Learn Scala in 3 Hours

Daniel Hinojosa

Contact Information

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Twitter: http://twitter.com/dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext

Conventions in the slides

The following typographical conventions are used in this material:

Italic
Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
Shows commands or other text that should be typed literally by the user.

`Constant width italic`
Shows text that should be replaced with user-supplied values or by values determined by context.

Shell Conventions

All shells (bash, zsh, Windows Shell) are represented as `%`

```
% calendar
```

All SBT shells are represented as `>`

```
> compile
```

All Scala REPL and SBT Consoles are represented as `scala>`

```
scala>
```

What is Scala?

- Multi-paradigm programming language
  - Functional
    - Every function is an object and a value
    - Capable of anonymous and higher order functions
  - Object Oriented
    - Everything can considered an object, including integers, floats
    - Inheritance through mixins and subclasses

Statically Typed Language

- Every value contains a type
- Expressive type system
- Types can be inferred
  - Cleaner
  - Less Physical Typing

Why bother?

- Contains all the features that Java 8 has now.
- Fully vetted by community
- Large Community
- Financial Backing
- It is good to learn a new language every year – The Pragmatic Programmer

What are the advantages of Scala?

- JVM Based
- Highly Productive Language
- Expressive Language
- Concise Language
  - Type Inference
  - No `return` required
  - No semicolons (`;`) required
  - All methods `public` by default
- Above all, highly functional!

What are the disadvantages of Scala

- Hiring pool can be constrained
- Higher learning curve, until function programming becomes more prominent
- Non-backwards compatibility

Non-backwards compatibility

com.typesafe.akka	akka-actor_2.12	2.5.2 all (14)	24-May-2017
com.typesafe.akka	akka-actor_2.11	2.5.2 all (59)	24-May-2017

From: search.maven.org

REPL

About the REPL (Read-Eval-Print-Loop)

- Scala's REPL (Read-Eval-Print-Loop) allows you to evaluate expressions in Scala
- Invoke the Scala REPL using the `scala` command at your command prompt

```
% scala
```

- Contains tab completion to complete code
- If code has not been assigned a variable, one will be assigned for you.
- `lastException` will be bound to the last exception that occurred

`:help`	Show help commands
`:paste`	Allows you to paste large amounts of code onto the REPL
`:history`	Show history
`:type`	Show the type of an assignment
`:reset`	Erase all bindings
`:quit`	Quit the REPL

Creating a Script

Creating a Script

- A script can be created and run like any scripting language
- No `main` method required
- Some overhead required

In a file called *ScalaScript.scala* anywhere on a system, and outside of an actual project:

```
println("Hello, Scala")
```

You can also invoke the following on the command line:

For MacOSX/Linux:

```
% echo 'println("Hello, Scala")' >> ScalaScript.scala
```

For Windows (No ticks required):

```
% echo println("Hello, Scala") >> ScalaScript.scala
```

Some items you should know:

- `println` prints to the console
- No semicolons are required in Scala

Running a Script

The script *ScalaScript.scala* can be executed with the following command in either your terminal (MacOSX, Linux) or command prompt (Windows)

```
% scala ScalaScript.scala
```

The `CompileServer` and learn to avoid it when needed

- The `CompileServer` is a Scala thread that caches compile code to subsequent runs called the *fsc offline compiler*
- You can `kill` the `CompileServer` process at anytime
- You can also tell `scala` to not use the `CompileServer` with the `-nc` flag

- Windows processes will not show up in `jps` (May change in the future)

Finding the `CompileServer`

- Find the `CompileServer`/`MainGenericRunner` using the following in your MacOSX or Linux terminal:

```
% jps
```

Should return something like the following:

```
34626 Jps
34579 MainGenericRunner
```

- To avoid using the cached bytecode in the `MainGenericRunner` use the `-nc` switch
- This may come in handy if you are doing a lot of scripting and you get unwanted results

```
% scala -nc ScalaScript.scala
```

Creating Scala Applications

About Applications in Scala

- Applications can be compiled just like Java
- The name of the file does not need to match the class in the file
- You can also have multiple `class` (and other constructs) in the same file

Creating an application

- Assuming inside a file, *Entities.scala*, we have two classes
- The property names come first before the type

```
class Employee(firstName:String, lastName:String, department:Department)
class Department(name:String)
```

Compiling the file with `scalac`

- `scalac` compiles the code

```
% scalac Entities.scala
```

Viewing the class files

- Using `javap -p` allows us to view the internals of the bytecode that Java produced.
- `-p` stands for the showing the `private` fields in the class

```
% javap -p Employee
```

```
% javap -p Department
```

You cannot compile a script!

- Adding `println("Hello, Scala")` to *Entities.scala* will keep the code from compiling
- It is neither a `class`, `object`, or `trait`

This file will not compile

```
class Employee(firstName:String, lastName:String, department:Department)
class Department(name:String)
println("Hello, Scala")
```

To compile it needs to be in an `object`

```
class Employee(firstName:String, lastName:String, department:Department)
class Department(name:String)
object MyRunner {
   def main(args:Array[String]) {
        println("Hello, Scala")
   }
}
```

- To run you can invoke `MyRunner` using `scala` at the command line

```
% scala MyRunner
```

About `object`

- `object` is a singleton
- It is how we avoid `static`
- Scala doesn't have the `static` keyword
- All methods are `public` by default
- Therefore the following is the same as `public static void main(String[] args)`

```
object MyRunner {
   def main(args:Array[String]) {
        println("Hello, Scala")
   }
}
```

More on `object` later

Using `App`

- Turns `object` into an executable program
- No need for a `main` method
- `args` can be referenced to object the arguments

```
object MyRunner extends App {
  println("Hello, Scala")
}
```

Adding a package to our code

- Packaging can be just like in Java

```
package com.xyzcorp
class Employee(firstName:String, lastName:String, department:Department)
class Department(name:String)
object MyRunner {
   def main(args:Array[String]) {
        println("Hello, Scala")
   }
}
```

Adding a package with containment

```
package com {
  package xyzcorp {
    class Employee(firstName:String, lastName:String, department
:Department)
    class Department(name:String)
  }
  package abccorp {
    object MyRunner {
      def main(args:Array[String]) {
        println("Hello, Scala")
      }
    }
  }
}
```

`val` and `var`

`val`

- `val` is called a *value*
- It is immutable, therefore unchangeable
- No reassignment
- Since Scala is functional, it is *preferred*

```
val a = 10
```

Reassignment of `val`

A reassignment is out of the question with a `val`

```
val a = 10
a = 19 //error: reassignment to val
```

`var`

- Called a *variable*
- It is mutable, therefore changeable
- Reassignable
- Used sparingly
- Usually kept from being changed from the outside

```
var a = 10
a = 19
println(19)
```

`println` is how you print to console with carriage return. `print` with no carriage return

Bending `val` and `var` to your will!

Scala's Flexibility

- Scala is also flexible with what character to use.
- As long as the unicode character is within the Opchar range: `\u0020-\u007F`

See: http://unicode.org/charts/PDF/U0000.pdf

- Symbols can also be a mathematical operator:

See: http://www.fileformat.info/info/unicode/category/Sm/list.htm

- Symbols can also from the other symbol category:

http://www.fileformat.info/info/unicode/category/So/list.htm

- Lab: Checkout the URLs for a feel of the kinds of characters you can use in Scala

Using any special characters after the underscore

- Given knowledge of the special characters we can use them after the underscore _

```
val isIncluded_? = true // Note: The Question Mark!
```

- We can also use spaces and reserved words within the backticks:

```
val `This value is the value I was telling you about!` = 40
```

- Of course this is not a license to be cruel to your coworkers:

```
val `true` = false
```

Invocation of the above still requires still that you use the backticks when referencing to avoid confusion.

Primitives

- All primitives are that of the JVM (`byte`, `short`, etc)
- JVM primitives are used as objects in Scala.
- At compile time, depending on the use, may compile to a primitive or an object.

`Byte`

- A byte is an 8 bit number, which is the same in Java.
- The maximum for each number in the JVM is defined as 2 ^{n-1} - 1 where n is the number of bits.
- The minimum is defined as -2^{n-1}
- That means that the maximum size for a byte, which is 8 bits, 2 ^{8-1} - 1, or 2^7 - 1 or 127.
- The minimum would be negative 2^{8-1} - 1, or -2^{7} or -128

```
val b:Byte = 127
```

or if we were to employ coersion

```
val b = 127:Byte
```

`Byte` continued

Here we establishing a negative with the minimum size of a byte.

```
val b = -128:Byte
```

If you care to express yourself in hexadecimal the following is the same as 127

```
val b = 0x7F:Byte
```

Any increase would either be an error in form of a type mismatch or cause an overflow depending on the operation.

```
val maxByte:Byte = 128
val minByte:Byte = -127
```

Short

- `Short` being 16 bits, is 2^{16-1}-1 or 32,767 and the minimum is -2^{16-1}, or -32,768

```
val maxShort = 32767:Short
val minShort = -32768:Short
```

Int

- Integers in Scala, and they are the default number type.
- Maximum: 2^{32-1}-1
- Minimum: -2^{32-1}

```
val a = 301202
```

- You can be explicit. But often times this is unnecessary

```
val a:Int = 301202
```

Long

- 64 bit and requires either the type declaration.
- Maximum: 2^{64-1}-1
- Minimum: -2^{64-1}

```
val g:Long = 30010200
```

or with coersion:

```
val g = 30010200:Long
```

- You can use an capital L using Java's style

```
val g = 30010200L
```

- a small l as a suffix.

```
val g = 30010200l
```

A small `l` looks like a `1` and can lead to confusion.

`Float` and `Double`

- `Float` and `Doubles` are IEEE 754 Standard Floating Arithmetic Values.
- `Float` can be explicitly stated as type

```
val f:Float = 19.0
```

Or coerced

```
val f = 19.0:Float
```

- Float with a capital or small F.

```
val f = 19.0f

val f = 19.0F
```

`Float` Exponents

Floats can also be expressed with exponents with a small e or a capital E.

```
val f = 93e-9F

val f = 93E-9F
```

`Double`

`Double` on the other hand are the default when dealing with decimal point

```
val d = 19.0
```

- You can affix a type or a capital `D` or a small `d` at the end to ensure a correct type

```
val d:Double = 19.0
val d = 19.0:Double
val d = 19.0D
val d = 19.0d
```

- Of course, you can also have double exponents

```
val d = 93.0E-9
```

Char

- Characters literals are much like java. Here is the character 'k'

```
val c:Char = 'k'
```

Unicode in characters, can be done with a preceding backslash u.

```
val c2 = '\u03B8' //theta Θ
```

Boolean

- `Boolean` also derive from Java

```
val b = true
val b2 = false
```

Scala treats primitives like objects

- Unlike Java, Scala's primitives may be treated like objects.
- When calling an operation that works perfectly fine as a primitive, Scala will not box or wrap the primitive
- Only when calling a method that requires an object will an object be created.
- You don't have to concern yourself how or when a wrapping occurs since the compiler will do that for you.

Consider the following statement:

```
1 + 4
```

In other words this operation looks just like `+` but is a method on the object `1` with a method parameter of `4`.

```
1.+(4)
```

Yes, there is operator overloading in Scala

Primitive Wrappers

- There are times in Scala there objects are wrapped to add functionality. For example:

```
-5.abs
```

- If you are familiar with Java, you know that when you take the absolute value of something you require the static method call, `Math.abs`.
- In Scala, it is in inherit part of `Int` through a trick called an implicit wrapper.
- In this case there is an adapter called `RichInt` that wraps around a regular `Int` that provides a this method called `abs` among others.
- Every primitive type in Scala, has a corresponding wrapper.
  - `Char` there is `RichChar`
  - `Boolean` there is `RichBoolean`
  - `Long` there is `RichLong`, etc.

Conclusion

- All primitives works much like Java with varying differences.
- Primitive assignments can be inferred by the type system, or you can add types manually as you see fit.
- Primitives may be wrapped by a rich wrapper depending on which method is called
- Scala primitives gives more methods and functionality than it had with Java.

