

(1) Scala supports type inference in many situations.
- Collections are just one example of that.
- Generally, it works very well, and is more concise than having to specify the type all the time.
- If you need to specify the type explicitly, then you always have that option.

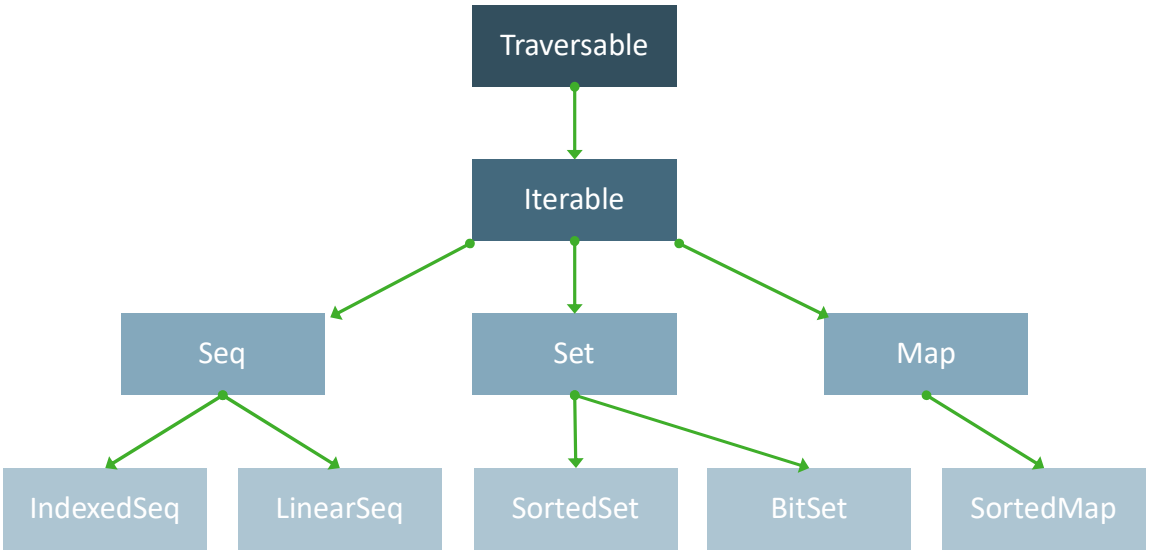
Scala Sequence types include – `Array`, `ArrayBuffer`, `List`, `Vector`, `Seq`, etc.

Map Example:

```
> val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF")
colors: scala.collection.immutable.Map[String,String] = Map(red ->
#FF0000, azure -> #F0FFFF)
```

Core Collection Types

- In package `scala.collection`
  - Traversable defines many common boilerplate methods



Session 1: Scala Primer

`Traversable` and `Iterable` are traits.

Traits are equivalent to Interfaces in Java.
They specify the object type.

Creating Collection Instances

- Create an instance using the collection name
  - Followed by a list of elements in parentheses
  - Note that there are mutable (`scala.collection.mutable`) and immutable (`scala.collection.immutable`) versions
  - Scala defaults to immutable versions

```
> val pets = Set("dog", "dog", "cat")  // Duplicates filtered out
pets: scala.collection.immutable.Set[String] = Set(dog, cat)

// Create map of pets with their count
> val petCount = Map("dog"->2, "cat"->1, "mice"->0)
petCount: scala.collection.immutable.Map[String,Int] = Map(dog -> 2,
cat -> 1, mice -> 0)

// Create mutable set
> val petsNow = scala.collection.mutable.Set("dog", "dog", "cat")
petsNow: scala.collection.mutable.Set[String] = Set(cat, dog)
```


`Mutable`: Can be changed/updated

`Immutable`: Cannot be changed or updated

For example, if you write `Set` without any prefix, Scala will select the immutable version.

To get a mutable version, you need to write **`collection.mutable.`**`Set`.

