



Core Scala

<http://scalacourses.com>



Fundamentals of Scala

Section 1	scalaRunning	Installing and Running	9 lectures
Lecture 1-1	scalaOverview	Scala Overview and Philosophy	
	Everything is an Object		
	Scala is Functional		
	Scala is Succinct		
	Every Scala Statement Returns a Value		
	Packages		
Lecture 1-2	scalaSbtInstall	Installing Scala & SBT	
	NoCompDaemon option		
	Installing SBT		
Lecture 1-3	scalaCodeRunner	The Scala Code Runner	
Lecture 1-4	scalaREPL	The Scala Interpreter (REPL) and simple looping	
	Other Control Statements		
	while		
	Scala assignment returns Unit		
	do while		
	for		
Lecture 1-5	scalaScripts	Scala Scripts and Calling Java from Scala	
	Scala File		
	Shell Script		
Lecture 1-6	sbtIntro2	Introduction to SBT	
	Installation		
	Cygwin		
	Mac/Linux launch		
	Global Setup		
	Project Setup		
	Project Files		
	project/build.properties		
	build.sbt		
	Sample build.sbt		
	project/plugins.sbt		
	Muzzle SBT		
	Digging Deeper		
	Where is sbt-launch.jar Stored?		
	Custom resolver		
	Useful SBT Commands		

Lecture 1-7	scalaEclipse	Working With Scala IDE for Eclipse	00:23:23
	Eclipse Configuration File		
	Eclipse Configuration Settings		
	Creating a New Scala Project		
	Converting the Sample Code to Eclipse Projects		
	Scala Interpreter		
	Running Scala and Java Programs		
	Performance		
	Useful Eclipse Hot Keys		
	Hot Keys Also Available in IntelliJ		

Lecture 1-8	scalaIDEA	Working With IntelliJ IDEA
	Configuring IDEA	
	Tuning Memory Allocation	
	Converting SBT Projects Into IntelliJ Projects	
	Configuring IDEA Scala Projects	
	Helpful Hints	
	Useful IntelliJ Hot Keys	
	Hot Keys From Eclipse Key Bindings	

Lecture 1-9	worksheet	Worksheets
	Scala-IDE (Eclipse) Worksheets	
	IDEA Worksheets	

Section 2	scala00	Object-Oriented Scala	11 lectures
------------------	----------------	------------------------------	--------------------

Lecture 2-1	scalaClasses	Classes
	Scala Classes	

Lecture 2-2	scalaAuxCons	Auxiliary Constructors for Classes
-------------	--------------	------------------------------------

Lecture 2-3	scalaSettersGetters	Setters and Getters
-------------	---------------------	---------------------

Lecture 2-4	scalaCompanion	Companion Objects
	Unapply	

Lecture 2-5	scalaCaseClasses	Case Classes
	Standard Methods of Case Classes	

Lecture 2-6	scalaUniform	Uniform Access Principle
-------------	--------------	--------------------------

Lecture 2-7	scalaFunctions	Functions are First Class
-------------	----------------	---------------------------

Lecture 2-8	scalaMatch	Pattern Matching 101
	Aliases	
	Sequences	

Lecture 2-9 scalaTraits Scala Traits

Pure Trait

Trait with Implementation

Extending Multiple Traits

Self Traits and Dependency Injection

Self Traits and Structural Types

Lecture 2-10 scalaTypes Type Hierarchy

Object Equality

Lecture 2-11 scalaAccess Deceptively familiar: access modifiers

Section 3 scalaIntroMisc **Useful Stuff** **1 lectures**

Lecture 3-1 scalaUnit Unit testing With Specs2

Quick and Easy Test Specifications

SBT

IntelliJ IDEA

Scala IDE

Section 4 scalaFunctional **Functional Programming** **3 lectures**

Lecture 4-1 scalaOption Option, Some and None

Lecture 4-2 scalaEither Either, Left and Right

Setup

Solution 1 - Efficient But Clumsy

Solution 2 - Efficient, Simple, Elegant

Solution 3 - Overly Complex But Shows Scala Techniques

Lecture 4-3 scalaIntroReview Review So Far

1 Installing and Running

ScalaCourses.com / scalaIntro / ScalaCore / scalaRunning

1-1 Scala Overview and Philosophy

ScalaCourses.com / scalaIntro / ScalaCore / scalaRunning / scalaOverview

Scala is a Java-compatible language that is both object-oriented and functional, and was designed to support large-scale computing. It is succinct and is designed to be the basis of domain-specific languages (DSLs). Scala is strongly typed, yet can be used in an interpretive fashion in many circumstances.

Scala has a minimum of control structures, because its design encourages data-driven programming. Python's data structures resemble Scala's in this fashion. Unlike Python, Scala allows you to design your own control structures easily.

Everything is an Object

Numbers, strings and even `Unit` (the Scala equivalent of `null`) is an object. Objects are normally defined with methods, and Scala also provides a mechanism called *implicit*s that allows you to associate new methods with objects, without changing the object definitions. This means that your domain model need not get tangled up with your business logic.

Functions are objects too, and can be manipulated and passed around in interesting ways.

Scala is Functional

You can just use Scala as a better Java, and there would be significant benefit. However, if you also learn how to write functional code using Scala, your program could readily take advantage of the multi-core capabilities of today's computing platforms. Functional programs are also easier to reason about than stateful programs.

Scala is Succinct

Unnecessary punctuation and verbosity has largely been done away with. For example, semicolons at the end of statements are only necessary if you want to write more than one statement per line. Statements that span several lines are detected by the compiler, although sometimes you have to let the compiler know that the statement continues to the next line by ending the line with an operator.

Every Scala Statement Returns a Value

This means that the last statement in a block of code establishes the return value. For example, here is a method definition that returns the value of $\pi/2$ (90 degrees, expressed in radians):

```
def piBy2: Double = math.Pi / 2.0
```

As you can see, the method `piBy2` computes the value of dividing the predefined constant `Pi` by `2.0`. The value becomes the value returned by the method.

The value returned from a Scala `if/then/else` construct can be used to set a variable. For example, the following performs the same function as `math.max()`, except that `bigger()` is limited to working with `Int` arguments:

```
def bigger(x: Int, y: Int) = if (x>y) then x else y
```

The compiler is smart enough to know that the value returned by `bigger` must be an `Int`, but you can declare the return type if you are so inclined:

```
def bigger(x: Int, y: Int): Int = if (x>y) then x else y
```

An `if` statement without an `else` clause is subject to *type widening*. The non-existent clause has value `Unit`, which is the Scala equivalent of `null`. The Scala compiler realizes that type `AnyVal` (which we will learn about soon) is the most specific type that can represent all possible returned values from an `if/then` clause. For this reason, you should not write an `if/then` clause to return a value from a block of code.

Packages

Like Java, source files can be grouped into packages. Unlike Java, the directory structure is decoupled from the Scala packages. Furthermore, you can scope a package, so that only a portion of a source file resides within a package. Here is a sample file, called `Packager.scala`. Two classes are defined: `com.micronautics.Temporal` and `com.micronautics.alternate.Spatial`.

```
package com.micronautics

class Temporal(age: Int)

package alternate {

  class Spatial(latitude: Double, longitude: Double)

}
```

Scala also has package objects, which can contain package-level definitions. Let's put `demo.scala` in a package called `com/micronautics/`.

```
package com.micronautics

package object demo {
  def doSomething(x: Int): Unit = ???
}
```

Notice that the `doSomething` method is defined but not implemented, yet it is not abstract. Scala understands `???` as a placeholder for a method body that returns `Nothing`. As we shall see later, this means that you can fill in the body of the method when you are ready.

You can invoke `doSomething()` from anywhere in the `com.micronautics.demo` package without qualification.

package objects are automatically imported into the package of the same name.

```
package com.micronautics.demo

def blah: Unit = {
  doSomething(3)
}
```

To import from a different package:

```
package x.y.z

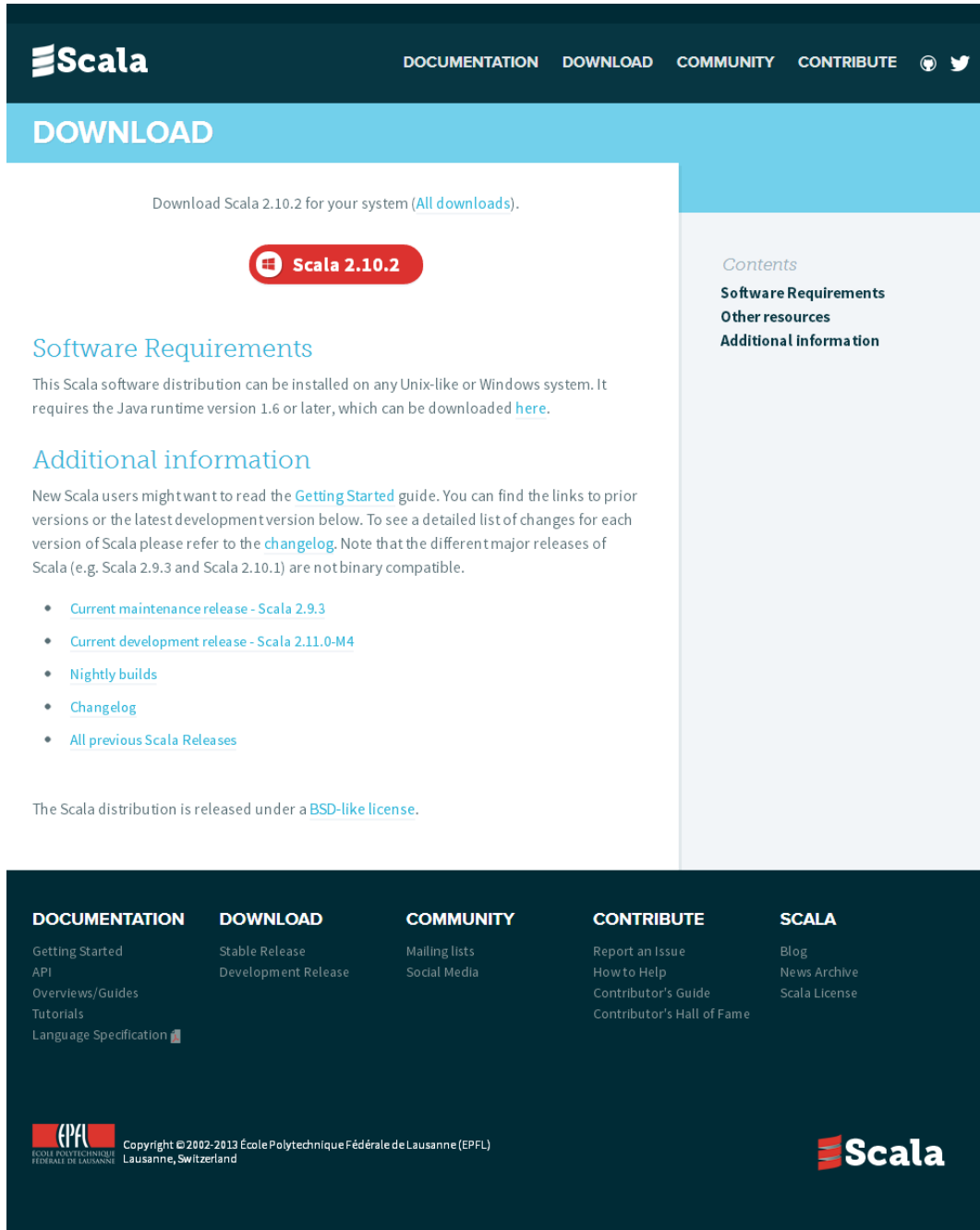
import com.micronautics.demo.doSomething

def blah: Unit = {
  doSomething(3)
}
```

1-2 Installing Scala & SBT

ScalaCourses.com / scalaIntro / ScalaCore / scalaRunning / scalaSbtInstall

Installing Scala is really easy, just navigate to <http://www.scala-lang.org> and click on the Downloads tab. Push the big red button and follow the instructions. For Windows, that is all you need to know. For other operating systems, read on.



The screenshot shows the Scala download page. At the top is the Scala logo and navigation links: DOCUMENTATION, DOWNLOAD, COMMUNITY, and CONTRIBUTE. Below the navigation bar is a large blue header with the word "DOWNLOAD" in white. The main content area has a white background. It starts with the text "Download Scala 2.10.2 for your system (All downloads)." followed by a red button with the Scala logo and "Scala 2.10.2". Below this is a section titled "Software Requirements" with the text: "This Scala software distribution can be installed on any Unix-like or Windows system. It requires the Java runtime version 1.6 or later, which can be downloaded [here](#)." This is followed by a section titled "Additional information" with the text: "New Scala users might want to read the [Getting Started](#) guide. You can find the links to prior versions or the latest development version below. To see a detailed list of changes for each version of Scala please refer to the [changelog](#). Note that the different major releases of Scala (e.g. Scala 2.9.3 and Scala 2.10.1) are not binary compatible." Below this is a bulleted list of links: "Current maintenance release - Scala 2.9.3", "Current development release - Scala 2.11.0-M4", "Nightly builds", "Changelog", and "All previous Scala Releases". At the bottom of the main content area, it says "The Scala distribution is released under a [BSD-like license](#)." On the right side of the page, there is a sidebar with the title "Contents" and three links: "Software Requirements", "Other resources", and "Additional information". At the bottom of the page is a dark blue footer with five columns of links: DOCUMENTATION (Getting Started, API, Overviews/Guides, Tutorials, Language Specification), DOWNLOAD (Stable Release, Development Release), COMMUNITY (Mailing lists, Social Media), CONTRIBUTE (Report an Issue, How to Help, Contributor's Guide, Contributor's Hall of Fame), and SCALA (Blog, News Archive, Scala License). In the bottom left corner of the footer is the EPFL logo and copyright information: "Copyright © 2002-2013 École Polytechnique Fédérale de Lausanne (EPFL) Lausanne, Switzerland". In the bottom right corner is the Scala logo.

1. You can download the packed file using the command line instead of a browser if you like. Here is how to download the version that was current when this lecture was written. First I changed to the /opt directory, where I want to store the unpacked contents. If you use Mac or Cygwin you might want to store them in Applications or some other directory. The contents of the zip and tgz files on the scala-lang.org web site are identical, but the tgz file is better because you won't have to set permissions on executable scripts if you uncompress it.


```
$ cd /opt
$ wget http://www.scala-lang.org/files/archive/scala-2.10.2.tgz
```

2. You need to unpack the file, which has a .tgz filetype. You can place the unpacked contents anywhere you like. You can use an archive tool or a command line. Since I am working with the Linux operating system, I'll unpack to the /opt directory. Here is the command I used at a console prompt to unpack the file which was downloaded to my Downloads directory. These commands create a directory called /opt/scala-2.10.2 which makes sense only for Linux - for Mac, you might want to unpack to another directory. Each version of Scala will create a similarly named directory:

```
$ tar xvzf scala-2.10.2.tgz
```

3. Define an environment variable called SCALA_HOME that points to the new directory (adjust the location of SCALA_HOME as required):

Run the following for changes to take effect without restarting Terminal:

```
$ source ~/.bash_profile
```

1. For Linux, define the variable in ~/.bashrc:

```
export SCALA_HOME=/opt/scala-2.10.2
export PATH=$SCALA_HOME/bin:$PATH
```

2. For Mac, define the variable in ~/.bash_profile:

```
export JAVA_HOME=$(/usr/libexec/java_home)
export SCALA_HOME=~/.Applications/scala-2.10.2
export PATH=$SCALA_HOME/bin:$PATH
```

Run the following for changes to take effect without restarting Terminal:

```
source ~/.bash_profile
```

4. Ensure that JAVA_HOME is set, and set it in ~/.bashrc (Linux) or ~/.bash_profile (Mac) if not:

```
$ echo "$JAVA_HOME"
```

1. Open a new shell and test:

```
$ scala -version
Scala code runner version 2.10.2 -- Copyright 2002-2013, LAMP/EPFL

$ java -version
java version "1.7.0_25"
Java(TM) SE Runtime Environment (build 1.7.0_25-b15)
Java HotSpot(TM) 64-Bit Server VM (build 23.25-b01, mixed mode)
```

You can read about the Scala runner's options like this:

\$ scala -help

Usage: scala <options> [<script|class|object|jar> <arguments>]
or scala -help

All options to scalac (see scalac -help) are also allowed.

The first given argument other than options to scala designates what to run. Runnable targets are:

- a file containing scala source
- the name of a compiled class
- a runnable jar file with a valid Main-Class attribute
- or if no argument is given, the repl (interactive shell) is started

Options to scala which reach the java runtime:

- Dname=prop passed directly to java to set system properties
- J<arg> -J is stripped and <arg> passed to java as-is
- nobootcp do not put the scala jars on the boot classpath (slower)

Other startup options:

- howtorun what to run <script|object|jar|guess> (default: guess)
- i <file> preload <file> before starting the repl
- e <string> execute <string> as if entered in the repl
- save save the compiled script in a jar for future use
- nc no compilation daemon: do not use the fsc offline compiler

A file argument will be run as a scala script unless it contains only self-contained compilation units (classes and objects) and exactly one runnable main method. In that case the file will be compiled and the main method invoked. This provides a bridge between scripts and standard scala source.

Options for plugin 'continuations':

- P:continuations:enable Enable continuations

The next few lectures will show you how to run Scala code in various ways.

```
$ scalac -X |& grep warn
```

- Xfatal-warnings Fail the compilation if there are any warnings.
- Xlint Enable recommended additional warnings.

```
$ scalac -Y |& grep warn
```

- Yinline-warnings Emit inlining warnings. (Normally suppressed due to high volume)
- Ywarn-adapted-args Warn if an argument list is modified to match the receiver.
- Ywarn-all Enable all -Y warnings.
- Ywarn-dead-code Warn when dead code is identified.
- Ywarn-inaccessible Warn about inaccessible types in method signatures.
- Ywarn-nullary-override Warn when non-nullary overrides nullary, e.g. `def foo()` over `def foo`.
- Ywarn-nullary-unit Warn when nullary methods return Unit.
- Ywarn-numeric-widen Warn when numerics are widened.
- Ywarn-value-discard Warn when non-Unit expression results are unused.

NoCompDaemon option

If you use the Scala compiler (scalac) on a laptop, be sure to always specify the `-nocompdaemon` option. It makes scalac run a little slower, but scalac won't hang. For Linux and Cygwin, put the following in `~/.profile`; for Mac put the following in `.bash_profile`:

```
alias scala='scala -nocompdaemon'
```

Installing SBT

The Simple Build Tool (which is far from simple!) is very useful when working with open source Scala projects. Version 1.3.0 was just released, but it is binary incompatible with the previous version (1.2.4), so download the previous version instead. Follow these instructions.

Once downloaded, type `sbt` at a command prompt and verify that `sbt` runs. It may download a lot of stuff the first time it runs.

```
$ sbt "show sbt-version"
[info] Loading global plugins from /home/mslinn/.sbt/plugins
[info] Loading project definition from /home/mslinn/work/training/projects/public_code/coreScala/course_
scala_intro_code/courseNotes/project
[info] Set current project to scalaIntroCourse (in build file:/home/mslinn/work/training/projects/public
_code/coreScala/course_scala_intro_code/courseNotes/)
[info] 0.12.3
```

1-3 The Scala Code Runner

ScalaCourses.com / scalaIntro / ScalaCore / scalaRunning / scalaCodeRunner

It is often helpful to be able to view the version of Scala that is installed on your computer:

```
$ scala -version
Scala code runner version 2.10.2 -- Copyright 2002-2013, LAMP/EPFL
```

You can read about the Scala runner's options like this:

```
$ scala -help
Usage: scala <options> [<script|class|object|jar> <arguments>]
    or  scala -help

All options to scalac (see scalac -help) are also allowed.
The first given argument other than options to scala designates
what to run. Runnable targets are:

- a file containing scala source
- the name of a compiled class
- a runnable jar file with a valid Main-Class attribute
- or if no argument is given, the repl (interactive shell) is started
```

Options to scala which reach the java runtime:

```
-Dname=prop  passed directly to java to set system properties
-J<arg>      -J is stripped and <arg> passed to java as-is
-nobootcp    do not put the scala jars on the boot classpath (slower)
```

Other startup options:

```
-howtorun    what to run <script|object|jar|guess> (default: guess)
-i <file>    preload <file> before starting the repl
-e <string>   execute <string> as if entered in the repl
-save        save the compiled script in a jar for future use
-nc          no compilation daemon: do not use the fsc offline compiler
```

A file argument will be run as a scala script unless it contains only self-contained compilation units (classes and objects) and exactly one runnable main method. In that case the file will be compiled and the main method invoked. This provides a bridge between scripts and standard scala source.

Options for plugin 'continuations':

```
-P:continuations:enable    Enable continuations
```

The next few lectures will show you how to run Scala code in various ways.

```
$ scalac -X |& grep warn
```

```
-Xfatal-warnings      Fail the compilation if there are any warnings.
```

```
-Xlint                Enable recommended additional warnings.
```

```
$ scalac -Y |& grep warn
```

```
-Yinline-warnings     Emit inlining warnings. (Normally suppressed due to high volume)
```

```
-Ywarn-adapted-args    Warn if an argument list is modified to match the receiver.
```

```
-Ywarn-all            Enable all -Y warnings.
```

```
-Ywarn-dead-code       Warn when dead code is identified.
```

```
-Ywarn-inaccessible    Warn about inaccessible types in method signatures.
```

```
-Ywarn-nullary-override Warn when non-nullary overrides nullary, e.g. `def foo()` over `def foo`.
```

```
-Ywarn-nullary-unit     Warn when nullary methods return Unit.
```

```
-Ywarn-numeric-widen   Warn when numerics are widened.
```

```
-Ywarn-value-discard   Warn when non-Unit expression results are unused.
```

1-4 The Scala Interpreter (REPL) and simple looping

ScalaCourses.com / scalaIntro / ScalaCore / scalaRunning / scalaREPL

REPL stands for: read, execute, print loop. In other words, a REPL is an interpreter. You can try out the Scala REPL by typing `scala`, at a command prompt:

```
$ scala
Welcome to Scala version 2.10.2 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_25).
Type in expressions to have them evaluated.
Type :help for more information.

scala> println("Hello, world!")
Hello, world!
```

We can define an immutable variable called `x` this way:

```
scala> val x = 3
x: Int = 3
```

The REPL first displays the name of the variable, followed by its type and then its value. You can also explicitly define the type of the variable, if you are concerned that someone reading your code might be unclear as to the type that might be assigned:

```
scala> val x: Int = 3
x: Int = 3
```

We can define a mutable value and change its value:

```
scala> var y = 4
y: Int = 4

scala> y = 36
y: Int = 36
```

We can also define a method and invoke it:

```
scala> def timesTwo(x: Int) = 2 * x
timesTwo: (x: Int)Int

scala> timesTwo(21)
res6: Int = 42
```

The Scala compiler deduces the return type of the method definition above as `Int`. It is generally a good habit to declare the return type of a variable, unless it is really obvious to a person reading your code:

```
scala> def timesTwo(x: Int): Int = 2 * x
timesTwo: (x: Int)Int
```

Exercise

Define a method that squares any `Int` passed to it, and test it on a variety of input values.

Let's define a range of integers containing values from 1 to three, inclusive. Note that the result is automatically stored in a variable called `res1`, of type `scala.collection.immutable.Range.Inclusive`:

```
scala> 1 to 3
res1: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3)
```

The range above was written with *infix notation*. This is possible because `to` is actually a method that takes a single parameter. Any method which takes a single parameter can be written with infix notation. We could have written the statement with postfix notation to get the same effect:

```
scala> 1.to(3)
res18: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3)
```

We could also do away with the syntactic sugar, and written the same range this way. Note that this is not any more efficient, and may be harder to read:

```
scala> scala.collection.immutable.Range.inclusive(1, 3)
res2: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3)
```

We could shorten the previous incantation using an `import` statement to import the `Range` class.

```
scala> import scala.collection.immutable.Range
import scala.collection.immutable.Range

scala> Range.inclusive(1, 3)
res19: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3)
```

We can also import the `inclusive` method from the `Range` class:

```
scala> import scala.collection.immutable.Range.inclusive
import scala.collection.immutable.Range.inclusive

scala> inclusive(1, 3)
res20: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3)
```

... and we can import all methods from a class:

```
scala> import scala.collection.immutable.Range._
import scala.collection.immutable.Range._
```

Now let's use the range to print out "Hello, world!" three times:

```
scala> 1 to 3 foreach { i => println("Hello, world!") }
Hello, world!
Hello, world!
Hello, world!
```

Notice that the Scala expression is parsed such that the range is computed before the foreach is executed. In other words, the above is equivalent to:

```
scala> (1 to 3) foreach { i => println("Hello, world!") }  
Hello, world!  
Hello, world!  
Hello, world!
```

Again, the above was written with infix notation. We could rewrite it using postfix notation (note the period between the range and foreach).

```
scala> (1 to 3).foreach { i => println("Hello, world!") }  
Hello, world!  
Hello, world!  
Hello, world!
```

The above is also equivalent to:

```
scala> Range.inclusive(1, 3).foreach { i => println("Hello, world!") }  
Hello, world!  
Hello, world!  
Hello, world!
```

Now let's display the value of `i` for each iteration of the loop:

```
scala> 1 to 3 foreach { i => println("Hello, world #" + i) }  
Hello, world #1  
Hello, world #2  
Hello, world #3
```

Here is a more convenient way of writing the same thing. The current instance of the `val i` is substituted for `$i` in the string because string interpolation is enabled for strings that are prefaced with the letter `s`.

```
scala> 1 to 3 foreach { i => println(s"Hello, world #${i}") }  
Hello, world #1  
Hello, world #2  
Hello, world #3
```

Other Control Statements

Scala has three control statements: `while`, `do while` and various types of `for` statements.

while

```
while (condition) {  
    statement(s)  
}
```

For example, we can write the previous example as:


```
var i = 1
while (i<=3) {
  println(s"Hello, world #$i")
  i = i + 1
}
```

Scala assignment returns Unit

Scala assignment returns `Unit`, unlike other languages like Java, where assignment returns the value assigned. This means `i = i + 1` cannot be used in a conditional expression:

```
while ((i = i + 1) <=3)
  println("THIS CODE WILL NOT COMPILE")
```

do while

```
do {
  statement(s)
} while (condition)
```

Here is the same example written using `do while`:

```
var i = 1
do {
  println(s"Hello, world #$i")
  i = i + 1
} while (i<=3)
```

for

For loops look a bit like for comprehensions, introduced later in this course. The key difference is that for loops do not contain the `yield` keyword:

```
for (i <- 1 to 3)
  println(s"Hello, world #$i")
```

Exercise

Define a method that accepts a `String` message and an `Int` that indicates the number of times to print it. Test the method with a variety of input values.

Exercise

What gets printed out? Why?

```
for (i <- 1 to 3) {  
  var i = 2  
  println(i)  
}
```

1-5 Scala Scripts and Calling Java from Scala

ScalaCourses.com / scalaIntro / ScalaCore / scalaRunning / scalaScripts

There are two ways to execute Scala code as a script. This lecture also shows how easy it is for Scala to call Java.

Scala File

This method requires that your Scala code be stored into a text file with the `scala` filetype. You can then execute that code by typing `scala filename.scala`. Let's try this by creating a file called `hello.scala` with the following inside:

```
println("Hello, world!")
```

To execute this file simply specify the full filename as a parameter to the Scala compiler driver.

```
$ scala hello.scala
Hello world!
```

If your Mac has a problem running the file, use `-nocompdaemon`:

```
$ scala -nocompdaemon hello.scala
```

Note that only Scala code was placed into the source file, and the source file was not made executable. Also note that the script was not defined as a program, it is just a collection of Scala source lines.

Exercise

Write a Scala script file that prints out the current directory each time it is run. You can obtain the current directory from the returned value of this incantation:

```
new java.io.File("").getAbsolutePath.toString
```

Try running the script from various directories.

Shell Script

Scala scripts can be used with *nix shells such as `bash`, `sh`, `zsh` and `csh`. They do not work from a Windows command prompt, however they do work from a Cygwin shell. Scala scripts can have any file type, or no file type. All you need to do is to put the following at the top of the file, and to make the file executable:

```
#!/bin/sh
exec scala "$@" "$@"
!#
```

Here is a simple example. Please make a file called `script2` somewhere on your computer and copy the following into it:

```
#!/bin/sh
exec scala "$@" "$@"
!#

println("Hello, world!")
```

Now make the file executable by typing the following at a shell prompt:

```
$ chmod a+x script2
```

Run the script like this:

```
$ ./script2
hello, world!
```

Exercise

Write a shell script that prints out the number of days to your birthday. If you had a birthday earlier this year, it is fine to print out the number of days since your birthday instead. As a hint, here is some code that computes the number of seconds to/since Christmas:

```
import java.util.Calendar

val xmas = Calendar.getInstance()
xmas.set(Calendar.MONTH, Calendar.DECEMBER)
xmas.set(Calendar.DAY_OF_MONTH, 25)

def secsUntilXmas: Long = (xmas.getTimeInMillis - System.currentTimeMillis) / 1000
```

Experiment in the REPL, then move your code into the shell script.

Here is a more complex example, which I won't explain in detail at this time. Note that the bash script is more complex. We also define a Scala console application and run it:

```
#!/bin/sh
SCRIPT="$(cd "${0%/*}" 2>/dev/null; echo "$PWD"/"${0##*/}")"
DIR=`dirname "${SCRIPT}"`
exec scala $0 $DIR $SCRIPT
::!#

import java.io.File

object App {
  def main(args: Array[String]): Unit = {
    val Array(directory,script) = args.map(new File(_).getAbsolutePath)
    println("Executing '%s' in directory '%s'".format(script, directory))
  }
}
```

Save the above into a file called script3. Make it executable and run it as follows:

```
$ chmod a+x script3
$ ./script3
Executing '/home/mslinn/work/training/projects/public_code/scalaCore/course_scala_intro_code/scripts/script3' in directory '/home/mslinn/work/training/projects/public_code/scalaCore/course_scala_intro_code/scripts'
```

1-6 Introduction to SBT

[ScalaCourses.com](#) / [scalaIntro](#) / [ScalaCore](#) / [scalaRunning](#) / [sbtIntro2](#)

Sbt uses a Maven-compatible directory structure:

```
+---project
+---src
|   +---main
|   |   +---java
|   |   +---resources
|   |   +---scala
|   +---test
|   |   +---java
|   |   +---resources
|   |   +---scala
```

As you can see, sbt can build projects containing both Scala and Java source code. Sbt can also use Maven repositories of Scala and Java libraries.

Git is almost always used with sbt projects.

Installation

Use these links for Linux, Mac and Windows. Feel free to read more if you are so inclined.

Cygwin

Adjust the path to `sbt-launch.jar` to suit your computer.

```
#!/bin/bash
export JAVA_HOME='e:/PROGRA~2/Java/jdk1.6.0_31_32bit'
export JAVA_OPTS="$JAVA_OPTS -Xss2m -Xmx512M -XX:MaxPermSize=128m -XX:+CMSClassUnloadingEnabled"
if [ $OSTYPE == cygwin ]; then
  # this is for colored output
  set CYGWIN=tty ntsec
  #export JAVA_OPTS="$JAVA_OPTS -Djline.terminal=jline.UnixTerminal"
  export JAVA_OPTS="$JAVA_OPTS -Djline.terminal=jline.UnsupportedTerminal"
fi
java $JAVA_OPTS -jar 'e:/storage/programming/scala/sbt-launch.jar' "$@"
```

Mac/Linux launch

Adjust the path to `sbt-launch.jar` to suit your computer. JDK7 works fine, so set `JAVA_HOME` accordingly if you prefer.

```
#!/bin/bash
export JAVA_HOME='/usr/lib/jvm/java-6-sun'
export JAVA_OPTS="$JAVA_OPTS -Xss2m -Xmx512M -XX:MaxPermSize=128m -XX:+CMSClassUnloadingEnabled"
java $JAVA_OPTS -jar '/opt/scala/sbt-launch.jar' "$@"
```

Global Setup

Sbt compiles a project builder program from .sbt and .scala files. Each time you run sbt, it executes all of the files in ~/.sbt, where you can put all of your global configuration for sbt. Although you can stuff everything in one file, it is better to split the commands across multiple little files so you can manage them better.

If you use Eclipse, create ~/.sbt/eclipse.sbt (for Windows including Cygwin, %USERPROFILE%/.sbt/eclipse.sbt) containing this line:

```
EclipseKeys.withSource := true
```

The contents of the ~/.sbt/plugins directory is separate from the preceding. Create ~/.sbt/plugins/build.sbt (for Windows including Cygwin, create %USERPROFILE%/.sbt/plugins/build.sbt), and put this inside

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "2.1.1")

addSbtPlugin("com.github.mpeltonen" % "sbt-idea" % "1.5.1")
```

Project Setup

I created a github project that contains all of this information, and more. You can use that git project as a template for your next sbt project.

```
git clone https://github.com/mslinn/sbtTemplate.git
```

Alternatively, you can create your project with this script:

```
#!/bin/bash

mkdir -p project src/{main,test}/{scala,java,resources}
touch src/{main,test}/{java,scala,resources}/.gitkeep

cat > build.sbt <<EOF
organization := "com.mycompany"

name := "changeMe"

version := "0.1.0-SNAPSHOT"

scalaVersion := "2.10.2"

scalacOptions ++= Seq("-deprecation", "-encoding", "UTF-8", "-feature", "-target:jvm-1.7", "-unchecked",
  "-Ywarn-adapted-args", "-Ywarn-value-discard", "-Xlint")
```

.sbt files must be double-spaced

```

javacOptions += Seq("-Xlint:deprecation", "-Xlint:unchecked", "-source", "1.7", "-target", "1.7", "-g:vars")

resolvers += Seq(
  "Typesafe Releases" at "http://repo.typesafe.com/typesafe/releases"
)

libraryDependencies += Seq(
)

//logLevel := Level.Error

// define the statements initially evaluated when entering 'console', 'console-quick', or 'console-project'
initialCommands := ""
"".stripMargin
EOF

echo 'addSbtPlugin("com.github.mpeltonen" % "sbt-idea" % "1.5.1")' > project/plugins.sbt

for d in `find src -type d`; do touch $d/.gitkeep; done

cat > .gitignore <<EOF
.classpath
*~
.cache
.idea/
.idea_modules/
.project
.settings/
*.iml
target/
EOF

git init
git add -A .gitignore *

```

You might want to save this as a script called `newSbt`. Don't forget to make it executable with `chmod a+x!`
 Use it like this:

```

$ mkdir myproj
$ cd myproj
$ newSbt
$ sbt gen-idea

```

Project Files

project/build.properties

This file is recommended. Use it to specify the version of SBT that works best with your project's dependencies:

```
sbt.version=0.13.0
```

build.sbt

The % is for cross-compiled libraries (which are compiled against multiple Scala versions) and automatically adds the Scala version to the artifact id.

Sample build.sbt

```
organization := "com.micronautics"

name := "MyProject"

description := "Blah blah blah"

version := "0.1.0"

scalaVersion := "2.10.2"

scalacOptions ++= Seq("-deprecation", "-encoding", "UTF-8", "-feature", "-target:jvm-1.7", "-unchecked",
  "-Ywarn-adapted-args", "-Ywarn-value-discard", "-Xlint")

javacOptions ++= Seq("-Xlint:deprecation", "-Xlint:unchecked", "-source", "1.7", "-target", "1.7", "-g:vars")

libraryDependencies ++= Seq(
  "org.scalatest" %% "scalatest" % "2.0" % "test" withSources(),
  "com.typesafe.akka" % "akka-actor" % "2.2.0" withSources()
)

// Optional settings from https://github.com/harrah/xsbt/wiki/Quick-Configuration-Examples follow
initialCommands := ""
""
```

project/plugins.sbt

```
addSbtPlugin("com.github.mpeltonen" % "sbt-idea" % "1.5.1")
```

You can add `offline := true` to your project settings to force sbt to not try to update any dependencies and use the ones it already has.

Exercise

Create an SBT project that prints out Hello, world!

Hints:

- Your project should not require any dependencies.
- You can run an SBT project by typing:

```
sbt run
```

- Your Scala program, a console app, could be called anything you like, so long as it has a .scala file type.
- Your scala program might look like this:

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

Update Eclipse and IDEA Projects From a New or Revised build.sbt.

See SBT / Global Setup for instructions on installing the sbteclipse and gen-idea plugins.gen-idea automatically runs update before emitting project files.

```
sbt gen-idea "eclipse with-source=true"
```

You can check the version of sbt by typing the following at an OS prompt. There is no need to change directory to an existing SBT project; this command can be typed in any time, except from your home directory:

```
sbt sbt-version
```

.sbtrc

Each line in .sbtrc and ~/.sbtrc is evaluated as a command before the project is loaded.

Muzzle SBT

Sbt is very noisy. You can suppress most of the noise by setting logLevel to Level.Error. Here is an example of how to generate an Eclipse project using the sbteclipse plugin, without spewing the usual reams of useless information.

```
sbt  
set logLevel := Level.Error  
eclipse with-source=true
```

Sbt allows everything to be specified on the command line:

```
sbt "; set logLevel := Level.Error; eclipse with-source=true"
```

Digging Deeper

Where is sbt-launch.jar Stored?

```
~/.sbt/.lib/$SBT_VERSION/sbt-launch.jar
```

For example, version 0.12.4 is stored here:

```
~/.sbt/.lib/0.12.4/sbt-launch.jar
```

Custom resolver

You can define the URL pattern for a new resolver and name it this way:

```
resolvers += {  
  val typesafeRepoUrl = new java.net.URL("http://repo.typesafe.com/typesafe/releases")  
  val pattern = Patterns(false, "[organisation]/[module]/[sbtversion]/[revision]/  
    [type]s/[module](-[classifier])-[revision].[ext]")  
  Resolver.url("Typesafe Repository", typesafeRepoUrl)(pattern)  
}
```

Useful SBT Commands

Following are commonly used commands. Here is a more complete list.

clean

Deletes files produced by the build, such as generated sources, compiled classes, and task caches. It does not remove all compiled artifacts. Here is deeper clean:

```
rm -rf target/*
```

compile

Compiles sources; automatically runs update first.

~compile

Continuously compiles sources each time a file changes.

console

Starts the Scala interpreter with the project classes on the classpath.

doc

Generates Scaladoc for src/main into target/api/index.html. See also test:doc.

offline

Configures SBT to work without a network connection where possible.

run

Runs a main class; compiles if necessary. Currently does not detect Java classes with static void main() methods.

~run

Runs a main class, recompiling each time you change a file. Currently does not detect Java classes

with static void main() methods.

run-main

Runs a main class, passing along arguments provided on the command line. For example:

```
sbt 'run-main com.micronautics.akka.dispatch.futureScala.Zip'
```

Windows does not understand single quotes, so you must use double quotes with Windows:

```
sbt "run-main com.micronautics.akka.dispatch.futureScala.Zip"
```

test

Executes all tests that are not marked with ignore.

test-only

Executes tests in one test class. For example:

```
sbt 'test-only com.blah.MyTest'
```

Windows does not understand single quotes, so you must use double quotes with Windows:

```
sbt "test-only com.blah.MyTest"
```

help command*

Displays help message or prints detailed help on requested commands.

update-classifiers

Download sources and javadoc for all dependencies

Playing with SBT Console

The courseNotes directory in the git repository provided with this course contains a sample project. That project defines several classes in the com.micronautics.scalaJava package, including one called ScalaClass3. Let's use the sbt console to play with that class. The first time you run that command it might take a long time to download many jars.

```
$ sbt console
[info] Loading global plugins from /home/mslinn/.sbt/plugins
[info] Loading project definition from /home/mslinn/work/training/projects/public_code/scalaCore/course_
scala_intro_code/courseNotes/project
[info] Set current project to scalaIntroCourse (in build file:/home/mslinn/work/training/projects/public
_code/scalaCore/course_scala_intro_code/courseNotes/)
Welcome to Scala version 2.10.2 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_25).
Type in expressions to have them evaluated.
Type :help for more information.

scala> import com.micronautics.scalaJava._
import com.micronautics.scalaJava._

scala> val sc3 = new ScalaClass3(1, 2.0)
sc3: com.micronautics.scalaJava.ScalaClass3 = prop1=01; prop2=2.0; prop3=02; prop4=6.0
```

Exercise

Now you can experiment live with your partially complete code base! Start the sbt console on the courseNotes project and see how it works.

Popular Plugins

- dependencyReport - Lists all dependencies (direct and transitive) of an SBT project
- sbt-assembly - Build executable jar with all dependencies
- sbt-dependency-graph - display an ASCII graph of hierarchical direct and transitive dependencies
- sbt-eclipse Eclipse integration • Google Group
- sbt-idea IntelliJ IDEA integration

```
addSbtPlugin("com.github.mpeltonen" % "sbt-idea" % "1.5.1")
```

1-7 Working With Scala IDE for Eclipse

ScalaCourses.com / scalaIntro / ScalaCore / scalaRunning / ScalaEclipse

Although you can install Eclipse and the Scala plug-in separately, we recommend that you install one of the preconfigured versions. The naming of the Scala plugin is rather odd - in mid-May 2013 the preconfigured Scala IDE was renamed to Scala SDK, which is misleading because this is an IDE, not an SDK. The preconfigured versions track the most recently official releases, and they are easy to work with. If you want to add the Scala plugin to an existing Eclipse installation, or you just like installing software as a way to pass time:

- Download any Indigo, Juno or Kepler Eclipse bundle with Java in it.
- Download one of the Scala 2.10.2+ plugins for Scala IDE 3.0.1. Make sure you select the appropriate Indigo, Juno or Kepler plugin from that page.

You are now ready to work with a Scala project using Eclipse. If you have a lot of memory, Eclipse will not use it unless you configure it to do so. The Scala compiler needs a lot more memory than the Java compiler, so you should configure the IDE right away.

Eclipse Configuration File

The preconfigured Eclipse/Scala bundle has settings increased from the default Eclipse memory configuration to the minimum suggested configuration for Scala. If you have more than 4GB RAM, you may want to increase the memory even further.

The configuration file is called `eclipse.ini` and for Windows and Linux it is found in your Eclipse directory; for Mac the file is within the `Eclipse.App/Contents/MacOS/` directory. Make sure you back up the file before making any changes. The highlighted lines are the only ones you should modify. Do not copy the entire file between Eclipse versions, because the plugins mentioned on the other lines change, and you will make Eclipse unbootable. Here is the file as provided with preconfigured Scala IDE for Indigo on Linux:

```
-startup
plugins/org.eclipse.equinox.launcher_1.2.0.v20110502.jar
--launcher.library
plugins/org.eclipse.equinox.launcher.gtk.linux.x86_64_1.1.100.v20110505
-vmargs
-Xmx1048m
-Xms100m
-XX:MaxPermSize=256m
```

Here is the setup I used when writing this course, which used a generic J2EE Juno Eclipse SR2 packaging with the Scala IDE added later. I added or modified the highlighted lines. You could make similar changes to your `eclipse.ini` file.

```
-startup
plugins/org.eclipse.equinox.launcher_1.3.0.v20120522-1813.jar
--launcher.library
plugins/org.eclipse.equinox.launcher.win32.win32.x86_64_1.1.200.v20120913-144807
-product
org.eclipse.epp.package.jee.product
--launcher.defaultAction
openFile
--launcher.XXMaxPermSize
256M
-showsplash
org.eclipse.platform
--launcher.XXMaxPermSize
256m
--launcher.defaultAction
openFile
-vmargs
-server
-Xmn128m
-Xss2m
-XX:+UseParallelGC
-XX:PermSize=256m
-XX:MaxPermSize=128m
-Dosgi.requiredJavaVersion=1.5
-Dhelp.lucene.tokenizer=standard
-Xms1512m
-Xmx2124M
```

You can use `jvisualvm`, provided with the Java Development Kit, to measure memory usage of the IDE, so you can optimize the memory settings. This lecture does not show how to do that, but the VisualVM site has many articles that discuss how to use the tool. I suggest that you do not use the VisualVM IDE plugin to measure the IDE that it runs from, and instead run the tool standalone!

Eclipse Configuration Settings

I recommend that you set `JAVA_HOME` right away. Under Mac and Linux, and Windows with Cygwin, you can launch Scala IDE like the following. Of course, the directory you installed into will likely be different:

```
/opt/eclipse/eclipse &> /dev/null &
```

You could add an alias to `.profile` for convenience:

```
alias eclipse='/opt/eclipse/eclipse &> /dev/null &'
```

Launch Eclipse. When it first starts it will ask you to select a workspace. The configuration settings that you select from within the program are stored here, along with project information. You should only use an Eclipse of the one vintage with a workspace. In other words, if you are using Eclipse Juno for Java work, and Eclipse Indigo for Scala work, you should use two workspaces. If you use a later version of Eclipse with a workspace created by an earlier Eclipse version, the workspace will no longer work properly with the earlier version of Eclipse. My workspaces are stored in `/home/mslinn/eclipseWorkspaces/{indigo, juno}`.

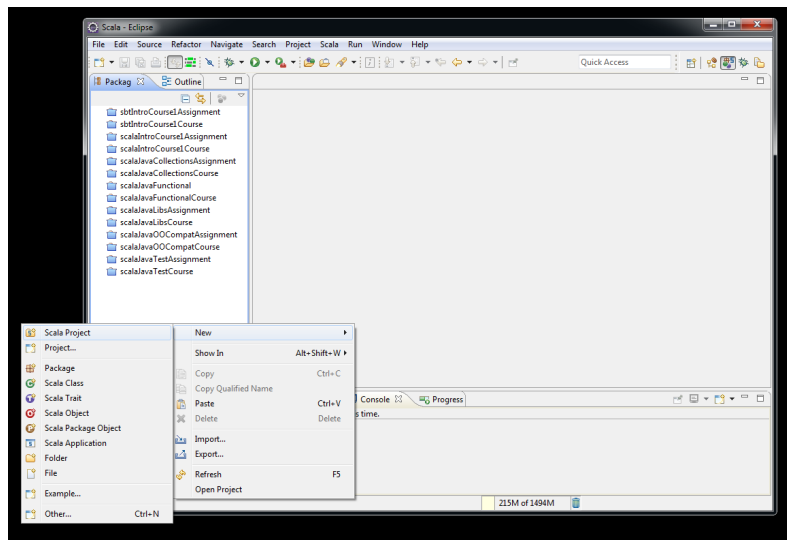
Use the **Window / Preferences** menu item to edit your Eclipse preferences (for Mac, use **Eclipse / Preferences** the menu item).

1. Enable **Always run in background** and **Show heap status**.
2. **General**
 1. **Editors**
 1. **Size of recently opened files list**: increase from 4 to something more reasonable, like 15.
 2. **Structured Text Editors / Task Tags**: enable **Enable searching for Task Tags**.
 3. **Text Editors**:
 1. Enable **Insert spaces for tabs**, **Show print margin** and **Show line numbers**. I set the print margin column to 150.
 2. **Spelling**: click the browse button to specify a file for your spelling dictionary. Eclipse and Thunderbird can use the same file.
 2. **Keys** - type in Scala in the search area and notice the Scala-related hotkeys that are predefined for you. You can change the definitions here.
 3. **Startup and Shutdown** - Enable **Refresh workspace on startup** and disable **Confirm exit when closing last window**.
 4. **Web Browser** - You can define your favorite browser here. For Mac, if you want to use Google Chrome, click the button and add the following:
 - Name: **Chrome**
 - Location: **/usr/bin/open**
 - Parameters: **-a "/Applications/Google Chrome.app" %url%**
 5. **Workspace**
 1. Enable **Refresh using native hooks or polling** and **Save automatically before build**.
 2. **Text file encoding** - Set to **Other: UTF-8**.
3. **Install / Update / Automatic Updates** - Enable **Automatically find new updates and notify me**. I also like to select **Download new updates automatically and notify me when ready to install them**.
4. **Scala**
 1. **Compiler** - Enable **deprecation** and **unchecked**. I also like to add the following to **Additional command line parameters**: **-Ywarn-adapted-args -Ywarn-value-discard -Xlint**
 2. You should explore the other Scala-related settings so you know what they are. I like the defaults just as they are.
5. **Scala Worksheet** - **Maximum number of output characters to be shown after evaluation**: 1000 characters is often not enough. Try 10000.

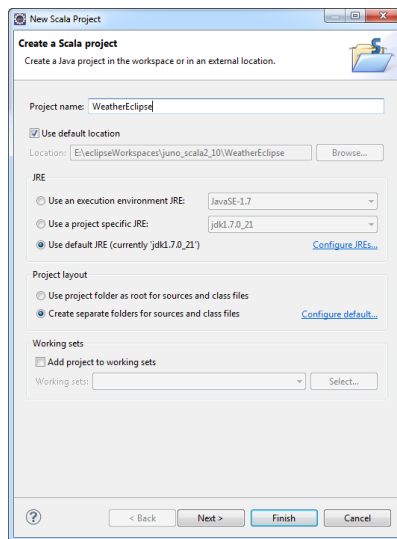
The preconfigured Eclipse with Scala in it does not have a git plugin. For most Scala programmers, git is essential. You should install the EGit plugin. Lars Vogel has an excellent tutorial on how to do this, and how to work with Git from Eclipse.

Creating a New Scala Project

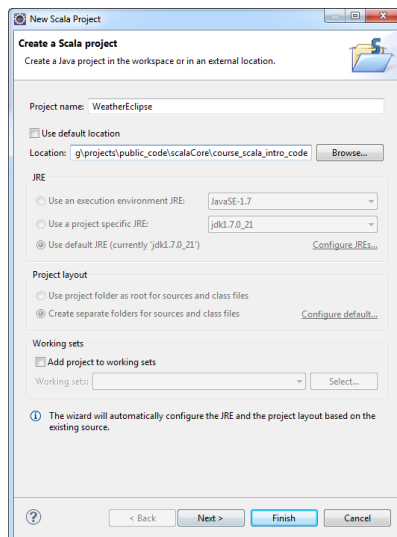
1) Start Eclipse / Scala IDE, and click on New / Scala Project .	
---	--



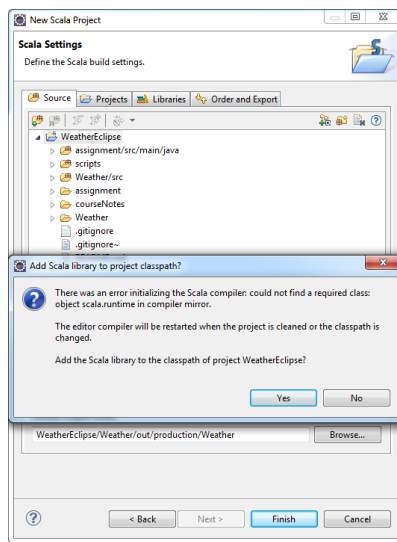
2) Give your new project a name (WeatherEclipse).



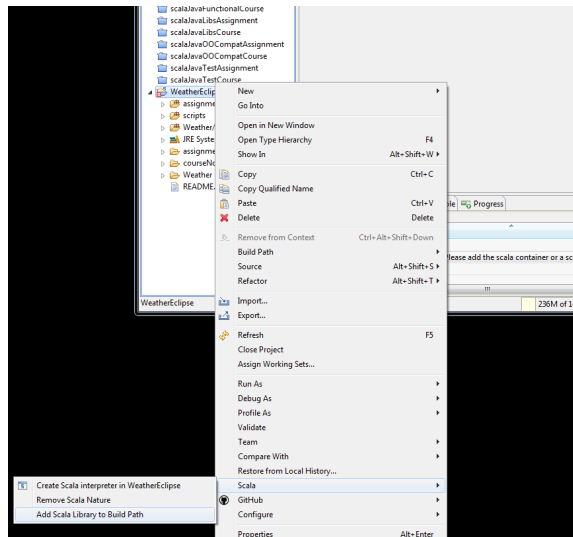
3) If there is a directory that you prefer to keep your work in (I recommend this!), uncheck **Use default location**, click **Browse...**, select the directory, and when you are back in this dialog, click **Next >**.



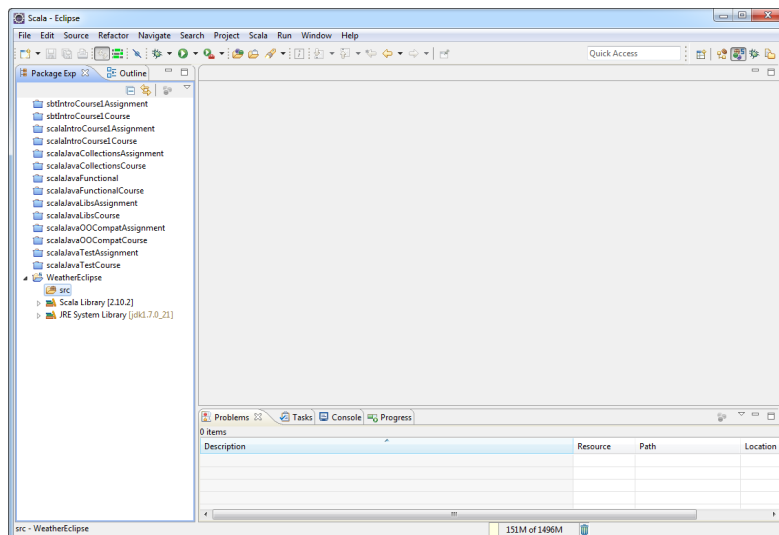
4a) If you did not define the `SCALA_HOME` environment variable you will be presented with this error message.



4b) To fix the missing Scala library problem, right-click on the project name in the Project panel, select the **Scala** item and click on **Add Scala Library to Build Path**.



6) The src directory is automatically created for you. This directory contains your source code, including Java and Scala code, and can also include resources.

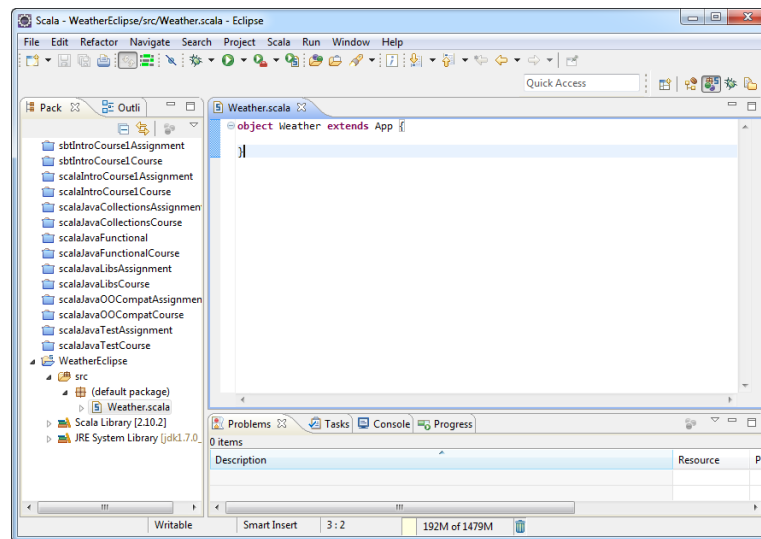


7) To make a source file in the src directory, which will contain the main program's

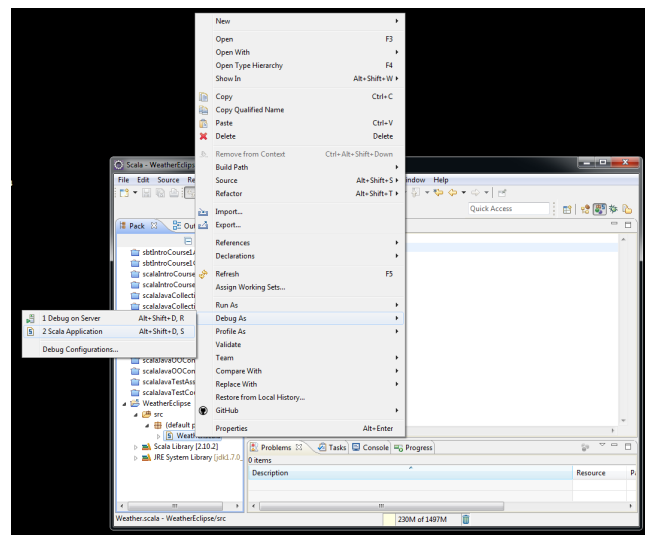
entry point, right-click on the src directory, select **New / Scala Object**.

8) Give your new Scala file a name (Weather). The Scala file will have a .scala filetype automatically added. Note that I defined the superclass to be App; this makes the constructor act as the main entry point. Click **Finish**

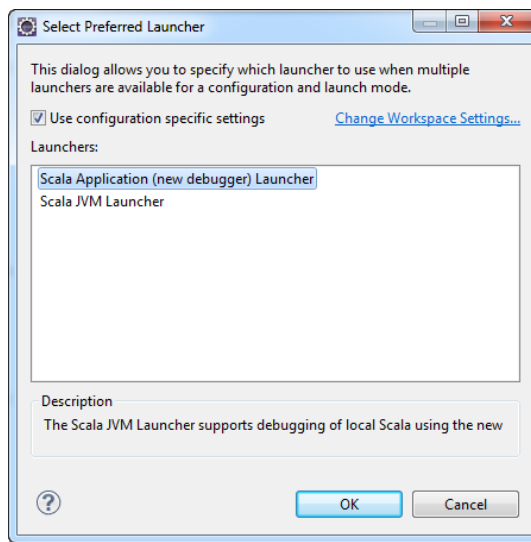
9) Eclipse creates a default Scala class with the name of the file.



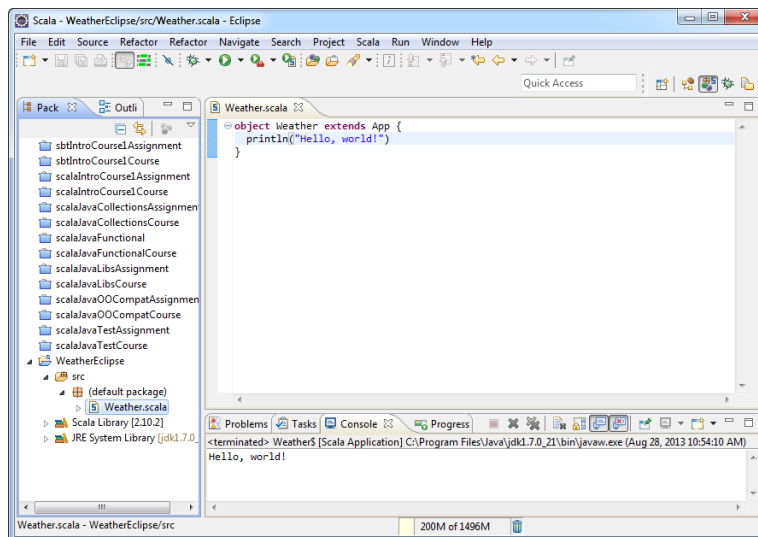
10) Let's run the newly created console app by right-clicking on it in the **Project** panel and selecting **Debug As / Scala Application**.



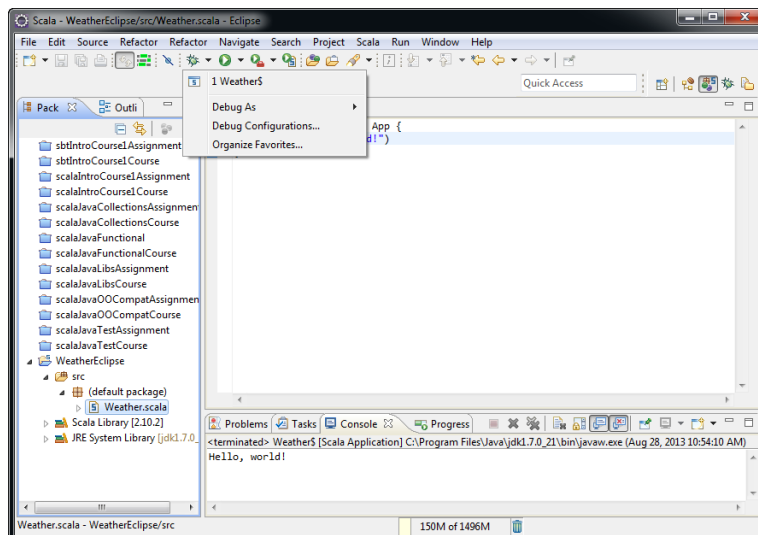
11) Enable **Use configuration specific settings** and select **Scala Application (new debugger) Launcher**.



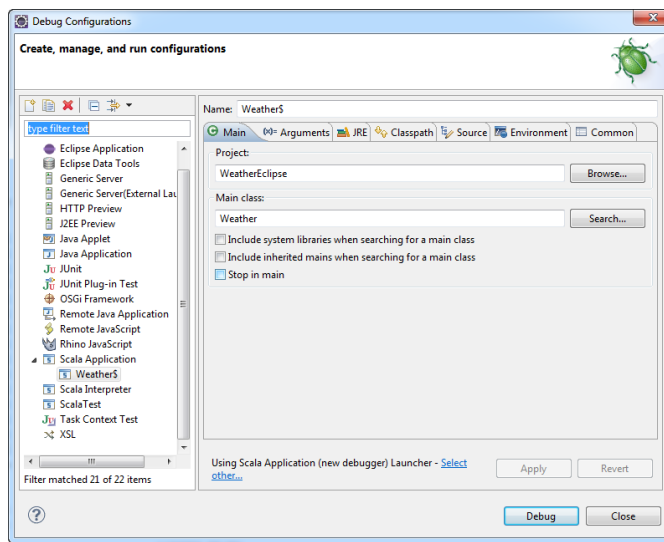
12) Output appears in the **Debug** panel which opens up at the bottom of the Eclipse window. You can click on the **Console** tab to show or hide the debug output. You can rerun the app by clicking on the little green bug at the top the Eclipse window.



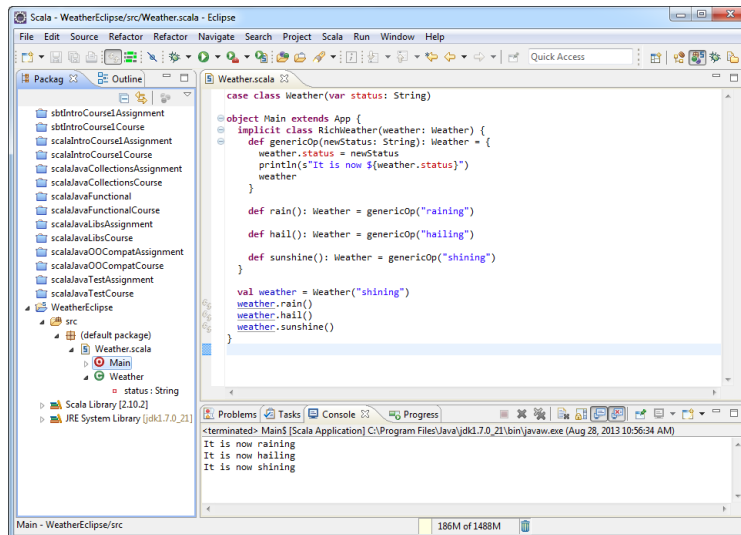
13) You can edit the debug configuration by clicking on the pull-down menu next to the little green bug at the top center of the Eclipse window.



14) You can add arguments for the JVM and your program, as well as tweak other settings.



15) A completed program.



Converting the Sample Code to Eclipse Projects

At the command line, navigate to the `course_scala_intro_code` directory. It contains two directories called `courseNotes` and `assignment`.

To convert the `courseNotes` sbt project to Eclipse format, open a bash shell or Windows `cmd` console, change to the `courseNotes` directory and type:

```
$ sbt eclipse
```

If this is the first time you run `sbt` with this project, you will have to wait several minutes while many dependencies are downloaded. Eventually you will see something like:

```
[info] Loading global plugins from C:\Users\Mike Slinn\.sbt\plugins
[info] Loading project definition from C:\scalaJavaInterop\ooCompat\courseNotes\project
[info] Set current project to scalaJavaOOCompatCourse (in build file:/C:/scalaJavaInterop/ooCompat/courseNotes/)
[info] About to create Eclipse project files for your project(s).
[info] Successfully created Eclipse project files for project(s):
[info] scalaJavaOOCompatCourse
```

Now change to the the `assignment` directory and run the same command:

```
$ sbt eclipse
```

Again, output should look something like:

```
[info] Loading global plugins from C:\Users\Mike Slinn\.sbt\plugins
[info] Loading project definition from C:\scalaJavaInterop\ooCompat\assignment\project
[info] Set current project to scalaJavaOOCompatAssignment (in build file:/C:/scalaJavaInterop/ooCompat/a
ssignment/)
[info] About to create Eclipse project files for your project(s).
[info] Successfully created Eclipse project files for project(s):
[info] scalaJavaOOCompatAssignment
```

Import the courseNotes Scala project into Eclipse:

1. Use the **File / Import / General / Existing Projects into Workspace** menu item (**Eclipse / Import / General / Existing Projects into Workspace** for Mac).
2. Click
3. Browse to the directory.
4. Click **Finish**.
5. You will see **Building workspace** on the Eclipse status line.

Scala Interpreter

To see the view in the current perspective, use **Window / Show View / Scala Interpreter**. To send selected code to the Scala Interpreter, highlight some Scala code and press - - .

Running Scala and Java Programs

You can run any of the Java programs in the project (.java files containing `public static main()` methods) by right-clicking on them in the **Package Explorer** and selecting **Debug As / Java Application**. Similarly, you can run any of the Scala programs in the project (.scala files containing `object extends App`) by right-clicking on them in the **Package Explorer** and selecting **Debug As / Scala Application**.

/ click on an item to see its definition.

Performance

To increase performance:

- Enable **Show Heap Status** in Preferences to see if used memory is close to the edge.
- Turn off **Mark occurrences** (- - on mac, - - on PC). This is one of the actions that happen on every keystroke.
- Turn off **semantic highlighting**
- Eclipse Indigo is faster than Juno. Kepler is somewhere in-between.
- If you experience slowness for specific files, you may have encountered a corner cases of the type checker (see <https://issues.scala-lang.org/browse/SI-5862>). There is no way to turn off type checking, and all type checking is done in the background thread. Sometimes the UI thread needs some result of type checking, and must wait.

Useful Eclipse Hot Keys

Ctrl + **Shift** + **W** **T** Shows the inferred type of the highlighted variable or expression.

Ctrl + **Space** Code completion.

Ctrl + **F** Search in current directory.

Hot Keys Also Available in IntelliJ

The following hot keys are defined in IntelliJ when you enable Eclipse key bindings.

Ctrl + **D** Delete line.

Ctrl + **E** and **Ctrl** + **Shift** + **E** Show list of recent files.

Ctrl + **H** Find in project, or other scope.

Ctrl + **/** Comment / uncomment current line (toggle).

Ctrl + **Shift** + **/** Comment / uncomment current selection (toggle).

F2 or **Ctrl** + **Shift** + **Space** Show type or method signature of method under cursor.

Alt + **Shift** + **K** Show default IntelliJ IDEA key bindings, requires **Shortcut Keys List** plugin.

Ctrl + **Shift** + **F** Reformat file or selected code.

Alt + **Shift** + **R** Rename artifact under cursor.

Ctrl + **Shift** + **R** List matching files anywhere in project, optionally open one. Can also specify a filter

1-8 Working With IntelliJ IDEA

ScalaCourses.com / scalaIntro / ScalaCore / scalaRunning / ScalaIDEA

You need the IDEA 12 Ultimate Edition to work with the projects provided with this course. JetBrains offers a 30 day trial for IDEA Ultimate. This lecture was produced with IDEA 12.1.3.

A word about the Scala compiler - scalac has been packaged many ways, and IDEA currently provides most of them. Regardless of how scalac is packaged, it needs a lot more memory than the Java compiler. In particular, large projects, or projects that use implicits heavily require more memory. The instructions below are intended to guide you through the most appropriate way of configuring the Scala compiler options. Some of the options currently available will be removed in the next major version of IDEA - this is good because the options that are being removed are not as effective as the newer options that have recently been added. The following instructions only discuss how to work with the Scala compiler configuration options that are expected to remain going forward. This lecture will be updated whenever configuration settings for Scala change with new releases of IDEA.

Configuring IDEA

Under Mac and Linux, and Windows with Cygwin, you can launch IDEA like the following. Be sure that `JAVA_HOME` is set. Of course, the directory you installed into will likely be different:

```
/opt/idea-IU-129.161/bin/idea.sh &> /dev/null &
```

When IDEA starts for the first time, it will run the **Initial Configuration Wizard**, and ask for the plugins that you want to enable. I have not noticed issues with having lots of unnecessary plugins enabled, and you can disable plugins easily at any time, so you should press the **Skip** button and move on.

1. When IDEA finally presents the **Welcome** panel, you will see an entry labeled **Configure**. Click on that button.
2. On the **Configure** panel click on **Plugins**. This is where you can download and install the Scala and sbt plugins.
3. If you are not using Play Framework 1.0, disable the **Playframework support** plugin.
4. Click on the **Browse Repositories...** button and scroll down until you see **Scala** and **SBT**. You could also click on **Install JetBrains plugin...**, which would not display the 3rd party SBT plugin.
5. Highlight both plugins and right-click, so you see **Download and Install**. If you are using Play 2.0, also install the **Play 2.0 Support** plugin (from the JetBrains repository) and the **RemoteCall** plugin (from the third party repository) and wait for the plugins to download. I do not recommend that you run plugin downloads in the background.
6. Close the **Browse Repositories** window.
7. Click **OK**.
8. Allow IntelliJ IDEA to restart.
9. Verify that the plugins were installed.
10. If you are running Ubuntu Linux, click on **Configure / Create Desktop Entry**.
11. You need to define a default project JDK before you can configure Scala. To do that:
 1. Click on **Project Defaults / Project Structure / Project**
 2. Click the button, then select **JDK**. Browse to your JDK. For Ubuntu Linux, this will be

under /usr/lib/jvm.

3. Click and then click on the back arrow.
12. **Project defaults / Project Structure** is where you specify the default JDK and Java language level.
 - a. I selected **JDK 1.7** and **language level 7.0 Diamonds, ARM, multi-catch, etc..** No need to specify anything else, just click **OK**.
13. Back up to the **Configure** menu, and click on **Settings**.
14. **Template Project Settings** - these settings can also be applied to an opened project by selecting **File / Project Structure** from an open project.
 - a. **Compiler**
 - I. Ensure that **Use external build** is set, which is the default.
 - II. It is safest (but a bit slower) to enable **Clear output directory on rebuild**.
 - III. You should enable **Compile independent modules in parallel**.
 - IV. The default **Compiler process heap size (Mbytes)** of 700MB is for non-Scala compilers. This memory is allocated on first use, so you can ignore this setting.
 - V. **Scala Compiler** - nothing to configure in external build mode.
 - b. **Inspections**
 - I. I turn these warnings off:
 - A. **HTML / File reference problems** (so Play templates do not have a lot of complaints over missing files)
 - B. **Scala / Method signature / Method with Unit result type defined like function**
 - C. **Scala / Method signature / Method with Unit result type defined with equals sign**
 - II. I turned these warnings on:
 - A. **Scala / General / Relative import**
 - c. **Scala**
 - I. **Imports**
 - A. I set **Class count to use import with '_'** to 9999 because I like explicit imports.
 - B. Set **Add full qualified imports**.
 - C. Unset **Import the shortest path for ambiguous references** because that setting can get you into trouble when you optimize imports.
 - II. **Worksheet**:
 - A. **Evaluation results length before line break**: I set this to 120.
 - B. **Output cutoff limit**: I set this to 100
 - d. **SBT**:
 - I. VM parameters can be whatever you want. Setting `Xmx` and `XX:MaxPermSize` will help compile large projects faster. My settings are:

`-Xmx2536M -XX:MaxPermSize=512M`
 - e. **Version Control**: Unset **Notify about VCS root errors**
15. **IDE Settings**
 1. **Appearance**:
 1. Enable **Show line numbers**
 2. Change the **Theme** to **Darcula**. IDEA will restart.

3. If you are used to the Eclipse IDE's shortcut keys, you can use them with IDEA by setting them from **File / Settings / Keymap**. Select the predefined **Eclipse** keymap.

2. **Editor** - uncheck **Allow placement of caret after end of line**.

1. **Appearance** - enable **Show method separators** so horizontal lines are drawn between methods
2. **Colors & Fonts** select **Scheme name Darcula**
3. If your monitor has a 16:9 or 16:10 aspect ratio, you may want to maximize vertical dimension for your editor. In that case, set **Editor Tab Appearance / Placement** to **Left** or **Right**. You can also reclaim some vertical space by disabling the toolbar by right-clicking on it and deselecting **Show Toolbar**. You can re-enable the toolbar with the **View / Toolbar menu** item.

3. **Scala**

1. Enable **Run compile server (in external build mode)**, which keeps the compiler 'warm' between compilations, thereby reducing the compile time.
2. Make sure the **JVM SDK** points to a valid JDK.
3. **JVM maximum heap size, MB**: This memory is used to convert the IDEA project model to something compatible with the Zinc server, and to parse error messages from the compiler. The lifespan of this memory allocation is only during a compilation, and should not need to be adjusted.

16. Click **OK**

17. Back arrow, you now see the Configure menu

Tuning Memory Allocation

You can use `jvisualvm`, provided with the Java Development Kit, to measure memory usage of the IDE, so you can optimize the memory settings. The VisualVM site has many articles that discuss how to use the tool. I suggest that you do not use the VisualVM IDE plugin to measure the IDE that it runs from, and instead run the tool standalone!

In the IDEA installation directory, edit `bin/idea.vmoptions`. My development machine has 32 GB RAM, and I use that memory to run several virtual machines simultaneously. Each VM has 7 GB RAM. Here are my settings:

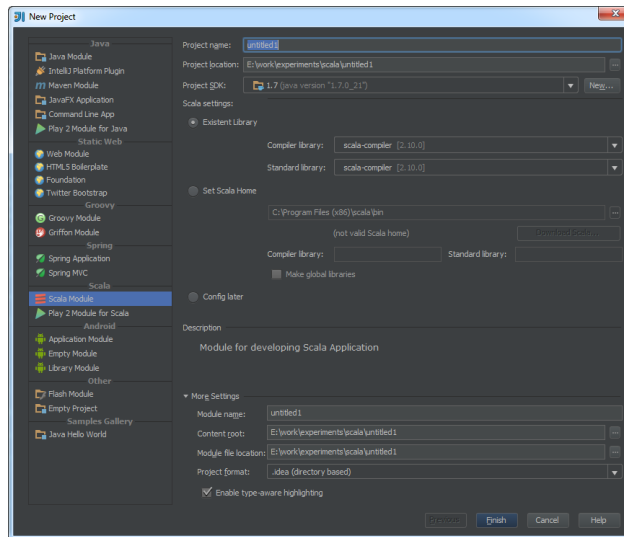
```
-Xms1128m
-Xmx2512m
-XX:MaxPermSize=500m
-XX:ReservedCodeCacheSize=164m
-XX:+UseCodeCacheFlushing
-ea
-Dsun.io.useCanonCaches=false
-Djava.net.preferIPv4Stack=true
```

Creating a Scala Project

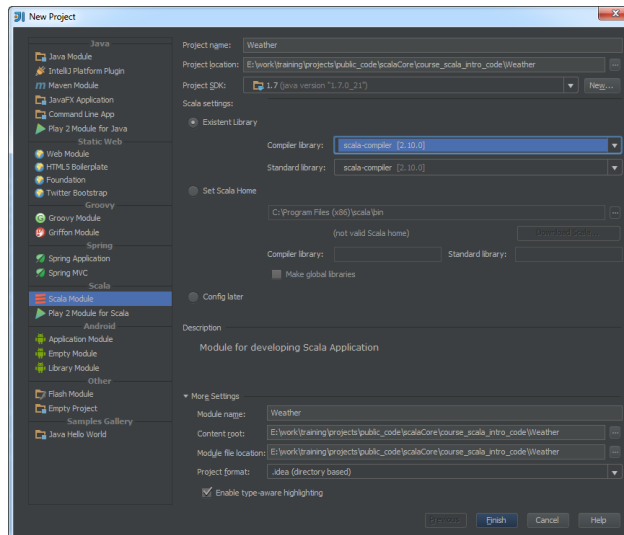
1) Start IntelliJ IDEA, and click on **Create New Project**.



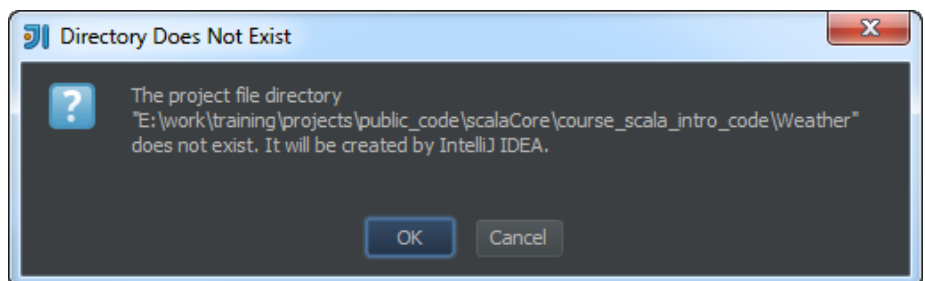
2) Give your new project a name.



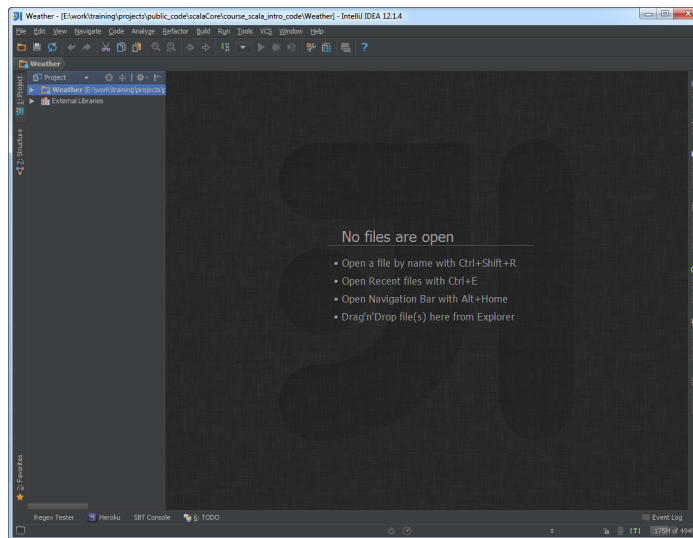
3) The project name will be used as the IntelliJ module name by default.



4) Click and you will be presented with this query. Click .

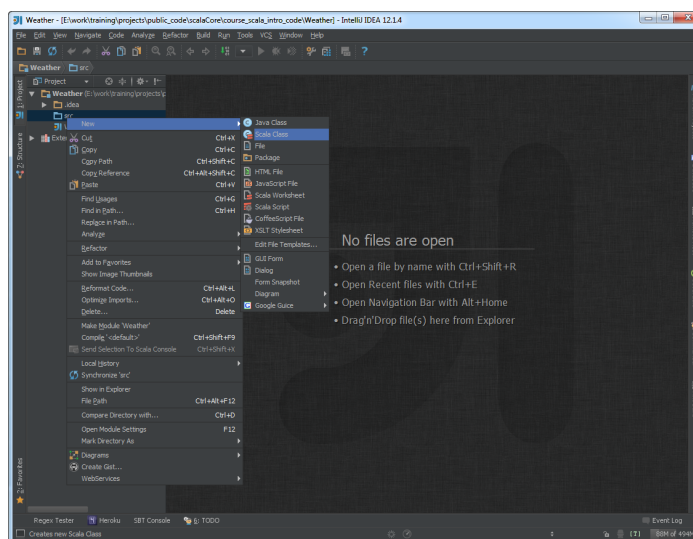


5) After a moment the **Project** panel will open up. Click on the module name to open it. The top one or two directories are the `.idea*` directories; normally you should not need to look inside.

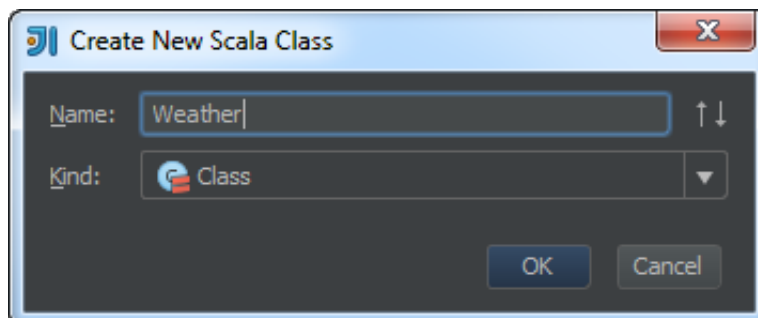


6) The `src` directory is automatically created for you. This directory contains your source code, including Java and Scala code, and can also include resources. To make a source file inside it, right-click on the `src` directory, select **New** and then **Scala Class**.

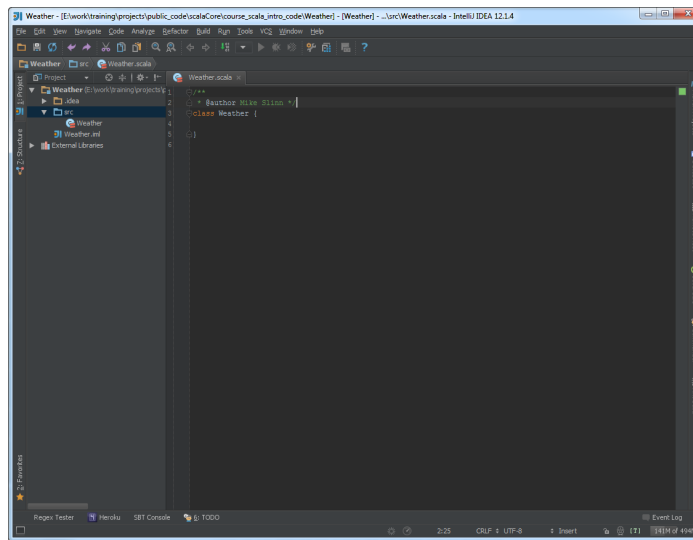
We select **Scala Class** even if we want to make a Scala object or a Scala case class.



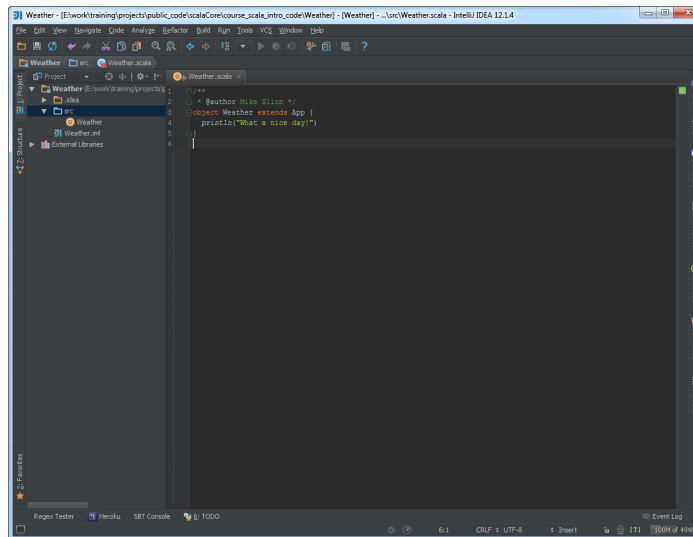
7) Give your new Scala file a name. The Scala file will have a `.scala` filetype automatically added.



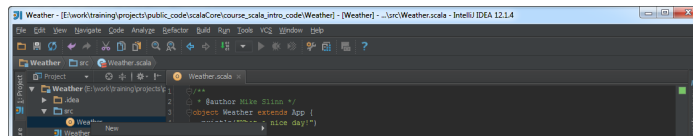
8) IDEA creates a default Scala class with the name of the file.



9) We need a main method for our console application. This means we need to extend an object (not a class) with the App trait. Change the word class to object and add extends App. The constructor for the object consists of only one line, which prints the traditional Hello, world!

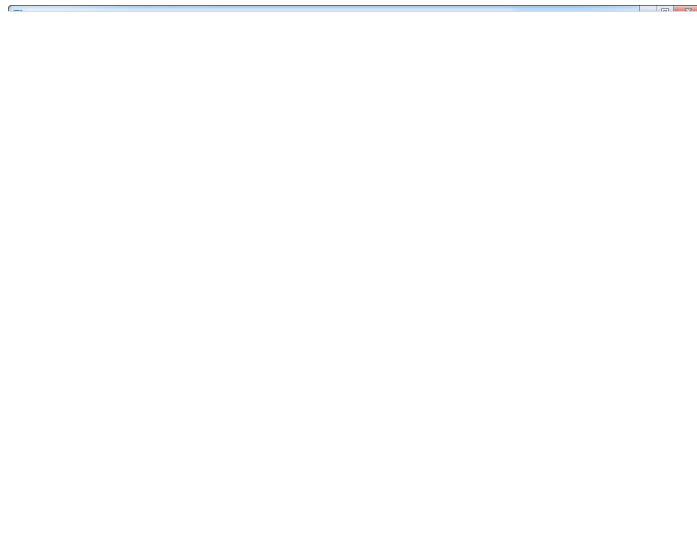


10) Let's run the console app by right-clicking on it in the **Project** panel and selecting **Debug Weather.main()**.

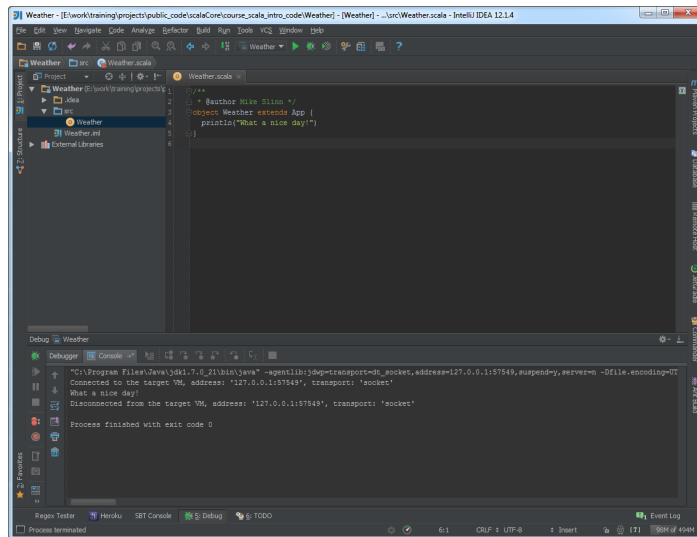


11) You are notified that an external Scala compiler is being used to compile the program. This is good. While

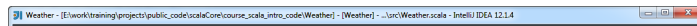
IDEA builds, you will see the status line at the bottom of the screen shows all the actions performed.



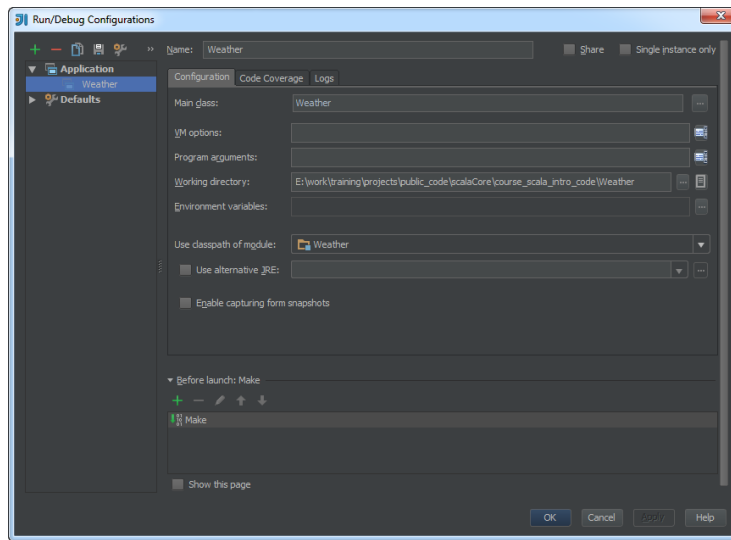
12) Output appears in the **Debug** panel which opens up at the bottom of the IDEA window. You can click on the **Debug** tab to show or hide the **Debug** panel. You can rerun the app by clicking on the little green bug on the left side of the **Debug** panel.



13) You can edit the debug configuration by clicking on the pull-down menu at the top center of the IDEA window.



14) You can add arguments for the JVM and your program, as well as tweak other settings.



15) If you enable **Share**, the debug configuration is written to a file in the .idea directory. You can check it into git if you want:

```
$ git add -fA .idea/runConfigurations/*  
$ git commit -m "New run configuration"  
$ git push
```

16) A completed program.

Converting SBT Projects Into IntelliJ Projects

At the command line, navigate to the `course_scala_intro_code` directory. It contains two directories, called `courseNotes` and `assignment`.

To export the `courseNotes` sbt project to IDEA, change to the `courseNotes` directory and run this command:

```
sbt gen-idea
```

If this is the first time you run sbt with this project, you will have to wait several minutes while many dependencies are downloaded. Eventually you will see something like:

```
[info] Loading global plugins from C:\Users\Mike Slinn\.sbt\plugins
[info] Loading project definition from C:\scalaJavaInterop\ooCompat\courseNotes\project
[info] Set current project to scalaJavaOOCompatCourse (in build
file:/C:/scalaJavaInterop/ooCompat/courseNotes/)
[info] Trying to create an Idea module scalaJavaOOCompatCourse
[info] Excluding folder target
[info] Created C:\scalaJavaInterop\ooCompat\courseNotes\.idea/IdeaProject.iml
[info] Created C:\scalaJavaInterop\ooCompat\courseNotes\.idea
[info] Excluding folder C:\scalaJavaInterop\ooCompat\courseNotes\target
[info] Created C:\scalaJavaInterop\ooCompat\courseNotes\.idea_modules\scalaJavaOOCompatCourse.iml
[info] Created
C:\scalaJavaInterop\ooCompat\courseNotes\.idea_modules\scalaJavaOOCompatCourse-build.iml
```

Now change to the the assignment directory and run the same command:

```
sbt gen-idea
```

Again, output should look something like:

```
[info] Loading global plugins from C:\Users\Mike Slinn\.sbt\plugins
[info] Loading project definition from C:\scalaJavaInterop\ooCompat\assignment\project
[info] Set current project to scalaJavaOOCompatAssignment (in build
file:/C:/scalaJavaInterop/ooCompat/assignment/)
[info] Trying to create an Idea module scalaJavaOOCompatAssignment
[info] Excluding folder target
[info] Created C:\scalaJavaInterop\ooCompat\assignment\.idea/IdeaProject.iml
[info] Created C:\scalaJavaInterop\ooCompat\assignment\.idea
[info] Excluding folder C:\scalaJavaInterop\ooCompat\assignment\target
[info] Created
C:\scalaJavaInterop\ooCompat\assignment\.idea_modules\scalaJavaOOCompatAssignment.iml
[info] Created
C:\scalaJavaInterop\ooCompat\assignment\.idea_modules\scalaJavaOOCompatAssignment-build.iml
```

The `gen-idea` command creates two directories, named `.idea` and `.idea_modules`. Both of these directories should be mentioned in the `.gitignore` file. You can examine the `.gitignore` files provided with the projects for this course.

When finished, go back to IDEA and **File / Open** the directory. The Scala project should load into IDEA. We can now finish setting up IDEA. You will notice a small rotating icon at the bottom of the IDEA window, and the word **Indexing...** next to it. The first time IDEA encounters a new jar, such as found with a new JDK, it will index the jar. This can take a while, and the process might slow down your computer noticeably.

Configuring IDEA Scala Projects

There is a vertical button at the upper left of the IDEA window, labeled **Project**. Click on that button and you will see the files and directories that comprise your project. You will see some directories that were not present when you created an IDEA project for Scala a moment ago. The project directory contains the sbt project files and directories. The External libraries directory is presented to you by IDEA so you

can browse the project dependencies. These files actually reside in `~/ivy2`. Note that there are two versions of the Scala compiler: 2.9.x and 2.10.x. The Scala 2.10.x build system still uses Scala 2.9.x but your code is compiled with Scala 2.10.x.

We need to configure the project structure before we can compile. **File / Project** structure opens a new window, and the Project SDK is probably invalid. You need to tell IDEA where your Java compiler is, so click the **New...** button, select **JDK** and navigate to the directory that you installed the JDK. Click the **Apply** button at the lower right of the window. We are done with Project Settings.

Helpful Hints

You may find that the editor does not parse complex Scala code or Play templates properly. If this happens you will see lots of good code highlighted as errors on the screen, and cutting and pasting may work improperly. You can temporarily toggle the Scala for the editor parser off and on by clicking in the small `[T]` symbol at the bottom right of the screen, in the status bar area. When Scala parsing is disabled, the symbol displays as `[_]`.

Useful IntelliJ Hot Keys

If you have set up Eclipse keyboard shortcuts as described, the following extra keyboard shortcuts are specific to IntelliJ IDEA. The **Key Promoter** and **Shortcut Keys List** plugins are helpful for learning IntelliJ keyboard hot keys.

- `Alt` + `=` Shows the inferred type of the highlighted variable or expression.
- `Ctrl` + hover Shows summary of all kinds of useful information about artifact under cursor.
- `Ctrl` + `Space` Code completion.
- `Ctrl` + `B` Rebuild project.
- `Ctrl` + `Shift` + `A` Find Action (learn key bindings, or do unbound actions)
- `Ctrl` + `Alt` + `V` Create a variable from an expression.
- `Ctrl` + `Shift` + `B` Toggle breakpoint on current line.
- `Alt` + `Shift` + `K` Show default IntelliJ IDEA key bindings, requires **Shortcut Keys List** plugin.
- `Ctrl` + `Shift` + `J` Join lines.
- `Ctrl` + `Shift` + `V` Past from 5 most recent copies.

Hot Keys From Eclipse Key Bindings

The following are some of the keys defined when you enable Eclipse key bindings.

- `Ctrl` + `D` Delete line.
- `Ctrl` + `E` and `Ctrl` + `Shift` + `E` Show list of recent files.
- `Ctrl` + `F` Highlight all occurrences of selected text, and optionally search in current directory.
- `Ctrl` + `H` Find in project, or if a directory is highlighted in the project pane, restrict search to that subdirectory tree.

`Ctrl` + `Alt` + `B` Pop up list of overrides or implementations, or go to implementation if there is only one.

`Ctrl` + `/` Comment / uncomment current line (toggle).

`Ctrl` + `Shift` + `/` Comment / uncomment current selection (toggle).

`F2` or `Ctrl` + `Shift` + `Space` Show type or method signature of method under cursor.

`Ctrl` + `Shift` + `F` Reformat file or selected code.

`Alt` + `Shift` + `R` Rename artifact under cursor.

`Ctrl` + `Shift` + `R` List matching files anywhere in project, optionally open one. Can also specify a filter

1-9 Worksheets

ScalaCourses.com / [scalaIntro](#) / [ScalaCore](#) / [scalaRunning](#) / [worksheet](#)

Scala worksheets are an imaginative melding of the Scala REPL and an IDE. Both the Eclipse and the IntelliJ worksheets are explored in this lecture. Note that IDEA worksheets and Scala-IDE worksheets are not interoperable unless you wrap an IDEA worksheet in an object `{ }`. Both types of worksheets have a `.sc` filetype, and both merely consist of the Scala code you entered.

Worksheets have all of the limitations of the REPL, however there is no `:paste` command to provide a workaround. This means you cannot define a companion object with a companion class in a worksheet. However, you can reference a companion class/object defined in your program because the worksheet inherits the project classpath. You can define worksheets in any type of IDE project, regardless if the project was created as a regular Eclipse or IDEA project, or the project was converted from an sbt project.

The `couresNotes` project contains a file called `ClassWithCompanion.scala` that defines a class called `ClassWithCompanion`, and a companion class. Those definitions are referenced in the worksheets.

It is good to save your worksheets in a separate directory as a reference. If you use worksheets you won't need to use the REPL much, if ever.

Scala-IDE (Eclipse) Worksheets

Scala-IDE worksheets contain the last evaluation values as comments for each line, and blank lines are inserted. Evaluation is automatic and quick.

IDEA Worksheets

IDEA worksheets do not contain the last evaluation values as comments, and evaluation is slow.

2 Object-Oriented Scala

ScalaCourses.com / scalaIntro / ScalaCore / scala00

2-1 Classes

ScalaCourses.com / scalaIntro / ScalaCore / scala00 / scalaClasses

Scala's object-oriented design is similar to Java. The same concept of classes and abstract classes exist. Java's interfaces have been made more powerful by Scala's traits; Java 8's interfaces will also be made more powerful in a similar fashion. Scala replaces Java's static fields with the concept of a companion object. Scala also has a convenient type of class definition, called a case class, which is just a class with a lot of methods added by default, and a companion object that has also been constructed with a standard set of methods.

Scala Classes

Let's explore how Scala classes work by using the REPL.

```
$ scala
Welcome to Scala version 2.10.2 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_21).
Type in expressions to have them evaluated.
Type :help for more information.
```

Next let's define an abstract class called `Animal` that accepts two parameters to the constructor.

```
scala> abstract class Animal(numLegs: Int, breathesAir: Boolean)

defined class Animal
```

This class is rather odd in that the parameters (`numLegs` and `breathesAir`) do not do anything, and then they are forgotten. BTW, these parameters are immutable by default. Let's redefine the `Animal` abstract class to compute a public property called `msg`. These two statements are actually the class constructor - in Scala, all statements in a class that are not a definition of some sort are part of the primary constructor. We'll talk more about constructors in the next lecture.

```
scala> abstract class Animal(numLegs: Int, breathesAir: Boolean) {
  private val breatheMsg = if (breathesAir) "" else " do not"
  val msg = s"I have $numLegs legs and I $breatheMsg breathe air"
}

defined class Animal
```

The parameters to the constructor of this abstract class are used to define a public property called `msg`. The public property `msg` is immutable, that is, one it assumes a value that value can never change. Immutable variables are defined with the `val` keyword. Mutable properties are defined with the `var` keyword.

We can extend this abstract class by defining an instance of an anonymous class using a syntax that reminds of Java. The following instance, stored in the `frog` variable, defines an immutable public property called `canSwim`.

```
scala> val frog = new Animal(4, true) { val canSwim: Boolean = true }  
frog: Animal{val canSwim: Boolean} = $anon$1@7cfcef1f
```

Now let's retrieve the value of the computed property `msg` from the `frog` instance.

```
scala> frog.msg  
res11: String = I have 4 legs and I breathe air
```

It is more common to define a concrete class that extends an abstract class. Let's define a concrete class called `Frog1` which extends the abstract `Animal` class. Note how two of the new `Frog1` class's constructor parameters are passed to the abstract class's constructor. `Frog1` also defines an immutable public property called `canSwim`.

```
scala> class Frog1(val canSwim: Boolean, numLegs: Int, breathesAir: Boolean) extends Animal(numLegs, breathesAir)
```

You can create an instance of a `Frog1` by passing in values for each constructor parameter.

```
scala> val frog1 = new Frog1(true, 4, true)  
frog1: Frog1 = Frog1@3b9e714c
```

Public properties are automatically defined with a setter and a getter. You can query a property like this:

```
scala> frog1.canSwim  
res6: Boolean = true
```

The other two constructor parameters cannot be queried because they are not defined to be properties. If you decorate them with `var` or `val` then they become properties and can be queried after an instance is created.

```
scala> class Frog2(val canSwim: Boolean, val numLegs: Int, val breathesAir: Boolean) extends Animal(numLegs, breathesAir)  
defined class Frog2  
  
scala> val tadpole = new Frog2(true, 0, false)  
tadpole: Frog2 = Frog2@794d6ef3  
  
scala> tadpole.numLegs  
res7: Int = 0  
  
scala> tadpole.breathesAir  
res8: Boolean = false
```

Note that when reading the code it is not obvious the mapping between constructor parameters and their values. What if you mixed up the boolean values, for example? That type of annoying bug is very common, especially when there are a lot of parameters to specify. Scala supports named parameters, which also allows the parameters to be specified in any order:

```
scala> val frog2b = new Frog2(breathesAir=true, numLegs=4, canSwim=true)
frog2b: Frog2 = Frog2@7af55600
```

None of the parameters was defined with default values, so they all must be specified. Here is what happens if you do forget to specify one of the constructor parameter values:

```
scala> val frog2c = new Frog2(breathesAir=true, numLegs=4)
<console>:9: error: not enough arguments for constructor Frog2: (canSwim: Boolean, numLegs: Int, breathe
sAir: Boolean)Frog3.
Unspecified value parameter canSwim.
    val frog2c = new Frog2(breathesAir=true, numLegs=4)
</console>
```

Now let's examine the properties of the `tadpole`. Note that we can use tab completion to list all the properties and methods of the instance.

```
scala> tadpole. Tab
asInstanceOf  breathesAir  canSwim        isInstanceOf  msg          numLegs      toString

scala> tadpole.canSwim
res0: Boolean = true
```

Now let's define a class with a mutable property, using the `var` keyword:

```
scala> class Frog3(val canSwim: Boolean, var numLegs: Int, breathesAir: Boolean) extends Animal(numLegs,
breathesAir)
defined class Frog3
```

Here is an instance of the class:

```
scala> val frog3 = new Frog3(canSwim=true, 4, breathesAir=true)
frog4: Frog4 = Frog4@6f97e1d1
```

And we can change the value of the mutable property:

```
scala> frog3.numLegs=4
frog6.numLegs: Int = 4
```

... but not the immutable properties:

```
scala> frog3.canSwim=false
<console>:10: error: reassignment to val
    frog3.canSwim=false
```

We can define an instance of an anonymous class that method using a syntax that is similar to Java's. Here we attempt to override a standard method called `toString()`. We get the error because `breathesAir` is merely a constructor parameter, not a public property, and is therefore not available once the class instance has been created.

```
scala> val frog3b = new Frog3(canSwim=true, 4, breathesAir=true) { override def toString() = s"$canSwim; $numLegs legs; $breathesAir" }
<console>:9: error: not found: value breathesAir
    val frog3b = new Frog3(true, 4, true) { override def toString() = s"$canSwim; $numLegs legs; $breathesAir" }

scala> val frog3c = new Frog3(canSwim=true, 4, breathesAir=true) { override def toString() = s"$canSwim; $numLegs legs" }
frog3c: Frog3 = Frog5@6f97e3b2

scala> frog3c
res7: Frog3c = true; 4 legs
```

This results in a similar class instance:

```
scala> class Frog4(val canSwim: Boolean, var numLegs: Int, breathesAir: Boolean) extends Animal(numLegs, breathesAir) { override def toString() = s"$canSwim; $numLegs legs" }
defined class Frog4

scala> val frog4 = new Frog4(canSwim=true, 4, breathesAir=true)
frog4: Frog4 = true; 4 legs
```

Let's define a similar class, but give each of the constructor parameters a default value.

```
scala> class Frog5(val canSwim: Boolean=true, var numLegs: Int=4, breathesAir: Boolean=true) extends Animal(numLegs, breathesAir) { override def toString() = s"$canSwim; $numLegs legs" }
defined class Frog5

scala> val frog5 = new Frog5
frog5: Frog5 = true; 4 legs
```

Exercise

Define a new `Animal` subclass called `Bird`. Give it a `Boolean` constructor property called `canFly`, and another `Double` property called `topSpeed`. Create a few `Bird` instances with different top speeds.

2-2 Auxiliary Constructors for Classes

ScalaCourses.com / scalaIntro / ScalaCore / scala00 / scalaAuxCons

Classes always have a primary constructor, which is simply the body of the class, and it can also have auxiliary constructors, which are methods called `this()`. Auxiliary constructors must call the primary constructor and thereby supply values for each of its parameters that do not have default values defined. The primary constructor is also referred to as `this()`, but with a different method signature. Let's modify the abstract class `Animal` from the previous lecture and add an auxiliary constructor:

```
abstract class Animal(numLegs: Int, breathesAir: Boolean) {  
  private val breatheMsg = if (breathesAir) "" else "do not"  
  val msg = s"I have $numLegs legs and I $breatheMsg breathe air"  
  
  def this() = this(2, breathesAir=true)  
}
```

Let's use the auxiliary constructor to create an anonymous class that is a concrete implementation of this abstract class:

```
scala> val animal = new Animal(){}  
res15: Animal = $anon$1@4af84f28  
  
scala> animal.msg  
res16: String = I have 2 legs and I breathe air
```

Note that an auxiliary constructor must call another constructor on its first line. To avoid huge one-liners in auxiliary constructors, you might want to swap the primary constructor with an auxiliary constructor. I also defined a helper class in the companion object.

```
object Animal2 {  
  def shazam(numLegs: Int, breathesAir: Boolean) = {  
    val breatheMsg = if (breathesAir) "" else "do not"  
    s"I have $numLegs legs and I $breatheMsg breathe air"  
  }  
}  
  
abstract class Animal2(val msg: String) {  
  def this(numLegs: Int, breathesAir: Boolean) = this(Animal2.shazam(numLegs, breathesAir))  
}
```

We need to learn about the REPL's paste mode in order to define a companion object interactively:


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

object Animal2 {
  def shazam(numLegs: Int, breathesAir: Boolean) = {
    val breatheMsg = if (breathesAir) "" else "do not"
    s"I have $numLegs legs and I $breatheMsg breathe air"
  }
}
abstract class Animal2(val msg: String) {
  def this(numLegs: Int, breathesAir: Boolean) = this(Animal2.shazam(numLegs, breathesAir))
}
^D
// Exiting paste mode, now interpreting.

defined module Animal2
defined class Animal2

scala> val animal2 = new Animal2("asdf"){ }
animal2: Animal2 = $anon$1@68199fc8

scala> animal2.msg
res21: String = asdf
```

Exercise

Add an auxiliary constructor to the Bird you defined in the previous lecture's exercise. The auxiliary constructor will accept the same parameters as the primary constructor, but if it is created at a time with an even minute (e.g. 3:00pm, 3:02pm, 3:04pm...) boost its maximum speed by 10%.

Hint: you can get the current minutes of the hour as a String with the following incantation:

```
new java.text.SimpleDateFormat("mm").format(new java.util.Date)
```

2-3 Setters and Getters

ScalaCourses.com / scalaIntro / ScalaCore / scala00 / scalaSettersGetters

Decorating a Scala class constructor argument with `var` or `val` causes it to become a public property of the class. Decorating with `val` makes the property immutable, while decorating with `var` makes it mutable.

The convention for Scala getters is to create a method named after the property with no arguments. Scala setters are methods that are also named after the property, with `_=` appended. Let's look at some sample Scala code:

```
class GetSetExample {  
  private var pn = ""  
  
  def name = pn // getter  
  
  // setter can be called three ways, shown below  
  def name_=(n: String) { pn = n }  
}
```

Let's define an entry point called `GetSetDemo` that can be invoked as a console application.

```
object GetSetDemo extends App {  
  val x = new GetSetExample  
  x.name_="fred"    // although this syntax is possible  
  x.name_$eq("steve") // and this syntax is also possible  
  x.name = "george"  // this syntax is more commonly used  
  println(x.name)  
}
```

Scala implements assignment (`=`) as a method, called `$eq()`, and also follows this convention for setters. The above is equivalent to the following Scala code:

```
class GetSetExample {  
  var name = ""  
}  
  
object demo {  
  val x = new GetSetExample  
  x.name_$eq("steve")  
  x.name = "fred"  
  println(x.name)  
}
```

2-4 Companion Objects

ScalaCourses.com / scalaIntro / ScalaCore / scala00 / scalaCompanion

Companion objects are singletons where you can define static fields and methods. In order for them to have their magic activated, they must be defined in the same file as a regular class of the same name. For example, let's redefine Frog1 and give it a companion object:

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

class Frog1(val canSwim: Boolean, numLegs: Int, breathesAir: Boolean)

object Frog1 {
  def apply(canSwim: Boolean=true, numLegs: Int=4, breathesAir: Boolean=true) = new Frog1(canSwim, numLegs, breathesAir)
}
^D
// Exiting paste mode, now interpreting.

defined class Frog1
defined module Frog1
```

If both of these definitions were placed in the same file then Frog1 would be a companion object of the Frog1 class, and the Frog1 class would be a companion class to the Frog1 object. In Scala, methods called apply are default methods, and are often used as constructors. Because they are default methods, you don't have to use their name when invoking them. For example, you could create a new Frog1 instance with either of the following statements; both are equivalent:

```
scala> val frog1a = Frog1(canSwim=true)
frog1a: Frog1 = Frog1@5041a39c

scala> val frog1b = Frog1.apply(canSwim=true)
frog1b: Frog1 = Frog1@6bd2b352
```

Notice that the apply method that I defined default values for all parameters, so the default values of numLegs and breathesAir were used because those parameters were not specified. Also notice that when a companion object is defined, the its name (Frog1) refers to the singleton instance.

If you want the methods and properties defined in the companion object to be available in the companion class, you must import them:

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

class Frog4(val canSwim: Boolean, numLegs: Int, breathesAir: Boolean) {
  import Frog4.croak

  def makeNoise = croak(3)
}

object Frog4 {
  def apply(canSwim: Boolean=true, numLegs: Int=4, breathesAir: Boolean=true) = new Frog4(canSwim, numLegs, breathesAir)

  def croak(times: Int): String = ("Croak " * times).trim
}

// Exiting paste mode, now interpreting.

defined class Frog4
defined module Frog4

scala> Frog4(canSwim=true, 4, breathesAir=true).makeNoise
res0: String = "Croak Croak Croak"
```

Unapply

Unapply is a method, defined in a companion object, that is automatically called when the Scala compiler needs to extract values from a Scala class or case class (we will talk about case classes soon). Unapply is therefore useful for pattern matching. Case classes automatically define `apply` and `unapply`, however you can define your own for regular classes, and you can define more for case classes:

```

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

class Fraction(var numerator:Int, var denominator:Int) {
  def *(fraction: Fraction) = Fraction(numerator*fraction.numerator, denominator*fraction.denominator)

  override def toString = s"$numerator/$denominator"
}

object Fraction {
  def apply(numerator: Int, denominator: Int) = new Fraction(numerator, denominator)

  def unapply(fraction: Fraction) = if (fraction==null) None else Some(fraction.numerator, fraction.denominator)
}
^D

// Exiting paste mode, now interpreting.

defined class Fraction
defined module Fraction

scala> println(Fraction(3,4) * Fraction(2,4))
6/16

scala> val Fraction(numer, denom) = Fraction(1,4) * Fraction(4,5)
numer: Int = 4
denom: Int = 20

```

Exercise

The following class definition has an incomplete `unapply` method in the companion object. Can you work with the REPL to figure out a way to parse the input `String` such that a `Some(Tuple2)` containing the first and last name is returned in the event of a successful parse, or `None` is returned if the parse fails? We will discuss tuples later in this course. For now, let's just accept that we need to create a tuple in the event of a successful parse.

Hints:

1. `String` has an `indexOf(search: String)` method that could be used to detect the space between the first and last name
2. `String` has a `split(delim: String)` method that could be used to create an `Array[String]` containing tokens from the original `String`
3. You can create a `Some(Tuple2)` this way: `Some("first", "last")`. Tuples are often represented as an extra level of parentheses. If you try this in the REPL you should see the following:

```

scala> Some("first", "last")
res9: Some[(String, String)] = Some((first,last))

```

Without further ado, here is the class and its companion object, which you need to complete:

```

class Name(val first: String, val last: String)

object Name {
  def unapply(input:String) = {
    // TODO write me!
  }
}

val Name(firstName, lastName) = "Fred Flintstone"

```

When the Scala interpreter encounters the last line above, it looks for an `unapply` method in `Name`'s companion object with the proper signature. In this case, it looks for an `unapply` method that accepts a `String`.

As a final hint, the REPL needs to receive a class and its companion object simultaneously in order to consider them as residing in the same file. Use the REPL's `:paste` command to accomplish this.

Answer:

```

class Name(val first: String, val last: String)

object Name {
  def unapply(input:String): Option[Tuple2[String, String]] = {
    val stringArray = input.split(" ")
    if (stringArray.length>=2)
      Some(stringArray(0), stringArray(1))
    else
      None
  }
}

object Main extends App {
  val Name(firstName, lastName) = "Fred Flintstone"
  println(firstName + " " + lastName)
}

```

2-5 Case Classes

ScalaCourses.com / scalaIntro / ScalaCore / scala00 / scalaCaseClasses

Case classes are great for building domain models and pattern matching. We won't explore those uses in this lecture - for now, we'll just take our first look at case classes. Each case class is merely a regular class with some standard methods defined, plus an automatically defined companion object with standard methods defined. In addition, each of the class's constructor parameters are also automatically defined as an immutable property, just as if the `val` keyword had prefixed each parameter. To define a case class, merely preface the `class` keyword with `case`. Let's use the REPL to experiment with case classes:

```
scala> case class Frog9(canSwim: Boolean, numLegs: Int, breathesAir: Boolean)
defined class Frog9
```

This is equivalent to:

```
case class Frog9b(val canSwim: Boolean, val numLegs: Int, val breathesAir: Boolean)
```

If you want a property to be mutable you must preface it with the `var` keyword:

```
case class Frog10(canSwim: Boolean, var numLegs: Int, breathesAir: Boolean)
```

The case class's automatically generated companion object defines the `apply` method such that you don't have to use the `new` keyword in order to create an instance:

```
scala> val frog9 = Frog9(canSwim=true, 4, breathesAir=true)
frog9: Frog9 = Frog9(true,4,true)
```

Notice that I named the boolean parameters. This is a good habit to get into. Even better, in this case, would be to name every parameter.

Case classes also automatically define an `unapply` method in the companion object:

```
scala> val Frog9(swimmer, legCount, airBreather) = frog9
swimmer: Boolean = true
legCount: Int = 4
airBreather: Boolean = true
```

Exercise

You can augment a case class's companion object simply by defining new methods. Your task is to write a Scala script called `frog9` that define another `unapply` method for `Frog9`'s companion object which accepts a `String` (or an `Array[String]`) and parses it into a `Option[Tuple3]`, which means a successful parse will return `Some((canSwim, numLegs, breathesAir))` or `None` if the parse fails:

```
object Frog9 {
  def unapply(input: String): Option[Tuple3] = // TODO write me

  def unapply(input: Array[String]): Option[Tuple3] = // TODO OR write me
}
```

Note that the above definition does not replace the automatically created companion object or its unapply method, it augments the existing definition. Test your code like this:

```
val Frog9(swimmer1, legCount1, airBreather1) = "true 4 true"
val Frog9(swimmer2, legCount2, airBreather2) = "true 0 false"
```

OR if you opted for the second version of unapply:

```
val Frog9(swimmer1, legCount1, airBreather1) = Array("true", "4", "true")
val Frog9(swimmer2, legCount2, airBreather2) = Array("true", "0", false)
```

The Scala script should read the arguments from the command line, and display the parsed values like this:

```
$ frog9 true 4 true
swimmer = true
legCount = 4
airBreather = true

$ frog9 true 0 false
swimmer = true
legCount = 0
airBreather = false
```

Standard Methods of Case Classes

Case classes are guaranteed to have at least the following methods defined:

- `copy` - Copies a case class instance while allowing named properties to be modified

```
scala> val frog9b = frog9.copy(canSwim=false)
frog9b: Frog9 = Frog9(false,4,true)
```

- `canEqual` - indicates if two objects can be compared.

```
scala> Frog9(true, 4, true).canEqual(Frog10(true,4,true))
res1: Boolean = false

scala> frog9 canEqual frog9b
res10: Boolean = true
```

- `equals` - compare two objects. No error is given if they should not be compared - in that case, `false` is quietly returned.


```
scala> frog9.equals(frog9b)
res8: Boolean = false

scala> frog9 equals frog9b
res9: Boolean = false
```

- hashCode - A digest stores a hash of the data from an instance of the class into a single hash value

```
scala> frog9.hashCode
res11: Int = 272580090
```

- productArity - a count of the number of constructor properties of the case class

```
scala> frog9.productArity
res15: Int = 3
```

- productElement - retrieve the untyped value of the nth case class constructor property

```
scala> frog9.productElement(0)
res17: Any = true

scala> frog9.productElement(1)
res18: Any = 4

scala> frog9.productElement(2)
res19: Any = true
```

- productIterator - return an iterator of the case class constructor properties

```
scala> frog9.productIterator.foreach(println)
true
4
true
```

- productPrefix - Simply the name of the case class

```
scala> frog9.productPrefix
res31: String = Frog9
```

- toString - string representation of the case class instance. Often overridden.

```
scala> frog9.toString
res32: String = Frog9(true,4,true)

scala> frog9
res33: Frog9 = Frog9(true,4,true)
```

Case class companion objects are guaranteed to have at least the following methods defined:

- apply - the default method for the companion object, which is a factory that calls new to construct new instances.
- toString - prints the name of the case class. You normally should use the instance's toString method instead.
- unapply - useful for pattern matching and value extraction. It is usually called behind the scenes so little gems like this can be written:

```
scala> val Frog9(x, y, z) = frog9
x: Boolean = true
y: Int = 4
z: Boolean = true

scala> val Frog9(a, _, b) = frog9
a: Boolean = true
b: Boolean = true
```

While an underscore is used as a wildcard for much of Scala, in this context an underscore means that parsed value need not be stored.

2-6 Uniform Access Principle

ScalaCourses.com / scalaIntro / ScalaCore / scala00 / scalaUniform

Declare an immutable property with the `val` keyword, like this:

```
scala> val x = 42
x: Int = 42

scala> x = 2
<console>:8: error: reassignment to val
      x = 2
      ^
```

Declare a mutable property with the `var` keyword, like this:

```
scala> var y = 42
y: Int = 42

scala> y = 3
y: Int = 3
```

Define a method with the `def` keyword, like this:

```
scala> def echo(what: String): Unit = println(what)
echo: (what: String)Unit

scala> echo("Hi there!")
Hi there!
```

It is convenient but incorrect to think of `var` and `val` as fields. Scala enforces the Uniform Access Principle by making it impossible to refer to a field directly. Wikipedia says:

The Uniform Access Principle was put forth by Bertrand Meyer. It states "All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation". This principle applies generally to object-oriented programming languages. In simpler form, it states that there should be no difference between working with an attribute, precomputed property, or method/query.

All accesses to any field are made through getters and setters. When the Scala compiler encounters a `val` or `var` in your source code it emits a field and a getter for the field. For a `var` the Scala compiler also emits a setter for the field. Let's look at an example:

```
scala> class Person1 { val name = "Fred Flintstone" }  
defined class Person1  
  
scala> val person1 = new Person1  
person1: Person1 = Person1@7ddaca36  
  
scala> println(person1.name)  
Fred Flintstone
```

The above outputs identical results as:

```
scala> class Person2 { def name = "Fred Flintstone" }  
defined class Person2  
  
scala> val person2 = new Person2  
person2: Person2 = Person2@26fda07e  
  
scala> println(person2.name)  
Fred Flintstone
```

The only difference is that the `def` is evaluated every time, while the `val` is evaluated once.