Control Statements

`if`, `else if`, `else` imperative

- `if` statements can be made imperatively but will often require a mutable variable `var`
- Scala programmers often do not use `var` although that is not quite exactly a hard rule.
- This is *imperative style code*

```
val a = 10
var result = "" // var is usually a code smell
if (a < 10) {
  result = "Less than 10"
} else if (a > 10) {
  result = "Greater than 10"
} else {
  result = "It is 10!"
}
```

`if`, `else if`, `else` functional

- What is different about `if` statements in Scala as well as other functions is that they are *assignable*
- In this case we are assigning to `result`, a `val`
- Arguably, cleaner and concise code.

```
val a = 10
val result = if (a < 10) "Less than 10"
             else if (a > 10) "Greater than 10"
             else "It is 10!"
```

There is no "ternary" operator in Scala

`while` loops

- Nearly the same as in Java
- Imperative Style
- Runs the code within the block until there the boolean condition becomes `false`
- Not used as much by Scala programmers
  - Unless you are writing APIs
  - Using a `mutable` collection

```
var a = 10
var result = "" // var is usually a code smell
while (a > 0) {
   result = result + a
   if (a > 1) result = result + ","
   a = a - 1
}
println(result)
```

For a taste of idiomatic Scala, the above can be rewritten as:

```
println((100 to 1 by -1).mkString(",")) //Deliciousness!
```

`do-while` loops

- Nearly the same as in Java
- Imperative Style
- Runs the code within the block until there the boolean condition becomes `false`
- At least runs once
- Not used as much by Scala programmers

```
var a = 10 // var is usually a code smell
var result = ""
do {
   result = result + a
   if (a > 1) result = result + ","
   a = a - 1
} while (a > 0)
```

`for` loops

- You can still perform the classic idea of a `for`-loop
- Often underused in the Scala community
- Replaced in favor of *for comprehensions*

```
var result = "" // var is usually a code smell
for (a <- 1 to 10) { // a for loop
   result = result + a
   if (a > 1) result = result + ","
}
```

Conclusion

- `if` statements exist like other languages except they are assignable to a `val` or `var` (preferably a `val`)
- `while`, `do-while` exists but are rarely used in Scala, because they cause the programmer to resort to variables (`var`)
- `for`-loops are also available, those too are underused, we use `for`-comprehensions in favor of `for`-loops

`String`

About `String`

- `String` is the same object as in the Java
- `StringOps` to provide added functionality

```
val s = "Scala"
```

- Can be declared, but unnecessary due to inference

```
val s:String = "Scala"
```

- Type can be added by coercion

```
val s = "Scala":String
```

`String` format

- `String` can be formatted with C-style/Java format flags

Here is the Java-Style before, which still works in Scala

```
String.format("This is a %s", "test")
```

Here is the Scala style:

```
"This is a %s".format("test")
```

For a reference on the types of flags: http://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html

Changing the order of arguments using `format`

Without specifying order, `format` will use the order provided:

```
println("Because you're %s, %s, %s times a lady".format("Three",
"Twice", "Once"))
```

The above will surely render incorrectly (if you know the song):

```
Because you're Three, Twice, Once times a lady
```

To specify order we can use the format `%n$s` where `n` is the which argument we wish to use and `s` is the type. In this case `String`

```
println("Because you're %3$s, %2$s, %1$s times a lady".format("Three",
"Twice", "Once"))
```

This will render:

```
Because you're Once, Twice, Three times a lady
```

The above can be trimmed to the following using `printf`

```
printf("Because you're %3$s, %2$s, %1$s times a lady", "Three", "Twice",
"Once")
```

Formatting Dates and Times

- Java Time came with Java 8 and compliments well with Scala

```
import java.time._
println("We will be eating lunch on %1$tB the %1$te in the year %1$tY"
.format(LocalDate.now))
```

Depending on today's date you should see something like the following:

```
We will be eating lunch on June the 26 in the year 2017
```

The underscore for the `import` (`_`) is analogous to the asterisk in (`*`) in Java

Smart Strings

- Smart Strings are surrounded 3 x `"`
- They allow multi-lines of code

```
val prose = """I see trees of green,
               red roses too
               I see them bloom,
               for me and you,
               and I think to myself,
               what I wonderful world"""
```

The problem with the above is that it that the margins are misaligned.

```
I see trees of green,
               red roses too
               I see them bloom,
               for me and you,
               and I think to myself,
               what I wonderful world
```

Smart Strings with `stripMargin`

- `stripMargin` can align the strings based on the pipe (`|`) by default

```
val prose = """I see trees of green,
               |red roses too
               |I see them bloom,
               |for me and you,
               |and I think to myself,
               |what I wonderful world""".stripMargin
```

This will render...

```
I see trees of green,
red roses too
I see them bloom,
for me and you,
and I think to myself,
what I wonderful world
```

Smart Strings with customized `stripMargin`

- `stripMargin` can align the strings based on a character of your choice.

```
val prose = """I see trees of green,
              @red roses too
              @I see them bloom,
              @for me and you,
              @and I think to myself,
              @what I wonderful world""".stripMargin('@')
```

This will render the same...

```
I see trees of green,
red roses too
I see them bloom,
for me and you,
and I think to myself,
what I wonderful world
```

Smart Strings with combination `format`

- Since Smart Strings are just `String` you can use all the same methods
- Including `format`

Here we will use format to include the colors

```
val prose = """I see trees of %s,
              |%s roses too
              |I see them bloom,
              |for me and you,
              |and I think to myself,
              |what I wonderful world""".stripMargin
                                       .format("green", "Red")
```

String Interpolation

- You can replace any variable in a string from it's environment or context with *string interpolation*
- The only thing that you require is that the letter `s` precedes the string.
- You can refer to an outside variable by using `$` to precede it, for example `$a`
- If you require an expression wrap the expression in a bracket, for example, `${a + 1}`

String Interpolation

- Given the following:

```
val a = 99 //Setting up a value within the context
println(s"$a luftballoons floating in the summer sky")
```

- The previous would render the following after `$a` is replaced

```
99 luftballoons floating in the summer sky
```

The `f` interpolator

- Used to combine `String.format` functionality with `String` interpolation

```
val ticketsCost = 50
val bandName = "Psychedelic Furs"
println(f"The $bandName%s tickets are probably $$$ticketsCost%1.2f")
```

- `$bandName%s` treats the interpolation as a `String`
- `ticketsCost%1.2f` treats the cost with a *width* of 1 if possible and two decimal points
- `$$` is used to escape the dollar sign

The above renders ...

```
The Psychedelic Furs tickets are probably $50.00
```

Extra decoration for the `f` interpolator

- The formats allowed after the `%` character are all part of the standard `Formatter`
- http://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html

Therefore, we can also try `%n` for a newline and `%%` for a percent.

```
val ticketsCost = 50
val bandName = "Psychedelic Furs"
val percentIncrease = 20
val musicGenre = "New Wave"

println(f"The $bandName%s tickets are probably $$$ticketsCost%1.2f%n
          That's a $percentIncrease%% bump because everyone
          likes $musicGenre")
```

The above renders ...

```
The Psychedelic Furs tickets are probably $50.00
That's a 20% bump because everyone likes New Wave
```

Smart Strings and Regexes

- Regular Strings are pretty terrible for creating regular expressions since you have to escape backslashes with to two backslashes:

```
val regex = "(\\d{3})-(\\d{4})".r //Yuck
```

the `.r` method creates a `scala.util.matching.Regex`

- A Smart String allows us to create a regex without having the two backslashes

```
val regex = """(\d{3})-(\d{4})""".r //Awesome!
```

Conclusion

- There are various ways to work with Strings in Scala
- You can use the standard String mechanisms you find in Java
- You can use smart string to create multilines.
- You can use the format method to do String style formatting
- You can also use string interpolation with varying flavors to do variable replacements in a String.

Methods

About Methods

- There is a differentiation between *methods* and *functions*
- Methods in Scala belong to context like a `class`, `object`, `trait`, a script, or another method.
- In Scala, a method starts with `def`
- Parameters are in reverse of what is expected in Java, value or variable before the type, e.g `age:Int`

A Basic Non-Concise Method

- The `:Int` is the return type
- If you expect something in return you need, an equal sign (`=`)
- If you do not then leave it out

```
def add(x: Int, y: Int):Int = {
      return x + y
}
```

Cleaning up our Method

- A method can make use of *type inference*
- The braces are optional
- `return` is optional, in fact, it is rarely used

Therefore...

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

Discuss: Why there no longer is `:Int` at the end by referencing the API at http://www.scala-lang.org/api/current/

When type inference is not good enough

- There are times when you need the type:
  - To make things clear
  - Because the type inferencer could be wrong
  - Your method might be overloaded and it would be needed disambiguate from other methods

- You're doing recursion
- You're doing method overloading

Figuring out the type

Given the following, what return type is inferred?

```
def numberStatus(a:Int) =
   if (a < 10) "Less than 10"
   else if (a > 10) "Greater than 10"
   else "It is 10!"
```

Conclusion

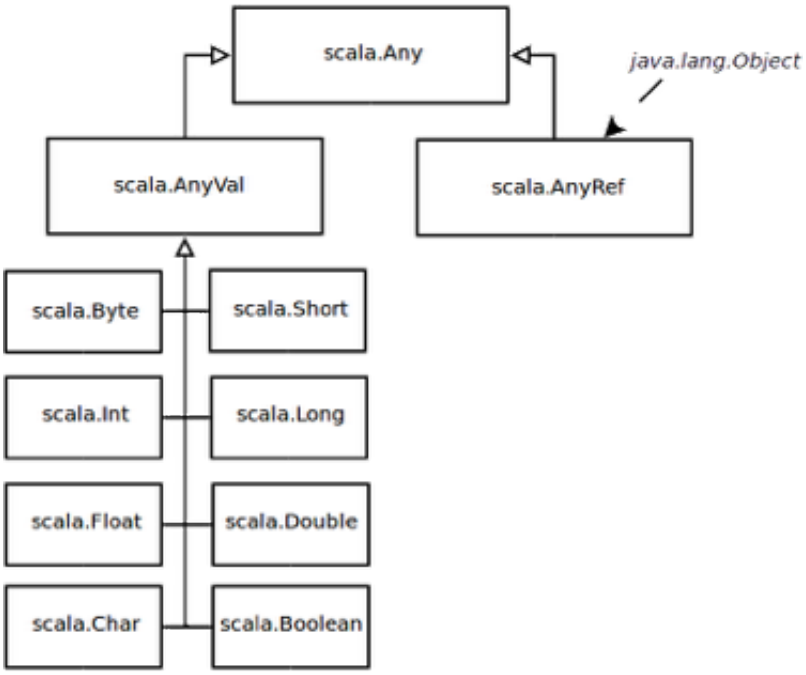
- Methods are defined using `def` and are always defined using `def`
- Methods are not to be confused with functions.
- The `return` keyword is unnecessary...because the last evaluated statement will be returned
- Most of the time you can omit the return type in method unless:
  - You need it for clarity
  - To override the type inferencer
  - You will be performing recursion
  - You will be performing method overloading

Methods Different Return Types

Reminder of Type Inference

- Scala's type "inferencer"
  - Will always make the best choice possible given the information that it has.
  - If it cannot find a match it will search for the parent of the two classes
  - This will apply to how a `List` is created and return types from an `if` Here is the class diagram from the previous session as a reminder.

Reminder of Type Hierarchy



Lab: Colliding types

When there is a chance that two or more types are being returned, the type inferencer will choose the parent of the types being returned.

Step 1: Try to guess what the return type for the following below will be.

Step 2: In the REPL using `:paste` mode, or http://scastie.scala-lang.org, copy and paste the following and determine if you were correct. You may need to exercise the method with some value to see the type.

```
def add(x: Int, y: Int) = {
  if (x > 10) (x + y).toString
  else x + y
}
```

Step 3: Try other combinations and ask questions

Conclusion

- Types inside of a method (this will also be applied to functions) will be inferred.
- Type inferencer will make its judgment based on what is available
- If types are different it will find a common ancestor and use that type

Lab: `isPrime`

Step 1: Go to https://scastie.scala-lang.org/

Step 2: In Scastie, create a method that is called `isPrime` that takes a `Int` and returns whether the number provided is a prime number.

A prime is a positive integer p > 1 that has no positive integer divisors other than 1 and p itself. More concisely, a prime number p is a positive integer having exactly one positive divisor other than 1, meaning it is a number that cannot be factored.

Hint: 1 is not prime, 2 is prime, and use something like a for loop to iterate from `2 to n`. and determine if any number is divisible

More Hint: Here is *a* solution in Python:

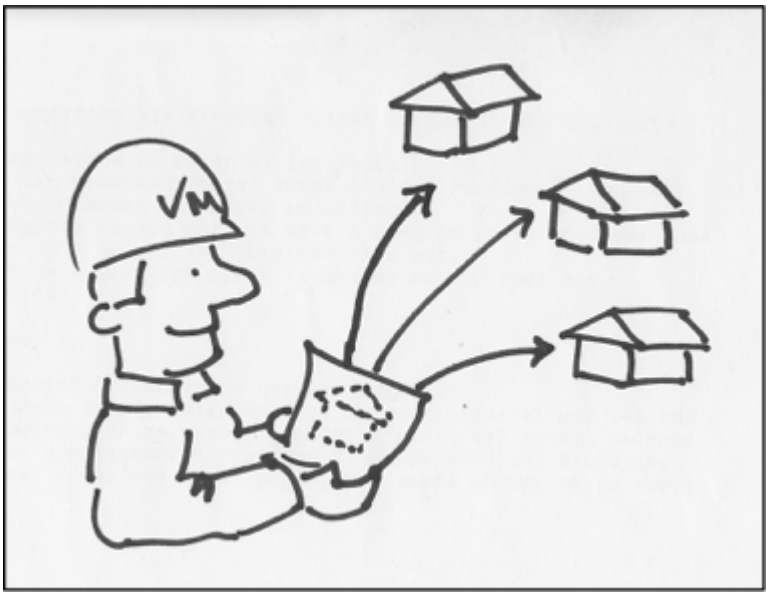
```
def test_prime(n):
   if (n==1):
      return False
   elif (n==2):
      return True;
   else:
      for x in range(2,n):
         if(n % x==0):
            return False
      return True
```

`test_prime` is not our style, use `isPrime` for Scala-style

Type Hierarchy

Reviewing the relationship of class and objects

- A type is a `class`, `trait`, a **primitive**, or an `object` in Scala.
- A `class` for those who don't know is a code template, or blueprint, that describes what the objects created from the blueprint will look like.