Accessing Collections

- Capabilities vary based on the type of collection
  - All collections support **iteration** (shown later)
  - **Sequences** (`scala.collection.Seq`) support indexed access
  - **Maps** provide access by key
  - There are many different access options for each type

```
> val pets = Set("mouse", "dog", "cat")
pets: scala.collection.immutable.Set[String] = Set(mouse, dog, cat)
> pets.tail
res7: scala.collection.immutable.Set[String] = Set(dog, cat)

> val a = 1 to 5
a: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)
> a(0)
res10: Int = 1

> val petCount = Map("dog"->2, "cat"->1, "mice"->0)
petCount: scala.collection.immutable.Map[String,Int] = Map(dog -> 2,
cat -> 1, mice -> 0)
> petCount("dog")
res11: Int = 2
```


`.tail` returns all but the 1st element in a set.
`.head` returns the 1s element in a set.

Collection Methods

- Many methods — we list several that will be of use to us
  - **Map operations** (`map, flatMap`): Apply a function to each element to produce a new collection
    - `flatMap` is a combination of map and flatten
  - **Sub-collection retrieval** (`take, filter, slice`): Return sub-collection identified by index range or predicate
  - **Folds/reductions** (`fold, reduce`): Apply a binary operation to successive elements
  - Many, many more — see the documentation
- Many are common when using Spark
  - We'll give some examples shortly
  - Once we know a bit more about functions

`flatten` collapses one level of nested collections, e.g.

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

Pairs and Other Tuples

- Tuples combine a fixed number of items together
  - e.g. a pair combines two items
  - Items can be of different types
  - Access them via notation `._n` (n is the 1-based index) (1 <= n <= 22)

```
> val pair = ("apple", 4)
pair: (String, Int) = (apple,4)

> pair._1
res149: String = apple

> println(pair._2)
4

> val wordCounts = Array(("apple",3), ("pear",2), ("grape",1))
wordCounts: Array[(String, Int)] = Array((apple,3), (pear,2), (grape,1))

> wordCounts(0)._1
res152: String = apple

> val edge = (1L,2L, 7)      // A triple
edge: (Long, Long, Int) = (1,2,7)
```

Session 1: Scala Primer

Tuple are objects containing a miscellaneous collection of elements.

They are instances of a class with name of the form **`TupleN`**.

N is the number of items in the tuple.

The `getClass` method will return the name of the tuple.

You can have anywhere from 2 to 22 items in a tuple.

What's the difference between a `List` and `Tuple`?

Tuples can contain any type while a list is of a specific single type.


APACHE
Spark™
Introduction
Collections
Functions/Methods
Class/Object/Trait

Function Definition

- At bottom, we define the function max
  - General form is

```
def function-name(arg1: Type …) : ReturnType =
                        { function-body }
```

  - The arguments always require a type spec
  - You can leave out the return type and braces (second example) (1)

```
> def max(x: Int, y: Int): Int = {
         if (x > y) x
         else y
       } // Returns the last expression evaluated(2)
max: (x: Int, y: Int)Int

> def max2(x: Int, y: Int) = if (x > y) x else y // Shorter version
max: (x: Int, y: Int)Int

>  max (1,2)    // Just call the function by name
res79: Int = 2
```

Session 1: Scala Primer

(1) You can often leave out the return type of a function.
- As long as the compiler can infer it, which it usually can.
- One place when it can't is for recursive functions, which require a return type.
- You can also leave out the {} braces if the function only has one statement.

(2) The return value of max depends on which expression is evaluated in the if
- The one that is evaluated becomes the return value of the function.