Scala getters and setters are methods, so they are defined in the same namespace as all other methods. This means you cannot have a property with the same name as a method. This also means that when subclassing, a `val` or a `var` can override a `def`. For example, here we see a `val` overriding a `def`:

```
abstract class AbstractPerson {  
  def name: String  
  def printName() = println(name)  
}  
  
class Person3 extends AbstractPerson { val name = "Jumping Jack Flash" }
```

Let's see what the REPL shows us:

```
scala> val person3 = new Person3  
person3: Person3 = Person3@19661df0  
  
scala> person3.printName  
Jumping Jack Flash
```

Here we see a `var` overriding a `def`:

```
abstract class AbstractPerson2 {  
  def name: String  
  def printName() = println(name)  
}  
  
class Person4 extends AbstractPerson2 { var name = "Jane Danger" }
```

Let's see what the REPL shows us:

```
scala> val person4 = new Person4
person4: Person4 = Person4@33d15244

scala> person4.printName
Jane Danger
```

So this means you should declare properties in base classes with `def`, so the implementations can optimize by overriding with `val` when appropriate. Note that `def` cannot override `var` or `val`.

2-7 Functions are First Class

ScalaCourses.com / scalaIntro / ScalaCore / scala00 / scalaFunctions

```
scala> def multiplyFive: Int => Int = 5 *  
multiplyFive: Int => Int
```

```
scala> def addThree: Int => Int = 3 +  
addTwo: Int => Int
```

```
scala> (addThree compose multiplyFive)(4)  
res1: Int = 23
```

Uniform access principle:

```
scala> val multiplySix: Int => Int = 6 *  
multiplyFive: Int => Int
```

```
scala> val addFour: Int => Int = 4 +  
addTwo: Int => Int
```

```
scala> (addFour compose multiplySix)(4)  
res1: Int = 28
```

2-8 Pattern Matching 101

ScalaCourses.com / scalaIntro / ScalaCore / scala00 / scalaMatch

Scala has a multi-way branch, just like many other languages. Let's see how simple value matches work, similar to a switch statement in other languages.

```
scala> val x = "b"
x: String = b

scala> x match {
  case "a" => 1
  case "b" => 2
  case "c" => 3
  case _ => 0
}
res3: Int = 2
```

You can match on any object, including numbers and Strings. Because every Scala statement returns a value, each of the clauses of the match statement should be of the same type, or the Scala compiler will widen the return type as necessary.

Notice that the last match value was a wild card, specified with an underscore, which means that the value is discarded. It is often a good practice to ensure that every possible match is provided for.

Exercise

Assign the result of a match to a variable. The input to the match could of different types - a String, an Int, a Double or a Char. Match against the type.

1. Return the `toString` value of the input. If you simply return `input.toString`, you are cheating.
2. What would be the type of the variable holding the value returned by the match statement, if the matched value was returned instead of the `toString` value?

You can also match against types, and extract values from those types. To see how this works, let's reuse some of the classes we defined earlier, and add one more. Note that I defined `Frog` and `Dog` as case classes for convenience when pattern matching.

```
abstract class Animal(numLegs: Int, breathesAir: Boolean) {
  private val breatheMsg = if (breathesAir) "" else " do not"
  val msg = s"I have $numLegs legs and I $breatheMsg breathe air"
}

case class Frog(canSwim: Boolean, numLegs: Int, breathesAir: Boolean) extends Animal(numLegs, breathesAir)

case class Dog(barksTooMuch: Boolean) extends Animal(4, true)
```

Now we can figure out what kind of `Animal` we have. `match` works by successively comparing the animal type against all the cases, in order. Notice that the first match case also checks the Frog's `numlegs` value; this value is available only if `unapply` succeeded and the match was made. Also notice that the last match (x) has no type - this means it will match against anything.

```
def classify(animal: Animal): Unit = animal match {  
  case frog: Frog if frog.numlegs>0 =>  
    println(s"Got a Frog with ${frog.numLegs} legs; canSwim=${frog.canSwim} and breathesAir=${frog.breat  
hesAir}")  
  
  case tadpole: Frog =>  
    println(s"Got a tadpole without legs; breathesAir=${tadpole.breathesAir}")  
  
  case dog: Dog =>  
    println(s"Got a Dog and $barksTooMuch=${dog.barksTooMuch}")  
  
  case x =>  
    println(s"Got an unexpected animal $x")  
}
```

The `classify` method simply prints the appropriate message. `Unit` is the Scala equivalent of Java's `null`, and `println` returns `Unit`. We explicitly defined the `classify` method to return `Unit` for the reader's convenience. Let's try it out:

```
scala> val frog1 = Frog(canSwim=true, 4, breathesAir=true)  
frog1: Frog = Frog(true,4,true)  
  
scala> classify(frog1)  
Got a Frog with 4 legs; canSwim=true and breathesAir=true  
  
scala> val tadpole = Frog(canSwim=true, 0, breathesAir=false)  
tadpole: Frog = Frog(true,0,false)  
  
scala> classify(tadpole)  
Got a Frog with 0 legs; canSwim=true and breathesAir=false  
  
scala> val bigDog = Dog(barksTooMuch=false)  
bigDog: Dog = Dog(false)  
  
scala> classify(bigDog)  
Got a Dog and barksTooMuch=false  
  
scala> val yapper = Dog(barksTooMuch=true)  
yapper: Dog = Dog(true)  
  
scala> classify(yapper)  
Got a Dog and barksTooMuch=true
```

Here is an example of how we can use pattern matching to extract values from case class instances. In this case, `match` uses the specified class's `unapply` methods to see if the values can be extracted. If `Frog.unapply` and `Dog.unapply` both fail, the last case will match because it has no class or extraction. You

could also specify an underscore instead of `x`, if you did not want to reference the value.

```
def extract(animal: Animal): Unit = animal match {  
  case Frog(canSwim, numLegs, breathesAir) if frog.numLegs>0 =>  
    println(s"Got a Frog with $numLegs legs; canSwim=$canSwim and breathesAir=$breathesAir")  
  
  case Frog(canSwim, numLegs, breathesAir) =>  
    println(s"Got a tadpole without legs; breathesAir=${breathesAir}")  
  
  case Dog(barksTooMuch) =>  
    println(s"Got a Dog and barksTooMuch=$barksTooMuch")  
  
  case x =>  
    println(s"Got an unexpected animal $x")  
}
```

Notice that the output for the `extract` method is the same as for the `classify` method.

Exercise

We accomplished the same task two different ways. When might you want to use an extractor, and when might you simply want to match against types or values? What is the difference between the matched values?

```
scala> extract(tadpole)  
Got a Frog with 0 legs; canSwim=true and breathesAir=false  
  
scala> extract(bigDog)  
Got a Dog and barksTooMuch=false  
  
scala> extract(yapper)  
Got a Dog and barksTooMuch=true
```

Aliases

There are some special notations used in the Scala's pattern matching. Scala allows to put aliases on patterns or on parts of a pattern. The alias is put before the part of the pattern, separated by `@`. For example, the expression `address @ Address(_, _, "Paris", "France")`, we define a filter that finds Addresses in Paris/France and returns the result in the alias `address`.

```

case class Address(street: String, street2: String, city: String, country: String)

val addresses = List(
  Address("123 Main St", "Apt 3", "Yourtown", "MD"),
  Address("234 Rue Blue", "Apt 5", "Fontaineblue", "France"),
  Address("543 Toulouse", "Apt 6", "Paris", "France")
)

addresses foreach { _ match {
  case address @ Address(_, _, "Paris", "France") => println(address.street)
  case _ =>
  }
}

```

Sequences

Scala provides a notation to match sequences elements with `_*`. It matches zero, one or more elements in a sequence to its end.

```

def hasLeadingZero(list: List[Int]) =
  list match {
    case List(0, _) => true
    case _ => false
  }

```

Of course, a better way to do this would be:

```

def hasLeadingZero(list: List[Int]) = list.head == 0

```

Putting aliases and sequence element matching together:

```

def isReadme(string: String): Boolean = string.toLowerCase.startsWith("readme")

val mergeStrategy = List("a", "b", "README.md")

mergeStrategy match {
  case List("reference.conf") => "MergeStrategy.concat"
  case List(ps @ _) if isReadme(ps.last) => "MergeStrategy.rename"
  case _ => ""
}

```

This is a simpler way of writing the same thing:

```
def isReadme(string: String): Boolean = string.toLowerCase.startsWith("readme")

val mergeStrategy = List("a", "b", "README.md")

mergeStrategy match {
  case List("reference.conf") => "MergeStrategy.concat"
  case ps: List[String] if isReadme(ps.last) => "MergeStrategy.rename"
  case _ => ""
}
```

2-9 Scala Traits

ScalaCourses.com / scalaIntro / ScalaCore / scala00 / scalaTraits

We say that traits are 'mixed in'. Multiple traits can be mixed into a class or object. Trait constructors cannot have parameters.

Pure Trait

A pure trait has no implementation and is equivalent to a Java Interface. Note that I have defined the property in the Checkable trait below with `def`, even though you might expect it to normally only require a `val`. Better safe than sorry!

```
import java.sql.Date

trait Checkable { def preFlight: Boolean }

case class Course(
  startDate: Date = new Date(System.currentTimeMillis),
  override val preFlight: Boolean = false
) extends Checkable
```

Now let's create some instances of the class `Course` which mixes in the pure trait `Checkable`:

```
scala> val course1 = Course(new Date(System.currentTimeMillis), false)
course1: Course = Course(2013-08-22,false)

scala> val course2 = Course()
course2: Course = Course(2013-08-22,false)
```

You can view the object as an instance of any of its traits, using `asInstanceOf`:

```
scala> course1.asInstanceOf[Checkable]
res12: Checkable = Course(2013-08-22,false)
```

Trait with Implementation

A trait with an implementation is similar to an abstract class, but traits cannot have a constructor that takes arguments.

```
import java.sql.Date

trait Checkable { def preFlight: Boolean }

trait HasId { def id: Option[Long] = None }

case class Lecture(
  override val id: Option[Long] = None,
  startDate: Date = new Date(System.currentTimeMillis),
  override val preFlight: Boolean = false
) extends Checkable with HasId
```

Now let's create some instances:

```
scala> val lecture1 = Lecture(Some(1), preFlight=true)
lecture1: Lecture = Lecture(Some(1),2013-08-22,true)

scala> val lecture2 = Lecture()
lecture2: Lecture = Lecture(None,2013-08-22,false)
```

Again, you can view an object as an instance of any of its traits, using `asInstanceOf`:

```
scala> lecture1.asInstanceOf[Checkable]
res13: Checkable = Lecture(Some(1),2013-08-22,true)

scala> lecture1.asInstanceOf[HasId]
res14: HasId = Lecture(Some(1),2013-08-22,true)
```

Extending Multiple Traits

Objects and classes can extend multiple traits. The first trait or class being extended is referenced with the keyword `extends`. All other traits are referenced with the keyword `with`. The order of the traits can be significant - in the following example, the

`type Shuttle extends Spacecraft with ControlCabin with PulseEngine` is not equivalent to the
`type Shuttle extends Spacecraft with PulseEngine with ControlCabin`.

```

abstract class Spacecraft { def engage: Unit }

trait Bridge {
  def speedUp: Unit
  def engage: Unit = 1 to 3 foreach { _ => speedUp }
}

trait Engine { def speedUp }

trait PulseEngine extends Engine {
  var currentPulse = 0;
  def maxPulse: Int

  def speedUp: Unit = if (currentPulse < maxPulse) currentPulse += 1
}

trait ControlCabin {
  def increaseSpeed
  def engage = increaseSpeed
}

class Shuttle extends Spacecraft with ControlCabin with PulseEngine {
  val maxPulse = 10
  def increaseSpeed = speedUp
}

trait WarpEngine extends Engine {
  def maxWarp: Int
  var currentWarp: Int = 0

  def toWarp( x: Int ): Unit = if (x < maxWarp) currentWarp = x
}

class Explorer extends Spacecraft with Bridge with WarpEngine {
  val maxWarp = 10

  def speedUp: Unit = toWarp(currentWarp + 1)
}

object Defiant extends Spacecraft with ControlCabin with WarpEngine {
  val maxWarp = 20

  def increaseSpeed = toWarp(10)

  def speedUp: Unit = toWarp(currentWarp + 2)
}

```

Self Traits and Dependency Injection

Self traits are used for dependency injection. Here is a great article covering many different forms of dependency injection in Scala. The difference between a self type and extending a trait is this: if you say B extends A, then B *is* an A. For dependency injection, you only want B to *require* A, not to *be* an A. For example, let's define a User trait, and a Tweeter trait that needs to be able to assume that the object is a User:

```
scala> trait User { def name: String }
defined trait User

scala> trait Tweeter { user: User => def tweet(msg: String) = println(s"$name: $msg") }
defined trait Tweeter

scala> case class Blabber(name: String) extends User with Tweeter
defined class Blabber
```

Some observations about this code:

- Because Blabber is a case class, the name parameter to the class constructor is also an immutable public property.
- Blabber's name property fulfills the contract of the User trait, in that a name property must be defined.
- The the User trait defined the name property as a def, in accordance with best practices for the uniform access principal
- The Tweeter trait does not extend User, so the Tweeter trait does not fulfill the User contract. In other words, a Tweeter is not a User, however it requires that the object which extends Tweeter must be a User. That means that the implementation details of User are not exposed in the Tweeter trait.

Self Traits and Structural Types

Self types can extend structural types:

```
type Openable = { def open(): Unit }
type Closeable = { def close(): Unit }

trait Door { self: Openable with Closeable =>
  def doSomething(f: () => Unit): Unit = try {
    open()
    f()
  } finally {
    close()
  }
}

class FrontDoor extends Door {
  def open(): Unit = println("Door is open")

  def walkThrough(): Unit = doSomething { () => println("Walking through door") }

  def close(): Unit = println("Door is closed")
}
```

Let's play with this in the REPL:

```
scala> val frontDoor = new FrontDoor
frontDoor: FrontDoor = FrontDoor@595b5d6b

scala> frontDoor.walkThrough()
Door is open
Walking through door
Door is closed
```

This trait assumes that the object which it extends, referred to as `self`, has methods called `open()` and `close()`. This allows for safe mixins for duck typing. Remember that structural types are somewhat inefficient, so don't write this type of code in a loop that gets called a lot.

Exercise

Given the following class that models how people express their feelings:

```
case class Person(name: String) { def speak(feelings: String) = println(feelings) }
```

We also have a trait that provides a mode of expression if the `Person` instance is angry. This trait adds a method called `growl` to the `Person` when mixed in:

```
trait Angry { self: Person =>
  def growl = self.speak("I'm having a bad day!!")
}
```

Use the REPL or write a Scala script to:

1. Mix the `Angry` trait into the `Person` case class, and get an instance of the `Person` class to `growl`.
2. Define a `Happy` trait and get the `Person` instance to `laugh`.

2-10 Type Hierarchy

ScalaCourses.com / scalaIntro / ScalaCore / scala00 / scalaTypes

All Scala objects are object, including numbers, strings and functions. The diagram shows the hierarchy. There are two main classifications of Scala types:

1. Value types, such as the primitive types `Boolean`, `Int` and `Unit`. The base class of all value types is `AnyVal`. Value types are immutable. `Unit` can also be written as `()`.
2. Reference types, such as Scala and Java objects. The base class of all reference types is `AnyRef`, which is equivalent to `java.lang.Object` and Scala's `System.Object`. Reference types are created using the `new` keyword. `Null` is a subtype of all reference types.

`Nothing` is a subtype of all types. In other words, `Nothing` is a subtype of everything!

`Any` is the base type of the Scala type system, and defines final methods like `==`, `!=` and `asInstanceOf`. `Any` also defines non-final methods like `equals`, `hashCode` and `toString`. Declaring an object to be of type `Any` effectively removes all type checking for that object.

Object Equality

`==` and `!=` use value equality as defined by `equals`, not reference equality like Java or C#. Let's see how that works. Below is a `Dog` case class that we want to define equality for. We use a case class because they define a method called `canEqual`, which informs us if two objects can be compared:

```
scala> case class Dog(name: String)
defined class Dog

scala> val dog1 = Dog("Fido")
dog1: Dog = Dog@787b059

scala> val dog2 = Dog("Fido")
dog2: Dog = Dog@3f990cad

scala> dog1.canEqual(dog2)
res4: Boolean = true

scala> dog1==dog2
res1: Boolean = false
```

Let's say we want dogs with the same name to be equivalent. We would use the following definition of `Dog`:

```
case class Dog(name: String) {
  override def equals(that: Any): Boolean = canEqual(that) && hashCode==that.hashCode
  override def hashCode = name.hashCode
}
```

Comparisons would then work as expected:

```
scala> val dog1 = Dog("Fido")
dog1: Dog = Dog(Fido)

scala> val dog2 = Dog("Fido")
dog2: Dog = Dog(Fido)

scala> dog1==dog2
res6: Boolean = true
```

If you wanted to compare two objects according to the value of their name property that were not defined as case classes, you will need to use `asInstanceOf` instead of `canEqual`, like this:

```
class Hog(name: String) {
  override def equals(that: Any): Boolean = that.isInstanceOf[Hog] && hashCode==that.hashCod
e
  override def hashCode = name.hashCode
}
```

2-11 Deceptively familiar: access modifiers

ScalaCourses.com / scalaIntro / ScalaCore / scala00 / scalaAccess

In Scala you can define many levels of visibility. Visibility rules for Scala classes also apply to Scala objects.

We'll cover this matrix through examples, instead of getting tangled up in byzantine technical details and the buggy implementation. Paul Phillips, the person who wrote most of the Scala compiler, says even he does not know "what is supposed to work and how it's supposed to work... The bugs persist in large part because I lack sufficient information to fix them." Historically, the early versions of the Scala compiler had a lot of input from graduate students, and some of that code lives on in the bowels of the Scala compiler. This issue is marked for resolution for Scala 2.11-M5, due to be announced 16/Sep/13, with final release due 03/Feb/13.

In this lecture we will merely demonstrate the behavior of the Scala 2.10.1 implementation by defining seven main Scala classes, each with a different type of visibility: `public`, `protected`, `protected[packageName]`, `protected[this]`, `private`, `private[packageName]`, and `private[this]`. All but the `private` class are defined with a group of properties and methods which exercise all possible visibilities, and these members are named after their visibility so we can keep track of the myriad combinations of class and member visibilities. Most of these classes also define inner classes, again with varying visibility, and with names that suggest their visibility.

All of the classes are defined in the `com.micronautics.scalaIntro` package.

public

There is no `public` keyword in Scala, however by default classes, properties and methods all have `public` visibility.

Scala `public` visibility means the same thing as Java `public`: software artifacts are visible in all valid scopes, for both languages.

Here is the Scala demonstration class, which contains members with varying visibility, and inner classes of varying visibility:

```

class Public {
  val aPublicProperty = 0
  protected val aProtectedProperty = 0
  protected[scalaIntro] val aProtectedPackageProperty = 0
  protected[Public] val aProtectedClassProperty = 0
  protected[this] val aProtectedThisProperty = 0
  private val aPrivateProperty = 0 // not visible from Java
  private[scalaIntro] val aPrivateClassProperty = 0
  private[this] val aPrivateThisProperty = 0 // not visible from Java

  class PublicInPublicClass
  protected[scalaIntro] class ProtectedPackageInPublicClass
  protected[Public] class ProtectedClassInPublicClass
  protected[this] class ProtectedThisInPublicClass
  private[scalaIntro] class PrivatePackageInPublicClass
  private[this] class PrivateThisInPublicClass
}

```

Note the unique Scala syntax necessary to create instances of the inner Scala classes; the inner class name is qualified by the outer class instance which will contain the inner class instance.

```

val publicInPublicInstance = new publicInstance.PublicInPublicClass

```

protected

When `protected` is applied to a Scala class, the class's visibility is restricted to subclasses, and it is not visible from other classes in the defining package. When applied to a property or method, the property or method's visibility is restricted to the defining Scala type and derived types.