- `Int` is a type
- `String` is a type
- If we created a `Car` class, `Car` would be a type
- If we created `InvoiceJSONSerializer`, `InvoiceJSONSerializer` would be a type

Matching Types

Given these two methods:

```
def add(x:Int,y:Int):Int = x + y
def subtract(x:Int, y:Int):Int = x - y
```

To use them together, you would just need to match the types:

```
add(subtract(10, 3), subtract(100, 22))
```

If we changed `subtract` to use `Double` instead:

```
def add(x:Int,y:Int):Int = x + y
def subtract(x:Double, y:Double):Double = x - y
```

We would just need to make sure that that type matches:

```
add(subtract(10.0, 3.0).round.toInt, subtract(100.0, 22.0).round.toInt)
```

Primitives Are Objects

- In Scala we treat everything like an object
- Therefore, you can treat all numbers, boolean, and characters as types
- Every primitive is a member of `AnyVal`

`java.lang.Object` has been demoted

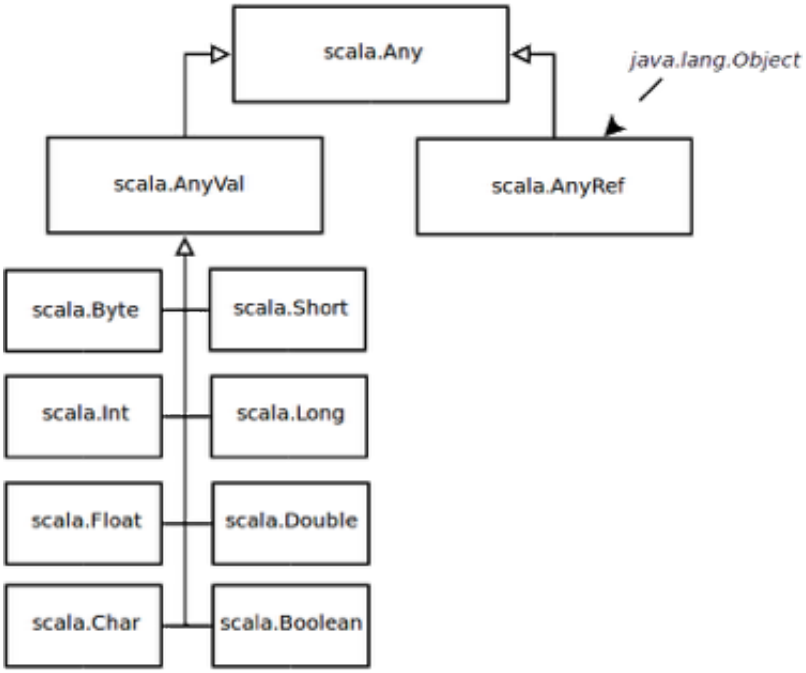
- `java.lang.Object` is called `AnyRef` in Scala
- `AnyRef` will be the super type of all object references
- This includes:
  - Scala API Classes
  - Java API Classes
  - Your custom classes

`Any` is the new leader

- `Any`
  - Super type of:
    - `AnyVal` (Primitive Wrappers)
    - `AnyRef` (Object References)

Scala Family Tree

Scala's Family Tree:



Lab: Polymorphism

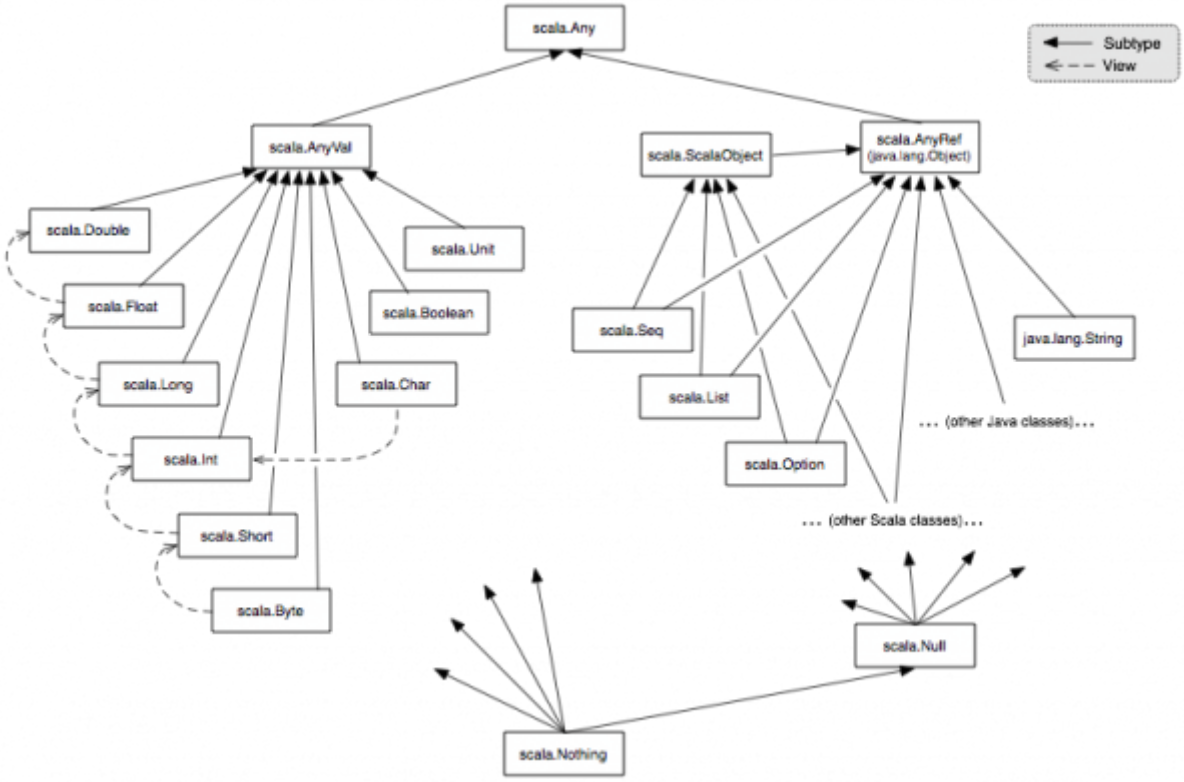
Step 1: Open a REPL or http://scastie.scala-lang.org

Step 2: Try various combinations of references and see how everything is match together. Try some of these combinations that may or may not work to be sure:

```
scala> val a:AnyVal = 40
scala> val b:Any = a
scala> val c:AnyRef = "A String"
scala> val d:Any = c
scala> val e:AnyRef = 40
scala> val f:AnyVal = "A String"
```

Step 3: Discuss and ask any questions.

The Complete Hierarchy



Nothing

- `Nothing` is the sub type of everything
- It is used for collections and other containers with no defined parameterized type
- Used to represent a "bottom", a type that covers among other things, methods, that only throw `Throwable`

`Nothing` in `List`

Given: A List with no parameterized type...

```
> val list = List()
```

This renders:

```
list:List[Nothing] = List()
```

Given: A `List` with a type, and note the difference

```
> val list2 = List[Int]()
```

This renders:

```
list2:List[Int] = List()
```

`Nothing` and throwing an `Exception`

Given an exception, thrown from a method, and that's all that is thrown:

```
> def ohoh(i:Int):Nothing = { throw new Exception() }
```

It shouldn't matter if it is an `Exception` or a `RuntimeException`

Link: http://www.scala-lang.org/api/current/scala/Nothing.html

`Null`

- `Null` is a type in Scala that represents a `null`
- `Null` is the sub type of all object references
- `null` is not used in pure Scala applications, but only for those that interop with Java

`Null` in use

Given:

```
val f = null
```

This renders:

```
f:Null = null
```

`Null` when a type is established

Given:

```
val x:String = null
```

This renders:

```
x:String = null
```

The reason the about is that `null` is the subtype of all `AnyRef`

Conclusion

- Types are templates that make up an object.
- Primitives are also types, `Int`, `Short`, `Byte`, `Char`, `Boolean`, etc.
- Types need to matched up like a puzzle. If it isn't the type system at compile time will tell you there is a type mismatch
- Every type is in a relationship.
- `Any` is the parent for all types
- `AnyVal` is the parent for all primitives
- `AnyRef` is the parent for all Scala, Java, and custom classes that you create
- `Nothing` is the subtype for everything
- `Null` is the subtype of all references

???

About ???

- The triple question mark `???` is a way to mark that a method, `val`, `var` is unimplemented
- The signature:

```
/** <code>???</code> can be used for marking methods
  *  that remain to be implemented.
  *  @throws NotImplementedError
  *  @group utilities
  */
def ??? : Nothing = throw new NotImplementedError
```

- It is of a type `Nothing`
- That also means that it is type safe
- Perfect for compilation without an actual implementation
- Perfect for Test Driven Development

Example of using ???

- The following works!
- `???` throws an `Exception`
- Therefore returns `Nothing`
- `Nothing` is the subtype of everything, including, `Int`

```
def add(x:Int, y:Int):Int = ???
add(10, 12) + 3
```

Conclusion

- `???` is a method that returns `NotImplementedError`
- It *can* be used for Test Driven Development
- It *can* also be used for any other reason
- `???` returns `Nothing` and therefore can be used in nearly any method as a placeholder until you have an implementation

`Unit`

- There is a `Unit` and it can be invoked using `()`
- A `Unit` is analogous to `void` in Java, C, C++
- Used as an interpretation of not returning a something
- Another way to interpret `Unit` is means there is nothing to give

About `Unit`

The following:

```
> val x = ()
```

Will render

```
x:Unit = ()
```

`println` uses `Unit`?

- There is a popular form of `Unit`
- It is called `println`,
- `println` doesn't return anything, therefore it is return `Unit`

Take a look at the signature of `Unit` at the following URL:

https://www.scala-lang.org/api/current/scala/Unit.html

Take a look at the signature of `println` at the following URL:

https://www.scala-lang.org/api/current/scala/Predef$.html

Assigning `println`

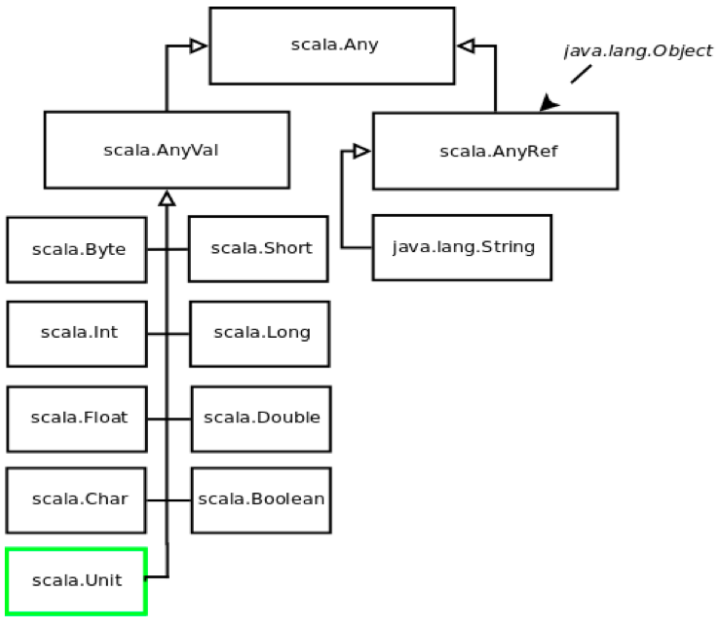
```
> println("Hello, Scala")
```

Now assign to a value! Yes, you can!

```
> val h = println("Hello, Scala")
```

The type `h` will resolve to `Unit`

Where is `Unit` in the Scala hierarchy?



Lab: What is the return type of this?

Step 1: Without running, what do you think the return type is for the following?

```
def add(x: Int, y: Int) = {
  if (x > 10) println(x)
  else x + y
}
```

Step 2: Paste it into a REPL, using the `:paste` mode or go to http://scastie.scala-lang.org and find out to see if you got it right

Step 3: Discuss the reasons why that returned what it did.

`Unit` can be used anywhere

- Unit is just an object that represents a void
- The following is a purely nonsensical method to prove that it can be treated like an object

```
def nonsense(g:Unit):Int = 50
```

Unplanned `Unit`

- In methods, if you do not use the `=` that will implicitly mean that you are returning `Unit`

Given the following which is correct, this will add `x` and `y`:

```
def add(x:Int, y:Int) = {
   x + y
}
```

If we didn't use `=` this will implicitly return `Unit`

```
def badAdd(x: Int, y: Int) {
   x + y
}

println(badAdd(4, 5)) //Returns a ()!
```

Explicitly putting `Unit`

If we can to write out the `Unit` explicitly we can do so, by stipulating `Unit` as the return type

```
def addUnit(x: Int, y: Int):Unit = {
   x + y
}
println(addUnit(4,5)) //Returns a ()!
```

It's called `addUnit` because we didn't create it on purpose.

