Beginning Scala

Daniel Hinojosa

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
- 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

Setup

Pre-Class Check

Before we begin it is assumed that all of you have the following tools installed:

- JDK 1.8 (latest java is 1.8.0_144)
- Scala 2.12.4
- SBT 1.0.2

To verify that all your tools work as expected

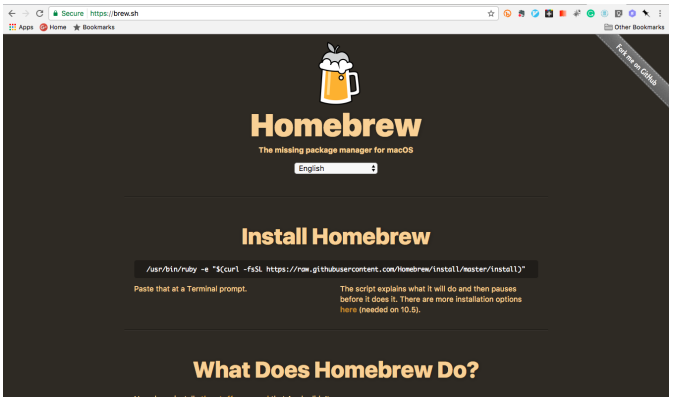
```
% javac -version
javac 1.8.0_144

% java -version
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_1.8.0_144-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)

% scala -version
Scala code runner version 2.12.4 -- Copyright 2002-2018, LAMP/EPFL

% sbt sbtVersion
[info] Set current project to scala (in build file:/<folder_location>)
[info] 1.0.2
```

Installing Java, Scala, and SBT on a Mac Automatically with Brew



If you have brew installed, you can run the following ***and be done***:

```
% brew update
% brew cask install java
% brew install scala
% brew install sbt
```

This will require an install of Homebrew. Visit https://brew.sh/ for details of installation if you want to use brew.

Depending on your company's software and security constraints, you may not be able to use brew

If you don't have Java 8 installed

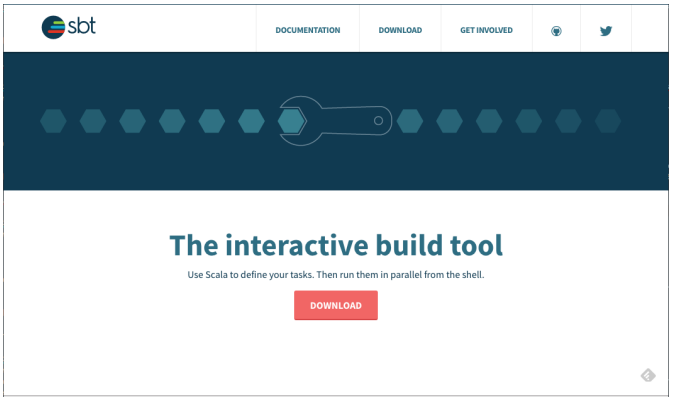
- Visit: http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html
- Select: *Accept License Agreement*
- Download the appropriate Java version based on your architecture.

Linux ARM 32 Hard Float ABI	Linux ARM 64 Hard Float ABI
Linux x86	Linux x86
Linux x64	Linux x64
Mac OS X	Solaris SPARC 64-bit
Solaris SPARC 64-bit	Solaris x64
Solaris x64	Windows x86

If you do not have Scala installed

- Visit http://scala-lang.org
- Click the *Download* Button
- Download the appropriate binary for your system:
  - Mac and Linux will load a *.tgz* file
  - Windows will download an *.msi* executable
- For Mac and Linux you can expand with `tar -xvf scala-2.12.4.tgz`

If you do not have SBT installed

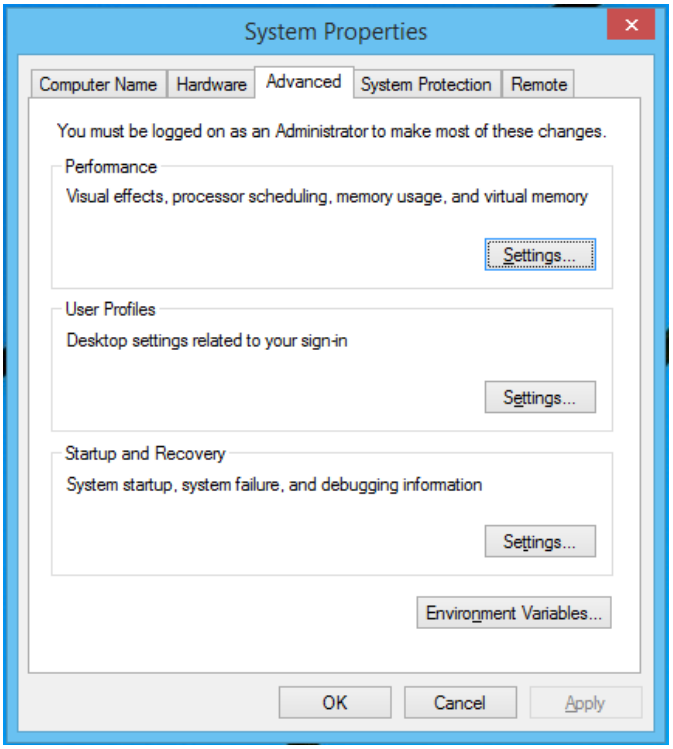


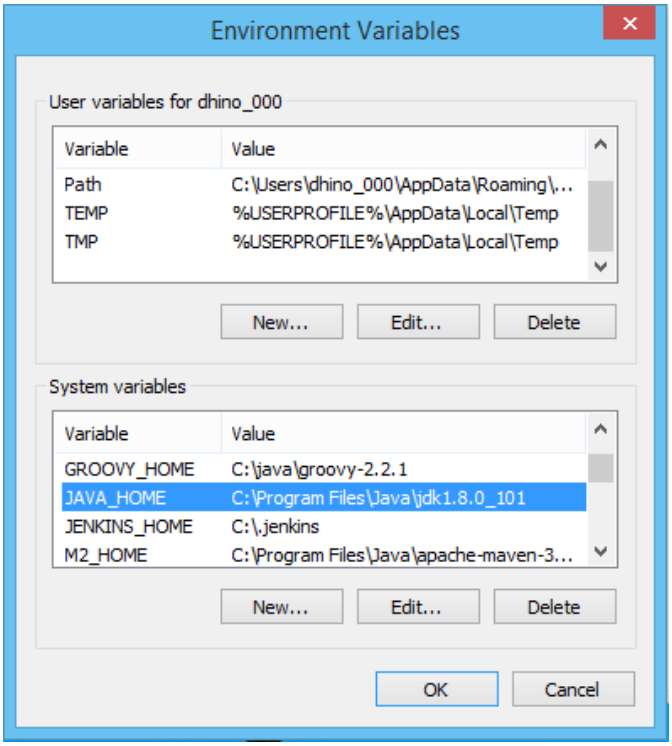
- Visit http://scala-sbt.org
- Click the *Download* Button
- Download the appropriate binary for your system:
  - Mac and Linux will load a *.tgz*, or a *.zip* file
  - Windows will download an *.msi* executable
- For Mac and Linux you can expand with `tar -xvf sbt-1.0.2.tgz`

SBT is not a requirement for Scala, it is just the most used build tool

Windows Users Only: Setting up the Windows Environment Variables for Java

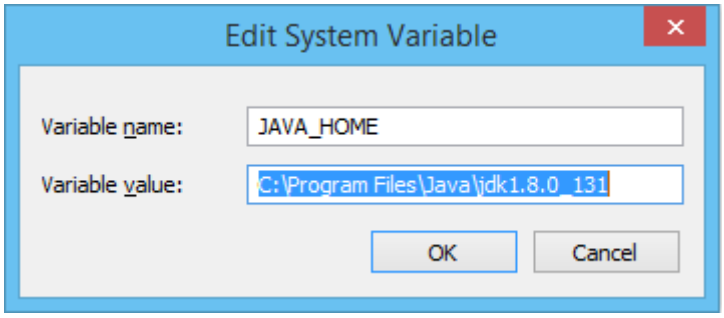
- Go to your *Environment Variables*, typically done by typing the Windows key(⊞) and type `env`





Windows Users Only: Setting up `JAVA_HOME`

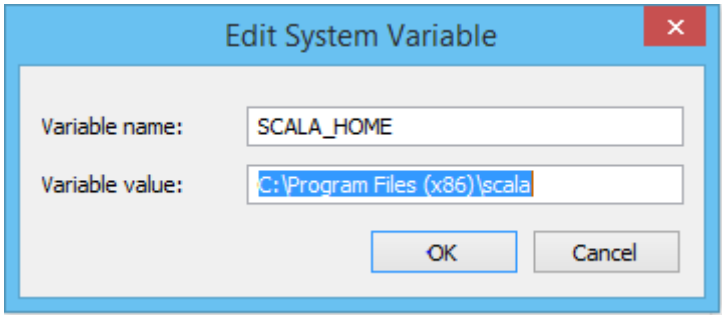
- Edit `JAVA_HOME` in the System Environment Variable window with the location of your JDK



Using `jdk1.8.0_131` in the image. Your version may vary.

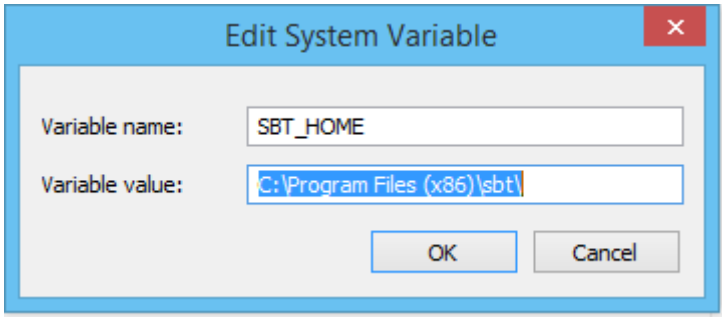
Windows Users Only: Setting up `SCALA_HOME`

- This setting is not necessary with Scala on Windows since the .msi file installs everything required
- If you do have problems where a tool is unable to locate Scala, set up an environment variable `SCALA_HOME`



Windows Users Only: Setting up SBT_HOME

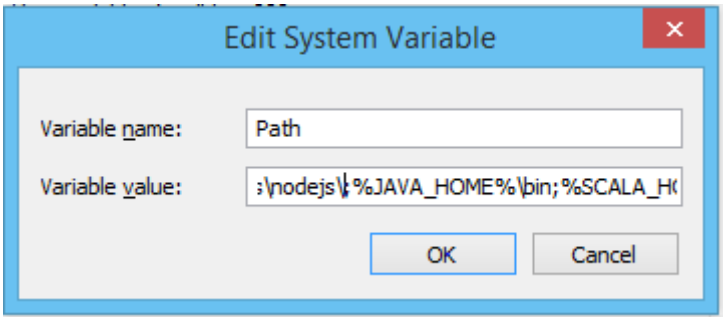
- This setting is not necessary since SBT on Windows since the .msi file installs everything required
- If you do have problems where a tool is unable to locate SBT, set up an environment variable `SBT_HOME`



Windows Users Only: Setting up PATH

- Once you establish `JAVA_HOME`, and possibly `SCALA_HOME` and `SBT_HOME`, ***append*** to the `PATH` setting the following:

```
;%JAVA_HOME%\bin;%SBT_HOME%\bin;%SCALA_HOME%\bin
```



Windows Users Only: Restart All Command Prompts And Try Again

```
% javac -version
javac 1.8.0_144

% java -version
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_{latest_java}-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)

% scala -version
Scala code runner version 2.12.4 -- Copyright 2002-2018, LAMP/EPFL

% sbt sbtVersion
[info] Set current project to scala (in build file:/<folder_location>)
[info] 1.0.2
```

Changes won't take effect until you open a new command prompt!

Mac Users Only: Editing your *.bash_profile* or *.zshrc*

- If you are using the Bash shell, edit the your *.bash_profile* in your home directory using your favorite editor
- If you are using the Zsh shell, edit the your *.zshrc* in your home directory using your favorite editor

For example, if using `nano`

```
% nano ~/.bash_profile
```

Replace *nano* with your favorite editor *vim*, *emacs*, *atom*, etc.

Make sure the following contents are in your *.bash_profile*

```
export SCALA_HOME= <location_of_scala>
export SBT_HOME= <location_of_sbt>
export JAVA_HOME=$(/usr/libexec/java_home)
export PATH=$PATH:$JAVA_HOME/bin:$SCALA_HOME/bin:$SBT_HOME/bin
```

If you used `brew`, many of these application will not require their `PATH` setup.

You can locate where `scala` and `sbt` is by either doing

```
% which scala
% which sbt
% whereis scala
% whereis sbt
```

When done open a new terminal or if already on an open terminal type:

- For bash: `source .bash_profile`
- For zsh: `source .zshrc`

Linux Users Only: Editing your *.bash_profile* or *.zshrc*

- If you are using the Bash shell, edit the your *.bash_profile* in your home directory using your favorite editor
- If you are using the Zsh shell, edit the your *.zshrc* in your home directory using your favorite editor

For example, if using `nano`

```
% nano ~/.bash_profile
```

Replace *nano* with your favorite editor *vim*, *emacs*, *atom*, etc.

Make sure the following contents are in your *.bash_profile*

```
export SCALA_HOME= <location_of_scala>
export SBT_HOME= <location_of_sbt>
export JAVA_HOME= <location_of_jdk>
export PATH=$PATH:$JAVA_HOME/bin:$SCALA_HOME/bin:$SBT_HOME/bin
```

Mac and Linux Users Only: `source` your *.bash_profile* or *.zshrc*

When done open a new terminal or if already on an open terminal type:

- For bash: `source .bash_profile`
- For zsh: `source .zshrc`

Verify the results on the command line

```
% javac -version
javac 1.8.0_144

% java -version
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_{latest_java}-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)

% scala -version
Scala code runner version 2.12.4 -- Copyright 2002-2018, LAMP/EPFL

% sbt sbtVersion
[info] Set current project to scala (in build file:/<folder_location>)
[info] 1.0.2
```

Installing IDEs

- **IMPORTANT Be sure to download and configure the latest version of your IDE!**
- Eclipse - Be sure to have Neon or Oxygen
- IntelliJ IDEA (Professional or Community): 2017-3, 2017-2, 2017-1
- **IMPORTANT Be sure to backup any IDE settings that you believe are critical**

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
- 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

Lab: REPL

Let's run the REPL!