Functions and methods are very similar:
- However, methods are defined as members of a class/object
- Functions are not

Anonymous Functions / Function Literals

- You can define unnamed functions
  - Most often to pass to other functions for processing
  - Below, we use an anonymous function to process a collection
  - It prints out the square of each element in the collection
- General syntax is:
  - `(argumentList) => { functionBody }`
  - Can leave out the `()` if there is only one argument
  - Can leave out the `{}` braces if the body is one statement
  - Can leave out type of argument if it can be inferred (1)

```
> val b = 1 to 2
b: scala.collection.immutable.Range.Inclusive = Range(1, 2)

> b.foreach( (x:Int) => { println(x*x) } )  // Anonymous function
1
4

> b.foreach( x => println(x*x) )       // Shorter form
```

Session 1: Scala Primer

(1) In the example, since b contains Int values, the compiler can infer that the type of x is Int.
You can therefore leave it out, as shown in the second version.

Function Literal Syntax

- There are many choices in function literal syntax (1)
  - We show several choices for declaring literal functions below
  - The functions test if the input argument is even (they return a `Boolean`)

```
// Declare argument type, not return type
> val f = (i: Int) => { i%2==0 }
> val f = (_:Int)%2==0
f: Int => Boolean = <function1>  // All variants have this signature

// Declare the argument and return type
> val f: (Int) => Boolean = i => { i % 2 == 0 }
> val f : Int => Boolean = i => i%2==0
> val f : Int => Boolean = _%2==0
```


(1) As with many things in Scala, there is a lot of capability and flexibility. And also, as with many things, the many choices to do something can lead to obscure and hard-to-follow code.

Using filter and map

- We now know how to pass functions to the collection methods
  - So lets try them out
- `filter`: Select all elements satisfying the supplied predicate
  - Filter out those that don't
- `map`: Create new collection by applying supplied function to each element

```
> val b = 1 to 4
b: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4)

// Filter out all odd values
> b.filter(a => a%2==0)
> b.filter(f) // Same thing using function literal from earlier slide
res2: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4)

// Create a new collection containing the squares of the values
> b.map(a => a*a)
res3: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 4, 9, 16)
```

Session 1: Scala Primer

filter takes the elements of a collection, and applies the passed in function to each element.

If the function returns true, the element is placed into the result collection.

If the function returns false, the element is not kept.

The function for the filter example is **`a => a%2==0.`**

filter will iterate through the collection, and for each element "a" in the list will calculate `a%2`

`a%2` is the modulo operator — it computes the remainder after dividing by the second operand.

It then checks to see if `a%2` is zero (i.e. it checks to see if there is no remainder).

If this is true, then the element is kept (it is an even element).

If it is not true, the element is discarded.

The function for the map example is `a => a*a.`

`map` will iterate through the collection, and for each element "a" in it will calculate `a*a.`

It returns a collection containing all the results it calculated (the squares

of the original collection).

Multiple Function Arguments and reduce

- Below, we define an anonymous function with two args
  - We also assign the function to a variable (1)
  - Below that, we use it to reduce an Int range (2)
  - We also show a version inlining the anonymous function

```
> val sum = (a:Int, b:Int) => a+b
sum: (Int, Int) => Int = <function2>

> (1 to 3).reduce(sum)
res21: Int = 6

// Anonymous function – types of arguments inferred
> (1 to 3).reduce( (a,b) => a+b )
res22: Int = 6
```

Session 1: Scala Primer

(1) Because functions are full-class objects, you can assign one to a variable, as shown in the example.

(2) When you call reduce on a collection, it reduces it to a single value by iteratively applying the passed in function on values in the collection.

- reduce will make multiple passes, first with the elements in the collection, next with the results of the first pass, etc. until a single value is left.