Side Effects and `Unit`

- When seeing `Unit` in functional programming when being returned from a method, it means it likely is creating a side effect
- A side effect is when some is changed to the outside world either
  - Printing to a screen
  - Printing to a printer
  - Saving to Storage
  - Changing State

The following is changing state. Notice that this is returning `Unit`!
Also, notice that there is `var`!
A `Unit` return will typically mean that there is a side-effect

```
var a = 0
def sideEffect() {
   a = a + 1
}
```

You don't have to be too stringent in avoiding state, although some FP purist will disagree

Conclusion

- `Unit` won't give you anything. (Those jerks)
- They are analogous to Java's `void`.
- `Unit` are actually objects
- Units have a type, `Unit`.
- Units have one value, `()`.
- Whenever you see a `()` that means you have a `Unit`.

Classes

About `class`

- Classes are the templates or blueprints of a construct that encapsulates state and manages behavior.
- Classes have been around for a long time and in every object oriented language.
- Since Scala is a half object oriented language, half functional language, there are naturally classes.

Our first class

- A `class` is public by default, so no need for a `public` modifier
- The `(firstName:String, lastName:String)` is the primary constructor!
- In Scala the primary constructor is "top-heavy" with a constructor that contains all the information
- Other constructors are smaller constructors that feed the top constructor
- The reason for the top heavy constructor is immutability

```
class Employee(firstName:String, lastName:String)
```

Instantiating the class

- Instantiating the `class` is fairly straightforward

```
val emp = new Employee("Dennis", "Ritchie")
```

Can’t access or modify member of a class?

- As it stands in our `class`, we can neither access or modify our `class`
- Most of the time we don’t want to modify our `class` for immutability purposes
- To be able to access the members, we will predicate each of the values with `val`
- To be able to mutate the member variables, we predicate each of the values with `var` (Not recommended)

As it stands, running `javap -p Employee` shows the following Java code:

```
public class Employee {
  public Employee(java.lang.String, java.lang.String);
}
```

Accessing and Mutating

- `val` will create a Scala-style "getter"
- `var` will create a Scala-style "setter"

```
class Employee(val firstName:String, var lastName:String)
```

```
val emp = new Employee("Dennis", "Ritchie")
emp.firstName           //Works because of val
emp.lastName            //Works because of var
emp.lastName = "Hopper" //Works because of var
emp.lastName
```

Viewing bytecode with `val` and `var`

- The bytecode generated from `javap -p Employee`
- Using `val` for `firstName`
- Using `var` for `lastName`

```
public class Employee {
  private final java.lang.String firstName;
  private java.lang.String lastName;
  public java.lang.String firstName();
  public java.lang.String lastName();
  public void lastName_$eq(java.lang.String);
  public Employee(java.lang.String, java.lang.String);
}
```

Conclusion

- Classes are templates or blueprints
- `val` creates accessors, methods that will allow to access the inner state
- `var` create mutators and accessors, mutators allow us to change inner state.
- **IMPORTANT:** We rarely use `var`, it would be best to avoid.
- Use `javap -p` as a utility to view how scala compiles to Java.

Case Classes

About Case Classes

- `case` class
  - Automatically creates a `toString` implementation
  - Automatically creates a `hashCode` implementation
  - Automatically creates a `equals` implementation
  - Automatically creates a `copy` implementation
  - Automatically sets up pattern matching based on the primary constructor
  - Gives us the ability to instantiate without `new`
  - Makes all member variables a `val`
- A child class or parent class can be a `case class` but not both

`case class` and automatic `toString`

Before:

```
class Employee(firstName:String, lastName:String)

val e = new Employee("Desmond", "Everett")
println(e)
```

Renders:

```
Employee@58f254b1
```

`case class` and automatic `toString`

After:

```
case class Employee(firstName:String, lastName:String)

val e = new Employee("Desmond", "Everett")
println(e)
```

Renders:

```
Employee("Desmond", "Everett")
```

`case class` and automatic `hashCode`

Before:

```
class Employee(firstName:String, lastName:String)

val e1 = new Employee("Desmond", "Everett")
val e2 = new Employee("Desmond", "Everett")
println(e1.hashCode)
println(e2.hashCode)
```

Renders:

```
1963269381
1108152893
```

`case class` and automatic `hashCode`

After:

```
case class Employee(firstName:String, lastName:String)

val e1 = new Employee("Desmond", "Everett")
val e2 = new Employee("Desmond", "Everett")
println(e1.hashCode)
```

Renders:

```
-2009758914
-2009758914
```

`case class` and automatic `equals`

Before:

```
class Employee(firstName:String, lastName:String)

val e1 = new Employee("Desmond", "Everett")
val e2 = new Employee("Desmond", "Everett")
println(e1 == e2) //same as equals
```

Renders:

```
false
```

`case class` and automatic `equals`

After:

```
case class Employee(firstName:String, lastName:String)

val e1 = new Employee("Desmond", "Everett")
val e2 = new Employee("Desmond", "Everett")
println(e1 == e2)
```

Renders:

```
true
```

`case class` and automatic pattern matching

```
case class Employee(firstName:String, lastName:String)

val Employee(fn, ln) = new Employee("Desmond", "Everett")
println(fn)
println(ln)
```

Renders:

```
Desmond
Everett
```

`case class` and `new` not required

- Notice in the following that the `new` keyword is not required
- There is a subtle trick that is happening that we will see later on

```
case class Employee(firstName:String, lastName:String)

val emp = Employee("Desmond", "Everett")
```

`case class` and `val` not required

- There is no requirement for `val` on a member variable when using a `case class`
- A Scala-Style getter is automatically created

```
case class Employee(firstName:String, lastName:String)

val emp = Employee("Desmond", "Everett")
println(emp.firstName)
println(emp.lastName)
```

Renders:

```
Desmond
Everett
```

`case class` and `copy`

- Since `Employee` is immutable, `copy` will create copy of the object with new values
- Simple name the properties you are changing

```
case class Employee(firstName:String, lastName:String)

val emp = Employee("Desmond", "Everett")
val empCopy = emp.copy(lastName = "Gillespie")
println(empCopy)
```

Renders:

```
Employee(Desmond,Gillespie)
```

Conclusion

- Automatically creates a `toString` implementation
- Automatically creates a `hashCode` implementation
- Automatically creates a `equals` implementation
- Automatically sets up pattern matching based on the primary constructor
- Gives us the ability to instantiate without `new`
- Makes all member variables a `val`
- Provides a copy method to generate a copy
- A child class or parent class can be a `case class` but not both

Using Infix Operators

Infix Operators

- In Scala, a method with one argument can be called as an *infix operator*
- Using that rule, that's what makes:

```
1.+(2)
```

Look like...

```
1 + 2
```

Creating your own infix operator

Therefore, in Scala, if we create our class like the following:

```
class Foo(x:Int) {
   def bar(y:Int) = x + y
}
```

and we want to invoke it, we can call it not only like this:

```
val foo = new Foo(40)
foo.bar(10) // Called non-infix
```

but like this:

```
val foo = new Foo(40)
foo bar 10  // Called infix
```

Conclusion

- If a method has one argument you can call it in an infix manner
- This is called an infix operator

The Magical `apply` method

Considering the following code once again:

```
class Foo(x:Int) {
   def bar(y:Int) = x + y
}
```

If we run it, it would like this and return 50

```
val foo = new Foo(40)
foo.bar(10)
```

Replacing `bar` with `apply`

Let's take the previous code and use `apply` instead of `bar`:

```
class Foo(x:Int) {
   def apply(y:Int) = x + y
}
```

If we run it, it would like this and return 50

```
val foo = new Foo(40)
foo.apply(10)
```

Where thing are different, is that **apply is not required method call and you can leave the word out!**

Therefore...since we used apply it looks like this:

```
val foo = new Foo(40)
foo(10)
```

Conclusion

- If the method is called apply, *no matter where it was defined*, you can leave the explict call out!
- This is probably one of the most important aspects to the language that few know about since it too many it is too obvious too mention

The magical "right-associative" colons

Right associative colons

- Right-associative colons works with operator overloading only
- It flips the way we call our methods using it as an infix operator

Let's take our famous example, given:

```
class Foo(x:Int) {
   def bar(y:Int) = x + y
}
```

Let's rename it with a right-associative colon

```
class Foo(x:Int) {
   def ~:(y:Int) = x + y
}
```

Invoking the right associative colon

Now how do invoke the right associative colon given the following?

```
class Foo(x:Int) {
   def ~:(y:Int) = x + y
}
```

We can call it just like any method and it will return `50`:

```
val foo = new Foo(10)
foo.~:(10)
```

Or we can call it as an infix operator and get a compiler error:

```
val foo = new Foo(10)
foo ~: 10 //Compiler error: value ~: is not a member of Int
```

To still use it as an infix operator let's flip it!

```
val foo = new Foo(10)
10 ~: foo
```

Mind trick! The "col"on always attaches to the "col"lection or the object, we will see with a list

Lab: Creating and manipulating a `List`

- Although we may not use it too much, it will still show up so where do you see it?
- You will see it as an alternative for creating a `List`
- You will see it when manipulating a `List`

Step 1: Open a REPL or go to http://scastie.scala-lang.org and create a `List` in the following manner

```
val list = List(1,2,3,4)
```

Step 2: Try the alternative method, by using `::` and `Nil` which is an empty `List`

```
val list2 = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
```

Step 3: Figure out how this is working knowing what you know about right associative colons

Step 4: Append an element to `list2` using the `:+` method

```
list2 :+ 6
```

Step 5: Prepend an element to `list2` using the `:+` method

```
0 +: list2
```

Step 6: Combine them to see what happens

```
0 +: list2 :+ 6
```

Step 7: Explain how all of this works

Conclusion

- Right associative colons, will flip how it is invoked
- Not often used but there will moments where you will need it
- It is used to flexibly do things to the language

`Option`

About `Option`

- If I ask you if you have a middle name you'd either answer:
  - Yes
  - No
- If it is yes, I might press further and ask "What is it?"
- That is how `Option[T]` works
- It is also a way that we avoid `null`
- `null` is still available to interoperate with Java and its libraries

The problem with `null`

Apologies from Tony Hoare 2009

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

The ambiguities of `null`

If in a database, the middle name field has a `null`, does it mean:

- The person doesn't have a middle name
- No one has entered the middle name

`Option[A]`

- `Option[+A]` is the super type of `Some[T]` and `None`
- `Option[+A]` is an `abstract` class

https://www.scala-lang.org/api/current/scala/Option.html

How it is used:

- Using `Some`

```
val middleName = Some("Antony") //type is inferred as Some
val middleName2:Option[String] = middleName
val middleName3:Some[String] = middleName
```

- Using `None`

```
val noMiddleName = None //Singleton
val noMiddleName:Option[Nothing] = noMiddleName
val noMiddleName:None.type = noMiddleName
```

How do we interact with an `Option`

- To interact with an `Option` we can either call:
  - `get` (Dangerous)
  - `getOrElse` (Safe)
  - Pattern Matching (Whoa)
  - Functions (Sweet!)

```
middleName.getOrElse("N/A") //Antony
noMiddleName.getOrElse("N/A") //N/A
```

If we chose get we run the danger of the last one of throwing an exception

```
middleName.get //Antony
noMiddleName.get // java.util.NoSuchElementException: None.get
```

Getting the info using Pattern Matching:

- If we had a method that received an `Option[T]` we can use pattern matching to break it apart and analyze it

```
def whatIsTheMiddleName_?(x:Option[String]):String = {
   x match {
     case Some(a) => a
     case None    => "N/A"
   }
}
```

- Here we can break up the `Option[T]` by specifically asking what particular type we are looking at, a `Some` or `None`
- If it is a `Some`, ***extract*** the value and return, if it is a `None` return `"N/A"`

Conclusion

- Scala programmers despise `null`, so we use `Option`
- Options are modeled as `Some` or `None`, and if `Some`, we can extract the answer.
- Extracting the answer can be done by calling `get`, `getOrElse`, or pattern matching, or functions (though we haven't discussed functions yet)
- Scala still has `null` to interoperate with Java, but in a pure Scala application, don't use `null`.

`object`

About `object`

- We know that objects are instantiated from classes
- We can also create an object, a single object, without a class
- This is called an `object`
- **IMPORTANT:** An `object` is a **singleton**. There is only one
- It is how we avoid `static` methods

Lab: Trying out the `object`

Step 1: Go to the REPL or http://scastie.scala-lang.org and type the following

```
object MyObject
```

Step 2: Verify that it is truly a singleton

```
val a = MyObject
val b = MyObject
a == b
a eq b
```

Step 3: Discuss why this works

How we avoid `static`

- We can add values, or variables, and methods to an `object`
- When we invoke those values, variable, and methods it looks and feels like invoking something `static`

An example `object` with a `def`

- You can have `val`, `var`, `class`, `object`, `trait` in an `object`
- Here is `MyObject` which is an `object` with a method `foo`

```
object MyObject {
   def foo(x:Int, y:Int) = x + y
}
```

- This can be invoked using:

```
MyObject.foo(4, 2) should be (6)
```

This has a look and feel of Java `static` method

When do you need objects?