Here is the Scala test class, which contains members with varying visibility, and inner classes of varying visibility:

```

protected class Protected {
  val aPublicProperty = 0
  protected val aProtectedProperty = 0
  protected[scalaIntro] val aProtectedPackageProperty = 0
  protected[this] val aProtectedThisProperty = 0
  private val aPrivateProperty = 0 // not visible from Java
  private[scalaIntro] val aPrivateClassProperty = 0
  private[this] val aPrivateThisProperty = 0 // not visible from Java

  class PublicInProtectedClass
  protected[scalaIntro] class ProtectedPackageInProtectedClass
  protected[Protected] class ProtectedClassInProtectedClass
  protected[this] class ProtectedThisInProtectedClass
  private[scalaIntro] class PrivatePackageInProtectedClass
  private[this] class PrivateThisInProtectedClass
}

```

protected[packageName]

When a Scala class is decorated with `protected[packageName]` it is visible from all other classes in the specified package. When `protected[packageName]` is applied to a property or method, the property or method's visibility is restricted to the defining Scala type and derived types in the specified package. Here is the Scala demonstration class, which contains members with varying visibility, and inner classes of varying visibility:

```
protected[scalaIntro] class ProtectedPackage {
  val aPublicProperty = 0
  protected val aProtectedProperty = 0
  protected[scalaIntro] val aProtectedPackageProperty = 0
  protected[this] val aProtectedThisProperty = 0
  private val aPrivateProperty = 0           // not visible from Java
  private[scalaIntro] val aPrivateClassProperty = 0
  private[this] val aPrivateThisProperty = 0 // not visible from Java

  class PublicInProtectedPackageClass
  protected[scalaIntro] class ProtectedPackageInProtectedPackageClass
  protected[ProtectedPackage] class ProtectedClassInProtectedPackageClass
  protected[this] class ProtectedThisInProtectedPackageClass
  private[scalaIntro] class PrivatePackageInProtectedPackageClass
  private[this] class PrivateThisInProtectedPackageClass
}
```

protected[className]

From the point of view of Scala code, decorating an artifact with `protected[className]` is supposedly equivalent to decorating with `private`, however the decoration does nothing. Here is the Scala demonstration class, which contains members with varying visibility, and inner classes of varying visibility:

```
class EnclosingClass {
  protected[EnclosingClass] class ProtectedInnerClass {
    protected[ProtectedInnerClass] class ProtectedClassInProtectedClass

    val aPublicProperty = 0
    protected val aProtectedProperty = 0
    protected[scalaIntro] val aProtectedPackageProperty = 0
    protected[this] val aProtectedThisProperty = 0
    private val aPrivateProperty = 0
    private[scalaIntro] val aPrivateClassProperty = 0
    private[this] val aPrivateThisProperty = 0 // not visible from Java

    class PublicInProtectedInnerClass
    protected[scalaIntro] class ProtectedPackageInProtectedInnerClass
    protected[ProtectedInnerClass] class ProtectedClassInProtectedInnerClass
    protected[this] class ProtectedThisInProtectedInnerClass
    private[scalaIntro] class PrivatePackageInProtectedInnerClass
    private[this] class PrivateThisInProtectedInnerClass
  }
}
```

protected[this]

From the point of view of a Scala program, `protected[this]` is equivalent to `private` and `private[this]`.

Here is the Scala demonstration class, which contains members with varying visibility, and inner classes of varying visibility:

```
protected[this] class ProtectedThis {
  val aPublicProperty = 0
  protected val aProtectedProperty = 0
  protected[scalaIntro] val aProtectedPackageProperty = 0
  protected[ProtectedThis] val aProtectedClassProperty = 0
  protected[this] val aProtectedThisProperty = 0
  private val aPrivateProperty = 0 // not visible from Java
  private[scalaIntro] val aPrivateClassProperty = 0
  private[this] val aPrivateThisProperty = 0 // not visible from Java

  class PublicInProtectedThisClass
  protected[scalaIntro] class ProtectedPackageInProtectedThisClass
  protected[ProtectedThis] class ProtectedClassInProtectedThisClass
  protected[this] class ProtectedThisInProtectedThisClass
  private[scalaIntro] class PrivatePackageInProtectedThisClass
  private[this] class PrivateThisInProtectedThisClass
}
```

`private[packageName]`

The decoration `private[packageName]` is known as package-private, and is the default visibility for Java.

Package-private access means that a class or member can only be accessed within its own package. Subclasses cannot access subclasses or members whose superclass was marked `private[packageName]`.

When applied to a property or method: For example, consider a Scala package called `com.micronautics.scalaJava` that defines a class called `ScalaClass3`:

```
package com.micronautics.scalaJava

class ScalaClass3(val prop1: Int, var prop2: Double=0.0) {
  val prop3: Int = prop1 * 2;
  var prop4: Double = prop2 * 3.0;
  private[scalaJava] val packagePrivate = "package private"

  def largestProp(other: ScalaClass3): ScalaClass3 =
    if (this.prop1 > other.prop1) this else other

  override def toString = "prop1=%02d; prop2=%.1f; prop3=%02d; prop4=%.1f".
    format(prop1, prop2, prop3, prop4)
}
```

Notice how the last part of the package name that contains the Scala class (`scalaJava`), qualifies the `private` modifier. This means the definitions made in this file are not visible to other objects in other packages.

`private[className]`

"package-private access without inheritance".

It is unclear exactly what this visibility setting actually means. The recommendation is not to use this visibility until it is properly defined in Scala 2.11, due for release Jan 24, 2014.

private

When applied to a class or member, `private` has same meaning in Scala and Java: class definitions and members are not visible from other classes.

private[this]

From the point of view of a Scala program, `private[this]` is equivalent to `protected[this]` and `private`. Here is the Scala demonstration class, which contains members with varying visibility, and inner classes of varying visibility:

```
private[this] class PrivateThis {
  val aPublicProperty = 0
  protected val aProtectedProperty = 0
  protected[scalaIntro] val aProtectedPackageProperty = 0
  protected[PrivateThis] val aProtectedClassProperty = 0
  protected[this] val aProtectedThisProperty = 0
  private val aPrivateProperty = 0 // not visible from Java
  private[scalaIntro] val aPrivateClassProperty = 0
  private[this] val aPrivateThisProperty = 0 // not visible from Java

  class PublicInPrivateThisClass
  protected[scalaIntro] class ProtectedPackageInPrivateThisClass
  protected[PrivateThis] class ProtectedClassInPrivateThisClass
  protected[this] class ProtectedThisInPrivateThisClass
  private[scalaIntro] class PrivatePackageInPrivateThisClass
  private[this] class PrivateThisInPrivateThisClass
}
```

sealed

A sealed Scala class may not be directly subclassed by Scala code, unless the subclass is defined in the same source file as the sealed class. However, subclasses of a sealed class can be defined anywhere by Scala code. Here is a simple example of a sealed Scala class:

```
sealed class Sealed
```

3 Useful Stuff

ScalaCourses.com / scalaIntro / ScalaCore / scalaIntroMisc

3-1 Unit testing With Specs2

ScalaCourses.com / scalaIntro / ScalaCore / scalaIntroMisc / scalaUnit

Test source files must be placed in your application's test folder. You can use sbt to run them from the command line using the test and test-only tasks. Scala-IDE and IntelliJ IDEA both offer integrated unit test support. You won't be using the REPL or Scala scripts when writing unit tests.

You need to add this import to the test file:

```
import org.specs2.mutable._
```

Specs 2 can be included into a build.sbt file this:

```
libraryDependencies += Seq(  
  "org.specs2" %% "specs2" % "2.1.1" % "test"  
)
```

When you run sbt gen-idea or sbt eclipse, the IntelliJ IDEA or Scala-IDE project definition is (re)created with Specs 2 support.

Quick and Easy Test Specifications

The test class must extend Specification, and contain should statements which themselves contain one or more in statements.


```
import org.specs2.mutable._
import org.junit.runner.RunWith
import org.specs2.runner.JUnitRunner

@RunWith(classOf[JUnitRunner])
class HelloWorldSpec extends Specification {

  "The 'Hello world' string" should {
    "contain 11 characters" in {
      "Hello world" must have size(11)
    }

    "start with 'Hello'" in {
      "Hello world" must startWith("Hello")
    }

    "end with 'world'" in {
      "Hello world" must endWith("world")
    }
  }
}
```

For more information.

SBT

Sbt has a good test runner. You can run all tests by using the `sbt test` command, like this:

```
$ sbt test
[info] Loading global plugins from /home/mslinn/.sbt/plugins
[info] Loading project definition from /home/mslinn/work/training/course_scala_intro_code/courseNotes/project
[info] Set current project to scalaIntroCourse (in build file:/home/mslinn/work/training/course_scala_intro_code/courseNotes/)
[success] Total time: 33 s, completed Sep 2, 2013 5:14:36 PM
```

You can also just run a specific test, like this (note the quotes):

```
$ sbt "test-only HelloWorldSpec"
[info] Loading global plugins from /home/mslinn/.sbt/plugins
[info] Loading project definition from /home/mslinn/work/training/projects/public_code/scalaCore/course_scala_intro_code/courseNotes/project
[info] Set current project to scalaIntroCourse (in build file:/home/mslinn/work/training/projects/public_code/scalaCore/course_scala_intro_code/courseNotes/)
[success] Total time: 2 s, completed Sep 2, 2013 5:16:42 PM
```

IntelliJ IDEA

Running a Specs 2 unit test with IDEA is really easy. Simply right-click on the unit test, and IDEA recognizes that it needs to create a Specs 2 run configuration, and it launches the test.

Scala IDE

You need to add these extra imports to source files containing Specs2 tests:

```
import org.junit.runner.RunWith
import org.specs2.runner.JUnitRunner
```

You also need to add JUnit as a dependency to your project for Scala IDE to be able to work with Specs2.

```
libraryDependencies += Seq(
  "org.specs2" %% "specs2" % "2.1.1" % "test",
  "junit"      % "junit"   % "4.8.1" % "test" // Scala IDE requires this; IntelliJ IDEA does not
)
```

The extra `import` and new `junit` dependency required by Scala IDE are compatible with IntelliJ IDEA.

To run the test, right-click on the unit test and select **Debug As / Scala JUnit Test**.

For more information on Scala IDE unit testing.

Exercise

Write a Specs2 unit test that fetches the contents of <http://www.cisco.com> and verifies that the word `cisco` is present in any case. Use any IDE, or no IDE.

4 Functional Programming

ScalaCourses.com / scalaIntro / ScalaCore / scalaFunctional

4-1 Option, Some and None

ScalaCourses.com / scalaIntro / ScalaCore / scalaFunctional / scalaOption

Option is a wrapper for a value or a lack of a value, which helps avoid null pointer exceptions. When it has a value the Some subclass is used, otherwise the None subclass is used. Uninitialized (null) objects are the source of many errors. Scala's Option is a solution to this problem. If an Option has Some value, its isDefined method will return true; otherwise, if the Option has None value isEmpty will return true.

```
scala> val object1 = Some("Hi there")
object1: Some[java.lang.String] = Some(Hi there)

scala> object1.isDefined
res11: Boolean = true

scala> object1.isEmpty
res12: Boolean = false

scala> val object2 = None
object2: None.type = None

scala> object2.isDefined
res13: Boolean = false

scala> object2.isEmpty
res14: Boolean = true
```

Here is where Option gets interesting. First imagine that you want to retrieve the value of an environment variable:

```
scala> Option(System.getProperty("os.name"))
res15: Option[java.lang.String] = Some(Windows 7)
```

Because we are sure that the result is wrapped in a Some instance, we call get to retrieve it (we will discover soon that it usually is better to use map, foreach or other iterators instead of calling get):

```
scala> Option(System.getProperty("os.name")).get
res16: java.lang.String = Windows 7
```

If the environment variable is not set, Option will return None instead of null:

```
scala> Option(System.getProperty("not.present"))
res17: Option[java.lang.String] = None
```

We can use orElse() to provide a default value if the Option's value was None.

```
scala> Option(System.getProperty("not.present")).orElse(Some("default"))
res3: Option[String] = Some(default)

scala> Option(System.getProperty("os.name")).orElse(Some("default"))
res4: Option[String] = Some(Linux)
```

The above returns an `Option` if the system property has a value or not. Now we can use `get` to retrieve whichever value comes back.

```
scala> Option(System.getProperty("not.present")).orElse(Some("default")).get
res19: Option[java.lang.String] = default
```

We can use `getOrElse()` to retrieve the value of the `Option`, or the value of the `getOrElse()` arguments if the `Option`'s value was `None`.

```
scala> Option(System.getProperty("not.present")).getOrElse("default")
res18: String = default

scala> Option(System.getProperty("os.name")).getOrElse("default")
res6: String = Linux
```

`Option` is actually a collection of up to one item. This means that collection operations work on `Option` instances. This leads to useful Scala idioms. More on that later.

Exercise

Write a Scala script that contains a `match` statement which checks a value against `Some(String)` or `None`. Print out an appropriate message.

4-2 Either, Left and Right

ScalaCourses.com / scalaIntro / ScalaCore / scalaFunctional / scalaEither

Many methods might return a value, throw exceptions or return a different type of value when an error occurs. Instead of declaring these exceptions, Either wraps both possibilities. Either is parametric in two types, one for `Either.left` and the other for `Either.right`. By convention, The Left flavor of Either wraps an exception, if one was thrown; the Right flavor wraps the results.

You can try typing each of the following into the Scala REPL. In Scala, one might declare an immutable result and assign it a successful String value like this:

```
scala> val result1: Either[Throwable, String] = Right("yay!")
result1: Either[Throwable,String] = Right(yay!)
```

Storing exceptions in Scala into `Either.left` is simple:

```
scala> val result2: Either[Throwable, String] = Left(new Exception("Ain't this fun?"))
result2: Either[Throwable,String] = Left(java.lang.Exception: Ain't this fun?)
```

You can test to see if an Either is holding a successful value (on the right) or an exception (on the left). You can also obtain the value or the exception:

```
scala> result1.isRight
res0: Boolean = true

scala> result1.isLeft
res1: Boolean = false

scala> result1.right.get
res9: String = yay!

scala> result2.isRight
res12: Boolean = false

scala> result2.isLeft
res13: Boolean = true

scala> result2.left.get
res14: Throwable = java.lang.Exception: Ain't this fun?
```

Convention dictates that Left is used for failure and Right is used for success, however don't get the idea that the left side of Either must always hold a `Throwable`. A common use of Either is as an alternative to `Option` for dealing with possible missing values. For example, you could use `Either[String, Int]` to decode a `String` into an `Int` on the Right, or return the unparseable string on the Left.

```
type IntString = Either[String, Int]

def parse(in: String): IntString = try {
  Right(in.toInt)
} catch {
  case e: Exception =>
    Left(in)
}

def show(either: IntString): Unit =
  println(either match {
    case Right(x) =>
      s"Parsed Int: $x"

    case Left(x) =>
      s"Unparseable String: $x"
  })
```

Now let's try parsing some strings:

```
scala> show(parse("1234"))
Parsed Int: 1234

scala> show(parse("12abc"))
Unparseable String: 12abc

scala> show(parse("abc123"))
Unparseable String: abc123
```

Exercise

The `Either` type in the Scala library can be used for algorithms that return either a result or some failure information. Write a method that takes two parameters: a list of words and an word to match. Return an `Either` containing the index of the matching word in the list on the right or the word that did not match on the left.

I'm going to show you three solutions. Try each one - you will learn something with each solution.

Setup

We will create a console application, so to define one we start by creating a directory called `~/work/ex1`, and then an sbt project in that directory using the `newSbt` script we created in the sbt lecture.

```
$ mkdir -p ~/work/ex1
$ cd ~/work/ex1
$ newSbt
$ sbt gen-idea
```

Now let's edit a new file in ~/work/ex1 called src/main/scala/WordSearch.scala using your favorite text editor. To define an entry point in that file, simply write:

```
object WordSearch extends App { }
```

Let's define a type alias for the return type, inside the body of WordSearch:

```
object WordSearch extends App {  
  type FunnyReturnType = Either[String, Int]  
}
```

Let's also decide on what the signature will be for the method that will do the work. Note that ??? defines a method with a body that does nothing - it is a placeholder for the body which we will write later. This allows the code to compile before it is completely written.

```
object WordSearch extends App {  
  type FunnyReturnType = Either[String, Int]  
  
  def search(list: List[String], word: String): FunnyReturnType = ???  
}
```

We will use this code to run the search method:

```
List("word", "oink", "blahbla", "another").foreach { w =>  
  println(s"$w: ${search(list, w)}")  
}
```

So we now have:

```
object WordSearch extends App {  
  type FunnyReturnType = Either[String, Int]  
  
  def search(list: List[String], word: String): FunnyReturnType = ???  
  
  List("word", "oink", "blahbla", "another").foreach { w =>  
    println(s"$w: ${search(list, w)}")  
  }  
}
```

Notice that the entry point application consists of a type alias, a method definition and some code that is executed.

Solution 1 - Efficient But Clumsy

```
def search(list: List[String], word: String): FunnyReturnType = {  
  val index = list.indexOf(word)  
  if (index < 0)  
    Left(word)  
  else  
    Right(index)  
}
```

Solution 2 - Efficient, Simple, Elegant

```
def search(list: List[String], word: String): FunnyReturnType =
  list.indexOf(word) match {
    case -1 => Left(word)
    case index => Right(index)
  }
```

Solution 3 - Overly Complex But Shows Scala Techniques

```
def search(list: List[String], word: String): FunnyReturnType = {
  val answers = for {
    item <- list if item == word
  } yield {
    list.indexOf(item)
  }
  val result: FunnyReturnType =
    answers.headOption match {
      case Some(i) => Right(i)
      case None => Left(word)
    }
  result
}
```

The results of running each of the three solutions are the same. Look at the [courseNotes](#) to see the complete solution.

```
Method 1 word: Right(0)
Method 2 word: Right(0)
Method 3 word: Right(0)
Method 1 oink: Left(oink)
Method 2 oink: Left(oink)
Method 3 oink: Left(oink)
Method 1 blahbla: Left(blahbla)
Method 2 blahbla: Left(blahbla)
Method 3 blahbla: Left(blahbla)
Method 1 another: Right(1)
Method 2 another: Right(1)
Method 3 another: Right(1)
```


4-3 Review So Far

[ScalaCourses.com](#) / [scalaIntro](#) / [ScalaCore](#) / [scalaFunctional](#) / [scalaIntroReview](#)

```
Some(x=3/z) match {
  case a: Int => { println("no") }
  case b: String => println(b)
}
```

```
1 to 3 foreach { i => println(i) }
```

```
trait asdf { self: Fee =>
  def help: String = """"You are so smart
This is line 2
This is line 3
"""".stripMargin
}
```

```
You are so smart
This is line 2
This is line 3
```

```
for ( x <- List(1, 2, 3)) println(x)
```

```
val asdf: List[List[String]] = for {
  x <- List(1, 2, 3)
  y <- Seq("a", "b", "c").toList
  z <- Some(3).toList
} yield {
  println(x + y + z)
  List(x.toString, y, z.toString)
}
```

```
val xs: Option[String] = Some(x)
val ys: Option[String] = Some(y)
val zs: Option[String] = Some(z)
val asdf: Option[List[String]] = for {
  x <- xs
  y <- ys
  z <- zs
} yield {
  println(x + y + z)
  List(x, y, z)
}
```

```
val asdf: List[Option[String]] = for {
  x <- List(1, 2, 3)
  y <- Seq("a", "b", "c").toList
  z <- Some(3).toList
} yield {
  println(x + y + z)
  if (x>2)
    Some(x + y + z)
  else
    None
}
```

© Copyright 2008-2013 Micronautics Research Corporation. All rights reserved.
Micronautics Research, Micronautics Cadenza and ScalaCourses.com are trademarks of Micronautics
Research Corporation.
Build 1264 created October 2, 2013 12:45:03 PM UTC