_ (Underscore) Placeholders

- _ (underscore) can be used for anonymous function parameters
  - As long as the parameters are used only once
- Below, we show examples without and with placeholders
  - Note how we use _ twice in the anonymous function with two params
  - The first use means the first param, the second use the second param

```
> (1 to 4).filter( a => a%2 == 0)
res30: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4)

> (1 to 4).filter(_%2 == 0)  // Placeholder equivalent – one param
res31: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4)

> (1 to 3).reduce( (a:Int, b:Int) => a+b ) // Anonymous function
res22: Int = 6

> (1 to 3).reduce( _+_ )   // Placeholder equivalent – two params
res113: Int = 6
```

LAB

Play with Collections and Functions

Mini-Lab — 10 minutes

- Do your work in the Scala interpreter as before
- Create some collection instances
  - Create a List instance, create a numeric Range instance using `"to"`
  - Store them in variables
  - Access some of the values (use the docs / tab completion as needed)
- Use some of the functions we've seen
  - Use `filter` to create a subset of one of your collections
  - Use `reduce` on your numeric Range to add up all the values
    - Do this without underscore placeholders and with underscore placeholders
  - **[Optional]** Use `map` on your numeric Range to map to a new collection containing the square of each value

Session 1: Scala Primer

Here are some examples to try:

```
val x = 1 to 10
val  x2 = x.map(a => a*2)   // can you come up with a short hand version
val  xsq = x.map( ???)

val even = x.filter ( a =>   a%2 == 0)  // try a shorter version

// add up all even numbers
even.reduce( (a,b) => a+b)  // try the shorter version

// find even numbers and sum them up in one liner
```


APACHE
Spark™
Introduction
Collections
Functions/Methods
Class/Object/Trait

Defining a Class

- Class is a blueprint for objects (of course)
  - Defining the type in terms of methods and data
- Below, we define a simple class — we'll give more detail soon

```
> class Square (val size : Int) {
     |      def area : Int = size * size
     |      def perimeter : Int = size*4
     |      def display : Unit = {
     |         println("Size = " + size)
     |         println("Area = " + area)
     |     }
     | }
defined class Square

> val sqr = new Square(6)
sqr: Square = $iwC$$iwC$Square@478eb50

> sqr.area
res145: Int = 36

> sqr.size
res146: Int = 6
```

Class Def Details

- The below defines a constructor and a data member — size
  - Accessible throughout the class
  - Also accessible as instance.size since we used val keyword

```
class Square (val size : Int) {
```

- Define accessor methods for area (a calculated value here)

```
def area : Int = size * size
```

  - Called without any parentheses when defined this way

```
sqr.area
```

- We define a method that does some work (trivial work here)

```
def display : Unit = {
  println("Size = " + size)
  println("Area = " + area)
}
```


Secondary constructors can be defined like this:

```
def  this(size: Int, color : String) {
   this(size)
   // set color
}
```

Singleton Objects

- You can define singleton objects with the keyword **object**
  - One and only instance — accessed via the object name
    - e.g. `Counter.currentCount`
  - Basically a container for non-instance related code

```
> object Counter {
     |   var count = 0
     |
     |   def currentCount(): Long = {
     |     count += 1
     |     count
     |   }
     | }
defined module Counter

> Counter.currentCount
res153: Long = 1

> Counter.currentCount
res154: Long = 2
```


`object` constructs replace a lot of the use of statics in Java.

Traits

- Traits are types defining fields and methods that you can mix into your classes
  - It defines a role played by objects
  - It may have some implementation, but doesn't have to
- Below we define a trait with three accessor methods
  - But no shared implementation

```
> trait Geometric {
     |   def height : Int
     |   def area   : Int
     |   def perimeter : Int
     | }
defined trait Geometric
```


Trait vs. Java interface

Traits can have methods with implementations.

Using a Trait

- Use a trait in the class def
  - Via the **extends** or **with** keywords (1)

```
> class Square (val size : Int) extends Geometric {
     |            def area : Int = size * size
     |            def height : Int = size
     |            def perimeter : Int = size*4
     |            def display : Unit = {
     |               println("Size = " + size)
     |               println("Area = " + area)
     |           }
     |       }
defined class Square

> val sqr = new Square(6)
sqr: Square = $iwC$$iwC$Square@4f90cf86

> val geom : Geometric = sqr
geom: Geometric = $iwC$$iwC$Square@4f90cf86

> geom.area
res155: Int = 36
```

Session 1: Scala Primer

(1) Use the **extends** keyword for a trait if a class does not extend any other class (as shown in the slide example).