- For Classes
  - Need to define a template to create multiple instances
  - Every instance has a state
- For Objects
  - You need a singleton
  - You need a factory pattern, which defined as: Creating families of related or dependent objects without specifying or hiding their concrete classes.
  - You need to implement pattern matching logic
  - You need create a utility that doesn't require and instance or a state.
  - You have some default values or constants

The `main` method

- The `main` method in Java requires all of this so that we can execute it as an application

```
public static void main(String[] args) {
   System.out.println("Hello, Scala")
}
```

- The same elements apply in Scala although the components are different
  - `object` instead of `static`
  - `Unit` instead of `void`
  - Everything is `public` by default
  - `Array[String]` instead of `String[]`

Therefore the `main` method in Scala looks like:

```
object Runner { //Call it whatever you want
  def main(args:Array[String]):Unit = println("Hello, Scala")
}
```

Alternative `main` method

- You can also create a `main` method doing the following:

```
object Runner extends App {
   println("Hello, Scala")
}
```

Conclusion

- Objects are Singletons
- Objects are Scala's replacement for `static`
- Objects are typically meant for factories, utilities, defining pattern matching, defining defaults, and main methods
- `main` methods are inside of objects...always
- You can forgo the main method declaration have the object extend App.

Companion Objects

Using Companion Objects

- Companion Objects
  - Still singletons
  - Service a `class`, `trait`, `abstract class`
- Companion Object Rule:
  - Must have the same name as the `class`, `trait`, `abstract class` it supports
  - The class and the object ***has to be in the same file***

`trait` is analogous to an `interface` in Java

Companion Object Benefit

- `class` and `object` can share `private` information
- This gives `object` perfect for being a factory of the corresponding `class`
- Stores logic for pattern matching

Accessing `private` shared state

```
class SecretAgent(val name: String) {
  def shoot(n: Int) {
    SecretAgent.decrementBullets(n) //Can be imported and shortened
                                    //using import SecretAgent._
  }
}

object SecretAgent {
  //This is encapsulated!
  private var b: Int = 3000 //only available privately

  private def decrementBullets(count: Int) {  //only available privately
    if (b - count <= 0) b = 0
    else b = b - count
  }

  def bullets = b
}
```

Running this would look like:

```
object SecretAgentRunner extends App {
    val bond = new SecretAgent("James Bond")
    val felix = new SecretAgent("Felix Leitner")
    val jason = new SecretAgent("Jason Bourne")
    val _99 = new SecretAgent("99")
    val max = new SecretAgent("Max Smart")

    bond.shoot(800)
    felix.shoot(200)
    jason.shoot(150)
    _99.shoot(150)
    max.shoot(200)

    println(SecretAgent.bullets) //How many bullets are left?
}
```

Creating a factory

- One of the main uses of a companion object is to use it as a factory
- To do so, lets say, direct invocation of `Department` is locked with `private`

```
class Department private(val name:String)
```

- We can then create a factory using the companion object to `Department` class

```
object Department {
   def create(name:String) = new Department(name)
}
```

The above really is just a *static factory pattern* that you would see in Java.

Calling the factory from the companion object

- How can we call the factory?

```
val department = Department.create("Toys")
department.name should be ("Toys")
```

Important Question: Is there a better name instead of `create`? Decide, then look up how you create `List`, `Set`, `Map`!

Conclusion

- Companion Objects have the same name as the class that they work for
- Companion Objects must be in the same file.
- Companion Objects have access to their classes private information.
- Classes have access to the companion's private information.

Tuples

Defining Tuples

- Tuples are dumb containers
- Perfect for grouping items
- Perfect for return two items or three or four particularly if they are different types

Creating a `Tuple`

```
val ta = (1, "cool", 402.00)
val tb:(Int, String, Double) = (1, "cool", 402.00)
val tc:Tuple3[Int, String, Double] = (1, "cool", 402.00)
```

There is a `Tuple1`, `Tuple2`, `Tuple3`,...`Tuple22`

Getting the values from a `Tuple`

```
val ta = (1, "cool", 402.00)
println(ta._1)
println(ta._2)
println(ta._3)
```

The `_1`, `_2` comes from Haskell's `first` and `second` functions

Swapping `Tuple`

- Only a `Tuple2` can be swapped

```
val t2 = ("Foo", 40.00)
println(t2.swap) //(40.00, "Foo")
```

Syntactic Sugar for Tuples

- `Tuple2` can also be created with `->`
- The `->` is a method that is on every object that accepts another object

```
val t2 = "Foo" -> 40.00 // (40.00, "Foo")
```

You can use the `-` and `>` or you can use the unicode rightwards arrow: `→`

Conclusion

- Tuples are just dummy containers, they just hold stuff.
- Tuples are typed
- There are tuples that go all the way to `Tuple22`
- `Tuple2` has `swap`
- Tuples are immutable, become an essential part not only in the Scala language, but functional programming as well.

Lists

- `List` are like `java.util.List` where they are indexed collections that hold usually homogeneous data
- `List` are unlike `java.util.List` where they are an immutable `List`
- `List` generally allows duplicates

Varying Ways to create a `List`

```
val a = List(1,2,3,4,5) //What is this call?
val b = 1 :: 2 :: 3 :: 4 :: 5 :: Nil
val c = Nil:List[String]
```

Some methods for `List`

```
println(a.head)
println(a.tail)
println(a.init)
println(a.last)

println(a(4)) //5 <--Wait what is this?
println(a.max)
println(a.min)
println(a.isEmpty)
println(a.nonEmpty)
println(a.updated(3, 100)) //Underused

println(a.mkString(",")) //available on all collections!
println(a.mkString("{", " ## ", "}"))
```

Conclusion

- Lists are a immutable collection, duplicates allowed
- Nil is an empty List
- Lists are created with the object and an apply factory
- List have all the functional properties as other collections have

Range

Range direct

- Obtaining `0…4`, exclusively

```
val exclusive = Range(0, 4) //0,1,2,3
```

- Obtaining `0…4`, inclusively

```
val inclusive = Range.inclusive(0, 4) //0,1,2,3,4
```

Source: https://www.scala-lang.org/api/current/scala/collection/immutable/Range$.html

Range with implicit trickery

- Obtaining `0…4`, exclusively

```
val exclusive = 0 until 4 //0,1,2,3
```

- Obtaining `0…4`, inclusively

```
val inclusive = 0 to 4 //0,1,2,3,4
```

Source: https://www.scala-lang.org/api/current/scala/collection/immutable/Range$.html

Range exclusively with Positive Steps

- Obtaining `1…20` with a positive step of 2, exclusively, using Standard API

```
Range(0, 20, 2).toVector //Vector(0, 2, 4, 6, 8, 10, 12, 14, 16, 18)
```

The above uses the `apply` method

- Obtaining `1…20` with a positive step of 2, exclusively, using Implicit Trickery

```
(0 until 20 by 2).toVector //Vector(0, 2, 4, 6, 8, 10, 12, 14, 16, 18)
```

Range inclusively with Positive Steps

- Obtaining `1…20` with a positive step of 2, inclusively, using Standard API

```
Range.inclusive(0, 20, 2).toVector //Vector(0, 2, 4, 6, 8, 10, 12, 14,
16, 18, 20)
```

The above uses the `apply` method

- Obtaining `1…20` with a positive step of 2, inclusively, using Implicit Trickery

```
(1 to 20 by 2).toVector //Vector(0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```

Range exclusively with Negative Steps

- Obtaining `20…0` with a negative step of -2, exclusively, using Standard API

```
Range(20, 0, -2).toVector //Vector(20, 18, 16, 14, 12, 10, 8, 6, 4, 2)
```

The above uses the `apply` method

- Obtaining `20…0` with a negative step of -2, exclusively, using Implicit Trickery

```
(20 until 0 by -2).toVector  //Vector(20, 18, 16, 14, 12, 10, 8, 6, 4,
2)
```

Range inclusively with Negative Steps

- Using Standard API
- Obtaining `20…0` with a negative step of -2, inclusively, using Standard API

```
Range.inclusive(20, 0, -2).toVector //Vector(20, 18, 16, 14, 12, 10, 8,
6, 4, 2, 0)
```

The above uses the `apply` method

- Obtaining `20…0` with a negative step of -2, inclusively, using Implicit Trickery

```
(20 to 0 by -2).toVector  //Vector(20, 18, 16, 14, 12, 10, 8, 6, 4, 2,
0)
```

Sets

- Just like underlying Java and most programming languages, a `Set` is:
  - `Collection` that doesn't have duplicate elements
  - Generally have more mathematical methods than `List`
  - Doesn't maintain order
- Some of the characteristics of `Set`
  - `head`, `tail` are not available
  - `apply` has a different behavior

Creating a `Set`

```
val set =  Set(1,2,3,4)
val set2 = Set.apply(1,2,3,4,5)
```

Calculating the differences of a `Set`

The following calculates the differences of a `Set`

```
Set(1,2,3,4) diff Set(1,2,3,4,5,6,7)
```

Returns no result

```

```

Whereas, the opposite...

```
Set(1,2,3,4,5,6,7) diff Set(1,2,3,4)
```

Will return...

```
Set(5,6,7)
```

Calculating the `union` of two `Set`

- A union will provide the combination of the two `Set`

```
Set(1,2,3,4) union Set(5,10)
```

Returns...

```
Set(5, 10, 1, 2, 3, 4)
```

Calculating the `intersect` of two `Set`

- `intersect` is the opposite of a `diff` and shows the commonality of two `Set`

```
Set(1,2,3,4) intersect Set(19,2,3,10)
```

Will Return...

```
Set(2, 3)
```

Using `apply` with a `Set`

- `apply` will only return the same as `contains` and that is whether the element is in the `Set` or not

```
val set = Set(1,2,3,4)
set.apply(4) //true
set.apply(10) //false
set.contains(4) //true
```

Conclusion

- Sets are collections with no duplicate elements
- More mathematically powerful than the counter part
- Sets have a hash order that is undetermined (if less than 5)
- `apply` will return `true` or `false`

Using Maps

Maps

- Called associative arrays, dictionary, or tables in other languages
- Table of keys and values
- Items are looked up by key

Creating a `Map`

- The following calls all create the same `Map`

```
val m = Map.apply((1, "One"), (2, "Two"), (3, "Three"))

val m = Map((1, "One"), (2, "Two"), (3, "Three"))

val t:(Integer, String) = 1 -> "One"

val m = Map(1 -> "One", 2 -> "Two", 3 -> "Three")
```

Note: The `->` is syntactic decorator that creates a `Tuple2`

`Map` safe retrieval by key

Given a `Map` we created previously.

```
val m = Map(1 -> "One", 2 -> "Two", 3 -> "Three")
```

To retrieve by key:

```
m.get(1)
```

Returns...

```
Option[String] = Some(One)
```

When no key is available, then the result will be `None`

```
m.get(4)
```

Returns...

```
Option[String] = None
```

`Map` unsafe retrieval by key

Given a `Map` we created previously.

```
val m = Map(1 -> "One", 2 -> "Two", 3 -> "Three")
```

Calling `apply` will retreive the value direct without wrapping it in an `Option`

```
m.apply(1)
```

```
One
```

The problem that you will have to be careful about when calling an `apply` on a `Map` that doesn't contain the `key`

```
m.apply(4)
```

At that point you will receive

```
java.util.NoSuchElementException: key not found: 4
  at scala.collection.MapLike$class.default(MapLike.scala:228)
  at scala.collection.AbstractMap.default(Map.scala:59)
  at scala.collection.MapLike$class.apply(MapLike.scala:141)
  at scala.collection.AbstractMap.apply(Map.scala:59)
  ... 32 elided
```

`Map` retrieval of key `Iterable`

Given a `Map` we created previously.

```
val m = Map(1 -> "One", 2 -> "Two", 3 -> "Three")
```

To retrieve an `Iterable` of keys

```
m.keys
```

```
Iterable[Int] = Set(1, 2, 3)
```

To retreive them as a `Set`

```
m.keySet
```

```
scala.collection.immutable.Set[Int] = Set(1, 2, 3)
```

Conclusion

- Maps are a table-like collection that store keys and values
- Internally, maps are a collection of tuples, and can be operated on as such

The Importance of a Clean API

- In Scala, methods names and parameters were curated with care
- Lesson: Once you learn most if not all methods of `List` you will also know
  - `Set`
  - `Map`
  - `Stream`
  - `String`
  - `Future`
  - `Option`
  - `Queue`
  - `Range`
  - `Vector`
- In some capacity that will also include `mutable` collections
- Knowing this the learning curve drops significantly

Using Functions

Functions

- Think of functions as something that take input and throws output
- Functions are based on traits, `Function1`, `Function2`, ..., `Function22`

Functions the hard way

Given: An anonymous instantiation

```
val f1:Function1[Int, Int] = new Function1[Int, Int] {
   def apply(x:Int) = x + 1
}

println(f1.apply(4)) //5
```

But since, the name of the method is `apply`....