Step 1: Run the REPL using `scala` on the command line

Step 2: Try `help` within the REPL

```
scala> :help

All commands can be abbreviated, e.g., :he instead of :help.
:edit <id>|<line>        edit history
:help [command]          print this summary or command-specific help
:history [num]           show the history (optional num is commands to
show)
:h? <string>             search the history
:imports [name name ...] show import history, identifying sources of
names
:implicits [-v]          show the implicits in scope

...
```

Step 3: Make some assignments in the REPL

```
scala> 40
res1: Int = 40

scala> res1 + 50
res2: Int = 90
```

Lab: REPL Copy from a website

Step 1: Run `:paste` in the REPL where it will await you to paste some code

```
scala> :paste
// Entering paste mode (ctrl-D to finish)
```

Step 2: Copy the following content from the following

```
abstract class A {
  val message: String
}
class B extends A {
  val message = "I'm an instance of class B"
}
trait C extends A {
  def loudMessage = message.toUpperCase()
}
class D extends B with C
val d = new D
println(d.message)
println(d.loudMessage)
```

Source: http://docs.scala-lang.org/tutorials/tour/mixin-class-composition.html

Step 3: Hit `CTRL+D`. Notice the messages

Step 4: Type `:quit` to quit the REPL

Creating a Script

Lab: Creating a Script

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

Step 1: Create a file called `ScalaScript.scala` anywhere in your system that you have access to with the following content

```
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

Step 2: Run the `ScalaScript.scala` with the following command in either your terminal (MacOSX, Linux) or command prompt (Windows)

```
% scala ScalaScript.scala
```

Lab: Find 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)

Step 1: Find the `CompileServer`/`MainGenericRunner` using the following in your MacOSX or Linux terminal:

```
% jps
```

Should return something like the following:

```
34626 Jps
34579 MainGenericRunner
```

Step 2: Run the script again with `-nc` to avoid the cached copy of the script (which may come in handy if you are doing a lot of scripting and you get unwanted results)

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

Creating an Application

About Applications in Scala

- Applications can be compiled just like Java
- The following is sufficient with what we need
- 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

Lab: Create an application

Step 1: Create a directory called scala_project

Step 2: Create a file called *Entities.scala* in scala_project

Step 3: Inside of *Entities.scala* enter the following information

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

The property name is first then the type

Step 4: Compile the file with `scalac`

```
% scalac Entities.scala
```

Step 5: View the `.class` files using `dir` on Windows or `ls` on Linux/MacOSX

```
% ls
```

Lab: Looking at the Java of the class files

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

Step 1: Inside of your Scala Project folder, use `javap -p` for `Employee` and `Department`. You do not need `.class`

```
% javap -p Employee
```

```
% javap -p Department
```

Lab: Organizing the code

- Having class files littered with source code is messy and dangerous

Step 1: Inside of your Scala Project folder, create a folder called `src/main/scala` the other called `target`.

On Linux/MacOSX:

```
% mkdir -p src/main/scala
```

On Windows:

```
% mkdir src\main\scala
```

Step 2: Delete all the `.class` files

Step 3: Create a `target` folder where we can store our compiled files

```
% mkdir target
```

Step 4: Use `tree` (Linux/MacOSX) or `tree /F` to verify your progress. If you don't have `tree` on MacOSX, you can either install it if you have Homebrew `brew install tree` or just use `find .` in the meantime

Step 5: Move the `Entities.scala` file into `src/main/scala`

Step 6: At the root of your project run `scalac` with the following commands

```
% scalac -d target src/main/scala/*.scala
```

Step 6: Verify the results again using `tree` or `find .`

Lab: You cannot compile a script!

Step 1: Add `println("Hello, Scala")` to *Entities.scala* and

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

Step 2: Try to compile again with

```
% javac -d target src/*.scala
```

Step 3: Try to just run it as a script and you will see it runs.

Lab: If you want to compile *and* run create an `object`

Step 1: Put `println("Hello, Scala")` in an object `MyRunner` inside *Entities.scala* like the following:

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

You don't have to call it `MyRunner`, you can call it whatever you like

Step 2: Try to compile again with the following and enjoy success:

```
% javac -d target src/main/scala/*.scala
```

Step 3: You may now run the application with `scala` inside of the Scala Project folder

```
% scala -cp target 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")
}
```

Lab: Adding a package to our code

Step 1: Add a package inside of *Entities.scala*

```
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")
   }
}
```

Step 2 (Optional): Convert `MyRunner` to `extends App` and delete the main block

Step 3: Compile and run the code again to test that all is well, just note that when we run the

```
% scalac -d target src/main/scala/*.scala
```

Step 3: You may now run the application with `scala` inside of the Scala Project folder

```
% scala -cp target com.xyzcorp.MyRunner
```

Packages are rather more dynamic in Scala, more on that later too!

SBT

SBT Basics

- Simple Build Tool
- Probably the most popular of build tools in Scala
- Scala based
- Scala Projects don't need to use SBT there are alternatives
  - Maven
  - Gradle

Creating Projects Automatically with Seeds

- A new project can be used to start a project, *Giter 8*
- *Giter 8* uses Github to create new projects

For a list of standard templates: http://www.scala-sbt.org/0.13/docs/sbt-new-and-Templates.html

Lab: Create Projects Manually

- We can retrofit our project with a build file
- SBT Projects require a build file, one way is to create a *build.sbt*
- In our Scala Project folder, create a file called *build.sbt* with the following content
- It is important to leave a carriage return between each of the lines
- It requires by default (overrideable)
  - Production code to be in `src/main/scala`
  - Test code to be in `src/test/scala`

```
name := "Scala Project"

version := "1.0_SNAPSHOT"

scalaVersion := "2.12.4"

libraryDependencies ++= Seq(
  "org.scalactic" %% "scalactic" % "{latest-scalatest}" % "test",
  "org.scalatest" %% "scalatest" % "{latest-scalatest}" % "test")
```

SBT Common Commands:

`clean`	Deletes all generated files (in the target directory).
`compile`	Compiles the main sources (in `src/main/scala` and `src/main/java` directories).
`test`	Compiles and runs all tests.
`testOnly`	Compiles and runs one test, requires class name after the command.
`console`	Starts the Scala interpreter with a classpath including the compiled sources and all dependencies. To return to sbt, type `:quit`, `Ctrl+D` (Unix), or `Ctrl+Z` (Windows).
`run <argument>*`	Runs the main class for the project in the same virtual machine as sbt.
`package`	Creates a jar file containing the files in `src/main/resources` and the classes compiled from `src/main/scala` and `src/main/java.`
`help <command>`	Displays detailed help for the specified command. If no command is provided, displays brief descriptions of all commands.
`reload`	Reloads the build definition (`build.sbt`, `project/.scala`, `project/.sbt` files). Needed if you change the build definition.
`update`	Download any of the dependencies that are required by your project

Running SBT

SBT can be run from your command line and actions can be chained with a space

```
% sbt clean compile
```

Going into the SBT shell

- SBT has a shell that you can enter into to execute any build commands
- Just run `sbt` in your terminal to enter the SBT shell

```
% sbt
```

- Once in the shell you can execute the commands you need

```
> clean
```

- To chain commands, use a `;` as a delimiter, including one `;` in the front

```
> ;clean;compile;test
```

Lab: Play around with SBT

Let's try some items in SBT

Step 1: Go into the `sbt` shell

Step 2: Try `;clean;compile`

Step 3: Try `help`

Step 4: Try `;reload;update` to download all the latest dependencies

Step 5: Try `console` and in the REPL create an `Employee` a class you created :

```
new Employee("<YOUR_FIRST_NAME>", "<YOUR_LAST_NAME>")
```

Step 6: Type `:quit` to escape from the REPL

Step 7: Type `exit` to escape from SBT

Step 8: Type `sbt package` in your terminal shell to create a jar file of your project

Triggered Execution in SBT

- You can trigger any command in SBT by prepending that command with a `~` (no space after)
- Can only run once you are inside of the SBT shell
- Most often you will use them with:
  - `compile`
  - `test`
  - `testOnly`

For example:

```
> ~testOnly com.xyzcorp.MyTest
```

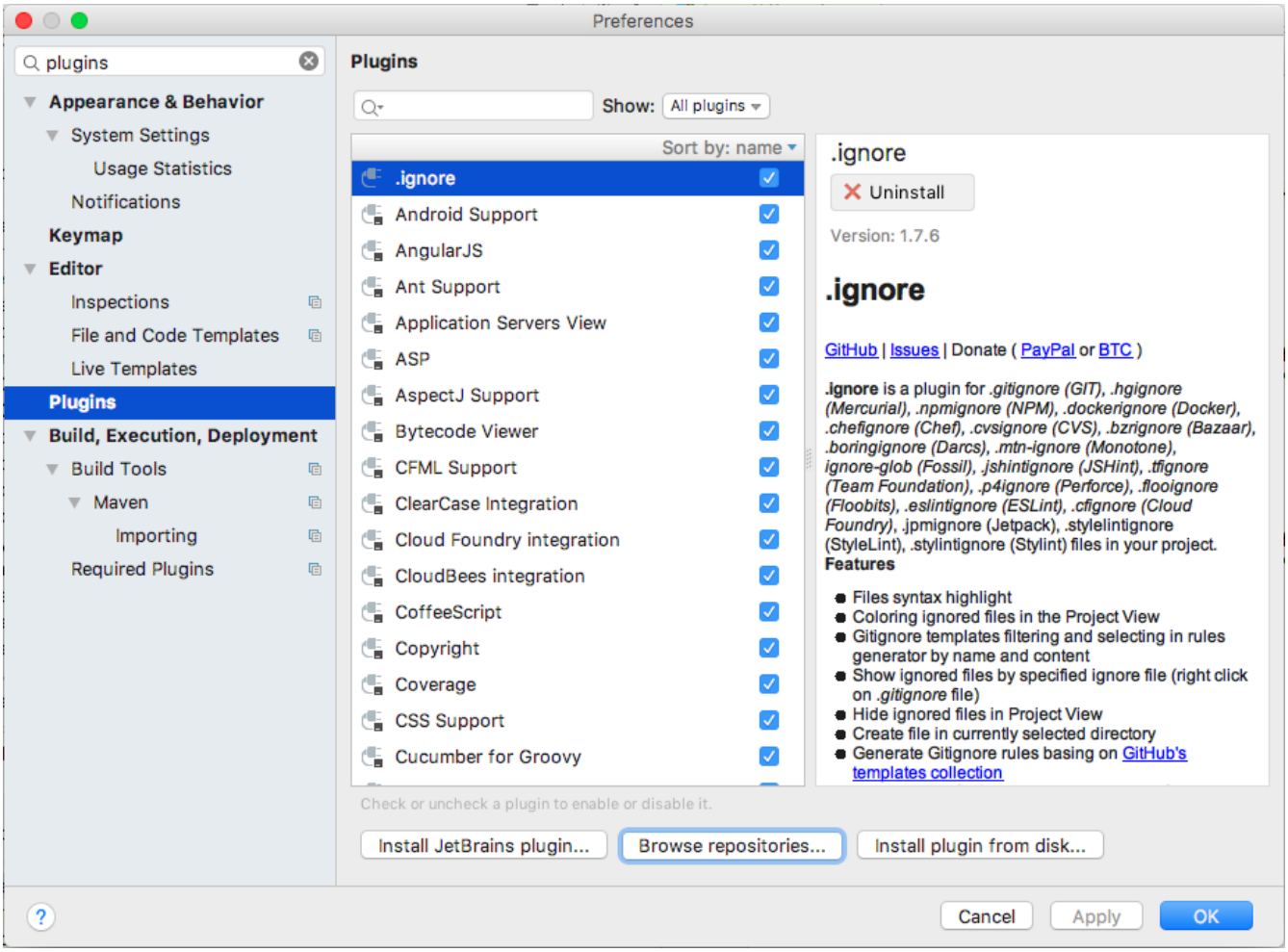
Scala & IntelliJ Idea

Lab (For IntelliJ Users): Install the IntelliJ Scala Plugin

IntelliJ from JetBrains offers support for Scala through one of its plugins

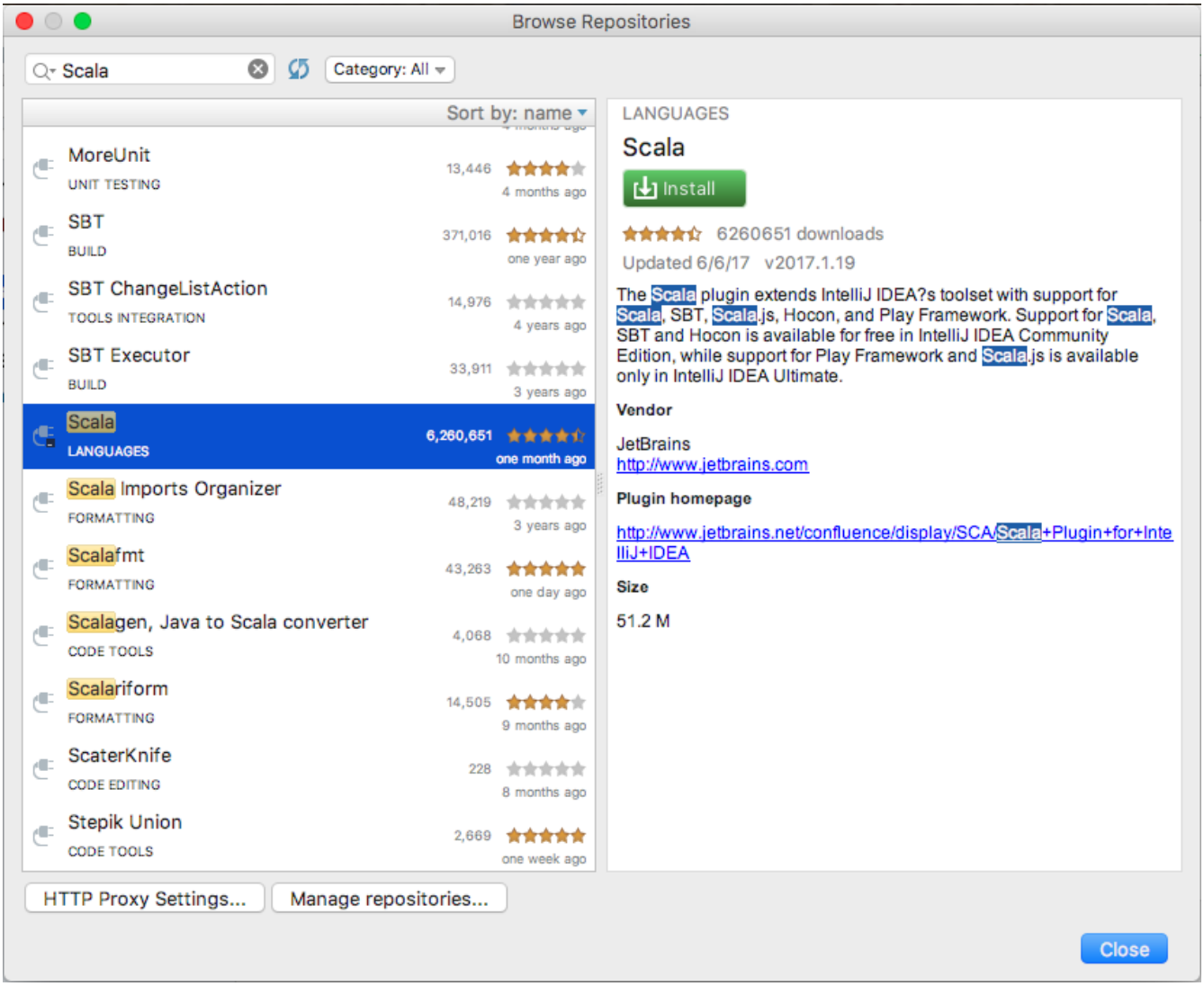
Step 1: Go to:

- For Windows and Linux: File | Settings | Plugins
- For MacOSX: IntelliJ IDEA | Preferences | Plugins
- Click on Browse Repositories



Lab (For IntelliJ Users): Install the IntelliJ Scala Plugin Continued

Step 2: Search for Scala in the filter on the upper-left hand corner



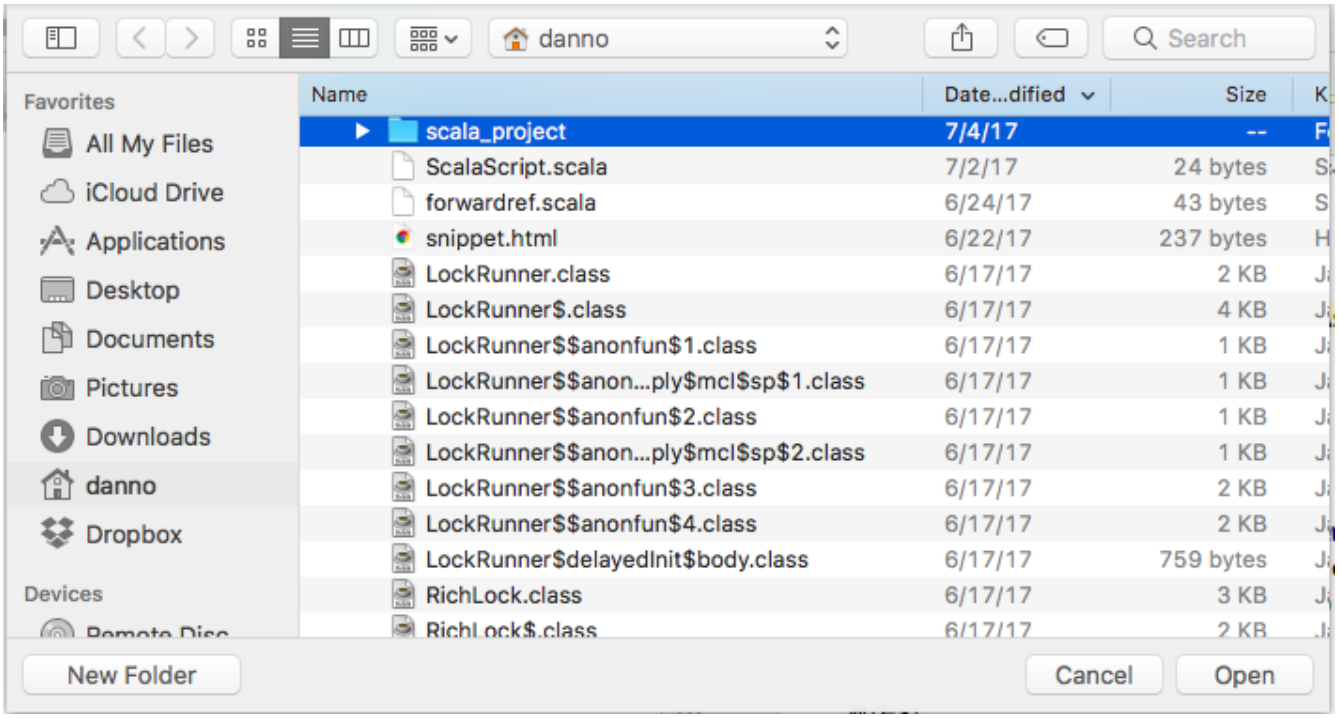
Step 3: Click *Install* on the right pane window

Step 4: Restart is necessary

Do not install the SBT plugin, the IntelliJ Plugin already has SBT support

Lab (For IntelliJ Users): Open your project in IntelliJ Continued

Step 1: Go to:



- File | Open
- Locate your project, :project_name:
- Click the Open button

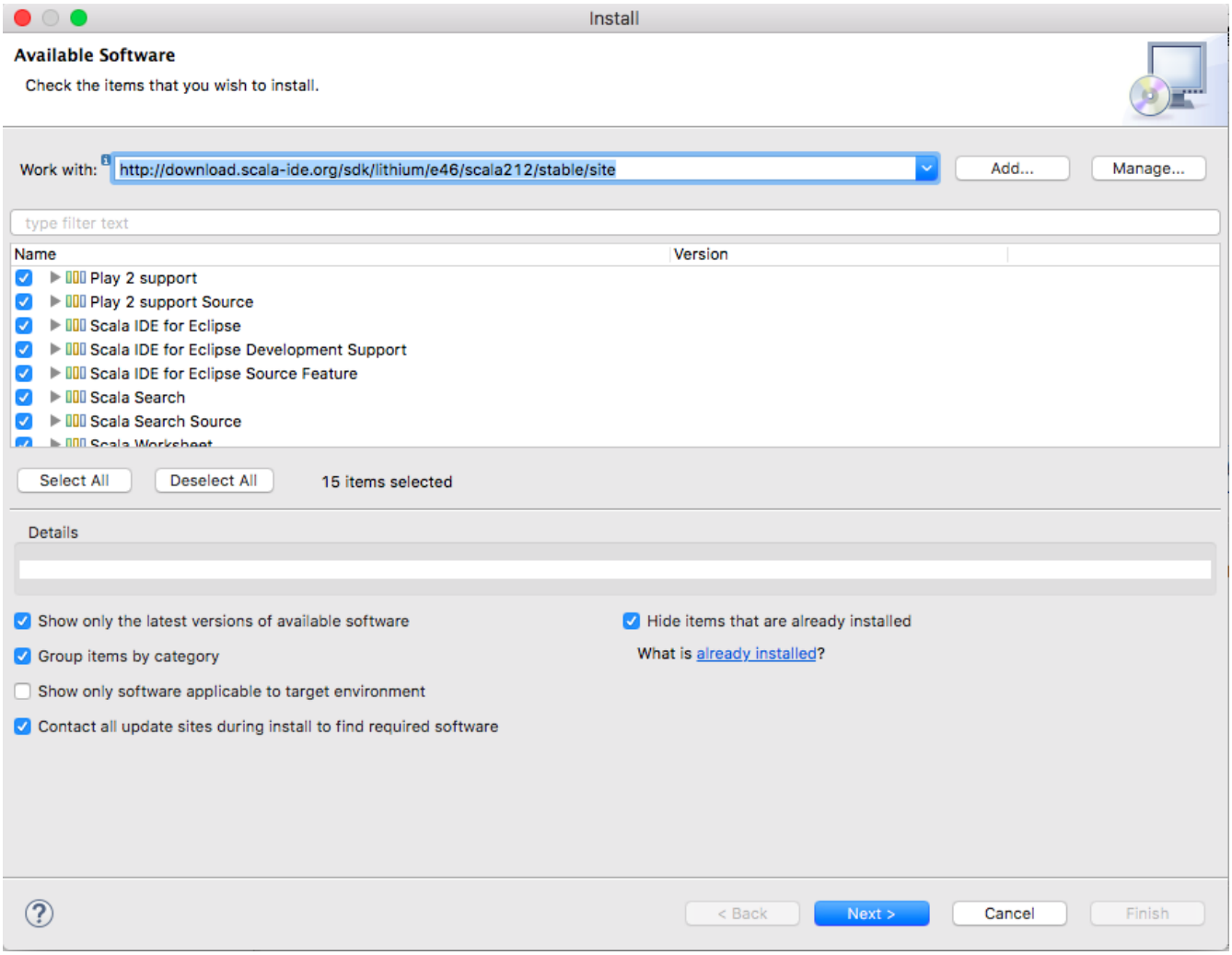
Scala & Eclipse

Eclipse Notes

- Be sure to have one of the latest Eclipse installation as plugins tend to not work with older plugins
- If you do not have any Eclipse you can just go straight to http://scala-ide.org and download a complete bundle and skip the Eclipse setup and jump to ***Installing the SBT Eclipse Plugin***

Optional Lab (Eclipse Users Only): Installing the Scala-IDE Plugin

If you wish to install the Scala-IDE plugin manually instead of downloading the complete Scala-IDE package...



Step 1: Go to Help | Install New Software

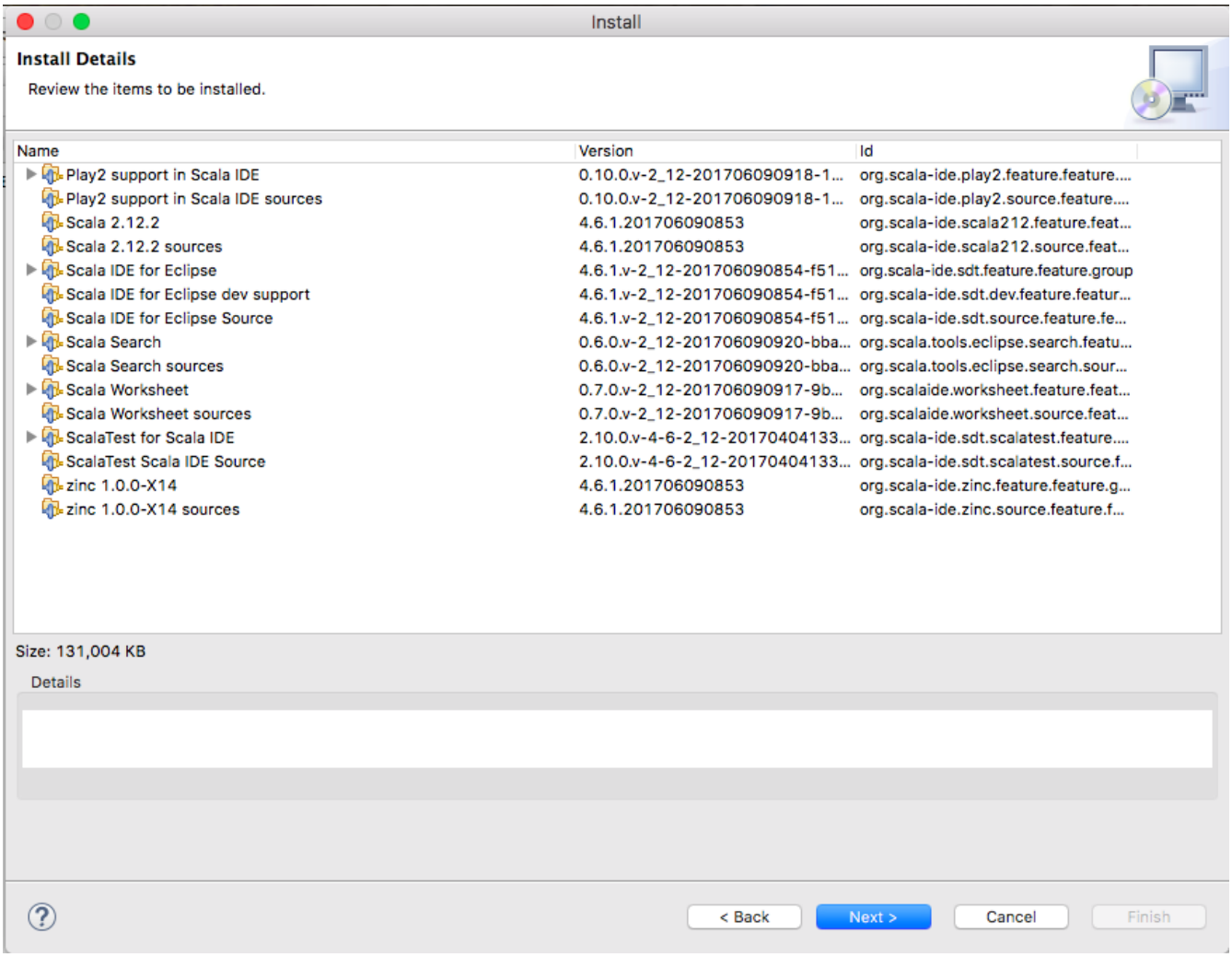
Step 2: In the *Work With:* field enter the URL of the Scala IDE Plugin found in http://scala-ide.org/download/current.html

Step 3: Hit Enter, and select all the Scala modules that appear

Step 4: Hit Next

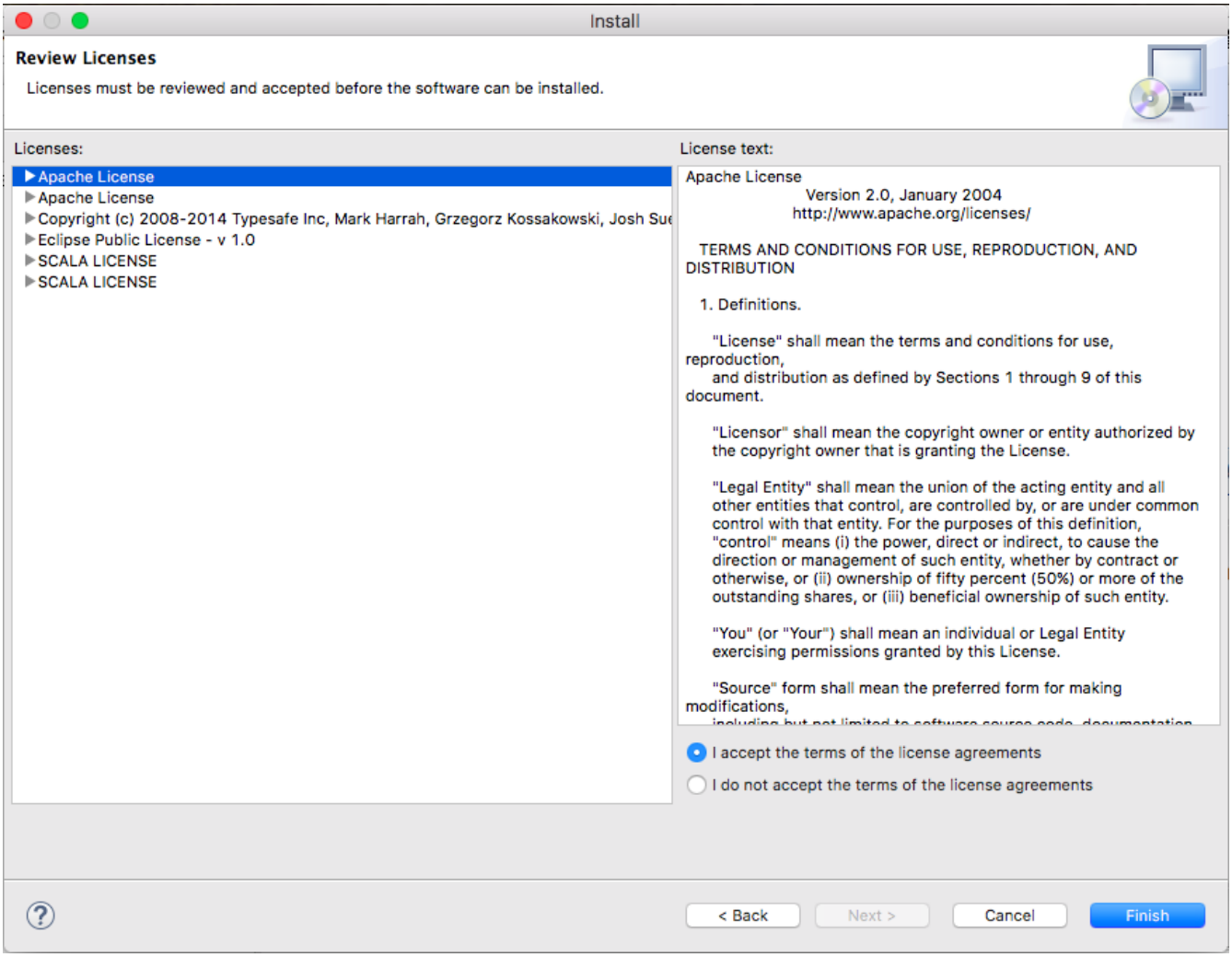
Optional Lab (Eclipse Users Only): Install Details

If you wish to install the Scala-IDE plugin manually instead of downloading the complete Scala-IDE package...



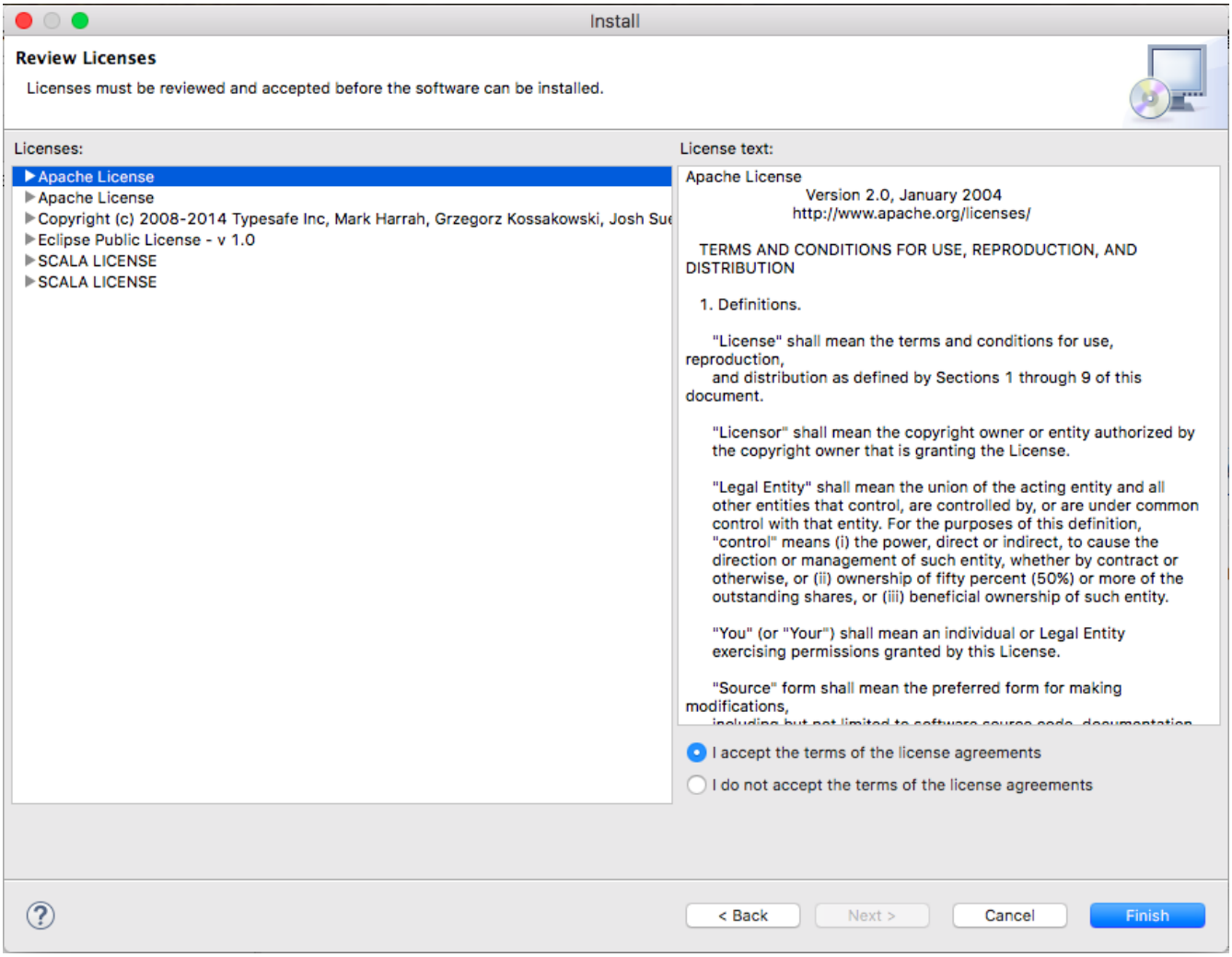
Step 1: Ensure the details for the plugin and hit *Next*

Optional Lab (Eclipse Users Only): License Details



Step 1: Ensure the details for the licenses used and hit *Next*

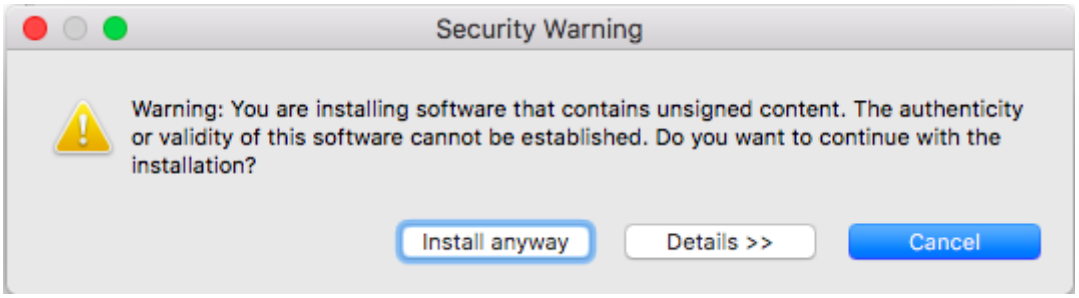
Optional Lab (Eclipse Users Only): Unsigned Content



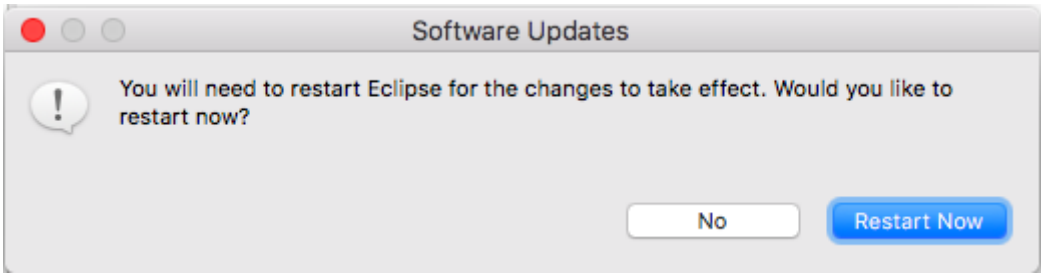
Step 1: Ensure the details for the licenses used and hit *Next*

Optional Lab (Eclipse Users Only): Last Details

Step 1: An image may appear warning about unsigned content, choose *Install Anyway*



Step 2: When prompted to Restart Machine, select *Restart Now*.



Lab (Eclipse Users Only): Installing the SBT Eclipse Plugin

Step 1: Create a file called *plugins.sbt* in the *project* folder located inside of the *scala_project* folder.

Step 2: In the *plugins.sbt* file, add the following text (in a single line):

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "5.2.4")
```