- If it does extend another class, use **with**

```
class Square (val size : Int) extends SomeClass with Geometric
```

- Similarly use **with** for any traits other than the first one

```
class Square (val size : Int) extends Geometric with Trait2
```

Case Class

- Plain, immutable types that are initialized via constructor only
  - Initialization simpler than regular class
  - Can use pattern matching on the class itself
  - Implicitly defines an equality operator (value based)
  - We illustrate definition and some usage below

```
> case class Person (name: String, gender: String, age: Long)
defined class Person

> val p1 = Person("John", "M", 45)
> val p2 = Person("Mary", "F", 50)
> val p3 = Person("John", "M", 45)

> if (p1==p2) "Equal" else "Non-Equal"
res0: String = Non-Equal

> if (p1==p3) "Equal" else "Non-Equal"
res1: String = Equal
```


For more info on case classes see:

http://docs.scala-lang.org/tutorials/tour/case-classes.html
https://stackoverflow.com/questions/2312881/what-is-the-difference-between-scalas-case-class-and-class

Packages and Imports

- Packages organize code and control namespace
  - Package names are hierarchical and dot-separated
    - e.g. `scala.math`
  - Packages generally hold related types
- Declare a package with a package statement, e.g.

```
package com.mycompany.time
```

- Import type names from other packages with import

```
import com.mycompany.time._  // Import all types
// Import Timepiece only
import com.mycompany.time.Timepiece
```

Packages and Imports Illustrated

- Below we define and use some types in different packages

```
// File Timepiece.scala
package com.mycompany.time;
import scala.math._;
trait TimePiece { /* … */ }
```

```
// File AlarmClock.scala
package com.mycompany.time;
import scala.math._;
class AlarmClock extends TimePiece { /* … */ }
```

```
// File Store.scala
package com.mycompany.products;
import com.mycompany.time.TimePiece

class Store {
  def timepieces : Array[TimePiece] = new Array[TimePiece](100)
}
```

A Standalone Scala Program

- Is an object with a main method, as shown below
  - If the main method has a parameter defined as shown in the example, it will receive the program arguments
- You compile this with the Scala compiler `scalac`

  > `scalac SimpleApp.scala`
- You run it with the scala command

  > `scala SimpleApp`

```
object  SimpleApp {
  def main(args: Array[String]) {  // args holds command line args
    for (cur <- args) println(cur) // Print out the args
  }
}
```

Session Summary

- Scala is a **functional** as well as OO language
- The **Scala Interpreter** is a quick and easy way to use Scala
  - The **REPL** (Read-Evaluate-Print Loop)
  - Helps in learning and testing code
  - Load external file with `:load <file>`
- All Scala data is **typed**
  - As specified in classes with methods and data
  - Declare singletons using the `object` keyword
  - **No** "primitive types" like Java and C
  - Scala can often **infer** a type (type inference)
- Scala is **garbage collected**
  - Automatically reclaims unused objects (never done manually)

Session Summary

- Values (`val`) are immutable
  - More stable than variables (`var`)
- A tuple is an ordered container of values of multiple types
  - Accessed by `_n` syntax (e.g. `myTuple._1`)
  - Can't iterate through tuple members
- Many useful **collection types** in Scala
  - `Set, List, Map`
  - Many useful functions, e.g. `filter, map, reduce`
- The `Unit` type is used for functions returning no data
  - Same as void in Java and C-like languages

Scala Resources

- Book: **Programming in Scala**
  - By Martin Odersky (Scala language Designer), et. al
  - 1st edition online at: http://www.artima.com/pins1ed/
- Scala website: www.scala-lang.org
  - The place for all things Scala, including API docs and tutorials
  - **Scala CheatSheet**: http://docs.scala-lang.org/cheatsheets/

Session 2: Introduction to Spark

- Introduction
- Spark in the Big Data Ecosystem
- First Look at Spark