```
println(f1(4)) //5
```

Note: In 2.12, it may not be like this, since they will integrate with Java 8's backed `java.util.function`

Different Signatures of Function (The long way)

- Here are some functions declared the long way
- We explicitly declare the types on the left hand side `Function1`, `Function0`
- We are instantiating the `traits`

```
val f1:Function1[Int, Int] = new Function1[Int, Int] {
     def apply(x:Int) = x + 1
}

val f0:Function0[Int] = new Function0[Int] {
     def apply() = 1
}

val f2:Function2[Int, String, String] = new Function2[Int, String,
String] {
     def apply(x:Int, y:String) = y + x
}
```

Signatures of Function (The medium way)

- Here are some functions declared the medium way
- We explicitly declare the types on the left hand side as:
  - `Int ⇒ Int` to represent `Function1[Int, Int]`
  - `() ⇒ Int` to represent `Function0[Int]`
  - `(Int, String) ⇒ String` to represent `Function2[Int, String, String]`
- Of course these functions are still anonymous instantiating of `traits`

```
val f1:(Int => Int) = new Function1[Int, Int] {
     def apply(x:Int) = x + 1
}

val f0:() => Int = new Function0[Int] {
     def apply() = 1
}

val f2:(Int, String) => String = new Function2[Int, String, String] {
     def apply(x:Int, y:String) = y + x
}
```

Trimming down the functions

- Give the functions from the previous page, we can clean up the functions
- The left hand side uses a short hand notation to declare the types as before
- The right hand side uses a short hand notation to create the function!

```
val f1:Int => Int = (x:Int) => x + 1

val f0:() => Int = () => 1

val f2:(Int, String) => String = (x:Int, y:String) => y + x
```

Making it concise: Using type inference with functions

- Since there is heavy type inference in Scala, we can trim everything down
- The left hand side can be left with little or no type annotation since the right hand side is declaring most of the information

```
val f1 = (x:Int) => x + 1

val f0 = () => 1

val f2 = (x:Int, y:String) => y + x
```

Choosing the type inference, left hand side or right hand side

Using right hand side verbosity, and left hand side type inference

```
val f1 = (x:Int) => x + 1

val f0 = () => 1

val f2 = (x:Int, y:String) => y + x
```

Using right hand side inferred, and left hand side verbosely declared

```
val f1:Int => Int = x => x + 1

val f0:() => Int = () => 1

val f2:(Int, String) => String = (x,y) => y + x
```

Returning more than one result

- Every function has one return value so how do we return multiple items?
- Use a `Tuple`!

```
val f3 = (x:String) => (x, x.size)  // Right hand side declaration
println(f3("Laser"))                // ("Laser", 5)
```

Trimming the result with _

- Given the function where the left hand side is declared with the type
- We can provide a shortcut using the _ that represents one of the arguments
- Seasoned Scala developers will recognize this shortcut
- Be aware that this may be confusing for novices

Given:

```
val f5: Int => Int = x => x + 1
```

Can be converted to:

```
val f5: Int => Int = _ + 1
```

Works the same when invoking the function

```
f5(40)  //41
```

Advanced Feature: Postfix Operators

- If the argument on the right most position is a `_`, it can be omitted
- This may require that an `import` feature may need to be turned on

Given:

```
val f5: Int => Int = x => x + 1
```

Can be converted to:

```
val f5: Int => Int = _ + 1
```

Since addition is commutative, it can be rearranged jj

```
val f5: Int => Int = 1 + _
```

Since, the `_` is at the right most position it can be dropped

```
val f5: Int => Int = 1+
```

Works the same when invoking the function

```
f5(40)  //41
```

Advanced Feature: Postfix Operators, Clearing Warnings

- If you attempted to clear out the most `_` on the right hand side you may have received this

warning:

```
warning: postfix operator + should be enabled
by making the implicit value scala.language.postfixOps visible.
This can be achieved by adding the import clause 'import
scala.language.postfixOps'
or by setting the compiler option -language:postfixOps.
See the Scaladoc for value scala.language.postfixOps for a discussion
why the feature should be explicitly enabled.
```

- This means that to avoid the warning you can turn that feature on by an `import` statement

```
import scala.language.postfixOps
val f5: Int => Int = 1+
```

Works the same when invoking the function, and you get no warnings

```
f5(40)  //41
```

Trimming the result with multiple _

- If a function has two arguments, that too can use the _ shortcut
- The only thing is that _ can only be applied to **one** argument

Here is the before:

```
val sum: (Int, Int) => Int = (x, y) => x + y
```

Here is the after:

```
val sum: (Int, Int) => Int = _ + _
```

Note in the above:

- The first _ represents the first argument, or `x` in the above example
- The second _ represents the second argument, or `y` in the above example

Conclusion

- Functions are traits that we instantiate anonymously.
- The `apply` method in the function means that you don't have to call apply explicitly.
- While you can only return one item, that one item can be a collection or a tuple.

- Shortcuts to the argument can be made with _
- The _ represents each argument being applied to the function

Using Higher Order Functions

Higher Order Functions

- A *higher order function* is a function or method that:
  - Takes other functions as parameters
  - Has a result as a function

A method that takes a function

```
def process(x:Int, y:Int, f:(Int, Int) => Int) = f(x,y)
```

The above example has three arguments: An `Int`, another `Int`, and a higher order function that takes two `Int` and returns an `Int`

To run the above we can run:

```
process(4, 6, (x, y) => x * y)
```

or

```
process(4, 6, _ * _)
```

A function that takes a function

Of course a function can also take a function

```
val process = (x:Int, y:Int, f:(Int, Int) => Int) = f(x, y)
```

The above example is a function that has three arguments: An `Int`, another `Int`, and a higher order function that takes two `Int` and returns an `Int`

To run the above we can run:

```
process(4, 6, (x, y) => x * y)
```

or

```
process(4, 6, _ * _)
```

A function that returns a function

- A function, or method can return a function
- This is also a higher order function

Function returning a function

```
val process = (x:Int, y:Int) => (z:Int) => x + y + z
val f2 = process(10, 10)
f2(3) //23
```

Method returning a function

```
def process(x:Int, y:Int) = (z:Int) => x + y + z
val f2 = process(10, 10)
f2(3) //23
```

Running both a `def` and a function as if they are they're the same semantics is no accident. Though mechanically different after a while the difference between the two becomes blurry

Closures

- Closures close around the environment
- In the below example
  - The `outer` inside of function `f` closes around the environment.
  - `x` is not a closure, it is the input to the function `f`

```
val outer = 99
val f = (x:Int) => outer + x
```

When used, you can then call `f` in the above example and outer will "come along for the ride" since it is a closed value.

```
f(4) //103
```

Lab: Closure by example

- Given the following it seems that `MyFunctions.lessThan` is reusable
- This returns a function that will receive a number and ask if it less than the previously defined value

Step 1: In the REPL or http://scastie.scala-lang.org copy the following code and implement

what `MyFunctions.lessThan` look like?

```
val isFreezingCelcius = MyFunctions.lessThan(0)
val isFreezingFahrenheit = MyFunctions.lessThan(32)
isFreezingFahrenheit.apply(25) // true
isFreezingCelcius.apply(25) // false
```

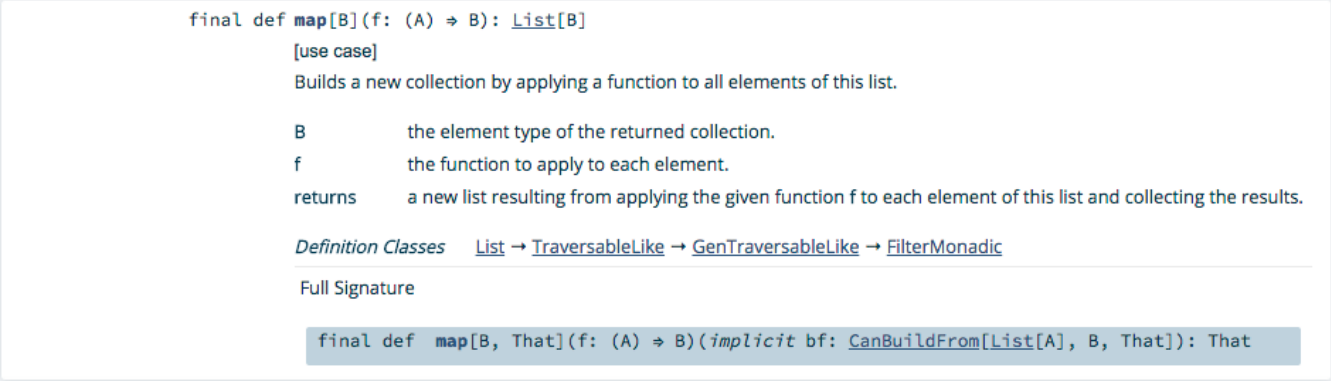
`map` function

About the `map` function

- `map` applies a higher order function to every element in a collection or container
- Available on nearly every collection or container in Scala
- The structure and properties for `map` are nearly identical across all collections

`List` and `map`

- This is the API Signature of `map` for `List`
- Note that the parameterized type of the `List` in this case is `[A]`



List and map

```
List(1,2,3,4).map(x => x + 1)
```

Renders:

```
List(2,3,4,5)
```

Link: https://www.scala-lang.org/api/current/scala/collection/immutable/List.html

`Set` and `map`

- This is the API Signature of `map` for `Set`
- Note that the parameterized type of the `Set` in this case is `[A]`

```
def map[B](f: (A) ⇒ B): Set[B]
```

[use case]

Builds a new collection by applying a function to all elements of this immutable set.

B	the element type of the returned collection.
f	the function to apply to each element.
returns	a new immutable set resulting from applying the given function f to each element of this immutable set and collecting the results.

Definition Classes SetLike → TraversableLike → GenTraversableLike → FilterMonadic

Full Signature

```
def map[B, That](f: (A) ⇒ B)(implicit bf: CanBuildFrom[Set[A], B, That]): That
```

Set and map

```
Set(5,10,13,12).map(x => x + 1)
```

Renders:

```
Set(6, 11, 14, 13)
```

Link: https://www.scala-lang.org/api/current/scala/collection/immutable/Set.html

`Map` and `map`

- This is the API Signature of `map` for `Map`
- Note that the parameterized type of the `Map` in this case is `[K,V]`
  - `K` is the key parameterized type
  - `V` is the value parameterized type

```
def map[B](f: (A) ⇒ B): Map[B]
```

[use case]

Builds a new collection by applying a function to all elements of this immutable map.

B	the element type of the returned collection.
f	the function to apply to each element.
returns	a new immutable map resulting from applying the given function f to each element of this immutable map and collecting the results.

Definition Classes TraversableLike → GenTraversableLike → FilterMonadic

Full Signature

```
def map[B, That](f: ((K, V)) ⇒ B)(implicit bf: CanBuildFrom[Map[K, V], B, That]): That
```

```
def mapValues[W](f: (V) ⇒ W): Map[K, W]
```

Transforms this map by applying a function to every retrieved value.

f	the function used to transform values of this map.
returns	a map view which maps every key of this map to f(this(key)). The resulting map wraps the original map without copying any elements.

Definition Classes MapLike → MapLike → GenMapLike

Map and map

```
Map(1 -> "One", 2 -> "Two").map(t => (t._1 + 100, t._2 + " Hundred"))
```

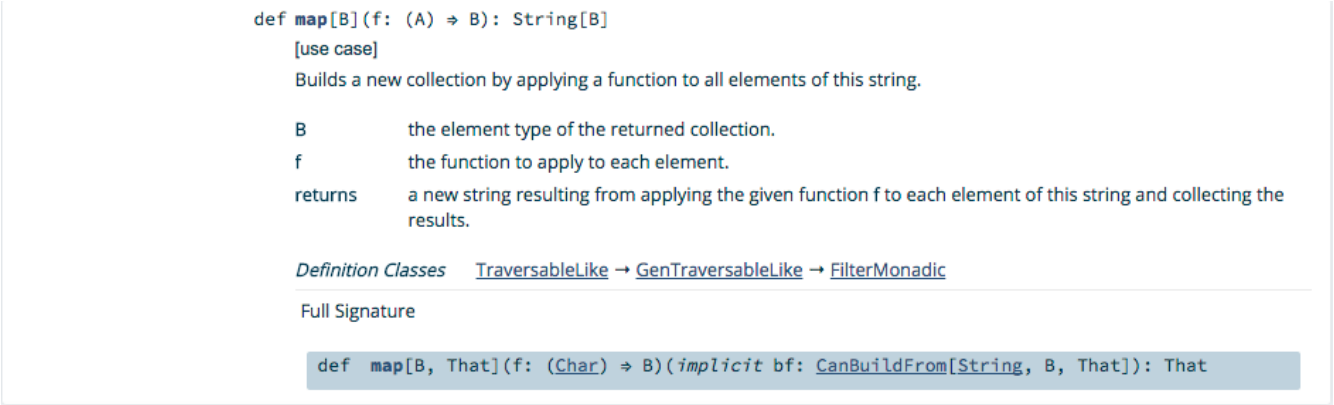
Renders:

```
Map(100 -> "One Hundred", 200 -> "Two Hundred")
```

Link: https://www.scala-lang.org/api/current/scala/collection/immutable/Map.html

`String` and `map`

- This is the API Signature of `map` for `StringOps`
- Note that the parameterized type of the `StringOps` in this case is actually `Char`



def map[B](f: (A) ⇒ B): String[B]

[use case]

Builds a new collection by applying a function to all elements of this string.

B	the element type of the returned collection.
f	the function to apply to each element.
returns	a new string resulting from applying the given function f to each element of this string and collecting the results.

Definition Classes TraversableLike → GenTraversableLike → FilterMonadic

Full Signature

```
def map[B, That](f: (Char) ⇒ B)(implicit bf: CanBuildFrom[String, B, That]): That
```

String with map

```
"Hello".map(c => (c + 1).toChar)
```

Renders:

```
Ifmmp
```

Conclusion

- `map` takes a function, and applies that function in every element in a collection.
- `map` can be applied to `List`, `Set`, `Map`, `Stream`, `String`, even `Option`!

`filter` function

Using the `filter` function

- `filter` removes elements from a collection based on a function or predicate
- A predicate is a function that returns `true` or `false`
- Available on nearly every collection in Scala
- The structure and properties for `filter` are nearly identical across all collections

`List` and `filter`

- This is the API Signature of `filter` for `List`
- Note that the parameterized type of the `List` in this case is `[A]`

```
def filter(p: (A) ⇒ Boolean): List[A]
```

Selects all elements of this traversable collection which satisfy a predicate.

p	the predicate used to test elements.
returns	a new traversable collection consisting of all elements of this traversable collection that satisfy the given predicate p. The order of the elements is preserved.

Definition Classes TraversableLike → GenTraversableLike

```
List(1,2,3,4).filter(x => x % 2 == 0)
```

Renders:

```
List(2,4)
```

Link: https://www.scala-lang.org/api/current/scala/collection/immutable/List.html

`Set` and `filter`

- This is the API Signature of `filter` for `Set`
- Note that the parameterized type of the `Set` in this case is `[A]`

```
def filter(p: (A) ⇒ Boolean): Set[A]
```

Selects all elements of this traversable collection which satisfy a predicate.

p	the predicate used to test elements.
returns	a new traversable collection consisting of all elements of this traversable collection that satisfy the given predicate p. The order of the elements is preserved.

Definition Classes TraversableLike → GenTraversableLike

```
Set(1,2,3,4).filter(x => x % 2 == 0)
```

Renders:

```
Set(2,4)
```

Link: https://www.scala-lang.org/api/current/scala/collection/immutable/Set.html

`Map` and `filter`

- This is the API Signature of `filter` for `Map`
- Note that the parameterized type of the `Map` in this case is `[K,V]`
  - `K` is the key parameterized type
  - `V` is the value parameterized type

```
def filter(p: ((K, V)) ⇒ Boolean): Map[K, V]
```

Selects all elements of this traversable collection which satisfy a predicate.

p	the predicate used to test elements.
returns	a new traversable collection consisting of all elements of this traversable collection that satisfy the given predicate p. The order of the elements is preserved.

Definition Classes TraversableLike → GenTraversableLike

```
Set(1,2,3,4).filter(x => x % 2 == 0)
```

Renders:

```
Set(2,4)
```

Link: https://www.scala-lang.org/api/current/scala/collection/immutable/Map.html

`String` and `filter`

- This is the API Signature of `filter` for `StringOps`
- Note that the parameterized type of the `StringOps` in this case is actually `Char`

```
def filter(p: (Char) ⇒ Boolean): String
```

Selects all elements of this traversable collection which satisfy a predicate.

p	the predicate used to test elements.
returns	a new traversable collection consisting of all elements of this traversable collection that satisfy the given predicate p. The order of the elements is preserved.

Definition Classes TraversableLike → GenTraversableLike

```
"Hello".filter(c => List('a', 'e', 'i', 'o', 'u').contains(c))
```

Renders:

```
"eo"
```

foreach

About foreach

- Like map, we apply a function to each element
- Unlike map
  - The return type for `foreach` is `Unit`
  - The function applied must also return a `Unit`
- `Unit` is an indication of a side effect

Using map instead of foreach

If we ran the following:

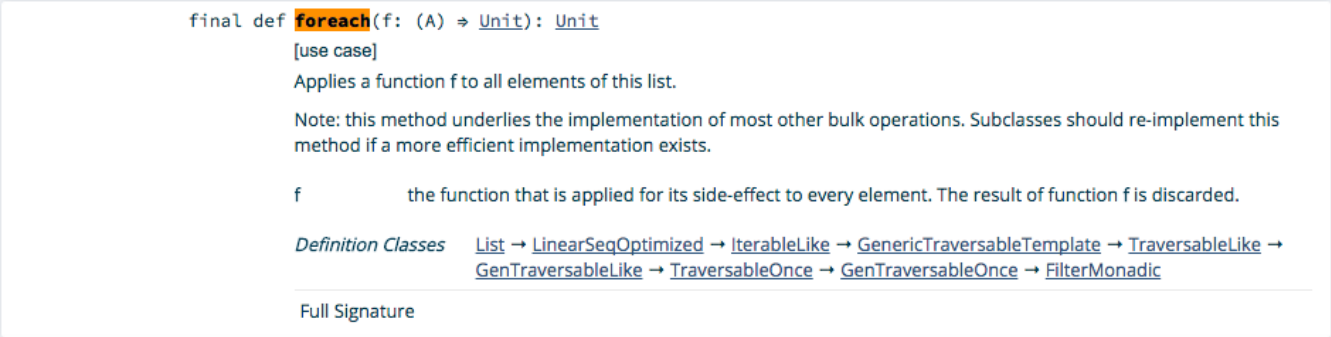
```
val a = (1 to 10)
val list = a.map(x => println(x))
println(list)
```

This would print each number but it will return a `Vector` of `()`:

```
1
2
3
4
5
6
7
8
9
10
Vector((), (), (), (), (), (), (), (), (), ())
```

Cleaner output with foreach

- Use `foreach` so that we don't have to see that messy end result
- Each item is given to the function returns a `Unit` (not mandatory)



Change `map` to `foreach`

- Changing the previous slide contents this will look like:

```
val a = (1 to 10)
val list = a.foreach(x => println(x))
println(list)
```

There we get the response we were looking for and the return is a `Unit`

```
1
2
3
4
5
6
7
8
9
10
()
```

Refactoring `foreach`

- Clearing out the assignment we just have:

```
val a = (1 to 10)
a.foreach(x => println(x))
```

- Let's just refactor `foreach` further using _ placeholder

```
val a = (1 to 10)
a.foreach(println _) //println is a perfect candidate, it returns Unit.
```

- The place holder _ is the last element within the parenthesis

- We can remove the placeholder

```
val a = (1 to 10)
a.foreach(println) //println is a perfect candidate, it returns Unit.
```

Final Refactoring with `foreach`

- Since `foreach` takes one argument we can call it as infix:

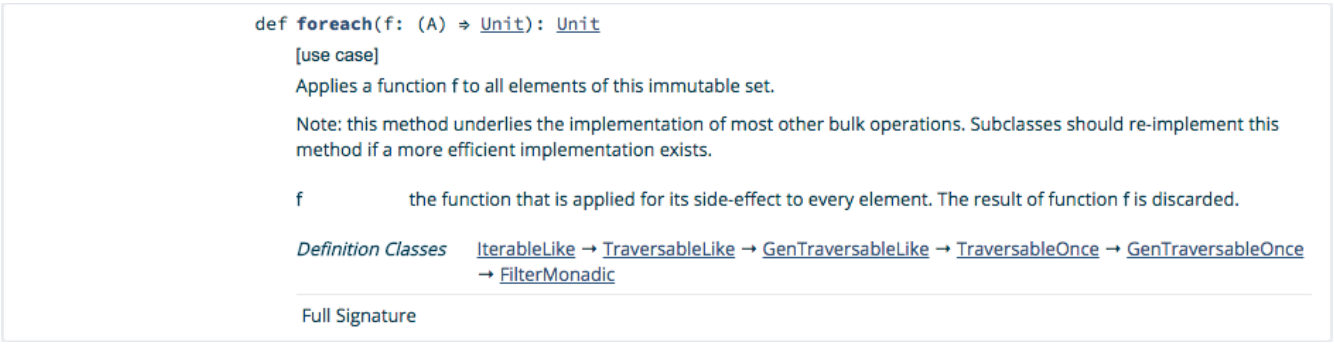
```
val a = 1 to 10
a foreach println
```

- Inlining the whole thing:

```
1 to 10 foreach println
```

`Set` and `foreach`

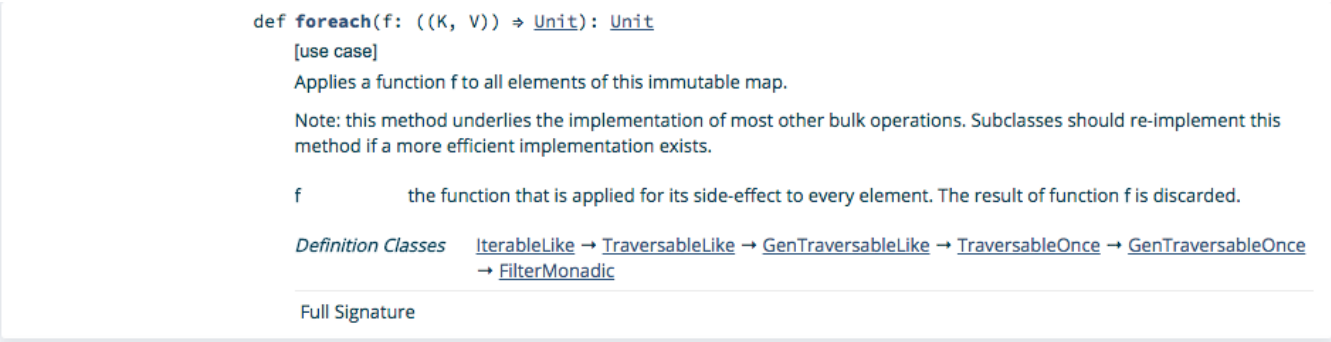
- This is the API Signature of `foreach` for `Set`
- Note that the parameterized type of the `Set` in this case is `[A]`



Link: http://www.scala-lang.org/api/current/scala/collection/immutable/Set.html

`Map` and `foreach`

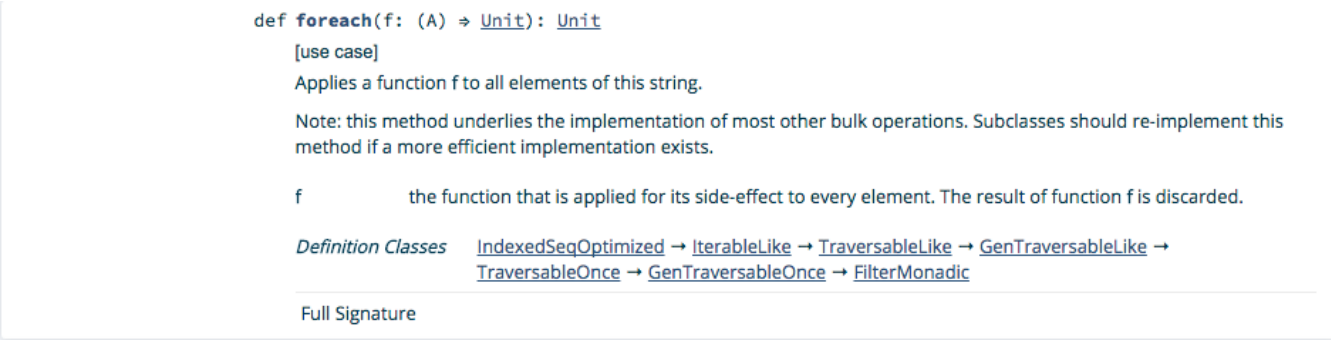
- This is the API Signature of `foreach` for `Map`
- Note that the parameterized type of the `Map` in this case is `[K, V]`
  - `K` is the key parameterized type
  - `V` is the value parameterized type



Link: http://www.scala-lang.org/api/current/scala/collection/immutable/Map.html

`String` and `foreach`

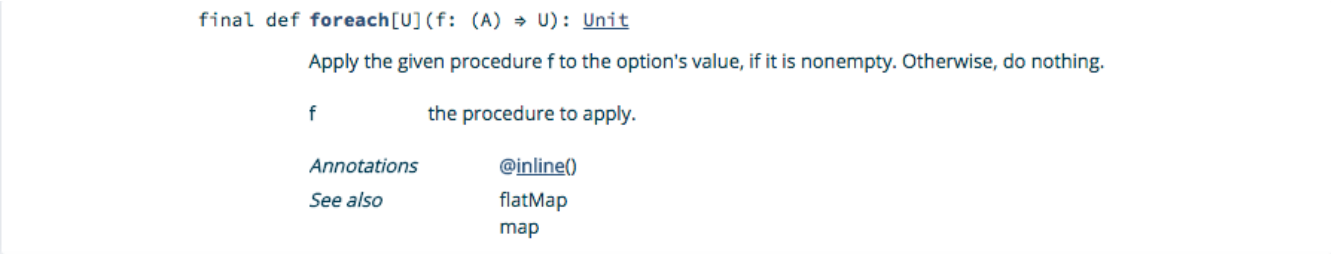
- This is the API Signature of `foreach` for `String`
- Note that the parameterized type of the `StringOps` in this case is `[A]` but is usually `Char`



Link: http://www.scala-lang.org/api/current/scala/collection/immutable/StringOps.html

`Option` and `foreach`

- This is the API Signature of `foreach` for `Option`
- Note that the parameterized type of the `Set` in this case is `[A]`



Link: http://www.scala-lang.org/api/current/scala/Option.html

Conclusion

- `foreach` is a method that takes a higher order `function` that will take an element and return `Unit`
- Perfect if you want to take an element and perform a side effect like print to screen

- `foreach` can be done to `List`, `Set`, `Stream`, `String`, `Array`, and more

`flatMap`

Using `flatMap`

- One of the most important functions/methods in functional programming
- Takes the value or values of one "container" and creates another "container" using the value.
- This is also important for *for comprehensions*

Starting from `map`

Let's say that we wish to `map` every element of a `List` into another `List`, seen below:

```
val a = List(1,2,3,4,5)
println(a.map(x => List(-x, 0, x)))
```

This is just a plain `map` that returns a `List[List[Int]]`

```
List(List(-1, 0, 1), List(-2, 0, 2), List(-3, 0, 3), List(-4, 0, 4),
List(-5, 0, 5))
```

Avoiding the `List[List[_]]` with `flatten` and `map`

But let's say that given all that we want to take that and `flatten` all that, this is where the `flatten` method is used.