Step 3: If you are already in an SBT session, `quit` out of it

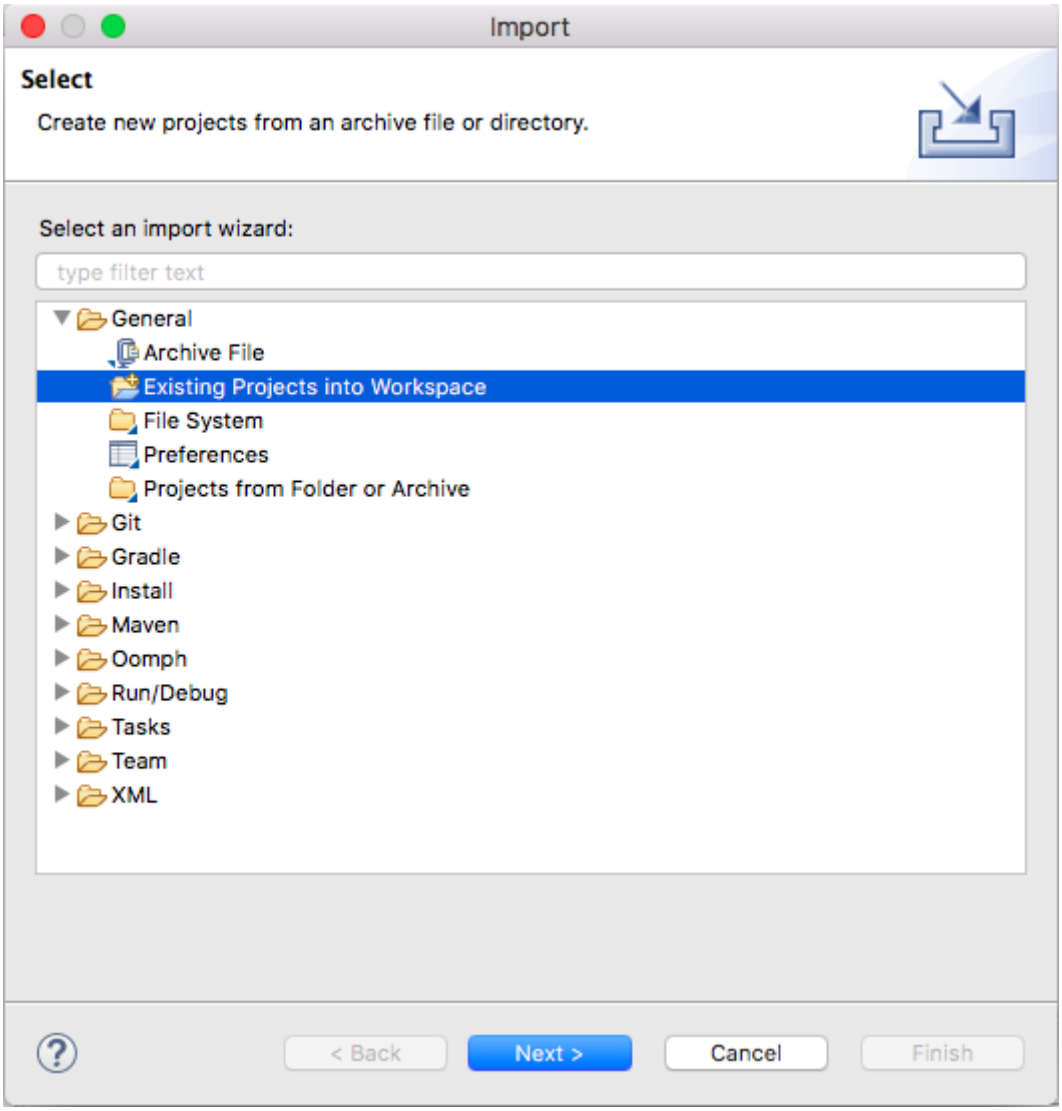
Step 4: Re-enter the `sbt` shell, and you should notice that the Eclipse plugin is installing (perhaps future versions of SBT may not display this)

Step 5: You should be able in the `sbt` shell to install eclipse in the shell (Hint: Please use tab completion). After you invoke this command in SBT, this will create all the eclipse files needed.

```
> eclipse with-source=true with-javadoc=true execution-
environment=JavaSE-1.8
```

Source: https://github.com/typesafehub/sbteclipse

Lab (Eclipse Users Only): Importing into Eclipse



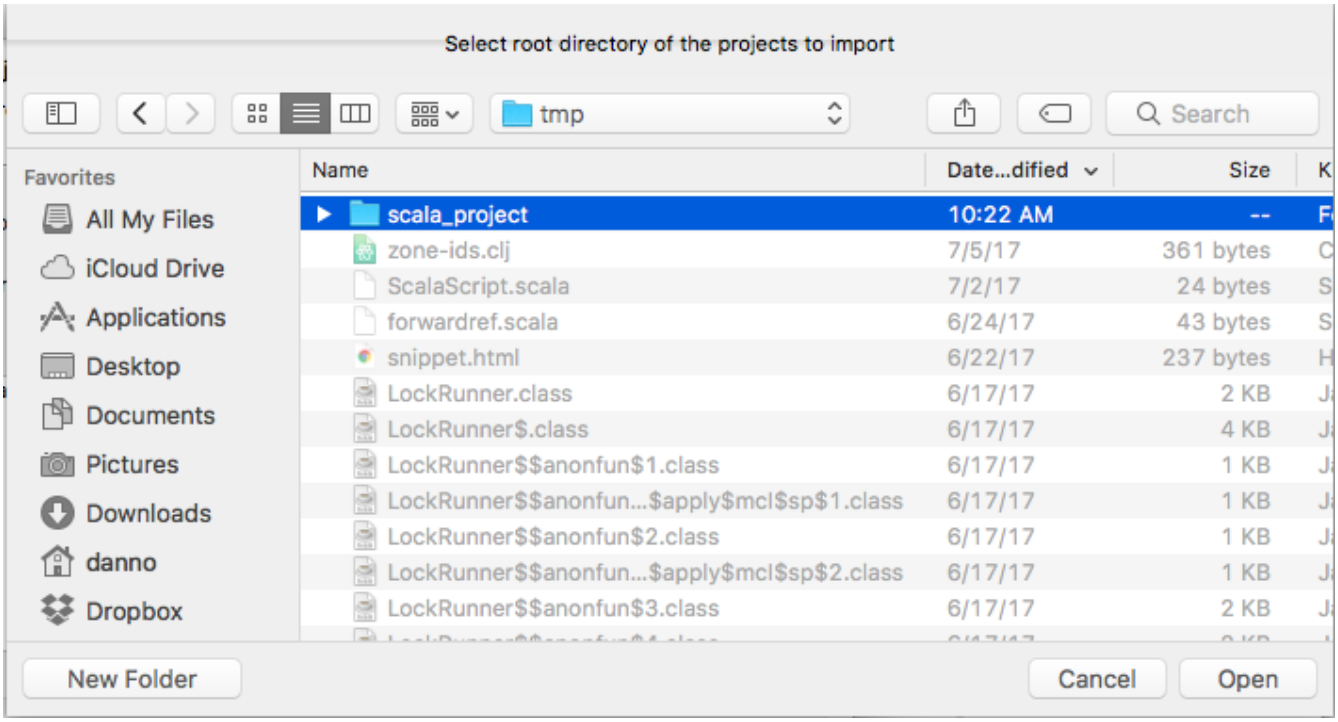
In Eclipse, in the menus, go to:

Step 1: File | Import

Step 2: Select: General | Existing Project in Workspace

Step 3: Click Next

Lab (Eclipse Users Only): Selecting the Project



Step 1: In *Select Root Directory* locate your project

Step 2: Select *Finish*

`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) //That's how you println
```

lazy val

About lazy val

- A `lazy val` is just like a `val` with the exception that the `lazy val` will not be evaluated until it is called.

```
lazy val a = {println "evaluated"; 5}
println(a)
```

Will be `println` "evaluated" one then evaluated `5` and will stay in the state.

Any subsequent calls to the lazy val will no longer be evaluated

```
println(a) // 5
```

No such thing as lazy var

- There is no such thing as `lazy var`
- In other words the following will not work

```
lazy var b = 200
```

```
error: lazy not allowed here. Only vals can be lazy
```

Lab: Forward referencing a lazy val

One thing that is possible with `lazy val` is the ability to forward reference.

Step 1: Create a new specification in *src/test/scala* called *LazyValsSpec.scala* in the `com.xyzcorp` package like the following:

```
package com.xyzcorp

import org.scalatest.{FunSuite, Matchers}

class LazyValsSpec extends FunSuite with Matchers {

}
```

Step 2: Inside the *LazyValsSpec.scala* that was just created add the following test.

```
test("lazy val forward reference") {
  val a = 10 + b
  val b = 19
  println(a)
}
```

Step 3: Run this test in either your IDE or at the command line with:

```
% sbt testOnly com.xyzcorp.LazyValsSpec
```

Step 4: Notice we will get `warning: Reference to uninitialized value b` with the wrong answer

Step 5: Fix this issue by making `a` a `lazy val`, and re-run the test

Exceptions with `lazy val`

`lazy val` will allow you to retry if you don't get a successful invocation.

```
var divisor = 0
lazy val quotient = 40 / divisor
println(quotient)
```

- Of course we get a division by zero exception.
- But since the first invocation failed, Scala is forgiving and gives you and attempt to do it again.
- If successful it will then lock the `val` quotient to the value.

```
divisor = 2
println(quotient)
```

- Now quotient is set and we see that it is set to 20.

Conclusion `lazy val`

- `lazy val` do not evaluate until called upon.
- Any subsequent calls will return the already established result
- You cannot do a `lazy var`
- `lazy val` is forgiving if an exception happens the first time.

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 where n is the number of bits.
- The minimum is the definite as negative 2^n.
- That means that the maximum size for a byte is 2^8-1 or 127.
- The minimum would be negative 2^8 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 //remember 2^n-1
```

`Short`

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

```
val maxShort = 32768:Short
```

Int

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

```
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⊠⊠ - 1
- Minimum: -2⊠⊠

```
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!"
```

`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

```
var a = 100
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 = 0 // 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 100) { // a for loop
   result = result + a
   if (a > 1) result = result + ","
   a = a - 1
}
```

Lab: `for` comprehensions

Since we discussed the topic of `for` comprehensions, let's do a lab for a sneak peek.

Step 1: Create a new specification in *src/test/java* called *ControlStatementsSpec.scala* in the `com.xyzcorp` package like the following:

```
package com.xyzcorp

import org.scalatest.{FunSuite, Matchers}

class ControlStatementsSpec extends FunSuite with Matchers {

}
```

Step 2: Create a `for` loop first inside of *ControlStatementsSpec.scala*

```
test("a for loop") {
   val xs = List(1,2,3,4)
   var result = List()
   for (a <- xs) {
      result = result :+ (a + 1)
   }
   result should be (List(2,3,4,5))
}
```

Step 3: Create a `for` comprehension next inside of *ControlStatementsSpec.scala*

```
test("a for comprehension") {
  val xs = List(1,2,3,4) //The list stays the same
  val result = for (a <- xs) yield (a + 1)
  result should be (List(2,3,4,5))
}
```

Step 4: Run this test in either your IDE or at the command line with:

```
% sbt testOnly com.xyzcorp.ControlStatementsSpec
```

Step 5: Discuss the differences

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
```

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 (`_`) is analogous to the asterisk in (`*`) in Java

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")
```

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}`

Lab: String Interpolation

Step 1: Create a new specification in *src/test/java* called *StringInterpolationSpec.scala* in the `com.xyzcorp` package like the following:

```
package com.xyzcorp

import org.scalatest.{FunSuite, Matchers}

class StringInterpolationSpec extends FunSuite with Matchers {

}
```

Step 2: Create the following test inside of *StringInterpolationSpec.scala*

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

Step 3: Run this test in either your IDE or command line to ensure victory with:

```
% sbt testOnly com.xyzcorp.StringInterpolationSpec
```

Step 4: Add `3` to the ***inside of the string***

The `f` interpolator

- Used to combine `String.format` functionality with 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%nThat'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

Lab: Figuring out the type

Step 1: Create a new specification in *src/test/java* called *MethodsSpec.scala* in the `com.xyzcorp` package like the following:

```
package com.xyzcorp

import org.scalatest.{FunSuite, Matchers}

class MethodsSpec extends FunSuite with Matchers {

}
```

Step 2: Create a method inside `MethodsSpec` with the following:

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

Step 3: Create a test to exercise the method, and fix the test

```
test("a method invocation") {
   val result = numberStatus(3)
   result should be ("Greater than 10")
}
```

Step 4: Copy the `numberStatus` code into a REPL to verify the type

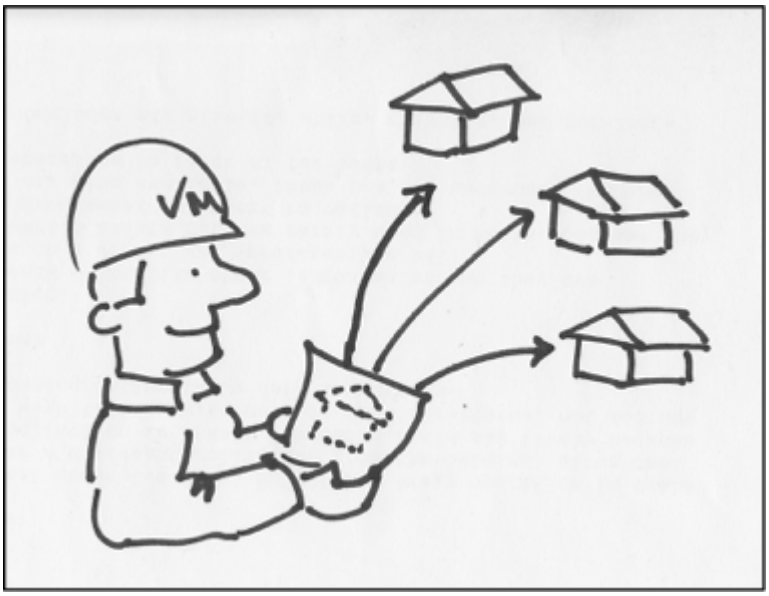
Conclusion

- 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

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

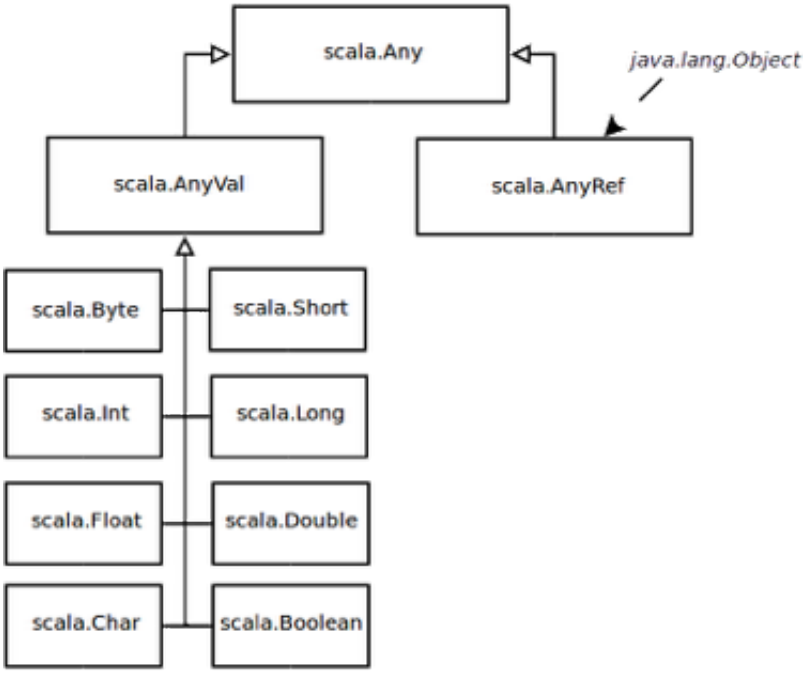
- `java.lang.Object` is called `AnyRef` in Scala
- `AnyRef` will be the super type of all object references
- This includes:
  - Scala API Classes
  - Java API Classes
  - Your custom classes

`Any` is the new leader

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

Scala Family Tree

Scala's Family Tree:



Lab: Polymorphism

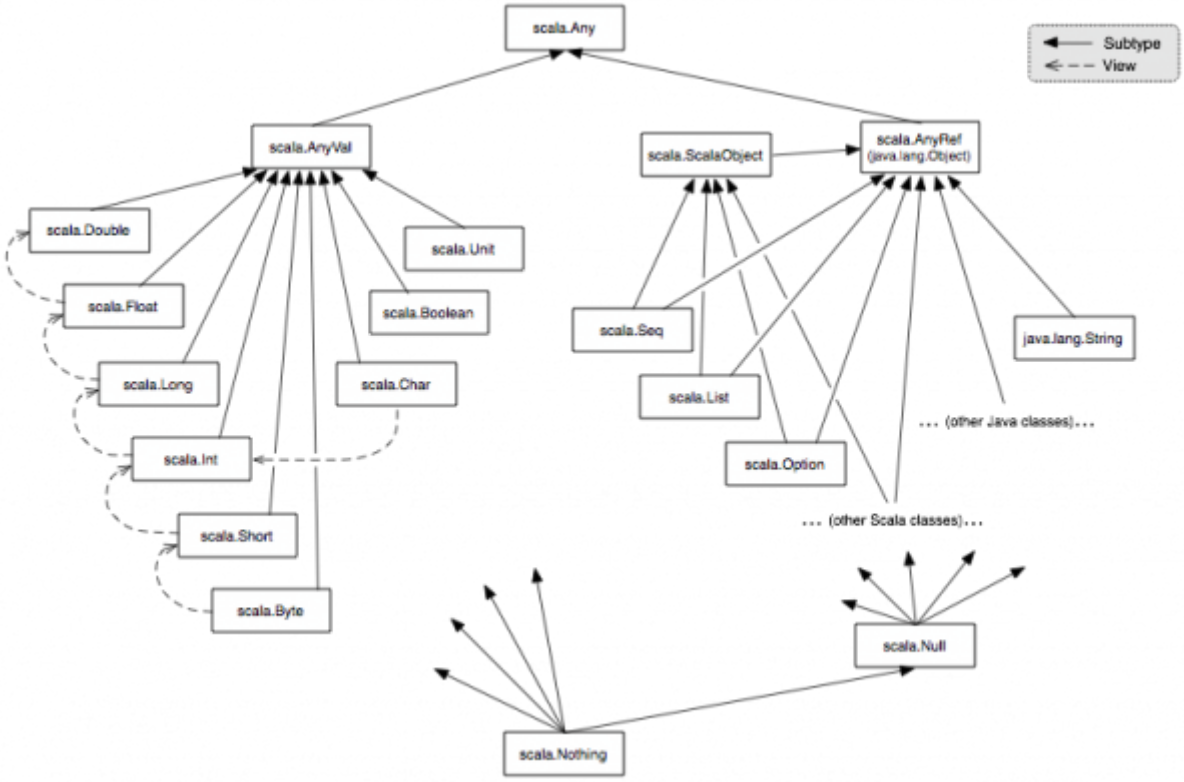
Step 1: Open a REPL either by typing `scala` or by running SBT's `console`.

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`

Lab: Where is `Nothing`?

Step 1: In a REPL try the following, create an empty `List` with no type:

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

Step 2: Create a `List` with a type, and note the difference

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

Step 3: Create a method called `ohoh` that returns only an `Exception`. Throwing an `Exception` in particular is always of type `Nothing` since nothing really ever gets returned.

```
> 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

Lab: `Null` type

Step 1: In a REPL, type the following, and noticed the type:

```
val f = null
```

Step 2: In the REPL, next try the following, and analyze the type:

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

Step 3: In the REPL, next try the following, and analyze the results, can you guess why there is a strange message?

```
val x:Int= null
```

Conclusion

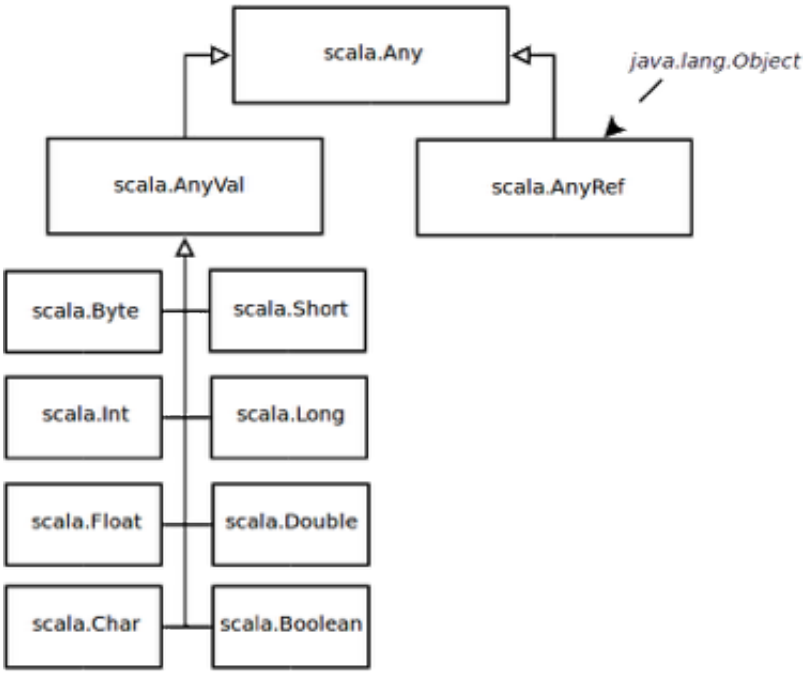
- Types are templates that make up an object.
- Primitives are also types, `Int`, `Short`, `Byte`, `Char`, `Boolean`, etc.
- Types need to matched up like a puzzle. If it isn't the type system at compile time will tell you there is a type mismatch
- Every type is in a relationship.
- `Any` is the parent 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

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, copy and paste the following using `:paste` mode and determine if you were correct

```
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

`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

Lab: `Unit`

Step 1: In the REPL, type the following

```
> val g = ()
```

Step 2: See that this is equal to `Unit` as a type

Lab: Where have we used `Unit`?

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

Step 1: Take a look at the signature of `Unit` at the following URL:

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

Step 2: Take a look at the signature of `println` at the following URL:

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

Step 3: Type the following in your REPL

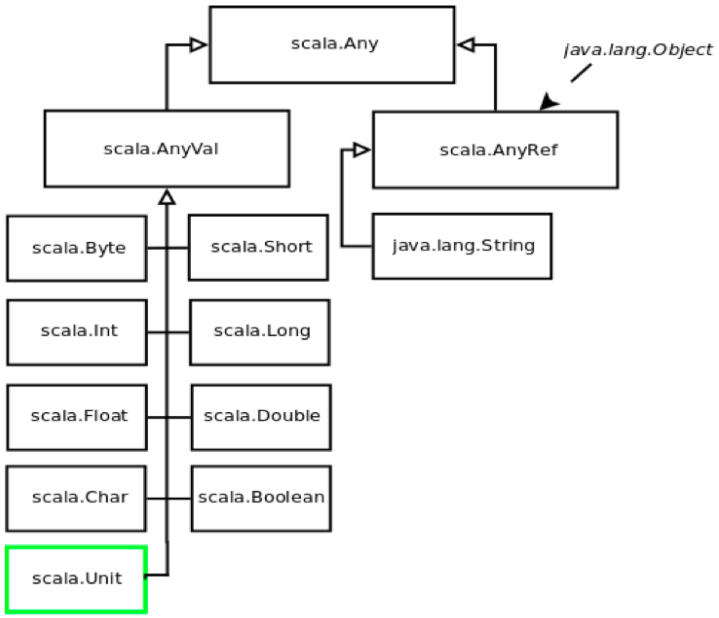
```
> println("Hello, Scala")
```

Step 4: Now assign to a value!

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

Step 5: Analyze the results

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 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

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

- 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

Lab: Creating and compiling the `class`

Step 1: In the *scala_project* project, create a new class in *src/main/scala* folder and in the `com.xyzcorp` package, and in the *Entities.scala* file, include the following content:

```
package com.xyzcorp

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

Step 2: Next, be sure you are in the *scala_project* project directory, and using sbt, run either `compile` from your command line or your SBT shell.

Reminder: How to compile from the command line:

```
% sbt compile
```

Reminder: How to compile in the SBT shell

```
% sbt
```

Then in the sbt shell

```
> compile
```

Investigating the Java byte code

Step 1: Now that you have compiled, you can view the byte code by running `javap -p` in the project folder.

```
% javap -p -cp target/classes com.xyzcorp.Employee
```

Step 2: View the Java method signature output and correlate the signature with that in your Scala file!

Lab: Instantiating the class

Step 1: Create a new specification in *src/test/java* called *ClassesSpec.scala* in the `com.xyzcorp` package like the following:

```
package com.xyzcorp

import org.scalatest.{FunSuite, Matchers}

class ClassesSpec extends FunSuite with Matchers {

}
```

Step 2: Create a test to instantiate the class inside of the file *ClassesSpec.scala*

```
test("instantiating an Employee") {
   val emp = new Employee("Dennis", "Ritchie")
}
```

Step 3: Run this test in either your IDE or at the command line with:

```
% sbt testOnly com.xyzcorp.ClassesSpec
```

Step 4: Verify the results

Can't access or modify 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)

Lab: Accessing and Mutating

Step 1: In your project scala_project and in the *src/main/java* folder and in the *Entities.scala* file and in the `com.xyzcorp` package, to access `firstName` and to access and mutate `lastName`, put a `val` in front of `firstName` and a `var` in front of `lastName`

```
package com.xyzcorp

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

Step 2: Compile

```
% sbt compile
```

Step 3: Show the bytecode and verify that you have an accessor and mutator

```
% javap -p -cp target/classes com.xyzcorp.Employee
```

Step 4: Rewrite the test "Instantiating an Employee" in *ClassesSpec.scala* to access and mutate the class

```
test("instantiating an Employee") {
   val emp = new Employee("Dennis", "Ritchie")
   emp.firstName should be ("Dennis") //Works because of val
   emp.lastName should be ("Ritchie") //Works because of var
   emp.lastName = "Hopper"            //Works because of var
   emp.lastName should be ("Hopper")
}
```

Step 5: Run this test in either your IDE or at the command line with:

```
% sbt testOnly com.xyzcorp.ClassesSpec
```

Step 6: Verify Results

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 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

Lab: Case Classes

Let's retrofit our `Employee` with a case class and fix the test

Step 1: In scala_project and in the *src/main/java* folder and in *Entities.scala* in the `com.xyzcorp` package, convert `Employee` to a case class, and get rid of the `val` and `var`.

```
package com.xyzcorp

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

Step 2: In *ClassesSpec.java*, adjust the "Instantiating an Employee" to use all the cool new features of a `case class`

```
test("instantiating an Employee") {
   val emp = Employee("Dennis", "Ritchie") //No new keyword
   emp.firstName should be ("Dennis")
   emp.lastName should be ("Ritchie")
   val emp2 = Employee("Dennis", "Ritchie")
   (emp == emp2) should be (true)
   (emp.hashCode == emp2.hashCode) should be (true)
   emp.toString should be ("Employee(Dennis, Ritchie)")
}
```

We removed `emp.lastName = "Hopper"` since `lastName` is now in a `case class` and all attributes are final because of the implied `val`. This class can no longer be mutated.

Step 3: Run this test in either your IDE or at the command line with:

```
% sbt testOnly com.xyzcorp.ClassesSpec
```

Step 4: Verify Results

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`
- A child class or parent class can be a `case class` but not both

Packages

About package

- package can be used in the same format as it is Java where *all* code in the file is under the package name

```
package com.xyzcorp;
```

- It can also be used as a stack as follows, this is Scala specific

```
package com {
   package xyzcorp {
   //The following line is in the xyzcorp package
   }
}
```

Containment of packages

- Packages can also be contained within one another for complete control
- All packages will inevitably be taken apart and store appropriately after compilation

```
package com {
   package xyzcorp {
   //The following lines is in the xyzcorp package
   }

   package othercorp {
   //The following line is in another othercorp package
   }
}
```

Flexible import Rules

- Scala has flexible `import` rules to obtain what you need, see the following table:

Format	Description
`import p._`	All members of `p` (this is analogous to `import p.*` in Java)
`import p.x`	The member `x` of `p`, for whatever `p`
`import p.{x ⇒ a}`	The member `x` of `p` renamed as `a`
`import p.{x, y}`	The members `x` and `y` of `p`
`import p1.p2.z`	the member `z` of `p2`, itself member of `p1`
`import p.{z ⇒ _}`	Everything except `z`

Renaming of import

- The following exemplifies the renaming of `import` statements
- This is typically used for Java package renaming within Scala

```
import java.lang.Math.{min => javaMin, max => javaMax}

javaMin(4, 10) //4
```

Flexibility of scoped import

- `import` can be strategically placed where needed
- It is not required that they be placed at the top of the file declaration

```
class Foo {
   class Bar {
       def baz(x:Int):Int {
          import scala.math._
          min(x, 10)
       }
   }
}
```

Importing a scala `static`/`object` import

- The following exemplifies the renaming of `import` statements
- This is typically used for Java package renaming within Scala

```
import java.lang.Math.{min => javaMin, max => javaMax}
import scala.math.{min => scalaMin, max => scalaMax}

javaMin(4, 10)  //4
scalaMin(5, 12) //5
```

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 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 //Compiler error: value ~: is not a member of Int
```

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 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 sbt console and type the following

```
scala> object MyObject
```

Step 2: Verify that it is truly a singleton

```
scala> val a = MyObject
scala> val b = MyObject
scala> a == b
scala> 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`

Lab: Adding to `MyObject`

Step 1: In your project, scala_project, and in the folder *src/main/java* and in the *Entities.scala* file in the `com.xyzcorp` package, create `MyObject` as follows:

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

Step 2: Create a new specification in *src/test/java* called *ObjectsSpec.scala* in the `com.xyzcorp` package like the following:

```
package com.xyzcorp

import org.scalatest.{FunSuite, Matchers}

class ObjectsSpec extends FunSuite with Matchers {

}
```

Step 3: Create a test to use `MyObject` inside of _ObjectsSpec.scala` called "calling an object"

```
test("calling an object") {
   MyObject.foo(4, 2) should be (6)
}
```

Step 4: Run the test in either the IDE or at the command line with:

```
% sbt testOnly com.xyzcorp.ObjectsSpec
```

Step 5: Verify the Results and discuss how much it looks like a `static` call

When do you need objects? When do you need singletons?

- 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?
}
```

Lab: Creating a factory

Step 1: In the scala_project project and *src/main/java* and in the *Entities.scala* file and in the `com.xyzcorp` package, create a class called `Department` and lock it up by placing a `private` in front of the primary constructor.

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

Step 2: In scala_project and *src/main/java* and in *Entities.scala* in the `com.xyzcorp` package create a companion object called `Department` that creates a `Department`

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

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

Step 3: In *ClassesSpec.java*, create a test "Calling a Department factory"

```
test("Calling a Department factory") {
  val department = Department.create("Toys")
  department.name should be ("Toys")
}
```

Step 4: Is there a better name instead of `create`? Decide and make a change. 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.

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`
- Knowing this the learning curve drops significantly

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

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

Using Symbols

Symbols

- Symbols are like a `String` except they are always interned and one symbol is always equal to another

Consider the following with a `String`:

```
val s = new String("Co")
val s2 = "Co"
println(s == s2) //true
println(s eq s2) //false
```

Now consider the what it would look like with a `Symbol`

```
val co = Symbol("Co")
val co2 = 'Co
println(co == co2) //true
println(co eq co2) //true
```

Using `Symbol` with a `Map`

- Since Symbols are perfect for `Map` we can use that instead of a `String`
- Quick constant lookup
- All interned so that there are never cases where they might be unequal, unlike `String`

```
val elements:Map[Symbol, String] = Map('Co -> "Cobalt",
                                       'H -> "Helium",
                                       'Pb -> "Lead")
elements.get('Co)
```

Conclusion

- Symbols, are like Strings, but guaranteed to be interned
- Symbols are also perfect for `Map` as keys since the same symbols are guaranteed to be `equal`

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, _ * _)
```

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

A function that returns a function

- The following is a method (or if you like to call it, a function), that takes two functions and returns a function
- This is a common signature called function composition
- This is obviously a higher order function
- This also uses parameterized types: `A`, `B`, `C`
- Parameterized types can represent anything

Here is the signature of the `compose` function

```
def compose[A, B, C](f: B => C, g: A => B):A => C
```

Here we will create two functions `add1` and `stringSize`

```
val add1:Int => Int = (x:Int) => x + 1
```

```
val stringSize:String => Int = (s:String) => s.length
```

By composing the functions we can create a conglomeration of the two, but the signatures have to be correct

Lab: Create and Implement `compose`

Step 1: In your *scala_project* and in the *src/main/java* folder as well as in the `com.xyzcorp` package, create an `object` titled `MyFunctions` that should look like the following

```
package com.xyzcorp

object MyFunctions {
   def compose[A, B, C](f: B => C, g: A => B):A => C = ???
}
```

Step 2: In *scala_project* and in *src/test/scala* and in the `com.xyzcorp` package, create an test called *FunctionsSpec.scala* with the following content.

```
package com.xyzcorp

import org.scalatest.{FunSuite, Matchers}

class FunctionsSpec extends FunSuite with Matchers {
    test("""the compose function takes two functions,
         and composes them into one""") {
       val add1 = (x:Int) => x + 1
       val stringSize = (s:String) => s.length
       val stringSizePlus1 = MyFunctions.compose(add1, stringSize)
       stringSizePlus1("Foo") should be (4)
    }
}
```

Step 3: Solve for `???` in the `object MyFunctions`

Step 4: You can clean up the imports by importing `com.xyzcorp.MyFunctions.compose` at the top so that you can only have to use `compose` when making the function call. Try it!

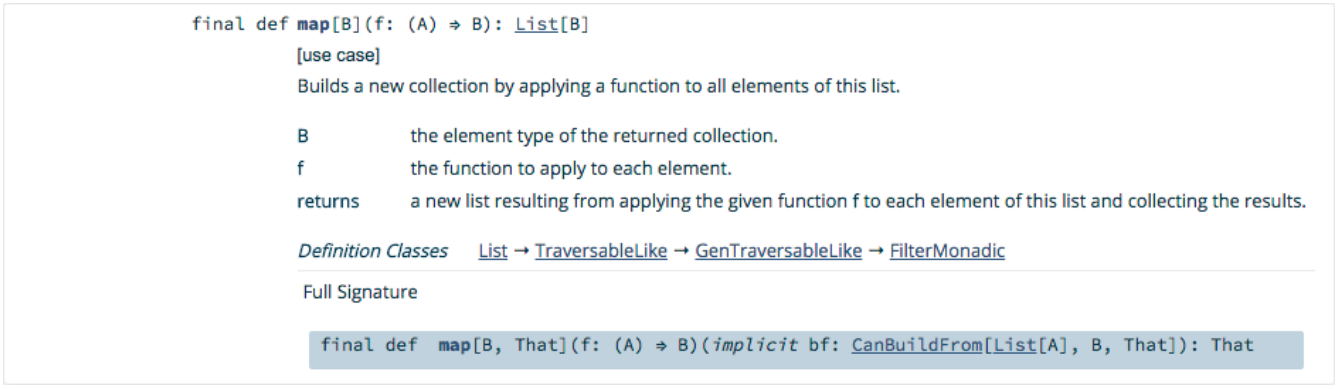
`map` function

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]`



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

[use case]

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

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 function f to each element of this list and collecting the results.

Definition Classes List → TraversableLike → GenTraversableLike → FilterMonadic

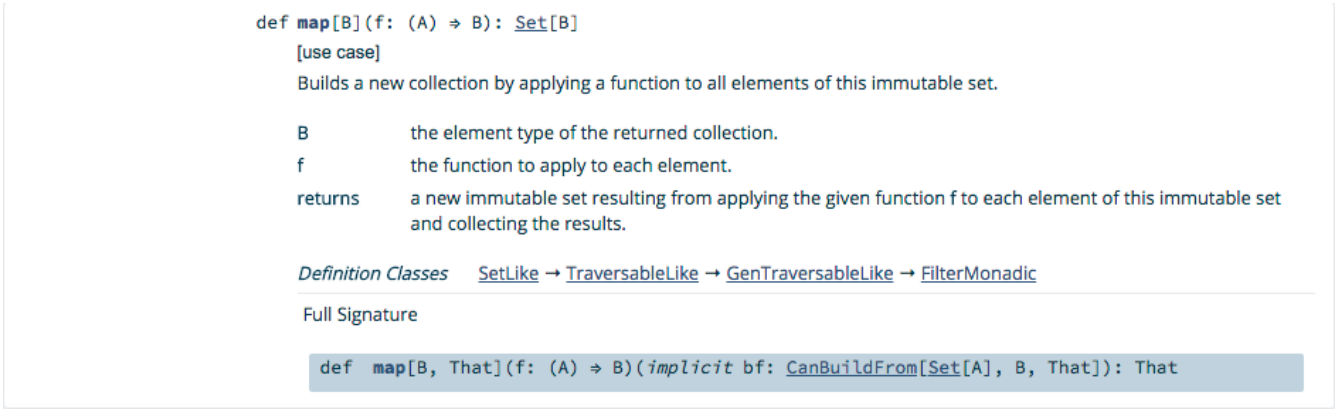
Full Signature

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

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

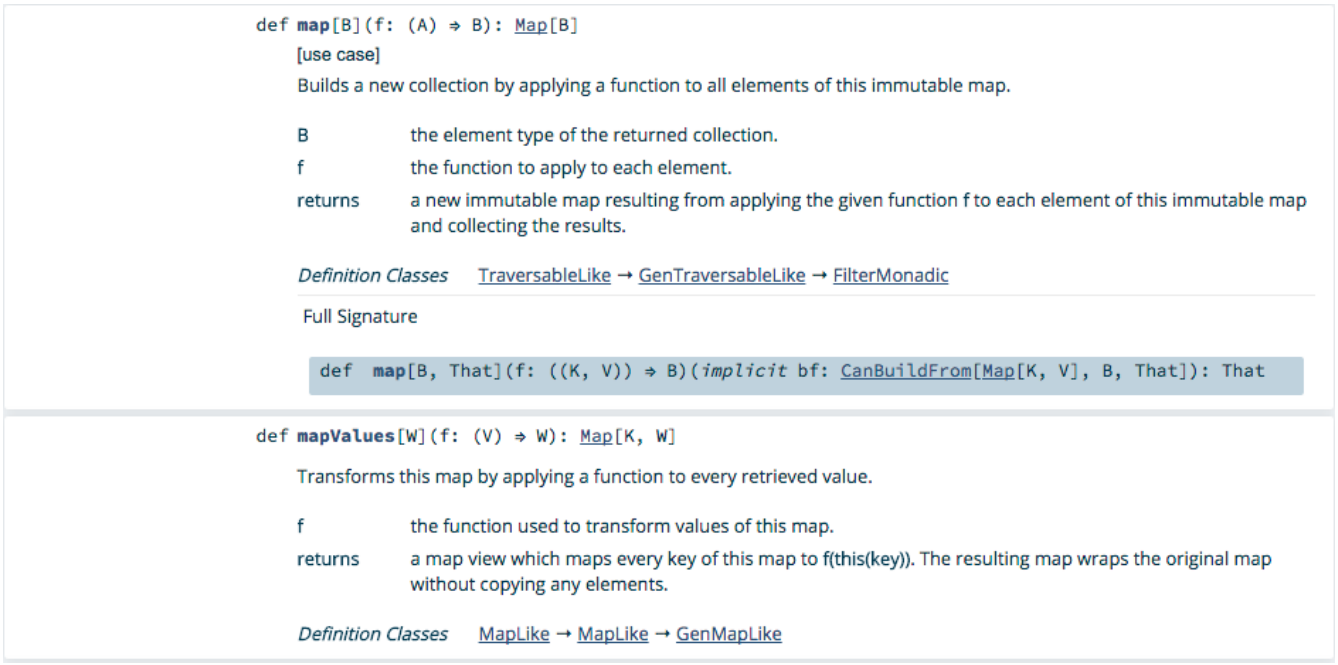
Full Signature

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

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

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` image::string_map_function.png[]

Lab: `map` function

Step 1: Create a new specification in *src/test/java* called *MapFunctionSpec.scala* in the `com.xyzcorp` package like the following:

```
package com.xyzcorp

import org.scalatest.{FunSuite, Matchers}

class MapFunctionSpec extends FunSuite with Matchers {

}
```

Step 2: Create a test like below to run a `map` example, notice the types along the way

```
test("""A map will take a function, and will apply that
       |function to every item on the List""".stripMargin) {
   val result = List("One", "Two", "Three").map(x => x.size)
   result should be (List(3, 3, 5))
}
```

Step 3: Run the test and verify the results

Step 4: Determine the type of `result` either by adding an explicit type to result, for example, `val result:List[???]` or you can use your IDE faculties to determine the type.

Lab: Manipulating an `Option[Tuple2[Long, String]]` with `map`

Step 1: In the *MapFunctionSpec.scala* Specification you just created, add the following test and setup:

```
test("""A map will take a function, and will apply that
       |function to every item in an Option""".stripMargin) {

   val before:Option[Tuple2[Long, String]] = Some((10L, "Foo"))
   val after:Option[Tuple2[Long, String]] = ???
   after should be (Some((20L, "FooBar")))
}
```

Step 2: In `after`, you should take `before` and manipulate it to where the first element of the `Tuple2` inside the `Some` is added to `10L` and the second element of the `Tuple2` inside of `Some` is appended with `"Bar"`. Solve for `???` so that assertion works

Step 3: Can all the parenthesis be cleaned up? If yes, do so.

Step 4: Do we need to declare `before` and `after`. If not, allow type inference to do its job.

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

About 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

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.

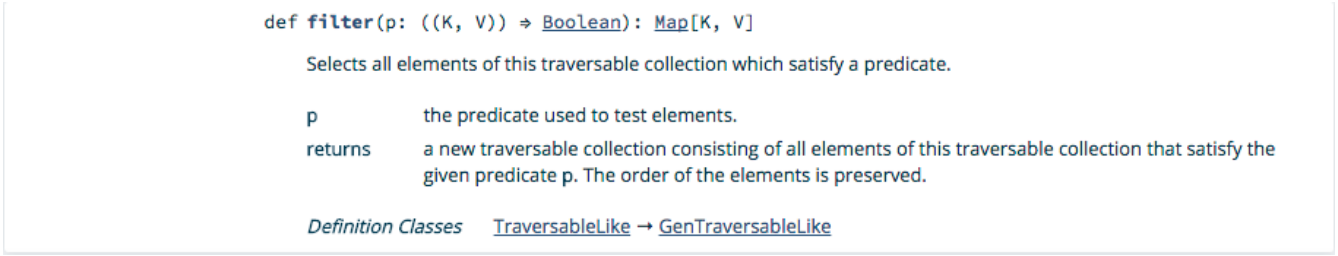
p	the predicate used to test elements.
returns	a new traversable collection consisting of all elements of this traversable collection that satisfy the given predicate p. The order of the elements is preserved.

Definition Classes TraversableLike → GenTraversableLike

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.

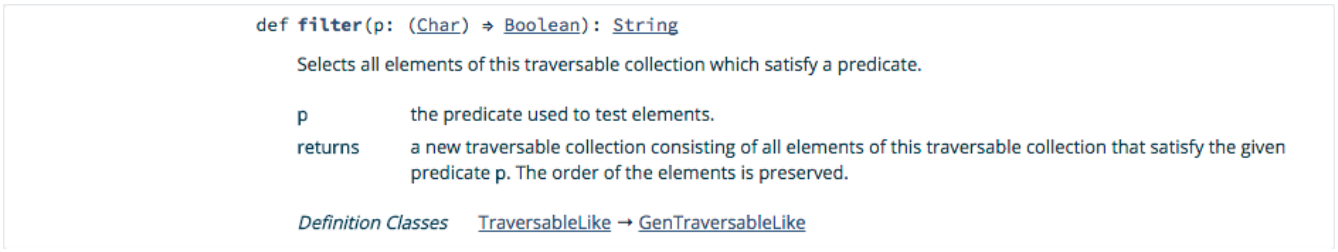
p	the predicate used to test elements.
returns	a new traversable collection consisting of all elements of this traversable collection that satisfy the given predicate p. The order of the elements is preserved.

Definition Classes TraversableLike → GenTraversableLike

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
```

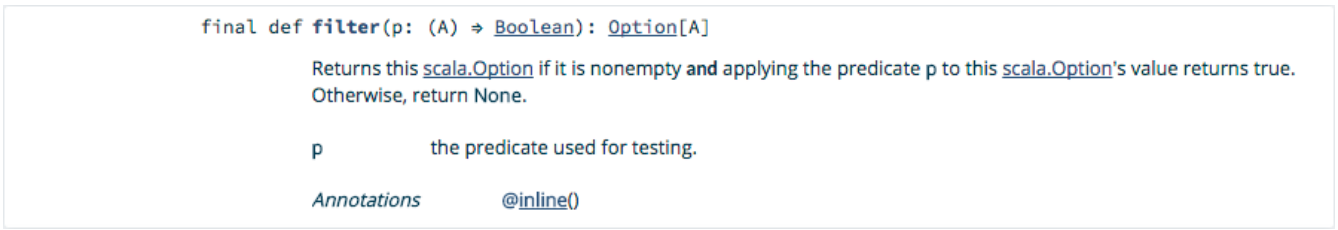
Selects all elements of this traversable collection which satisfy a predicate.

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

`Option` and `filter`

- This is the API Signature of `filter` for `Option`
- Note that the parameterized type of the `Option` in this case is actually `[A]`
- The behavior of what `filter` does here is quite unique, if the evaluation of the element is `false` then `filter` returns `None`



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

Returns this scala.Option if it is nonempty **and** applying the predicate p to this scala.Option's value returns true. Otherwise, return None.

p	the predicate used for testing.

Annotations @inline()

Lab: `filter` function

Step 1: In the *scala_project* project and in the `com.xyzcorp` package, create a new specification in *src/test/java* called FilterFunctionSpec.scala where it looks like the following:

```
package com.xyzcorp

import org.scalatest.{FunSuite, Matchers}

class FilterFunctionSpec extends FunSuite with Matchers {

}
```

Step 2: Create a test like below to run a `filter` example, notice the types along the way

```
test("""A filter will take a function that returns
     | a boolean, and will remove all elements
     | where the function evaluates to false""") {
   val result = List(1,2,3,4,5).map(x => x % 2 == 0)
   result should be (List(2,4))
}
```

Step 3: Run the test and verify the results

Step 4: Determine the type of `result` either by adding an explicit type to result, for example, `val result:List[???]` or you can use your IDE faculties to determine the type.

Lab: Manipulating an `List[Option]` with `filter`

Step 1: In the *FilterFunctionSpec.scala* Specification you just created, add the following test and setup:

```
test("""A filter will take a function that
      returns a boolean, and will remove all elements
      where the function evaluates to false, in a
      List of Options this will do something peculiar""") {

        val list = List(Some(4), Some(10), None,
                        Some(11), None, Some(50),
                        Some(3), None, Some(100))
        val result = list.filter(o => ???).map(o => ???)
        result should be (List(4, 10, 11, 50, 3, 100))

}
```

Step 2: Solve for the `???` so that assertion works and the test passes. As a hint, `Option[T]` has a method `isEmpty` that returns true if it is a `Some` and false if it is a `None`

Step 3: Clean up any inefficiencies that you can

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:

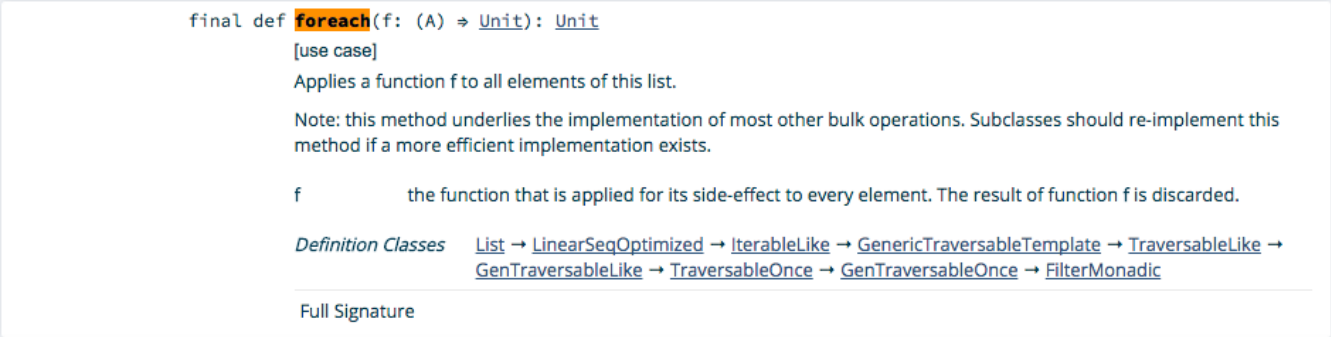
```
val a = (1 to 10)
val list = a.map(x => println(x))
println(list)
```

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:

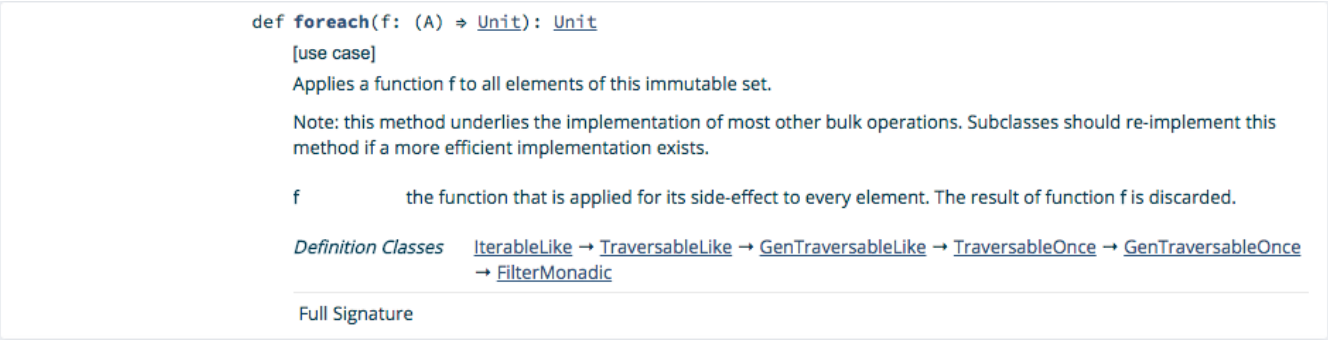
```
val a = 1 to 10
a foreach println
```

- Inlining the whole thing:

```
1 to 10 foreach println
```

`Set` and `foreach`

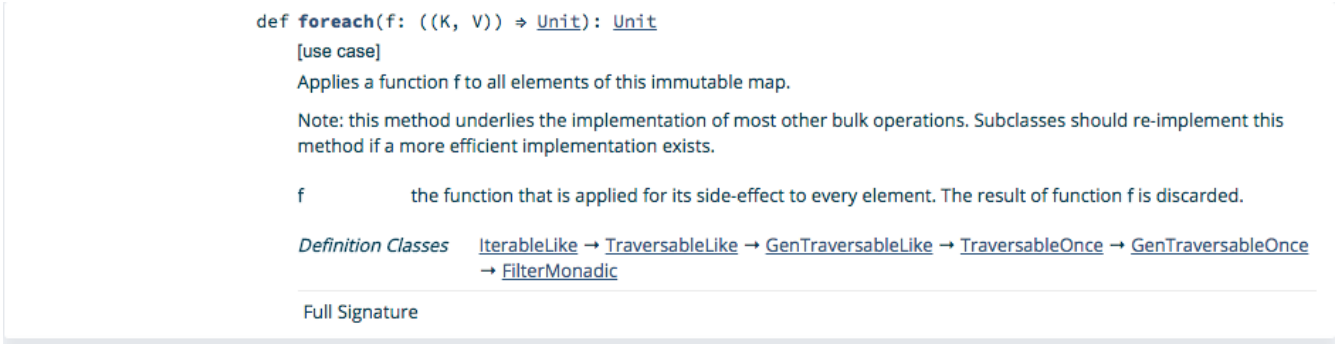
- This is the API Signature of `foreach` for `Set`
- Note that the parameterized type of the `Set` in this case is `[A]`



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

`Map` and `foreach`

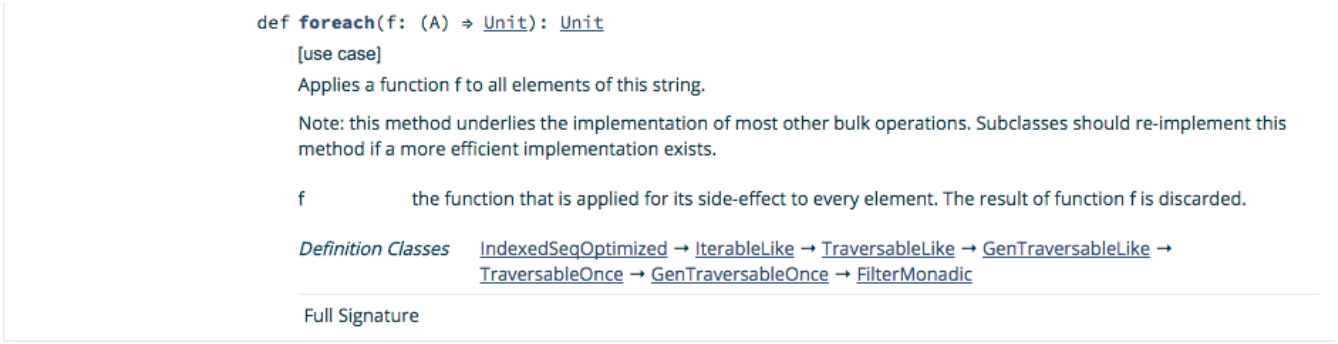
- This is the API Signature of `foreach` for `Map`
- Note that the parameterized type of the `Map` in this case is `[K, V]`
  - `K` is the key parameterized type
  - `V` is the value parameterized type



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

`String` and `foreach`

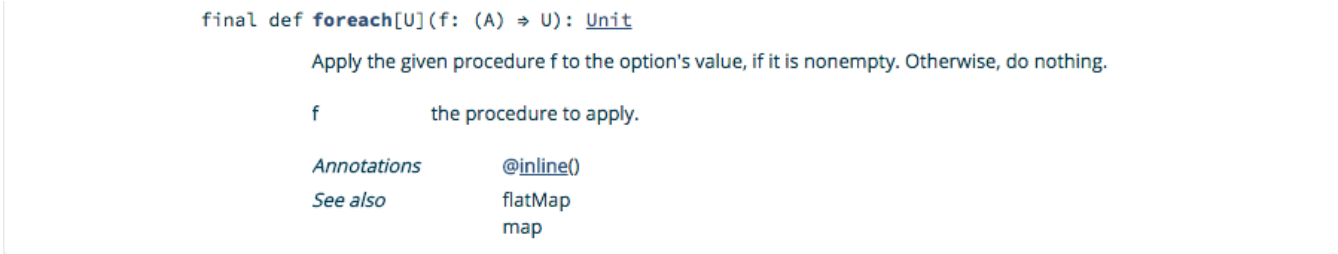
- This is the API Signature of `foreach` for `String`
- Note that the parameterized type of the `StringOps` in this case is `[A]` but is usually `Char`



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`

About `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`

Lab: The identity function and `flatMap`:

Step 1: In the project *scala_project*, create a new specification in *src/test/java* called *FlatMapFunctionSpec.scala* in the `com.xyzcorp` package like the following:

```
package com.xyzcorp

import org.scalatest.{FunSuite, Matchers}

class FlatMapFunctionSpec extends FunSuite with Matchers {

}
```

Step 2: Create a test below inside of `FlatMapFunctionSpec` to run a `flatMap` example, notice the types along the way

```
test("""A flatMap will take a function that returns
     | a TraversableOnce like List, Set, Map, and will
     | combine or flatten the results"""){

   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)))
   val result1 = list.flatMap(???)
   result1 should be (???)
}
```

Step 3: In the set of `???` plugin the identity function in which ever form, In the second set of `???` plugin the result.

Step 4: Clean up any inefficiencies that you can.

Lab: `flatMap` Another Layer

Step 1: In the project *scala_project*, open the specification called *FlatMapFunctionSpec.scala* in the `com.xyzcorp`, add another `flatMap`

Step 2: In the "A flatMap will take a function" test, `flatMap` result1 using the identity function, and determine the result. Use the following as guidance.

```
test("""A flatMap will take a function that returns
     | a TraversableOnce like List, Set, Map, and will
     | combine or flatten the results"""){

   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)))
   val result1 = list.flatMap(???)
   result1 should be (???)
   val result2 = result1.flatMap(???)
   result2 should be (???)
}
```

Step 3: Attempt to do it again, did it work? If it did, Why? If it didn't why not?

Hint: Use the List API: http://www.scala-lang.org/api/current/scala/collection/immutable/List.html

`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`

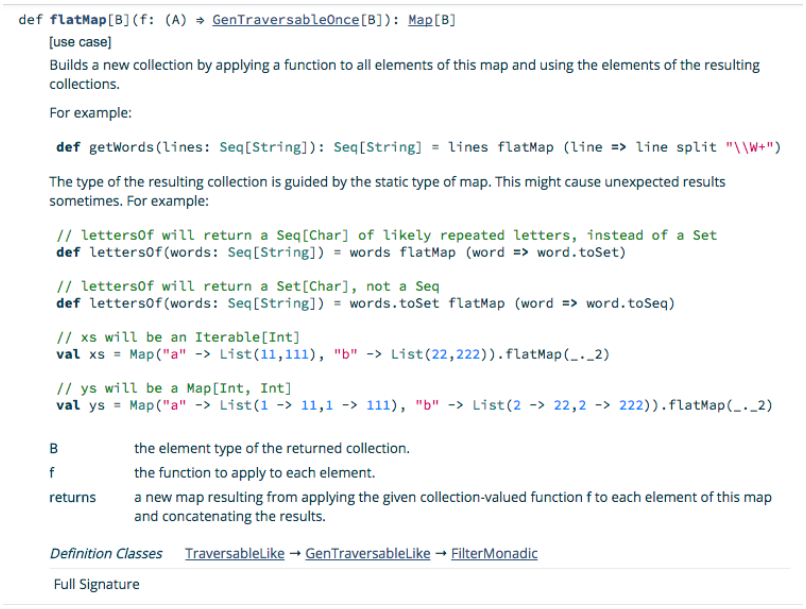
```
Set(2, 4, 10, 11).flatMap(x => Set(x, x*5))
```

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")

origMap.flatMap { t =>
  Map(t._1         ->  t._2,
     (t._1 * 100) -> (t._2 + " Hundred"))
}
```

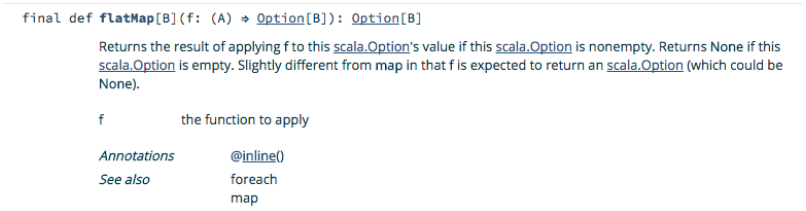
Will Render

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

`Option` and `flatMap`

`Option[A]` are flat mappable too and are they fun to `flatMap`

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



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

Returns the result of applying f to this scala.Option's value if this scala.Option is nonempty. Returns None if this scala.Option is empty. Slightly different from map in that f is expected to return an scala.Option (which could be None).

f the function to apply

Annotations @inline()

See also foreach
map

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

Using `Option` and `flatMap`

```
Some(4).map(x => Some(x + 19))
```

Will render:

```
Some(Some(23))
```

We see that the `map` with create a `Some` with a `Some`. That, again, is our cue that may we want a `flatMap`.

```
Some(4).flatMap(x => Some(x + 19))
```

Will render

```
Some(23)
```

`flatMap` with `None`

- In any functionial computation, a `None` in the functional chain will render the whole thing as `None` What if this is a `None`? Well first, let's make a terrible mistake

```
None.asInstanceOf[Option[Int]].flatMap(x => Some(x + 19))
```

Will render:

```
None
```

Why did we need `asInstanceOf`? Try it for yourself in the REPL.

Lab: Another `flatMap` with `None`

Step 1 In the REPL, type the following. What if we are performing a `flatMap` from a `Some` but the function used returns a `None`, type:

```
println(Some(10).flatMap(x => None))
```

You could put `asInstanceOf[Option[Int]]` but was not required since `Some(10)` can be inferred that it is of type `Option[Int]`.

Lab: `flatMap` a `List` of `Option`

Step 1: In the project *scala_project*, open the specification called *FlatMapFunctionSpec.scala* in the `com.xyzcorp`, add a new test like the following

```
test("""A flatMap can work with a List of Option with
        some dazzling effects""") {

     val result = List(Some(4), None, Some(5),
                  None, None, Some(10)).flatMap(x => x)
     result should be (???)

}
```

Step 2: Fill in the `???` with what you think the result should be. Run the test and verify.

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]]]`

Lab: Create a functional grocery list

Lab: Create a functional grocery list

Step 1: Create a grocery list: `List("Eggs", "Milk", "Naan", "Broccoli", "Salmon", "Apples", "Green Lettuce", "Peas")`

Step 2: Create a `String` that list all the groceries, sorted, enumerated by number followed by a period with a carriage return after each item and do so functionally

Step 3: The end result should look like the following:

```
1. Apples
2. Broccoli
3. Eggs
4. Green Lettuce
5. Milk
6. Naan
7. Peas
8. Salmon
```