```
val a = List(1,2,3,4,5)
println(a.map(x => List(-x, 0, x)).flatten)
```

That works out great, we see that we not longer have a list of list, we have a single list.

```
List(-1, 0, 1, -2, 0, 2, -3, 0, 3, -4, 0, 4, -5, 0, 5)
```

`flatten` is available in nearly all "containers", `List`, `Set`, `Map`, `Option`, `String`, `Stream`, etc.

Avoiding the `List[List[_]]` with `flatMap`

Here is a law that you may find helpful:

If you see a Collection in a Collection like we saw and you don't want it like that, use `flatMap`.

```
val a = List(1,2,3,4,5)
println(a.flatMap(x => List(-x, 0, x)))
```

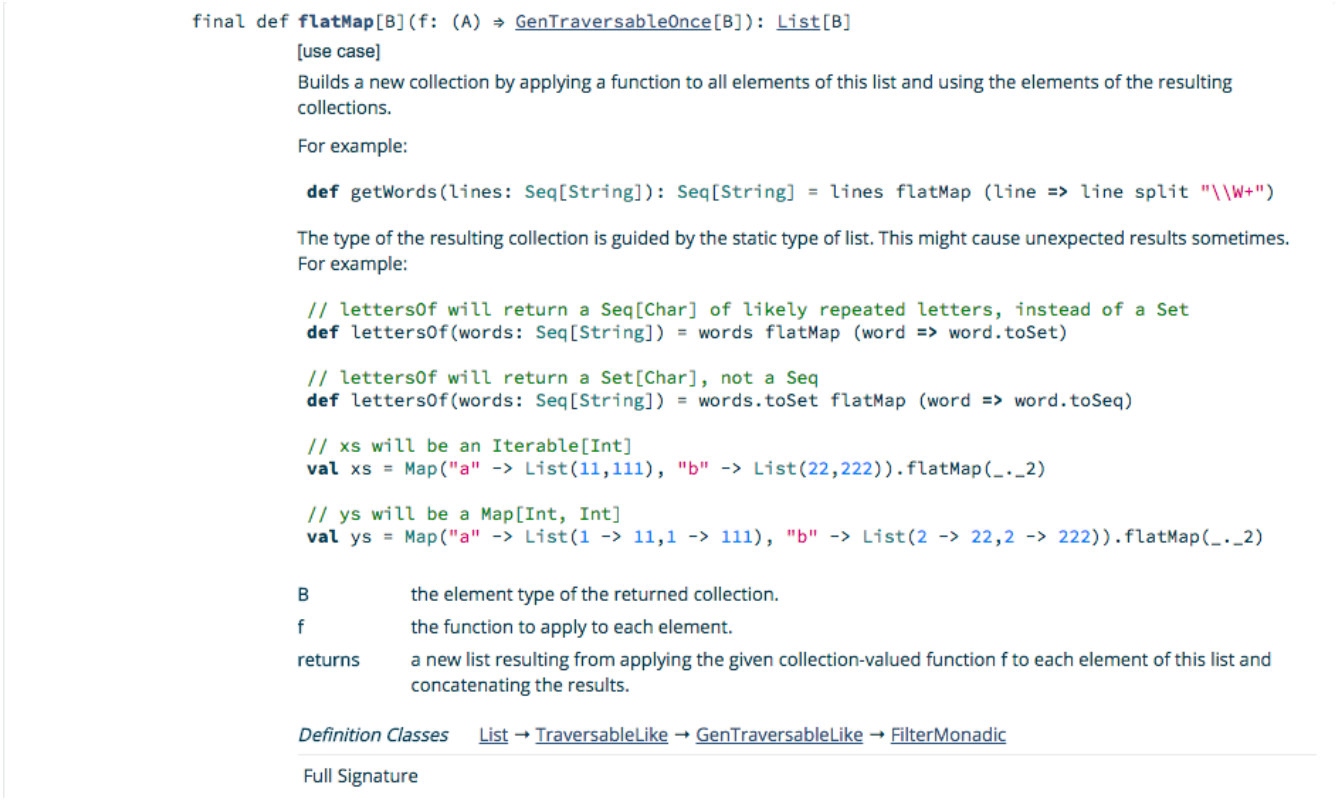
```
List(-1, 0, 1, -2, 0, 2, -3, 0, 3, -4, 0, 4, -5, 0, 5)
```

The previous result looks exactly the same as the `flatten` and `map` combination

`List` and `flatMap`

- Next slide is the API Signature of `flatMap` for `List`
- Note that the parameterized type of the `List` in this case is `[A]`
- Notice the signature of `flatMap`: `(A) ⇒ GenTraversableOnce[B]`
- Other `GenTraversableOnce[T]` subtypes include:
  - `Set`
  - `Map`
  - `Array`
  - `String`

`List` and `flatMap` API



```
final def flatMap[B](f: (A) ⇒ GenTraversableOnce[B]): List[B]
```

[use case]

Builds a new collection by applying a function to all elements of this list and using the elements of the resulting collections.

For example:

```
def getWords(lines: Seq[String]): Seq[String] = lines flatMap (line => line split "\\W+")
```

The type of the resulting collection is guided by the static type of list. This might cause unexpected results sometimes. For example:

```
// lettersOf will return a Seq[Char] of likely repeated letters, instead of a Set
def lettersOf(words: Seq[String]) = words flatMap (word => word.toSet)

// lettersOf will return a Set[Char], not a Seq
def lettersOf(words: Seq[String]) = words.toSet flatMap (word => word.toSeq)

// xs will be an Iterable[Int]
val xs = Map("a" -> List(11,111), "b" -> List(22,222)).flatMap(_._2)

// ys will be a Map[Int, Int]
val ys = Map("a" -> List(1 -> 11,1 -> 111), "b" -> List(2 -> 22,2 -> 222)).flatMap(_._2)
```

B	the element type of the returned collection.
f	the function to apply to each element.
returns	a new list resulting from applying the given collection-valued function f to each element of this list and concatenating the results.

Definition Classes List → TraversableLike → GenTraversableLike → FilterMonadic

Full Signature

`flatMap` with multiple layers

- Given the known signature of `flatMap`
- Even if the Collection are stacked in multiple layers we can `flatMap` until we get the collection we need.

```
val b:List[List[List[Int]]] =
   List(List(List(1,2,3), List(4,5,6)),
   List(List(7,8,9), List(10,11,12)))
```

- So what is the result of performing a `flatMap` on a `List[List[List[Int]]]`?
- Depends on the function

The Identity Function

- The identity function is take the input and make it the output
- Instead of writing `x ⇒ x`, you can opt for `identity(x)`
- `identity` comes from the `Predef`

The identity function and `flatMap`:

```
val list:List[List[List[Int]]] =
    List(List(List(1,2,3), List(4,5,6)),
    List(List(7,8,9), List(10,11,12)))
list.flatMap(x => x)
```

Renders:

```
List(List(1, 2, 3), List(4, 5, 6),
     List(7, 8, 9), List(10, 11, 12))
```

`Set` and `flatMap`

- This is the API Signature of `flatMap` for `Set`
- Note that the parameterized type of the `Set` in this case is `[A]`

```
def flatMap[B](f: (A) ⇒ GenTraversableOnce[B]): Set[B]
```

[use case]

Builds a new collection by applying a function to all elements of this set and using the elements of the resulting collections.

For example:

```
def getWords(lines: Seq[String]): Seq[String] = lines flatMap (line => line split "\\W+")
```

The type of the resulting collection is guided by the static type of set. This might cause unexpected results sometimes. For example:

```
// lettersOf will return a Seq[Char] of likely repeated letters, instead of a Set
def lettersOf(words: Seq[String]) = words flatMap (word => word.toSet)

// lettersOf will return a Set[Char], not a Seq
def lettersOf(words: Seq[String]) = words.toSet flatMap (word => word.toSeq)

// xs will be an Iterable[Int]
val xs = Map("a" -> List(11,111), "b" -> List(22,222)).flatMap(_._2)

// ys will be a Map[Int, Int]
val ys = Map("a" -> List(1 -> 11,1 -> 111), "b" -> List(2 -> 22,2 -> 222)).flatMap(_._2)
```

B	the element type of the returned collection.
f	the function to apply to each element.
returns	a new set resulting from applying the given collection-valued function f to each element of this set and concatenating the results.

Definition Classes TraversableLike → GenTraversableLike → FilterMonadic

Link: https://www.scala-lang.org/api/current/scala/collection/Set.html

Using `Set` and `flatMap`

```
Set(2, 4, 10, 11).flatMap(x => Set(x, x*5))
```

Will render

```
Set(10, 20, 2, 50, 11, 55, 4)
```

`Map` and `flatMap`

- This is the API Signature of `flatMap` for `Map`
- Note that the parameterized type of the `Map` in this case is `[K, V]`
  - `K` is the key parameterized type
  - `V` is the value parameterized type

```
def flatMap[B](f: (A) ⇒ GenTraversableOnce[B]): Map[B]
```

[use case]

Builds a new collection by applying a function to all elements of this map and using the elements of the resulting collections.

For example:

```
def getWords(lines: Seq[String]): Seq[String] = lines flatMap (line => line split "\\W+")
```

The type of the resulting collection is guided by the static type of map. This might cause unexpected results sometimes. For example:

```
// lettersOf will return a Seq[Char] of likely repeated letters, instead of a Set
def lettersOf(words: Seq[String]) = words flatMap (word => word.toSet)

// lettersOf will return a Set[Char], not a Seq
def lettersOf(words: Seq[String]) = words.toSet flatMap (word => word.toSeq)

// xs will be an Iterable[Int]
val xs = Map("a" -> List(11,111), "b" -> List(22,222)).flatMap(_._2)

// ys will be a Map[Int, Int]
val ys = Map("a" -> List(1 -> 11,1 -> 111), "b" -> List(2 -> 22,2 -> 222)).flatMap(_._2)
```

B	the element type of the returned collection.
f	the function to apply to each element.
returns	a new map resulting from applying the given collection-valued function f to each element of this map and concatenating the results.

Definition Classes TraversableLike → GenTraversableLike → FilterMonadic

Full Signature

Link: https://www.scala-lang.org/api/current/scala/collection/Map.html

Using `Map` and `flatMap`

```
val origMap = Map(1 -> "One",
  2 -> "Two",
  3 -> "Three")

val result:Map[Int, String] = ???
```

```
Map(1 -> "One", 2 -> "Two", 3 -> "Three",
    300 -> "Three Hundred", 200 -> "Two Hundred",
    100 -> "One Hundred")
```

Conclusion

- `flatMap` is the combination of `flatten` and `map`.
- `flatMap` can be used with `List`, `Set`, `Maps`, `String`, `Stream`, and `Option`
- Your cue is when you see a `List[List[A]]`, `Set[Set[A]]`, etc.
- `flatMap` can also be used with massive layering, `List[List[List[A]]]`

mkString

- Concatenates all elements into one string given a delimiter
- Can be given a before and after `String` to flank the result string

mkString By Example

```
List(1,2,3,4).mkString(":") //1:2:3:4

Vector('z','a','b','c').mkString("~~") //z~~a~~b~~c

Set("Seattle", "Los Angeles", "Denver").mkString("$$", "#", "$$")
   //$$Seattle#Los Angeles#Denver$$
```

Under four elements, the order of a `Set` is maintained

Lab: Doing a better `isPrime`

Step 1: In the REPL or http://scastie.scala-lang.org

Step 2: Knowing what you know now, is there a better `isPrime`? If so implement it!

Hint: Start with a range like (`1 to n`) and work from there. Look at the range API, specifically at the method `forall` and see if that will work! You can also try something else, lots of methods to choose from! There should be no for loops!

Solution: https://bit.ly/2uk6rnH

Thank You

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Twitter: http://twitter.com/dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext