Introduction to Scala

Core Scala

http://scalacourses.com

Introduction to Scala

Printed in the United States of America.

Published by Micronautics Research Corporation, 840 Main Street, Half Moon Bay, CA, USA 94019.

scalaIntro

Introduction to Scala

This "Introduction to Scala" course covers the fundamentals and is designed to give you a variety of hands-on experiences with Scala. The unique features of Scala are introduced. Exercises are provided throughout that reinforce the lecture material. The material builds from lecture to lecture, so be sure to go through the material from start to finish. The course consists of 6 hours of video, over 150 pages of lecture notes, more than four dozen sample programs and about a dozen exercises.

In addition to self-paced online learning, Micronautics Research offers on-site and remote delivery and/or tutoring for this course. A combination of online training with videoconferencing for office hours is available. Please contact us for further details.

Section 1 `scalaRunning` **Installing and Running** **11 lectures**

Lecture 1-1 `scalaOverview` Scala Overview and Philosophy 00:05:13

- Scala is Compatible with Java
- Everything is an Object
- Scala is Strongly Typed
- Scala supports Functional Programming
- No More Null Pointer Exceptions!
- Scala is Succinct
- Extractors and Pattern Matching
- Scala Scales Horizontally
- ... And Much More!

Lecture 1-2 `scalaInstall` Installing Scala 00:12:24

- Install a Terminal Emulator
- Installing Java
- Installing Scala
  - Mac
  - Windows
  - Linux
  - Manual Installation
  - Mac Laptops: NoCompDaemon option
  - Check the Version of Scala
- Scala Runner Command Line Options
  - Scalac options

Lecture 1-3 `scalaREPL` The Scala Interpreter (REPL) and simple looping 00:16:05

- 95% Runtime Fidelity
- Every Statement Returns a Value
- Exploring Scala with the REPL
- Other Scala Control Statements
  - while
  - Scala assignment returns Unit
  - do while
  - for

Lecture 1-4 `scalaSublime` Sublime Text 00:12:22

- Setup

Creating a Scala Project
Opening SBT projects
Converting SBT Projects Into IntelliJ Projects
Working With IDEA
Fixing the No 'scala-library*.jar' in Scala compiler library error
Helpful Hints
Worksheets
Useful IntelliJ Hot Keys
Hot Keys From Eclipse Key Bindings

Lecture 1-10 `worksheet` Worksheets 00:11:38

Scala-IDE (Eclipse) Worksheets
IDEA Worksheets
Worksheet Interoperability

Lecture 1-11 `scalaImportsPackages` Scala Imports and Packages 00:14:11

Packages
Package Objects
Import Statements
Locally Scoped Imports
Imports Can Enable Advanced Scala Language Features

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

Lecture 2-1 `scalaTypes` Type Hierarchy and Equality 00:15:59

Type Widening
Object Equality
Calling canEqual Prevents Bugs
REPL's :type

Lecture 2-2 `scalaClasses` Classes 00:13:19

Scala Classes
Named Parameters
Anonymous Classes
Operator Overloading

Lecture 2-3 `scalaUniform` Setters, Getters and the Uniform Access Principle 00:09:03

Quick Review
Setters and Getters
Uniform Access Principle

Lecture 2-4 `scalaAccess` Deceptively familiar: access modifiers 00:09:32

Primary constructor visibility
Syntax for creating inner classes

Lecture 2-5 `scalaCompanion` Companion Objects 00:05:37

Lecture 2-6 `scalaDocReading` Reading ScalaDoc 00:05:47

Lecture 2-7 `scalaAuxCons` Auxiliary Constructors 00:07:11

Return type rules

Parentheses rules when defining methods
Parentheses rules when invoking methods

Lecture 2-8 scalaTraits Traits / Mixins 00:08:38

Pure Trait
Trait with Implementation
Extending Multiple Traits
Traits Should Not Extend Classes

Lecture 2-9 scalaTraits2 More Traits 00:15:19

Stackable Traits
Another Example
Self Traits and Dependency Injection

Lecture 2-10 scalaCaseClasses Case Classes 00:13:41

Object Equality
Arity 22 Limitation
Subclassing Traits and Classes into Case Classes
Do Not Subclass Case Classes
Operator Overloading Revisited
Standard Methods
Companion Classes
Companion Objects

Lecture 2-11 scalaTuples1 Tuples Part 1 00:05:33

Section 3 scalaFunctional **A Small Taste of Functional Programming** **8 lectures**

Lecture 3-1 scalaOption Option, Some and None 00:09:27

Wrapping Nulls with Option
Exercise:
Solution

Lecture 3-2 scalaMatch Pattern Matching 00:14:20

Multiway Branch/Match on Value
Providing a guard
Matching On Type
Example: UnWrapping Java nulls Wrapped with Option
Matching On Type With a Guard
Another Example

Lecture 3-3 scalaUnapply Unapply 00:16:51

Unapply Return Types
Overloading Unapply
Partial match
Case Classes
Overloading unapply

Lecture 3-4 scalaExtract Sealed Classes and Extractors 00:07:14

Sealed Classes and Traits Provide Exhaustive Matching

Extracting Values
Quiz: Extractors vs. Matching types or values
Solution
Aliases

Lecture 3-5 scalaTry Try and try/catch/finally 00:07:29

Exercise: Yoda He Is
Solution
NoStackTrace
NoStackTrace Improved
try / catch / finally

Lecture 3-6 scalaEither Either, Left and Right 00:12:54

Type Alias
Exercise
Solution

Lecture 3-7 scalaFunctions Functions are First Class 00:13:08

Binding to a variable
Placeholder Syntax
Desugared Syntax
Methods vs. Functions
No-Argument Function Literals
Lifting a Method to a Function
Quiz: Passing Functions as parameters
Solution
Composition

Section 4 scalaIntroMisc **Useful Stuff** **1 lectures**

Lecture 4-1 scalaUnit Unit testing With ScalaTest and Specs2 00:05:50

Project Setup
Specs2
ScalaTest
ScalaTest and Specs2 In the Same Project
Quick and Easy Test Specifications
Specs2
ScalaTest
SBT
IntelliJ IDEA
Scala IDE

1. See the "**Sections & Lectures**" tab for the table of contents.
2. View the "**Printable transcripts**" tab and press `Ctrl`-`P` (`Cmd`-`P` on Mac) and select a PDF printer to make a PDF containing the course notes.
3. The "**Course Details**" tab shows a link for downloading the course project as a Zip file and the `git` command to clone the course project as a git repository. The project home contains information about the courseNotes, and issue log.

This is a hands-on course. Please don't just read the notes, try every code example yourself. Type along with the REPL. If you encounter a problem or have a question, and you are taking a live class, please talk with the instructor right away. Otherwise, please log an issue for this course.

Prerequisites

Java familiarity or other object-oriented programming language is recommended.

Please Install

Please install the following before starting the course, or if attending an in-person course, before coming to class:

- Your laptop or desktop should have 4GB RAM or more, 6+ GB is recommended. A laptop with only 2GB will probably not be usable for this course.
- JDK 6 or later should be installed. JDK 7 is recommended. Mac instructions. Windows instructions. Linux instructions.
- Linux and Mac laptops should work well; Windows poses a challenges that take extra time and effort to overcome.
- Shell:
  - Windows users should have a virtual machine installed, such as VMware Workstation, and a Linux client OS should be installed; XUbuntu 13.10 64-bit is recommended. Note this increases memory requirements; 8GB+ is recommended for VMware. Be sure to test network connectivity from the VM over wireless before coming to class. If Windows users opt for Cygwin instead (be sure to install all packages) then they will encounter limitations, but Cygwin is better than nothing. Note that Cygwin may take up to 8 hours to install.
  - Mac users should install a shell such as iTerm 2.
  - Linux users will already have a shell.
- Mac laptops should have Brew installed. Enter the following incantation, taken from the Brew home page, at a shell prompt:

  ```
  ruby -e "$(curl -fsSL https://raw.github.com/mxcl/homebrew/go)"
  ```

- Adobe Acrobat Reader or other PDF display software such as Foxit or Nitro should be installed.
- A text editor should be installed. Many choices exist. Sublime Text is a particularly good choice because you can open an entire directory, however any text editor will do. If you run Linux, here are some good installation instructions.

Students are encouraged to download the following software in advance, and to follow the installation instructions if they are able. The course lectures will discuss each of these software packages and will help students install them.

- One of the following two IDEs: IntelliJ IDEA (recommended) or Scala IDE. It is fine to install both.

- SBT v0.12.1 or later (all versions up to and including v0.13.1 have been tested and work well).
- Scala 2.10.3.

Problems?

If you are attending an in-person course, the instructor will attempt to deal with issues on the spot. Otherwise, and for unresolved problems and for suggestions, please use the Issue Tracker.

Course Evaluations

We would appreciate you taking the time to fill out the evaluation at the end of the course.

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.

Scala is Compatible with Java

Because Scala runs on the Java virtual machine (JVM), version 6 or later, Scala code can interoperate with Java code. That means all of the Java code ever written can be directly invoked from Scala, and Scala functionality can be accessed from Java. If your company has a big investment in Java, then you can enhance that investment by augmenting it with Scala code. Scala can also interoperate with other JVM languages.

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 *implicits* 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 supports concepts like higher-order functions, partially bound functions and functions that are only defined for ranges of input. We will cover this topic in detail in this course.

Scala is Strongly Typed

A type defines a set of values which a variable can possess and a set of functions that can be applied to these values. Not only is Scala strongly typed, but it has a very sophisticated type system, which is much more capable than most other strongly typed languages such as Java and C++. This means that many kinds of errors are caught by the compiler, instead of manifesting at runtime. This also means that IDEs can offer many more features than they can when working with untyped languages such as JavaScript and Ruby. Scala's type system supports the same kinds of polymorphic behavior that Java and C++ supports, but because Scala's type system is more powerful, the polymorphic behavior is correspondingly enhanced. In case you are not familiar with polymorphic behavior, here is what Wikipedia has to say:

In programming languages, polymorphism is the provision of a single interface to entities of different types. A polymorphic type is a type whose operations can also be applied to values of some other type, or types. There are several fundamentally different kinds of polymorphism:

- If a function denotes different and potentially heterogeneous implementations depending on a limited range of individually specified types and combinations, it is called ad-hoc polymorphism. Ad-hoc polymorphism is supported in many languages using function overloading.
- If the code is written without mention of any specific type and thus can be used transparently with any number of new types, it is called parametric polymorphism. In the object-oriented programming community, this is often called generic programming.
- In object-oriented programming, subtyping or inclusion polymorphism is a concept wherein a name may denote instances of many different classes as long as they are related by some common superclass.

Interaction between parametric polymorphism and subtype polymorphism leads to the concepts of variance and bounded quantification.

This course and the next course (Intermediate Scala) will explore all of these concepts in detail.

Scala supports Functional Programming

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 todays computing platforms. Functional programs are also easier to reason about than stateful programs, and are able to scale horizontally much more easily. Functional programming enables effective use of parallel collections, futures and actors. This course will give you only a small taste of Scala's functional programming capabilities - the next course will delve into this aspect quite deeply.

No More Null Pointer Exceptions!

`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. We will cover this concept in detail in this course.

Scala is Succinct

Scala does away with unnecessary punctuation. 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 a semicolon. Class definitions are also much shorter, combining getters, setters, and the primary constructor in the class definition.

Extractors and Pattern Matching

Part of learning to think in Scala idioms is learning how to use Scala's powerful pattern matching features. Scala provides many types of containers in its runtime library. Pattern matching allows you to extract values, if present from those containers, and to handle cases where values do not exist. This feature is introduced in this course, and is covered in much more detail in the following Intermediate Scala course.

Scala Scales Horizontally

Scala handles multicore programming with ease. For example, you can convert a series of transformations of a collection into a multicore operation by adding the four characters `.par` to an existing expression. The data for the map/reduce paradigm is automatically parceled out into units of work across multiple CPU cores, and the results are also gathered in parallel. Asynchronous programming does not require complex callbacks. The Intermediate Scala course following this one shows how to work with the half-dozen multicore programming options that Scala provides.

... And Much More!

Some of the most interesting features are difficult to express in a sound bite. I'll save those discussions for the course.

1-2 Installing Scala

ScalaCourses.com / scalaIntro / ScalaCore / scalaRunning / scalaInstall

Install a Terminal Emulator

Because a lot of your work with Scala tools is via the command line, you should install a terminal emulator and get used to working with the keyboard, if you are not in that mode already.

- Mac users should install iTerm2. Note that the version installed from the iTerm2 front page is not actually the recommended version. Instead, install the recommended test release.
- Linux users have a variety of terminal applications to work with. I like Xubuntu's Xfce Terminal emulator.
- Windows users should use Cygwin's `mintty` terminal emulator.

Installing Java

1. You need Java installed before you can install Scala. Check the version that is installed on your computer by opening a new shell and typing:

   ```
   $ java -version
   java version "1.7.0_45"
   Java(TM) SE Runtime Environment (build 1.7.0_45-b18)
   Java HotSpot(TM) 64-Bit Server VM (build 24.45-b08, mixed mode)
   ```

   If that did not give you the result you expected, then please install Java. JDK 6 or later is required. JDK 7 is recommended.

   - Mac instructions.
   - Linux instructions. For Linux, the directories that Java is installed into vary widely, however Ubuntu often uses subdirectories of `/usr/lib/jvm`, like `/usr/lib/jvm/java-7-oracle`.
   - Windows instructions. You need Cygwin in order to work with Scala at the command line effectively with Windows. The default directory that Java is installed is a subdirectory of `/cygdrive/c/PROGRA~1/Java`, like `/cygdrive/c/PROGRA~1/Java/jdk1.7.0_45`

2. Ensure that `JAVA_HOME` is set by typing:

   ```
   $ echo "$JAVA_HOME"
   /Library/Java/JavaVirtualMachines/jdk1.7.0_45.jdk/Contents/Home
   ```

   If not set, or set improperly:

   - For Mac, add this incantation to `~/.bash_profile` to set `JAVA_HOME` to the installed directory for Java 7:

```
export JAVA_HOME="$(/usr/libexec/java_home -v 1.7)"
```

- For Linux, add the following to `~/.bashrc`:

```
export JAVA_HOME="$(readlink -f /usr/bin/javac | sed "s:bin/javac::")"
```

- For Cygwin, edit `~/.bashrc` manually.

3. Ensure that `JAVA_OPTS` is set so Scala has enough memory. Again, put this line in `~/bash_profile` or `~/.bashrc`, depending on your OS. If you have more than 6GB RAM and you need to compile large Scala programs, increase values to suit.

```
export JAVA_OPTS="-Xms512M -Xmx2048M -Xss1M -XX:+CMSClassUnloadingEnabled -XX:+UseConcMarkSweepGC -XX:MaxPermSize=512M
"
```

Installing Scala

Installing Scala is really easy.

Mac

If you do not already have Brew, install it by typing:

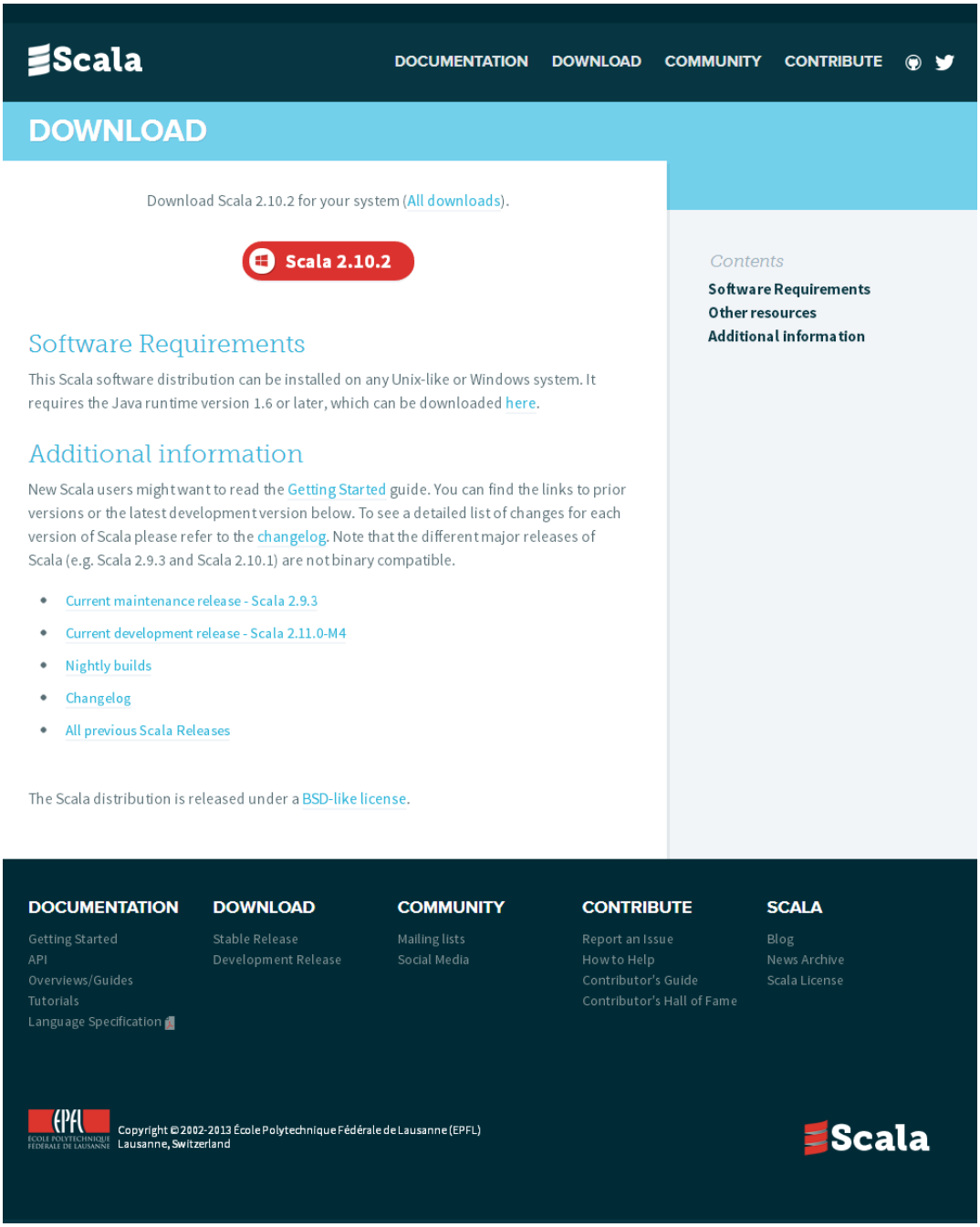
```
$ ruby -e "$(curl -fsSL https://raw.github.com/mxcl/homebrew/go/install)"
```

Now install Scala:

```
$ brew install scala
```

Windows

Navigate to `http://www.scala-lang.org` and click on the Downloads tab. Push the big red button and follow the instructions.



Linux

If you have a Debian-derived distro like Ubuntu, just type this one command:

```
$ sudo apt-get install scala
```

Otherwise, navigate to `http://www.scala-lang.org` and click on the Downloads tab. Push the big red button and follow the instructions.

Manual Installation

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 a Mac you might want to store the contents in `Applications`, or if you use Linux or Cygwin you might want to

place them in `/opt`. 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-2.10.3.tgz
```

2. You need to unpack the downloaded file, which has a `.tgz` file type. 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-2.10.3` 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-2.10.3.tgz
```

3. Define an environment variable called `SCALA_HOME` that points to the new directory (adjust the location of `SCALA_HOME` as required):
   1. For Linux, define the variable in `~.bashrc`:

      ```
      export SCALA_HOME=/opt/scala-2.10.3
      export PATH=$SCALA_HOME/bin:$PATH
      ```

      Run the following for changes to take effect without starting a new shell:

      ```
      source ~/.bashrc
      ```

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

      ```
      export JAVA_HOME=$(/usr/libexec/java_home)
      export SCALA_HOME=~/Applications/scala-2.10.3
      export PATH=$SCALA_HOME/bin:$PATH
      ```

      Run the following for changes to take effect without starting a new shell:

      ```
      source ~/.bash_profile
      ```

Mac Laptops: 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 at least it won't hang. For Linux and Cygwin, put the following in `~/.profile`; for Mac put the following in `~/.bash_profile`:

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

Check the Version of Scala

View the version of Scala that is installed on your computer like this:

```
$ scala -version
Scala code runner version 2.10.3 -- Copyright 2002-2013, LAMP/EPFL
```

Scala Runner Command Line Options

You can read about the Scala command-line 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
```

Scalac options

You can discover the Scala compiler (`scalac`) options like this:

```
$ 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 supressed 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.
```

1-3 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. The Scala REPL provides 95% fidelity of the Scala runtime environment. There are some important differences, however, and I will point them out as we go along. You can try out the Scala REPL by typing `scala` at a shell prompt:

```
$ scala
Welcome to Scala version 2.10.3 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_45).
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 displays 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. The type of argument `x` is defined to be `Int`, and the implementation of the method follows the equals sign.

```
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
```

95% Runtime Fidelity

Here is one of the ways that the Scala REPL distorts the Scala runtime environment. Even though we defined x to be immutable, we can redefine it:

```
scala> val x: String = "hi there"
res7: String = "hi there"
```

This is possible because the Scala REPL actually creates a new runtime context each time you issue a statement. A real Scala program would not allow the following to compile:

```
val x: Int = 3
val x: String = "hi there"
```

As you experiment with the Scala REPL, realize that definitions might behave differently than they would when compiling a real Scala program. Here is another difference: variables and methods are always associated with an instance of a class or an `object`. We'll talk about objects more a bit later, but for now just know that they are singleton instances of a class. You can define an `object` like this:

```
object Blah {
  val x = 3
  def timesTwo(i: Int) = 2 * i
}
```

For the REPL, you can paste in all of the lines together, like this:

```
scala> object Blah {
   val x = 3
   def timesTwo(i: Int) = 2 * i
 }
defined module Blah
```

Notice that the REPL printed vertical bars between the opening and closing braces. We could also use a semicolon to separate the statements if they are written on the same line, like this:

```
scala> object Blah { val x = 3; def timesTwo(i: Int) = 2 * i }
defined module Blah
```

Unknown to us, the Scala REPL wrapped the variables and methods we defined earlier in this lecture in an object. We don't have access to the object itself, just its properties and methods. You do have access to the `Blah` object we just created, however:

```
scala> Blah.x
res8: Int = 3

scala> Blah.timesTwo(21)
res9: Int = 42
```

Exercise

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

Every Statement Returns a Value

This means that the last statement in a block of code establishes the return value. For example, here is a statement that computes the value of Pi/2 (90 degrees, expressed in radians):

```
scala> math.Pi / 2.0
res10: Double = 1.5707963267948966
```

As you can see, the method `piBy2` computes the value of dividing the predefined constant Pi by 2.0. The REPL stores the value returned by the computation in a new variable it creates called res1 and displays the value.

Here is a method definition that encapsulates the computation:

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

The computed value becomes the value returned by the method.

When we call the method the REPL stores the value from the method in a new variable and displays the value:

```
scala> piBy2
res11: Double = 1.5707963267948966
```

Only the value of the last statement is returned. We can take advantage of this to insert a println. This technique is often used for debugging:

```
def piBy2: Double = {
  val result = math.Pi / 2.0
   println(s"result=$result")
  result
}
```

If we run the above we get the following:

```
scala> piBy2
 result=1.5707963267948966
res12: Double = 1.5707963267948966
```

The value returned from a Scala `if/then/else` construct can also be used to set a variable or as a return vale. For example, let's create two variables, `x` and `y`, and return the larger of the two:

```
scala> val x = 13
x: Int = 13

scala> val y = 14
y: Int = 14

scala> if (x>y) x else y
res13: Int = 14
```

Notice that the value of the `else` clause became the value of the entire `if` expression. We can wrap this computation in a method called `bigger`. The following performs the same function as `math.max`, except that `bigger` is limited to working with `Int` arguments:

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

scala> bigger(13, 14)
res14: Int = 14
```

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) x else y
```

Exploring Scala with the REPL

The REPL is useful for interactively experimenting with an incantation before you write some code. This is a much more productive way of working than the edit / compile / debug loop that you would otherwise have to perform.

Let's explore how to 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)
```

The above seems like the `1`, an `Int` value object, has a method called `to` that accepts the argument `3`. In fact, this is not actually true - instead, the `Int` type has been enriched with a variety of methods through the use of predefined implicits. The next course (Intermediate Scala) teaches you how to enrich classes through the use of implicits.

We could also do away with the syntactic sugar, and write the same range as follows. Note that this is not any more efficient, and is 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 parsed from left to right, so 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 Scala Control Statements

Scala has three other 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
}
```

Let's try this in the REPL:

```
scala> var i = 1
i: Int = 1

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

We could also add a semicolon after the `println` so the entire `while` statement can be expressed on a single line:

```
scala> var i = 1
i: Int = 1

scala> while (i<=3) { println(s"Hello, world #$i"); i = i + 1 }
Hello, world #1
Hello, world #2
Hello, world #3
```

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)
```

Again, we will try this in the REPL, adding a semicolon after the println so the entire while statement can be expressed on a single line:

```
scala> var i = 1
i: Int = 1

scala> do { println(s"Hello, world #$i"); i = i + 1 } while (i<=3)
Hello, world #1
Hello, world #2
Hello, world #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, which will not be discussed in this course. Formatting is optional; we could write this way:

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

Let's look at this in the REPL:

```
scala> for (i <- 1 to 3)
   println(s"Hello, world #$i")
Hello, world #1
Hello, world #2
Hello, world #3
```

Or we could put it all on one line:

```
scala> for (i <- 1 to 3) println(s"Hello, world #$i")
Hello, world #1
Hello, world #2
Hello, world #3
```

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)
}
```

1-4 Sublime Text

`ScalaCourses.com` / `scalaIntro` / `ScalaCore` / `scalaRunning` / `scalaSublime`

Sublime Text 2 is a great tool to have available, even if you usually use an IDE. Some of its more powerful features are:

- Can open an entire directory, and edit files randomly with ease
- Syntax coloring
- Extensive plugin selection, including those that support Scala and Play
- Look up definitions and browse their source
- Integrates with build systems
- Excellent documentation (Mac key bindings, PC and Linux key bindings)

Setup

Package Control

The Package Control plugin is handy for installing other packages, including sublimeSBT, SublimeREPL, Theme Soda and Tomorrow Color Schemes. The remainder of this lecture assumes that you have installed all of these packages. The key binding for Package Control on Mac is `⌘` - `Shift` - `P`, and on Windows and Linux the key binding is `Ctrl` - `Shift` - `P`.

Also recommended: ColorPicker, Color Highlighter, GitGutter, Hex-to-RGBA, SublimeGit, Pretty JSON, Local History, LiveReload, Markdown Preview, Terminal and changeList.

LiveReload also needs a browser plugin.

Additional Setup

It is handy to be able to invoke Sublime Text from the command line. For Mac, simply type:

```
ln -s "/Applications/Sublime Text 2.app/Contents/SharedSupport/bin/subl" ~/bin/subl
```

For Linux, if you installed to `/opt/sublime`, simply type:

```
sudo ln -s /opt/sublime/sublime_text /usr/local/bin/subl
```

For Cygwin:

```
ln -s /cygdrive/c/Program\ Files/Sublime\ Text\ 2/sublime_text.exe /usr/local/bin/subl
```

Now you can invoke Sublime Text using the `subl` command. Additional documentation describing command-line options is here.

User Settings

Preferences / Settings - User allows you to override the default settings. Although you should not edit the default settings, it is helpful to refer to them when establishing your user settings. You can use my user settings if you like. If you do, be sure to either install the same `color_scheme` and `theme` first by using Package Control as described above, or remove the `color_scheme` and `theme` settings before saving.

By default on Mac the `⌘`-`,` (comma) key binding opens user preferences. My key bindings also set up the Mac `⌘`-`.` (period) key binding to open default preferences, so you can refer to that information when customizing your user preferences. My key bindings also work in Windows and Linux, so `Win`-`.` opens default preferences and `Win`-`,` opens user preferences on both of those OSes.

```
{
  "enabled_extensions": "github"
}
```

MarkdownPreview.sublime-settings hosted with ❤ by **GitHub** **view raw**

```
{
        "auto_complete_commit_on_tab": true,
        "auto_complete_with_fields": true,
        "bold_folder_labels": true,
        "color_scheme": "Packages/Tomorrow Color Schemes/Tomorrow-Night.tmTheme",
        "default_line_ending": "unix",
        "detect_slow_plugins": true,
        "fallback_encoding": "UTF-8",
        "file_exclude_patterns":
        [
        ".cache",
        ".classpath",
        ".project",
        "*.iml",
        "*.sublime-*",
        "*.*~",
        "*.pyc",
        "*.pyo",
        "*.exe",
        "*.dll",
        "*.obj",
        "*.o",
        "*.a",
        "*.lib",
        "*.so",
        "*.dylib",
        "*.ncb",
        "*.sdf",
        "*.suo",
        "*.pdb",
        "*.idb",
        ".DS_Store",
        "*.class",
        "*.psd",
        "*.db"
        ],
```

```
        "folder_exclude_patterns":
        [
        ".worksheet",
        ".settings",
        ".idea*",
        ".svn",
        ".git",
        ".hg",
        "CVS"
        ],
        "highlight_line": true,
        "match_brackets_angle": true,
        "remember_open_files": false,
        "shift_tab_unindent": true,
        "spell_check": false,
        "tab_size": 2,
        "theme": "Soda Dark.sublime-theme",
        "translate_tabs_to_spaces": true,
        "trim_trailing_white_space_on_save": true,
        "word_wrap": "off"
}
```

Preferences.sublime-settings (All OSes) hosted with ❤ by **GitHub** **view raw**

```
[
  { "keys": ["super+,"], "command": "open_file", "args": {"file": "${packages}/User/Pref
  { "keys": ["super+."], "command": "open_file", "args": {"file": "${packages}/Default/P
  { "keys": ["shift+super+."], "command": "open_file", "args": {"file": "${packages}/Def
  { "keys": ["shift+super+,"], "command": "open_file", "args": {"file": "${packages}/Use
  { "keys": ["ctl+alt+m"], "command": "markdown_preview", "args": {"target": "browser",
  { "keys": ["super+k, super+m"], "command": "toggle_minimap"}
]
```

sublime-keymap Linux hosted with ❤ by **GitHub** **view raw**

```
[
  { "keys": ["super+,"], "command": "open_file", "args": {"file": "${packages}/User/Pref
  { "keys": ["super+."], "command": "open_file", "args": {"file": "${packages}/Default/P
  { "keys": ["shift+super+."], "command": "open_file", "args": {"file": "${packages}/Def
  { "keys": ["shift+super+,"], "command": "open_file", "args": {"file": "${packages}/Use
  { "keys": ["ctl+alt+m"], "command": "markdown_preview", "args": {"target": "browser",
  { "keys": ["super+k, super+m"], "command": "toggle_minimap"}
]
```

sublime-keymap Mac hosted with ❤ by **GitHub** **view raw**

```


```

```
[
  { "keys": ["super+,"], "command": "open_file", "args": {"file": "${packages}/User/Pref
  { "keys": ["super+."], "command": "open_file", "args": {"file": "${packages}/Default/P
  { "keys": ["shift+super+."], "command": "open_file", "args": {"file": "${packages}/Def
  { "keys": ["shift+super+,"], "command": "open_file", "args": {"file": "${packages}/Use
  { "keys": ["ctl+alt+m"], "command": "markdown_preview", "args": {"target": "browser",
  { "keys": ["super+k, super+m"], "command": "toggle_minimap"}
]
```

sublime-keymap Windows hosted with ❤ by **GitHub** **view raw**

Play Support

If you are working with Play Framework, you should install the Play plugin for Sublime Text.

Scala Support

The `sbtTemplate` project includes the `sbt-sublime` plugin. The `courseNotes` project for this course also includes the `sbt-sublime` plugin. If this plugin is installed in your project or is globally installed, you can use `sbt-sublime` on an SBT project by simply typing:

```
sbt gen-sublime
```

The SBT Global Setup lecture discusses how to set up SBT and plugins, and it describes how to install `sbt-sublime` globally.

Once you have run `gen-sublime`, you can then open the project and browse the source code of all dependencies in Sublime Text by typing:

```
subl --project *.sublime-project &
```

Even better, use the script from the gist in the previous section to open the project.

Opening a Directory

This is easily the best feature of Sublime Text. To do this from a command line, change directory to the project folder and then open the current folder with

```
subl .
```

This gist shows a script that causes Sublime Text to open a file, a directory or a Sublime project created with `gen-sublime`. I recommend you save the script as `bin/s` (for Mac) or `/usr/local/bin/s` (for Linux and Cygwin).

```
1
2
3
```

```
#!/bin/bash

# Assumes you have defined subl command
# See http://www.sublimetext.com/docs/2/osx_command_line.html
# See http://www.sublimetext.com/forum/viewtopic.php?f=4&t=10473

if [ "$1" ]; then  # open given file(s) or directory/directories
  subl "$@" &
else  # no arguments
  PRJ=`ls -1 *.sublime-project 2> /dev/null`
  if [ "$PRJ" ]; then  # At least one project exists in current folder, open them all
    for P in "$PRJ"; do
      subl --project "$P" &
    done
  else  # no file specified and no projects available so open current directory
    subl . &
  fi
fi
```

s hosted with ❤ by **GitHub** **view raw**

Don't forget to give the script execute permission. For Mac, type:

```
chmod a+x bin/s
```

For Linux and Cygwin, type:

```
chmod a+x /usr/local/bin/s
```

Running the Scala REPL in Sublime Text

This is extremely cool!

1. You can run `gen-sublime` first if you wish, but that is not necessary for `sublimeREPL` to work.
2. Open a project directory in Sublime Text. The `s` command I provided earlier is a good way to load an SBT project into Sublime Text.
3. Open a new column or row in Sublime Text for the REPL.
   a. Mac:
      i. To switch to 2 column mode: `⌘` - `2`.
      ii. To switch to 2 row mode: `⌘` - `Shift` - `2`.
   b. Windows and Linux:
      i. To switch to 2 column mode: `Alt` - `Shift` - `2`.
      ii. To switch to 2 row mode: `Alt` - `Shift` - `8`.
4. Click in the pane where you want to SBT REPL to appear, and select the menu option **Tools** / **Sublime REPL** / **Scala** / **SBT for opened folder**. SBT will start up, download whatever dependencies are outstanding, and eventually present you with a `>` prompt on Mac, or a `$` prompt on Windows/Linux This is the SBT command prompt. Type in commands like `run`, `~run`, `run-main`, `test`, `testQuick` or `console`. For example, with the `courseNotes` project, try typing

```
~run-main PackageDemo
```

5. If you type `console`, the SBT project will update, compile, and the REPL will load your project classpath. You can then type in anything you wish to be evaluated. The `~console` command will only recompile a modified project after you type `exit` to get back to the SBT command prompt; once recompiled the console will automatically be re-entered.

If you accidently load a source file into the REPL window, double-click on the tab to restore the REPL.

To exit the REPL:

1. Click on the pane that has your source code.
2. For Mac, press `⌘` - `1`; for Windows and Linux: `Alt` - `Shift` - `1`.

1-5 Hello, World in Scala

ScalaCourses.com / scalaIntro / ScalaCore / scalaRunning / scalaScripts

There are two ways to execute Scala code as a script.

Scala File

This method of writing a Scala script requires that your Scala code be stored into a text file with the `scala` file type. 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. You can use Sublime Text, `vi` or your favorite text editor to create this file.

1. Using the `s` script that we saw at the end of the last lecture to launch Sublime Text and edit a new file:

```
s hello.scala
```

2. Using `vi`:

```
vi hello.scala
```

The entire file should contain only this one line:

```
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!
```

Remember, if your Mac laptop 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

You can skip this section without missing anything.

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 to is to put the following at the top of the file, and to make the file executable:

```
#!/bin/sh
exec scala "$0" "$@"
!#
```

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 "$0" "$@"
!#

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 SBT Global Setup

`ScalaCourses.com` / `scalaIntro` / `ScalaCore` / `scalaRunning` / `sbtGlobalSetup`

Git is almost always used with SBT projects. I recommend that you get to know the git command line, if you are not already using it.

SBT uses a Maven-compatible directory structure for your project:

```
+---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. SBT uses Apache Ivy to manage dependencies, which are placed in `~/.ivy` by default.

SBT performs a two-stage build: first it compiles the project builder files, then it uses the project builder to compile and perhaps run the project code. The project builder meta-project consists of some or all of these files:

- If you are using SBT 0.13, all `.sbt` files in `~/.sbt/0.13/`; if that directory does not exist or you are using an older version of SBT, all `.sbt` files in `~/.sbt/`
- All `.sbt` files in the top-level directory of your project, plus `Build.scala` if it exists in that directory.
- All files in your project's `project/` directory.

`.sbt` files must be double-spaced. This is because of the contents are parsed as a domain-specific language (DSL) which is then compiled as a Scala program. SBT combines all of the `.sbt` files together and loads them as one when it starts up, so you can put the project information into as few or as many files as you like. By convention, `project/plugins.sbt` defines the SBT plugins you want to make available to your project builder. We will look at some popular plugins in a minute.

`project/build.properties` contains the version of SBT to be used with the project. You should specify this value, so builds do not break as SBT and Scala continue to evolve. Here are typical contents:

```
sbt.version=0.13.1
```

Installation

You can use these instructions, or follow along below.

Mac

MacPorts and Homebrew both work. When this video was made, the sbt script installed via Homebrew and Macports was deficient; it did not support the `-mem` option. Here is a workaround.

```
brew install sbt
```

Linux

For a Debian distro like Ubuntu, type:

```
sudo apt-get install sbt
```

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 $JAVA_OPTS -jar 'e:/storage/programming/scala/sbt-launch.jar' "$@"
```

Manual install

Adjust the path to `sbt-launch.jar` to suit your computer. The manual installation for SBT does not provide the `-mem` option which allocates JVM memory appropriately, given the total amount of memory that you wish to allocate to it. You can install the scripts from `sbt-launcher-package` if desired.

For Mac:

```
#!/bin/bash
export JAVA_HOME=`/usr/libexec/java_home`
export JAVA_OPTS="$JAVA_OPTS -XX:+CMSClassUnloadingEnabled"
java $JAVA_OPTS -jar '/opt/scala/sbt-launch.jar' "$@"
```

For Linux:

```
#!/bin/bash
export JAVA_HOME="$(readlink -f /usr/bin/javac | sed "s:bin/javac::")"
export JAVA_OPTS="$JAVA_OPTS -XX:+CMSClassUnloadingEnabled"
java $JAVA_OPTS -jar '/opt/scala/sbt-launch.jar' "$@"
```

Checking the SBT Version

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
```

Unfortunately, if your SBT project fails to build, this command will not display the SBT version. This is frustrating if you need to discover the version SBT that is failing to build your project. The solution is to run this command from another directory.

Global Setup

SBT compiles a project builder program from the global SBT configuration in `~/.sbt/`, project-specific files `Build.scala`, `*.sbt` and the contents of `project/`. Each time you run SBT, it executes all of these files. Although you can store all the project configuration settings in one file, it is often better to split the commands across multiple files so you can manage them better.

.sbt files must be double-spaced

Unversioned Setup

Prior to SBT 0.13, there was only one global SBT setup, in `~/.sbt/`. Globally installed plugins were located in `~/.sbt/plugins/`. For Windows, including Cygwin, use `%USERPROFILE%/.sbt/` and `%USERPROFILE%/.sbt/project/`.

Here is my `~/.sbt/plugins/build.sbt`, set up for SBT 0.12.3 and Scala 2.10. This file provides plugins for all SBT projects.

```
resolvers += Resolver.url("artifactory", url("http://scalasbt.artifactoryonline.com/scalasbt/sbt-plugin-releases"))(Resolver.ivyStylePatterns)

resolvers += "sbt-idea-repo" at "http://mpeltonen.github.com/maven/"

addSbtPlugin("com.github.mpeltonen"    % "sbt-idea"          % "1.5.2")

addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "2.1.0")

addSbtPlugin("com.orrsella" % "sbt-sublime" % "1.0.9")
```

Here is my `~/.sbt/eclipse.sbt`, which causes `sbteclipse-plugin` to download source files for all dependencies when converting an SBT project for use with Eclipse:

```
EclipseKeys.createSrc := EclipseCreateSrc.Default + EclipseCreateSrc.Resource

EclipseKeys.withSource := true
```

SBT 0.13 Versioned Setup

Globally installed plugins (those installed in `~/.sbt/plugins/`) might work with one version of SBT but might not work with another version of SBT. Because SBT is able to automatically install any older version of SBT as required for the project you are currently working with, the plugins did not behave in a predictable manner. SBT 0.13 introduced the capability for you to configure SBT plugins differently for each version of SBT called for by your projects. This means that a project that requires SBT 0.12.3, for example, can invoke an older version of the Eclipse plugin, while a newer project that uses SBT 0.13.1 can use a newer version of that same plugin – with

different settings, if desired. To support this, versioned SBT setup was introduced. Global settings specific to SBT 0.13.x go in `~/.sbt/0.13/` and global plugins specific to that version of SBT go in `~/.sbt/0.13/plugins/`. For Windows, including Cygwin, use `%USERPROFILE%/.sbt/0.13/` and `%USERPROFILE%/.sbt/0.13/project/`.

If you do not create a `~/.sbt/0.13/` directory, SBT 0.13+ runs in compatibility mode, and issues this warning that suggests you should create the directory:

```
[warn] The global sbt directory is now versioned and is located at /Users/mslinn/.sbt/0.13.
[warn]   You are seeing this warning because there is global configuration in /Users/mslinn/.sbt but not in /Users/mslinn/.sbt/0.13.
```

The sentence actually says the opposite of what was intended. A better wording might be:

```
There is no versioned global SBT directory. If you want to make one, you could put it at ~/.sbt/0.13/
```

If you already have `~/.sbt/` set up, and you want to create an equivalent versioned setup for SBT 0.13, the easiest way to proceed is as follows:

```
$ cp -a ~/.sbt ~/.sbtTemp
$ mv ~/.sbtTemp ~/.sbt/0.13
```

Now edit the contents of `~/.sbt/0.13` to suit.

Here is my `~/.sbt/0.13/plugins.sbt` (for Windows including Cygwin, this file is found at `%USERPROFILE%/.sbt/0.13/plugins.sbt`):

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "2.4.0")

addSbtPlugin("com.github.mpeltonen"    % "sbt-idea"          % "1.5.2")

addSbtPlugin("com.orrsella"            % "sbt-sublime"       % "1.0.9")
```

Plugins

Here some popular SBT plugins:

- dependencyReport - Lists all dependencies (direct and transitive) of an SBT project
- sbt-assembly - Build executable jar with all dependencies
- sbt-dependency-graph - Displays an ASCII graph of hierarchical direct and transitive dependencies
- sbt-eclipse - Converts SBT projects to Eclipse. If you use Eclipse, create `~/.sbt/0.13/eclipse.sbt` containing this line:

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

- sbt-idea - Converts SBT projects for use with IntelliJ IDEA; this plugin is no longer needed for IDEA 13 for most SBT projects because IDEA 13 can now load `build.sbt` directly.
- sbt-sublime - Converts SBT projects for use with Sublime Text.

.sbtrc

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

Here are some handy definitions which can be placed in either of these files. They cause SBT to continually monitor your project's source code tree, and whenever a file changes, clear the console and then compile or run the project. There are a few tricks used:

1. Tilde (~) causes whatever follows it to be performed each time a file in the project changes.
2. A semicolon (;) is equivalent to a newline.
3. ANSI escape codes are output.

```
# Clear the screen and recompile the program each time a change is made
alias cc = ~; eval "\u001B[2J\u001B[0\u003B0H"; compile

# Clear the screen and rerun the program each time a change is made
alias rc = ~; eval "\u001B[2J\u001B[0\u003B0H"; run

# Clear the screen and only rerun changed unit tests specific to the modified code
alias tc = ~; eval "\u001B[2J\u001B[0\u003B0H"; testQuick
```

Use as follows:

```
$ sbt cc
```

Now open the `courseNotes` project in an editor and make a small change to `src/main/scala/Hello.scala`. When you save that file, SBT will recompile the changed project. SBT remains running in the background, keeping the compiler 'warm'. Let's try out the `rc` alias. Exit SBT by typing `Ctrl`-`D` and then restart with the `rc` alias:

```
$ sbt rc
```

`Hello.scala` should run. Make another small change to the file, and the program will recompile and rerun.

1-7 SBT Project Setup

ScalaCourses.com / scalaIntro / ScalaCore / scalaRunning / sbtProjectSetup

I created a GitHub project for use as a template for new SBT 0.13 projects.

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

This is the main `build.sbt` file from `sbtTemplate` which specifies how your project should be built.

```
organization := "com.micronautics"

name := "changeMe"

version := "0.1.0-SNAPSHOT"

scalaVersion := "2.10.3"

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

scalacOptions in (Compile, doc) <++= baseDirectory.map {
  (bd: File) => Seq[String](
     "-sourcepath", bd.getAbsolutePath,
     "-doc-source-url", "https://github.com/mslinn/changeMe/tree/master€{FILE_PATH}.scala"
  )
}

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(
// "org.scalatest" %% "scalatest" % "2.0" % "test" withSources,
// "com.github.nscala-time" %% "nscala-time" % "0.6.0" withSources
)

logLevel := Level.Warn

// define the statements initially evaluated when entering 'console', 'consoleQuick', or 'consoleProject'
initialCommands := """
""".stripMargin

// Only show warnings and errors on the screen for compilations.
// This applies to both test:compile and compile and is Info by default
logLevel in compile := Level.Warn
```

Notes:

- If you only have JDK 6 installed and not JDK 7, change all three instances of the number 7 above (highlighted in yellow) to 6.
- SBT launches the Scala compiler to build Scala code, and it also uses the Java compiler to compile Java code. Options for each of these compilers are specified in `scalacOptions` and `javacOptions`, respectively. I recommend that you always use these settings in every project.

- The second instance of `scalacOptions` is the magic incantation to locate ScalaDoc in a GitHub project. Yes, that is a Euro symbol. You can delete this statement if this is not a GitHub project.
- `resolvers` is a comma-delimited list of URLs where SBT should look for dependencies. Each item in the list consists of an arbitrary friendly name of the dependency, followed by `at` and then the URL to check.
- SBT is very noisy. You can suppress most of the log output by setting `logLevel` to `Level.Warn` as shown in the above `build.sbt`. Here is an example of how to temporarily increase logging verbosity in order to track down a problem when compiling:

  ```
  $ sbt
  > set logLevel := Level.Info
  > compile
  ```

  SBT allows everything to be specified on the command line. Here we see that a semicolon is treated like a newline, so the above is equivalent to the following:

  ```
  $ sbt "; set logLevel := Level.Info; compile"
  ```

Dependencies

Your source code normally has dependencies. SBT allows these dependencies to be specified in the top-level `build.sbt`. `libraryDependencies` are specified using the following fields. I am using BNF to express the syntax:

```
groupId % dependencyName % | %% version [% test | % compile] [withSources]
```

For example, you could specify that this project uses the Akka library, like this:

```
"com.typesafe.akka" %% "akka-actor" % "2.2.3" withSources
```

In the above the dependency is specified as:

- Maven group id is `com.typesafe.akka`
- Maven artifact id is `akka-actor`
- Version is `2.2.3`
- Source code is fetched in addition to the executable JAR.

Because Scala is not binary compatible between releases, a separate version of every library needs to be created for each version of the Scala compiler that programmers might want to link with. The `%%` in the above incantation performs name mangling to fetch the proper version of the dependency. The name is mangled by appending the version of the Scala compiler used in your project to the Maven artifact id. Double percent signs should only be used with dependencies written in Scala. Dependencies written in Java are always written with only one percent sign.

You could also specify the Akka dependency without using the double percent characters as follows, so the version that was compiled with Scala 2.10 is specifically pulled in:

```
"com.typesafe.akka"  %  "akka-actor_2.10" % "2.2.3"
```

`withSources` is optional; feel free to specify it after the version of the dependency if you wish.

When you update the project to build with the Scala 2.11 compiler, you will need to update this dependency because you did not use the double percent sign:

```
"com.typesafe.akka" % "akka-actor_2.11" % "2.2.3"
```

Several dependencies could be specified one at a time, double-spaced. Note the operator used to append a single value to `libraryDependencies` is `+=` (with only one plus sign):

```
libraryDependencies += "org.scalatest" %% "scalatest" % "2.0" % "test" withSources

libraryDependencies += "com.github.nscala-time" %% "nscala-time" % "0.6.0" withSources
```

The following does the same thing, and may be easier to read. Note that the operator is `++=` (with two plus signs):

```
libraryDependencies ++= Seq(
 "org.scalatest"          %% "scalatest"   % "2.0"   % "test" withSources,
 "com.github.nscala-time" %% "nscala-time" % "0.6.0" withSources
)
```

In this format, notice:

- Zero or more dependencies can be wrapped within a `Seq()`
- Each dependency is delimited from the next by a comma
- The entire statement needs no blank lines inside
- I used spaces to format the fields of the dependencies so they line up

Discovering Dependencies and Versions

Open source projects normally contain a `README` file that indicates the preferred version to use. Library dependencies can also be found at `http://mvnrepository.com`. Let's look for `nscala-time`, which is a good Scala wrapper around the Joda time and date utility package.

1. Point your browser to `http://mvnrepository.com`.
2. Enter `nscala-time` into the search box and press `Enter`.
3. The resulting page shows these results:
   nscala-time
   nscala-time
   com.github.nscala-time » nscala-time_2.10

   nscala-time
   nscala-time
   com.github.nscala-time » nscala-time_2.9.2
4. Click on the top link, which is for Scala 2.10. You now see all the versions of the library that have been cross-compiled for that version of the compiler.
5. Select 0.6.0 and then click on the **SBT** tab. You should see the incantation that you should add to `build.sbt`.
   ```
   libraryDependencies += "com.github.nscala-time" % "nscala-time_2.10" % "0.6.0"
   ```
6. Notice that ithe incantation does not use double-percent. You should probably change the incantation as follows:
   ```
   libraryDependencies += "com.github.nscala-time" %% "nscala-time" % "0.6.0"
   ```

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`, `gen-idea` and `gen-sublime` plugins. All of these plugins automatically run `update` before emitting project files.

To create an IntelliJ IDEA compatible project definition from your SBT project, type:

```
$ sbt gen-idea
```

To create an Eclipse-compatible project definition from your SBT project, type:

```
$ sbt eclipse
```

You can also specify that source code jars should be downloaded for all dependencies that provide them:

```
$ sbt "eclipse with-source=true"
```

You can do all of the above in one line, plus add Sublime Text compability:

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

SBT Command Line Tasks

Following are commonly used command-line tasks. 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.

runMain

Runs a main class, passing along arguments provided on the command line. This task can also be specified as `run-main`. For example:

```
sbt 'runMain com.micronautics.akka.dispatch.futureScala.Zip'
```

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

```
sbt "runMain com.micronautics.akka.dispatch.futureScala.Zip"
```

test

Executes all tests that are not marked with `ignore`.

~testQuick

Like test, but uses incremental compiler to only run test classes that are affected by your latest change. This task can also be specified as `~test-quick`.

testOnly

Executes tests in one test class. This task can also be specified as `test-only`. For example:

```
sbt 'testOnly com.blah.MyTest'
```

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

```
sbt "testOnly com.blah.MyTest"
```

help command*

Displays help message or prints detailed help on requested commands.

updateClassifiers

Download sources and Javadoc for all project dependencies

updateSbtClassifiers
Download sources and Javadoc for all meta-project dependencies

Playing with SBT Console

The SBT console is a REPL that provides your entire project, including dependencies, on the `classpath`. This allows you to manually invoke the code in your project in an interactive session.

The `courseNotes` directory in the git repository provided with this course contains a sample project. That project defines several classes in the `com.micronautics.scalaIntro` 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/scalaCore/course_scala_intro_code/courseNotes/project
[info] Set current project to scalaIntroCourse (in build file:/home/mslinn/work/scalaCore/course_scala_intro_code/courseNot
es/)
Welcome to Scala version 2.10.3 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_45).
Type in expressions to have them evaluated.
Type :help for more information.

scala> import com.micronautics.scalaIntro._
import com.micronautics.scalaIntro._

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! Run `sbt console` on the `courseNotes` project and see how it works.

:load

You can load files containing SBT command into an SBT console session. This is good for performing a complex setup while trying out different ideas. If you have a file called `repl/blah`, you can load it into an SBT console session like this:

```
scala> :load repl/blah
```

`repl/blah` might contain:

```
import java.io._
val file = new File("blah.txt")
// more stuff could follow
```

SBT commands

You can enter SBT commands when the SBT REPL is open. The `:help` command gives you a list of all available commands. Tab completion is nicely implemented, so try it!

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

:cp <path>                add a jar or directory to the classpath
:help [command]           print this summary or command-specific help
:history [num]            show the history (optional num is commands to show)
:h? <string>              search the history
:imports [name name ...]  show import history, identifying sources of names
:implicits [-v]           show the implicits in scope
:javap <path|class>       disassemble a file or class name
:load <path>              load and interpret a Scala file
:paste                    enter paste mode: all input up to ctrl-D compiled together
:power                    enable power user mode
:quit                     exit the interpreter
:replay                   reset execution and replay all previous commands
:reset                    reset the repl to its initial state, forgetting all session entries
:sh <command line>        run a shell command (result is implicitly => List[String])
:silent                   disable/enable automatic printing of results
:type [-v] <expr>         display the type of an expression without evaluating it
:warnings                 show the suppressed warnings from the most recent line which had any

scala> :power
** Power User mode enabled - BEEP WHIR GYVE **
** :phase has been set to 'typer'.          **
** scala.tools.nsc._ has been imported      **
** global._, definitions._ also imported    **
** Try  :help, :vals, power.<tab>           **

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

:cp <path>                add a jar or directory to the classpath
:help [command]           print this summary or command-specific help
:history [num]            show the history (optional num is commands to show)
:h? <string>              search the history
:imports [name name ...]  show import history, identifying sources of names
:implicits [-v]           show the implicits in scope
:javap <path|class>       disassemble a file or class name
:load <path>              load and interpret a Scala file
:paste                    enter paste mode: all input up to ctrl-D compiled together
:power                    enable power user mode
:quit                     exit the interpreter
:replay                   reset execution and replay all previous commands
:reset                    reset the repl to its initial state, forgetting all session entries
:sh <command line>        run a shell command (result is implicitly => List[String])
:silent                   disable/enable automatic printing of results
:type [-v] <expr>         display the type of an expression without evaluating it
 :warnings                 show the suppressed warnings from the most recent line which had any
:phase <phase>            set the implicit phase for power commands
```

... And one more thing!

If you've ever wanted to build truly executable (i.e., no `java -jar` needed) JARs with sbt, now you can, using the `sbt-assembly` plugin.

1-8 Working With Scala IDE for Eclipse

ScalaCourses.com / scalaIntro / ScalaCore / scalaRunning / ScalaEclipse

Although you can install Eclipse and the Scala plug-in separately, I 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 the Kepler Eclipse bundle with Java in it.
- Download one of the Scala 2.10.3+ plugins for Scala IDE 3.0.2. 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. *Update: Eclipse Kepler is now the default. Do not copy the entire Juno config file for use with Kepler - just consider the yellow highlighted lines.*

```
-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, kepler}`.

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

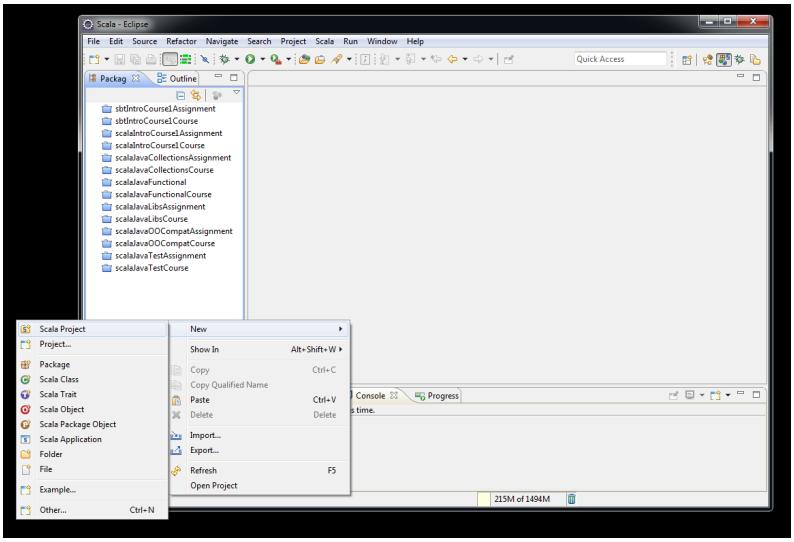
1. Enable **Always run in background** and **Show heap status**.
2. **General**

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 `New...` 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.

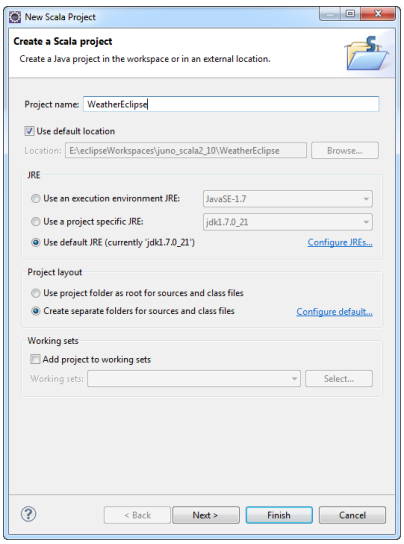
The preconfigured Eclipse with Scala in it does not have a git plugin. For most Scala programmers, git is essential. You should install the EGit plugin. Lars Vogel has an excellent tutorial on how to do this, and how to work with Git from Eclipse.

Creating a New Scala Project

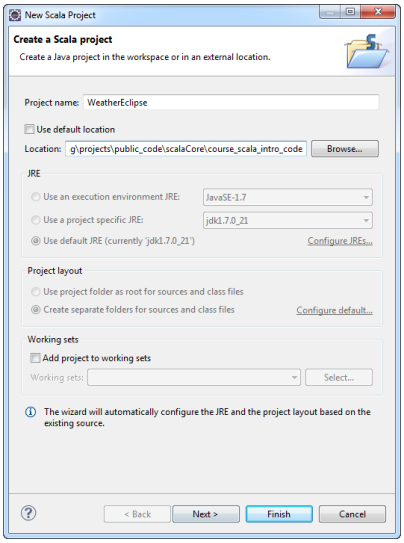
1) Start Eclipse / Scala IDE, and click on **New / Scala Project**.	



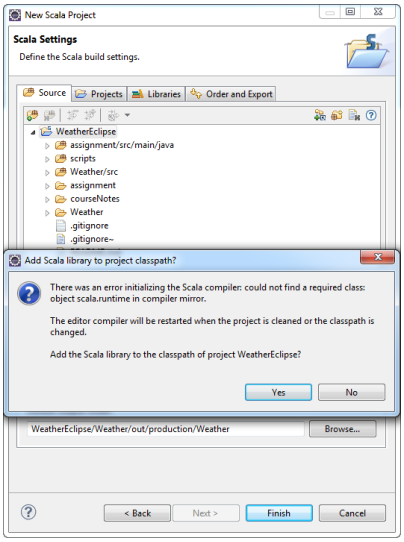
2) Give your new project a name (`WeatherEclipse`).



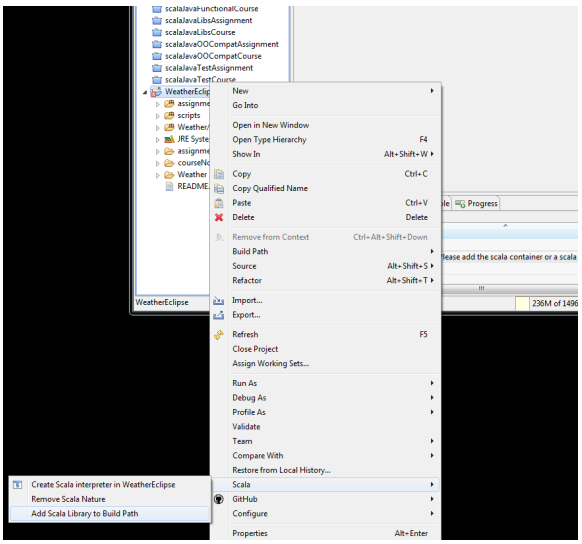
3) If there is a directory that you prefer to keep your work in (I recommend this!), uncheck **Use default location**, click `Browse...`, select the directory, and when you are back in this dialog, click `Next >`.



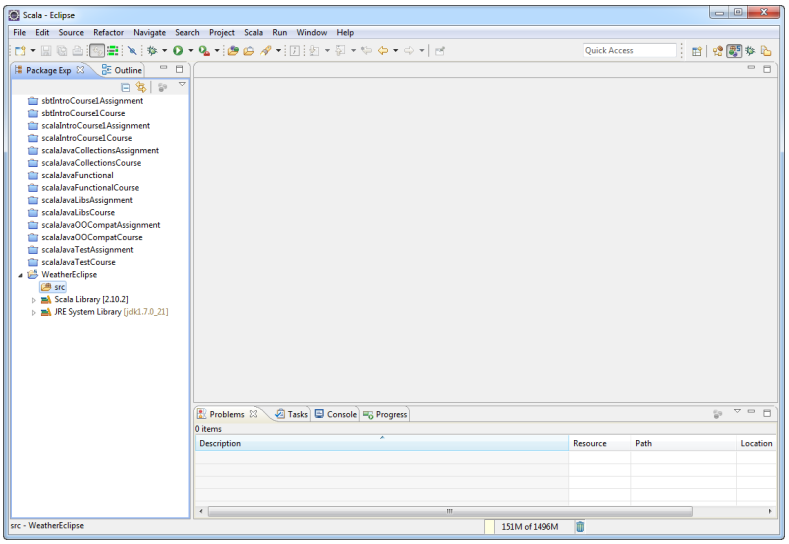
4a) If you did not define the `SCALA_HOME` environment variable you will be presented with this error message.



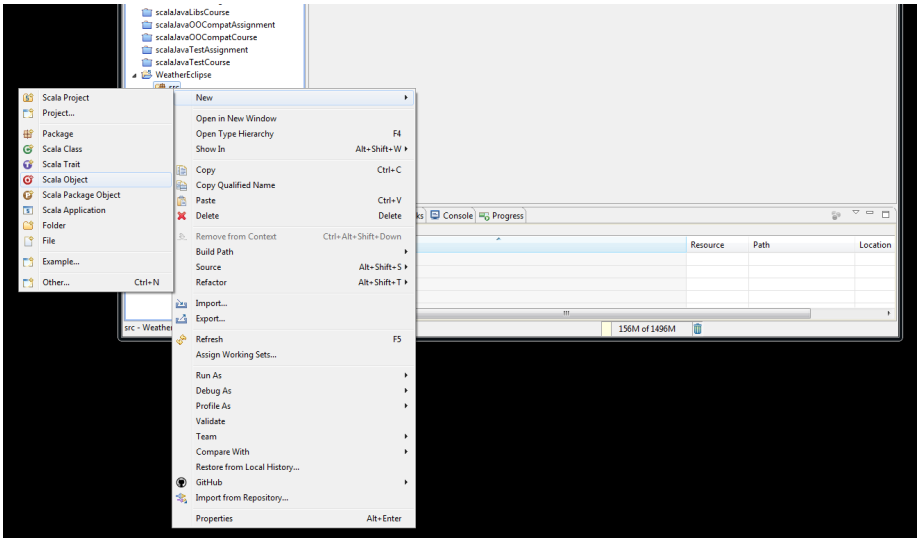
4b) To fix the missing Scala library problem, right-click on the project name in the Project panel, select the **Scala** item and click on **Add Scala Library to Build Path**.



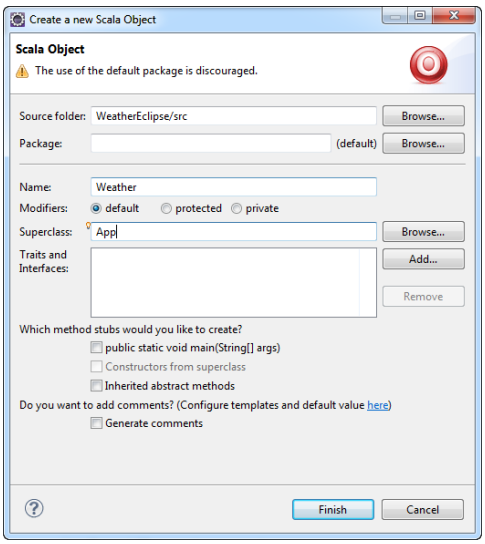
6) The `src` directory is automatically created for you. This directory contains your source code, including Java and Scala code, and can also include resources.



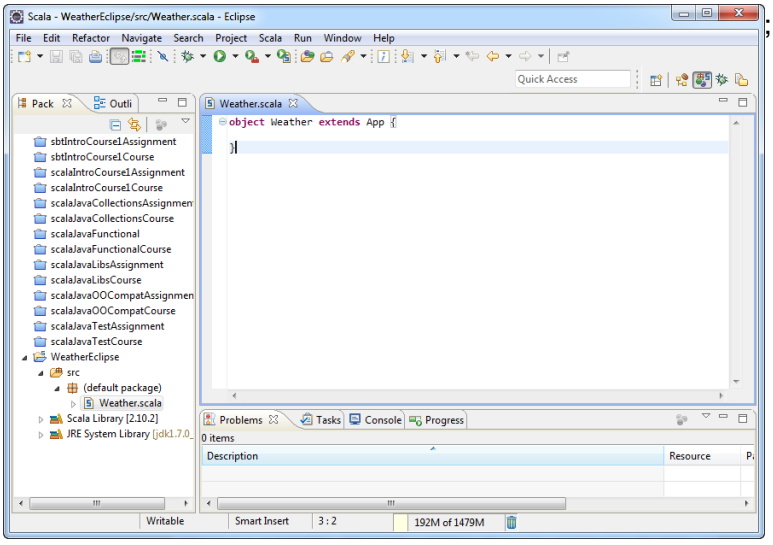
7) To make a source file in the `src` directory, which will contain the main program's entry point, right-click on the `src` directory, select **New** / **Scala Object**.



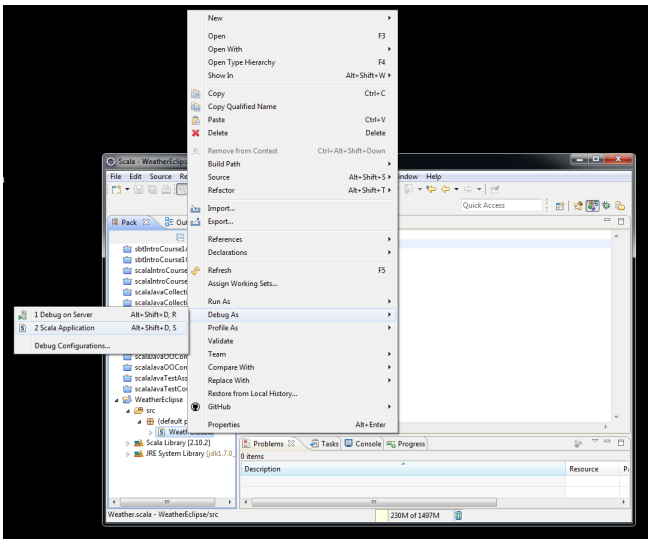
8) Give your new Scala file a name (`Weather`). The Scala file will have a `.scala` file type automatically added. Note that I defined the superclass to be `App`; this makes the constructor act as the main entry point. Click `Finish`



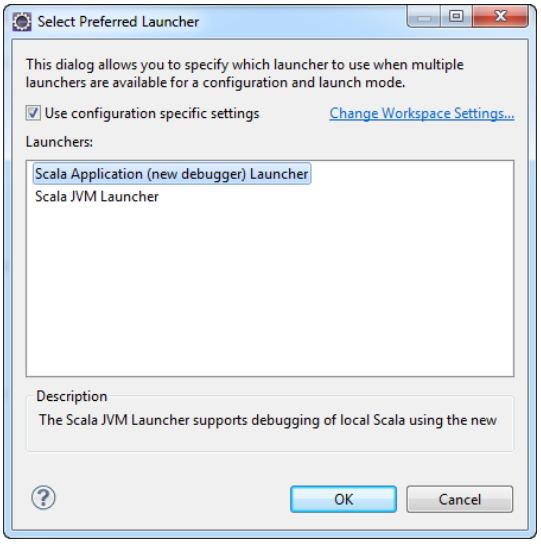
9) Eclipse creates a default Scala class with the name of the file.



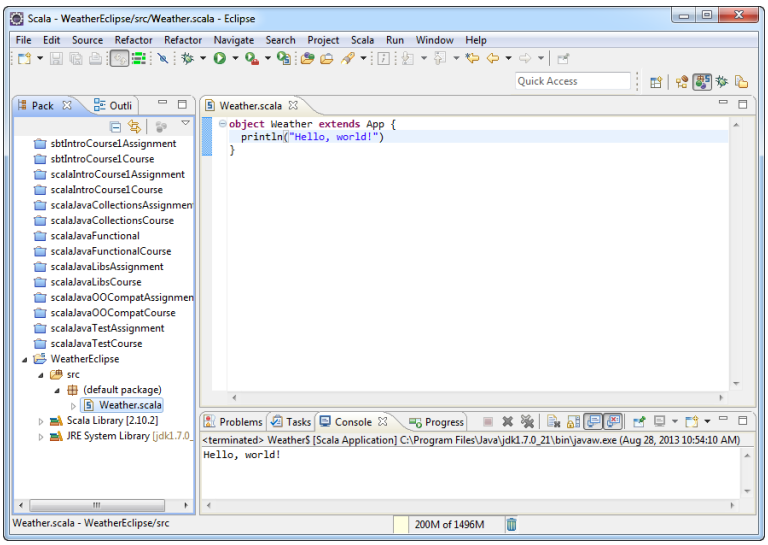
10) Let's run the newly created console app by right-clicking on it in the **Project** panel and selecting **Debug As / Scala Application**.



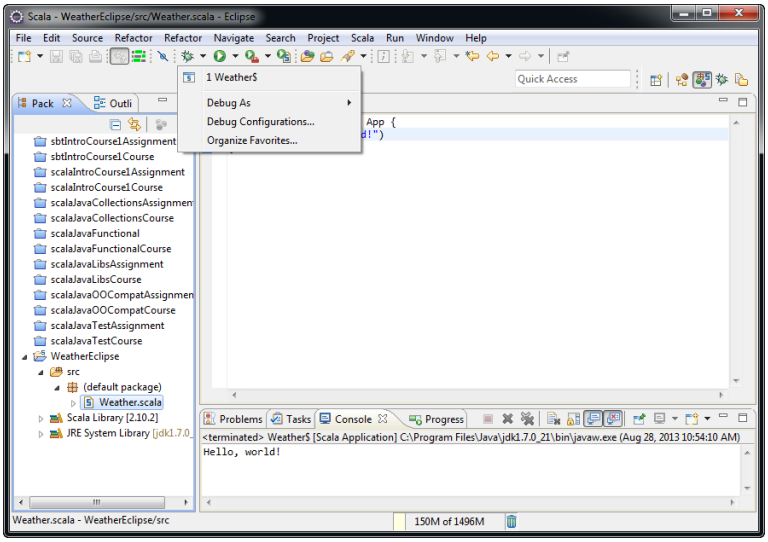
11) Enable **Use configuration specific settings** and select **Scala Application (new debugger) Launcher**.



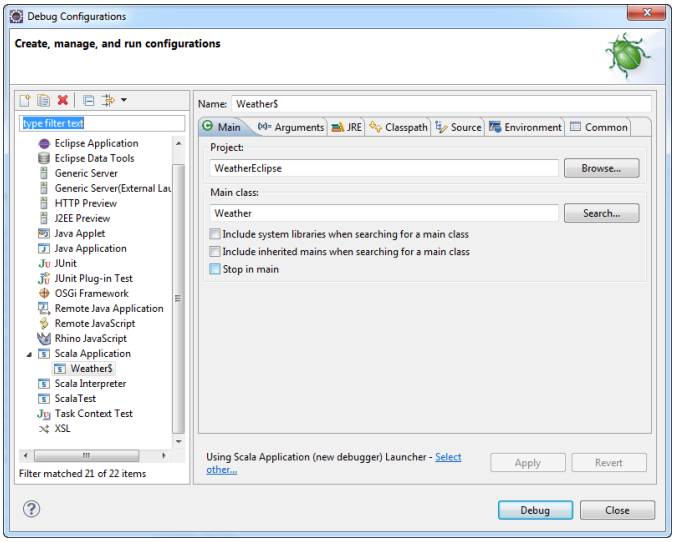
12) Output appears in the **Debug** panel which opens up at the bottom of the Eclipse window. You can click on the **Console** tab to show or hide the debug output. You can rerun the app by clicking on the little green bug at the top the Eclipse window.



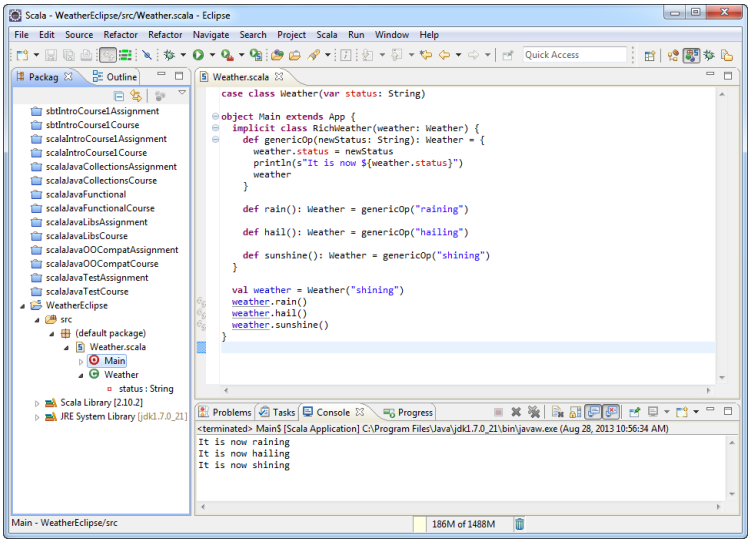
13) You can edit the debug configuration by clicking on the pull-down menu next to the little green bug at the top center of the Eclipse window.



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:\scalaCore\scalaIntro\courseNotes\project
[info] Set current project to scalaIntroCourse (in build file:/C:/scalaCore/scalaIntro/courseNotes/)
[info] About to create Eclipse project files for your project(s).
[info] Successfully created Eclipse project files for project(s):
[info] scalaIntroCourse
```

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:\scalaCore\scalaIntro\assignment\project
[info] Set current project to scalaIntroAssignment (in build file:/C:/scalaCore/scalaIntro/assignment/)
[info] About to create Eclipse project files for your project(s).
[info] Successfully created Eclipse project files for project(s):
[info] scalaIntroAssignment
```

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 `Next >`
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 `Ctrl` - `Shift` - `X`.

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**.

`Ctrl` / `⌘` 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** (⌘-Alt-o on Mac, Ctrl-Alt-o 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. This is a truly awkward key sequence! Change it to Alt + = using **Preferences** / **General** / **Keys**, click on **Show Type** command, then enter new value in **Binding** text box.

Ctrl + Space Code completion.

Ctrl + F Search and perhaps replace in current file.

Ctrl + Shift + L Show key bindings.

Eclipse 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 current selection.

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 Working With IntelliJ IDEA

`ScalaCourses.com` / `scalaIntro` / `ScalaCore` / `scalaRunning` / `ScalaIDEA`

You need IDEA 12 or IDEA 13 to work with the projects provided with this course. JetBrains offers a 30 day trial for IDEA Ultimate, but the free version should also work. You need Ultimate in order to work with Play. This lecture was originally produced with IDEA 12.1.3 and updated with IDEA 13.0.

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 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 &
```

Under Mac:

```
/Applications/IntelliJ\ IDEA\ 13.app/Contents/MacOS/idea &
```

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.

If you import a settings file (from the Welcome to IntelliJ IDEA screen: **Configure / Import settings**), by default called `settings.jar`, then all of the following settings could be set.

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 (and you should not!), disable the **Playframework support** plugin because that only works with Play Framework version 1.
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 `New...` button, then select **JDK**. Browse to your JDK. For Ubuntu Linux, this will be under `/usr/lib/jvm`.
    3. Click `OK` 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 Contro**l: 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** key map. Unless you and your entire team are committed to programming exclusively on the Mac, I suggest you do not use the Eclipse Mac key bindings, because they do not map to Linux or Windows key bindings for Eclipse.

    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

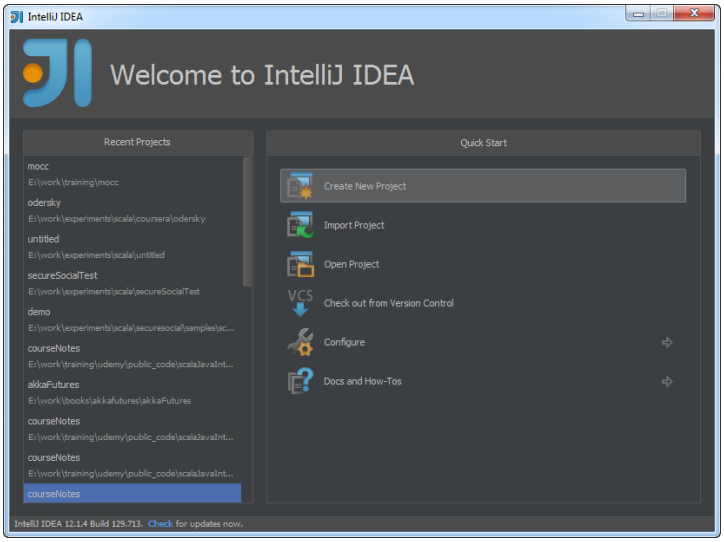
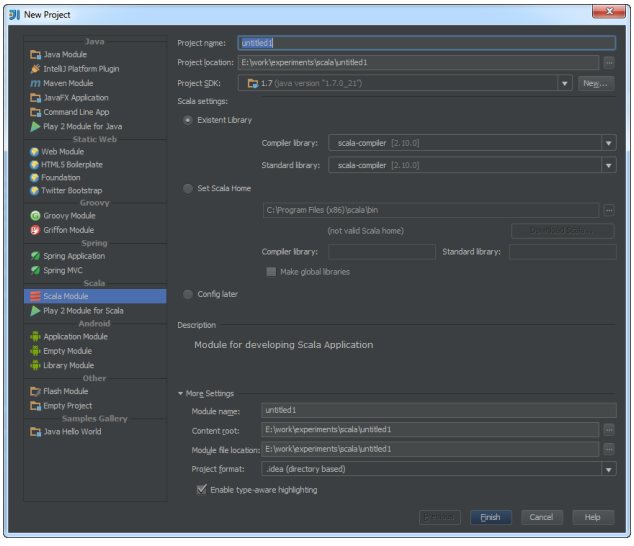
You can use `jvisualvm`, provided with the Java Development Kit, to measure memory usage of the IDE, so you can optimize the memory settings. The VisualVM site has many articles that discuss how to use the tool. I suggest that you do not use the VisualVM IDE plugin to measure the IDE that it runs from,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

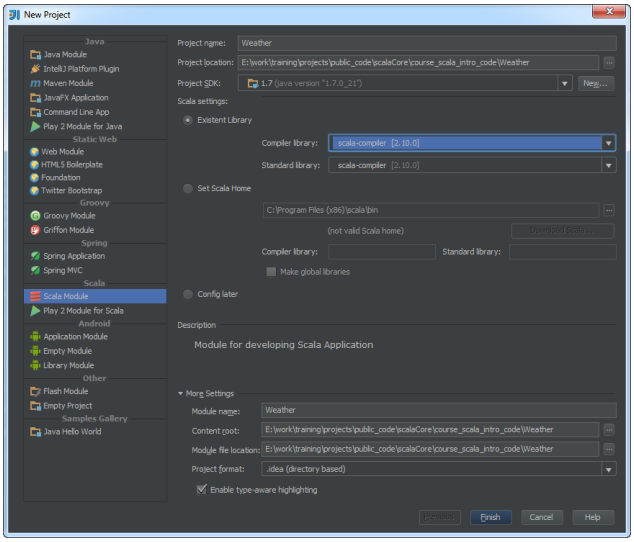
In the IDEA installation directory, edit `bin/idea.vmoptions`. My development machine has 32 GB RAM, and I use that memory to run several virtual machines simultaneously. Each VM has 7 GB RAM. Here are my settings:

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

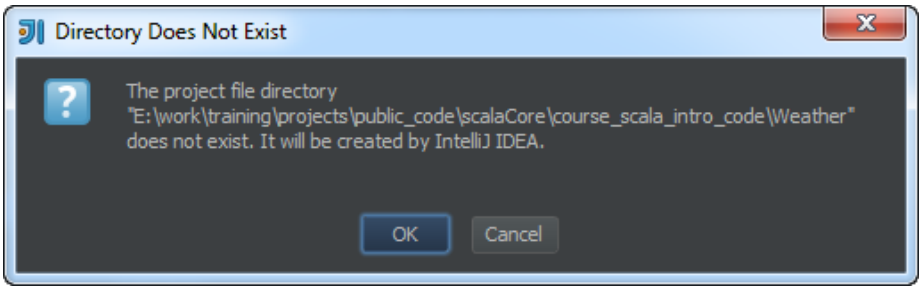
Creating a Scala Project

Recommendation - do not follow this process to create a scala project from IDEA, instead create an sbt project (copy `sbtTemplate` to make your life easier) then convert it to IDEA using `gen-idea`, described next.

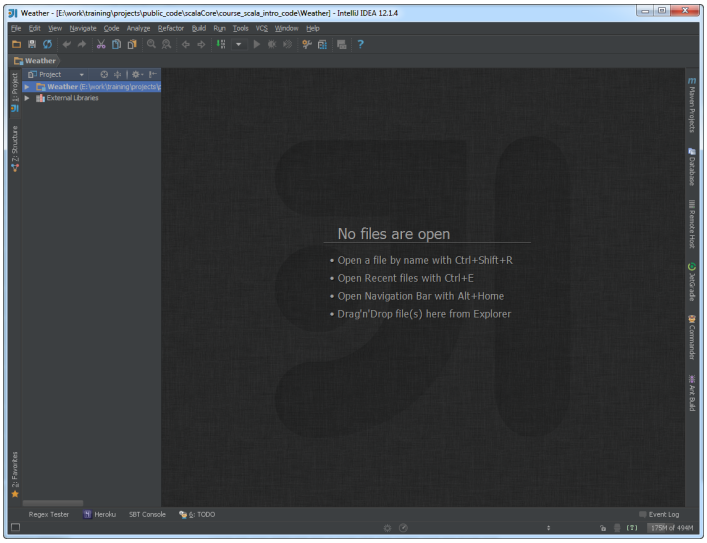
1) Start IntelliJ IDEA, and click on **Create New Project**.	
2) Give your new project a name.	
3) The project name will be used as the IntelliJ module name by default.	



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



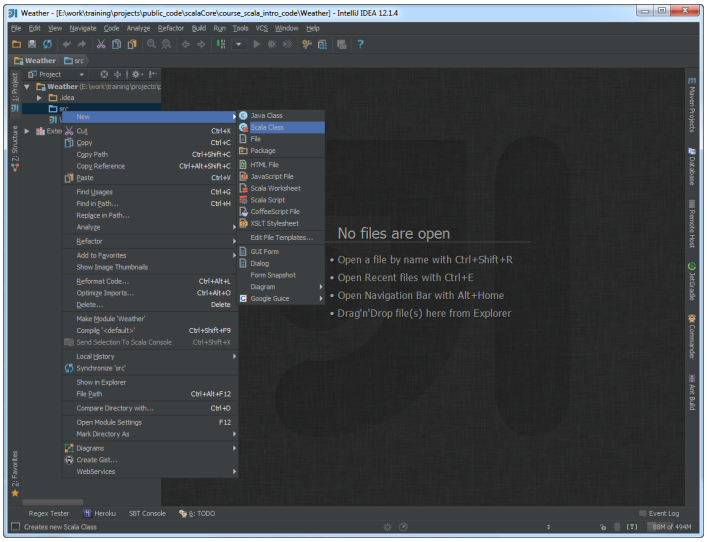
5) After a moment the **Project** panel will open up. Click on the module name to open it. The top one or two directories are the `.idea*` directories; normally you should not need to look inside.



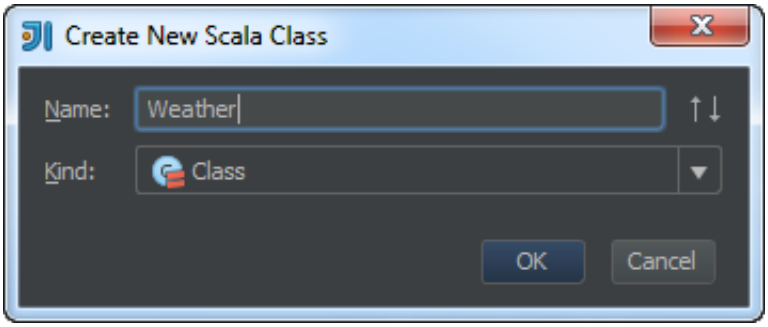
6) The `src` directory is automatically created for you. This directory contains your source code, including Java and Scala code, and can also include resources. To make a source file

inside it, right-click on the `src` directory, select **New** and then **Scala Class**.

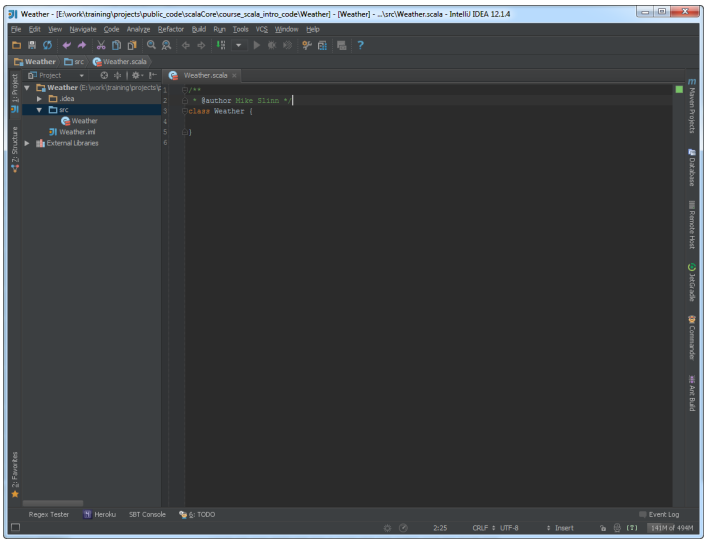
We select **Scala Class** even if we want to make a Scala object or a Scala case class.



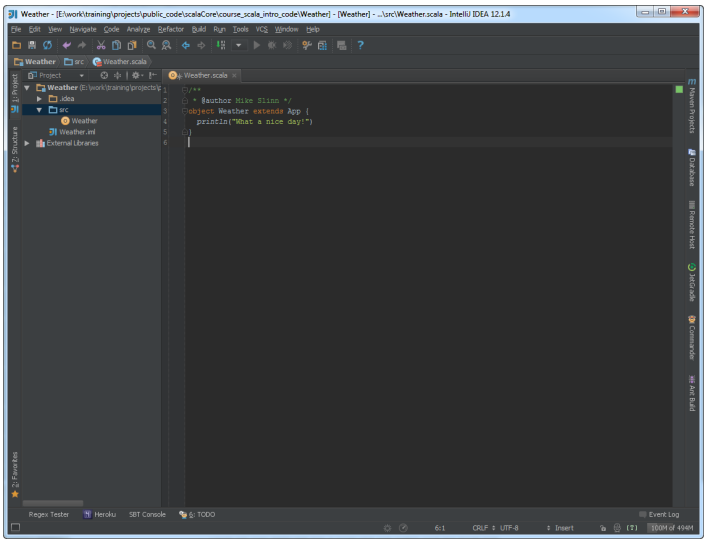
7) Give your new Scala file a name. The Scala file will have a `.scala` file type automatically added.



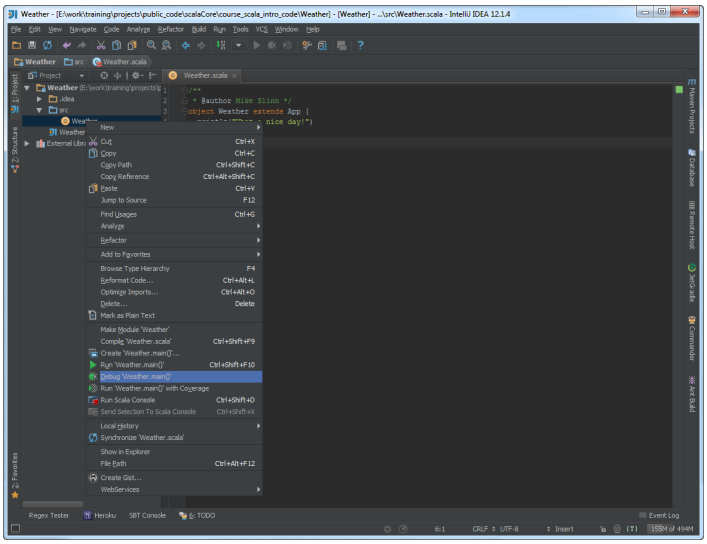
8) IDEA creates a default Scala class with the name of the file.



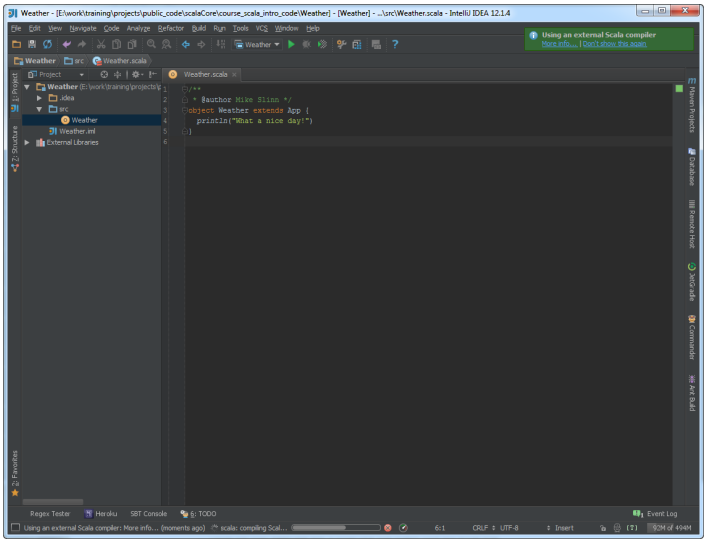
9) We need a main method for our console application. This means we need to extend an `object` (not a class) with the `App` trait. Change the word `class` to `object` and add `extends App`. The constructor for the object consists of only one line, which prints the traditional `Hello, world!`



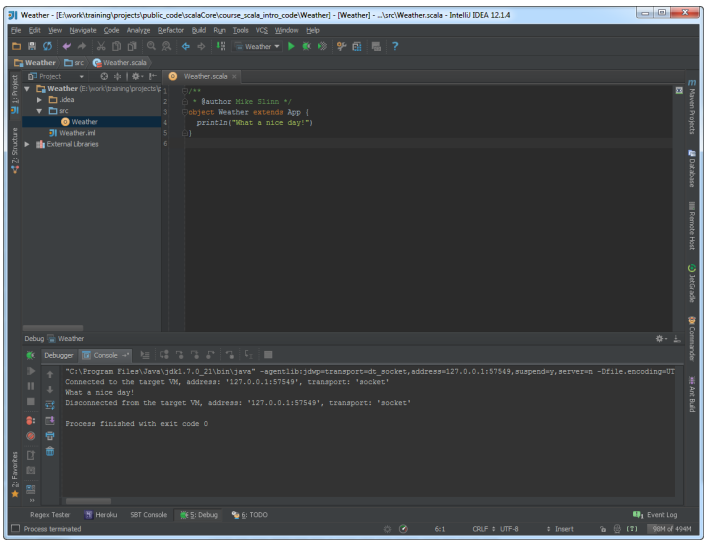
10) Let's run the console app by right-clicking on it in the **Project** panel and selecting **Debug Weather.main()**.



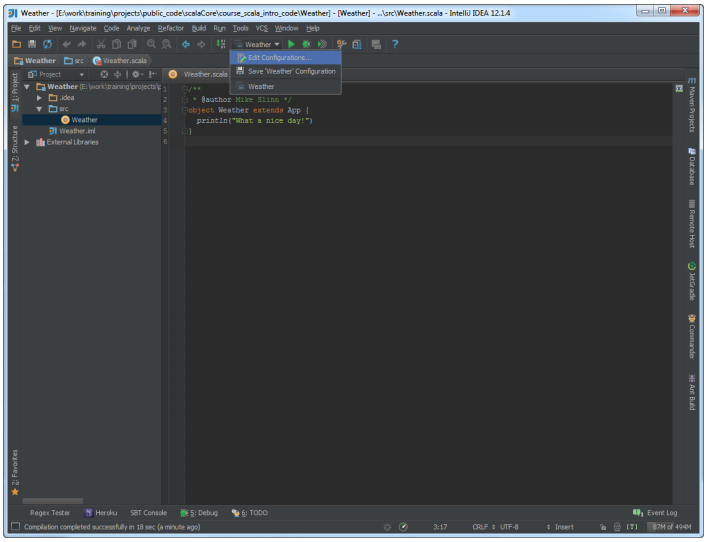
11) You are notified that an external Scala compiler is being used to compile the program. This is good. While IDEA builds, you will see the status line at the bottom of the screen shows all the actions performed.



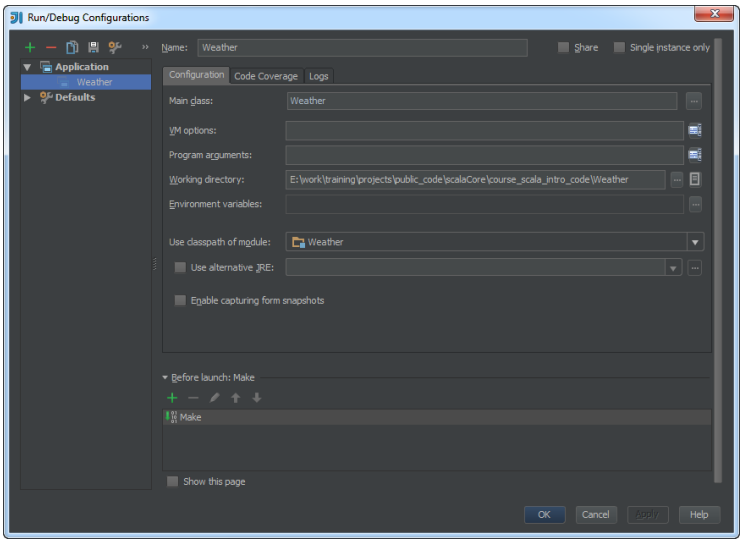
12) Output appears in the **Debug** panel which opens up at the bottom of the IDEA window. You can click on the **Debug** tab to show or hide the **Debug** panel. You can rerun the app by clicking on the little green bug on the left side of the **Debug** panel.



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

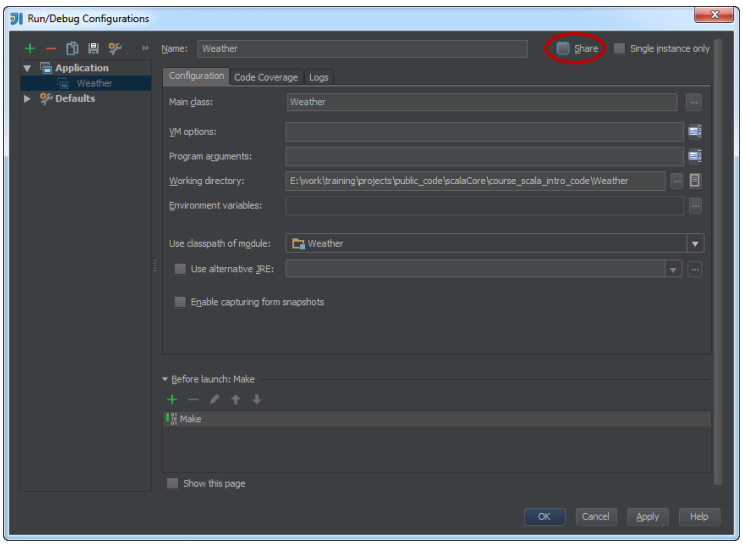


14) You can add arguments for the JVM and your program, as well as tweak other settings.

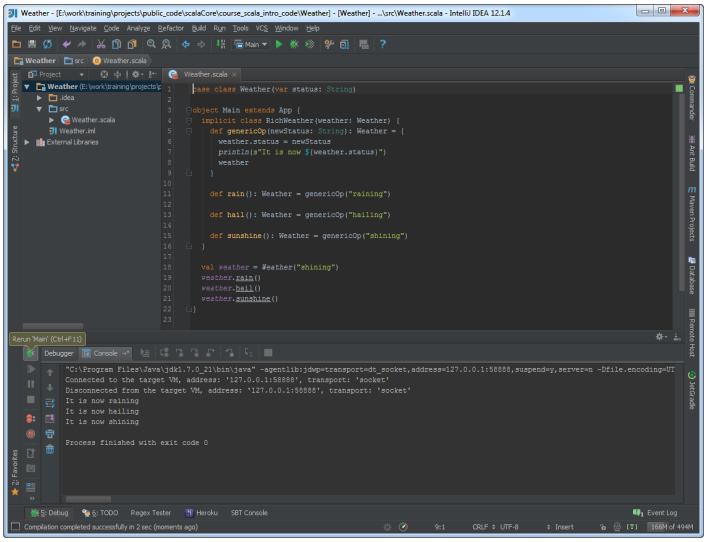


15) If you enable **Share**, the debug configuration is written to a file in the .idea directory. You can check it into git if you want:

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

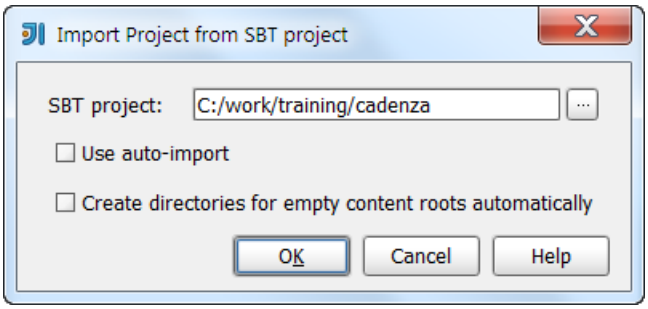


16) A completed program.



Opening SBT projects

IntelliJ 13 for Scala has some new options which are non-obvious. You can now open a `build.sbt` file instead of running `sbt gen-idea`, discussed in the next section. When you do so, you will see the following dialog box. You should always enable both options.



"**Use auto-import**" option turns on subsequent monitoring of relevant project configuration files (`*.sbt, ./project/**`) and project refreshing on changes in those files.

Create directories ... " checkbox can be used to automatically create source and test directories (like `main/src/java/`) that are reported by SBT but not yet present in a project (so that **Project Structure / [Module] / Sources** contained no red entries.

The JetBrains Scala team indicated to me that they will soon add the means to explicitly update projects. They also told me that the checkboxes will probably be removed from the project import dialog in a future release, possibly in mid-January 2014. To modify these settings afterwards, you may access them via **Project Settings / SBT**.

Converting SBT Projects Into IntelliJ Projects

This section discusses how to work with IntelliJ IDEA on Scala projects in a more traditional manner, instead of opening SBT projects directly. This might be required if you are using IDEA 12, or if you have a complex SBT setup.

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:\scalaCore\scalaIntro\courseNotes\project
[info] Set current project to scalaJavaOOCompatCourse (in build
file:/C:/scalaCore/scalaIntro/courseNotes/)
[info] Trying to create an Idea module scalaJavaOOCompatCourse
[info] Excluding folder target
[info] Created C:\scalaCore\scalaIntro\courseNotes/.idea/IdeaProject.iml
[info] Created C:\scalaCore\scalaIntro\courseNotes\.idea
[info] Excluding folder C:\scalaCore\scalaIntro\courseNotes\target
[info] Created C:\scalaCore\scalaIntro\courseNotes\.idea_modules/scalaJavaOOCompatCourse.iml
[info] Created
C:\scalaCore\scalaIntro\courseNotes\.idea_modules/scalaJavaOOCompatCourse-build.iml
```

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

```
sbt gen-idea
```

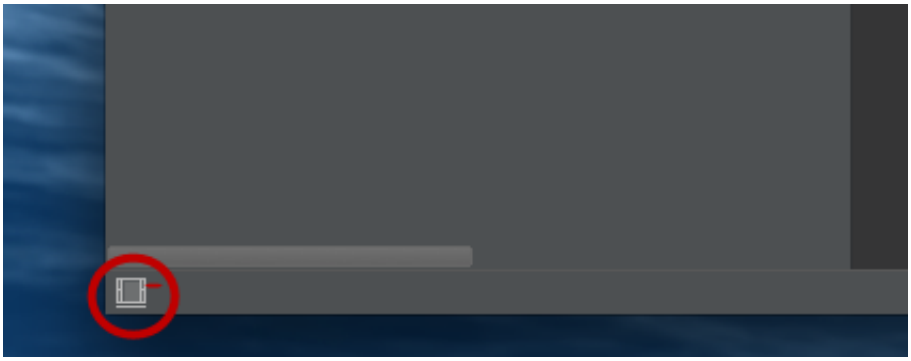
Again, output should look something like:

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

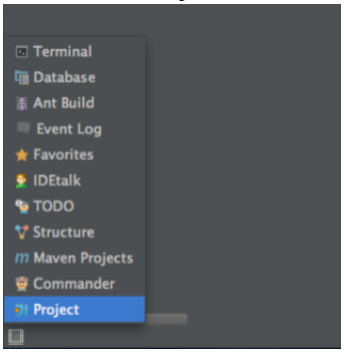
The `gen-idea` command creates two directories, named `.idea` and `.idea_modules`. Both of these directories should be mentioned in the `.gitignore` file. You can examine the `.gitignore` files provided with the projects for this course.

Working With IDEA

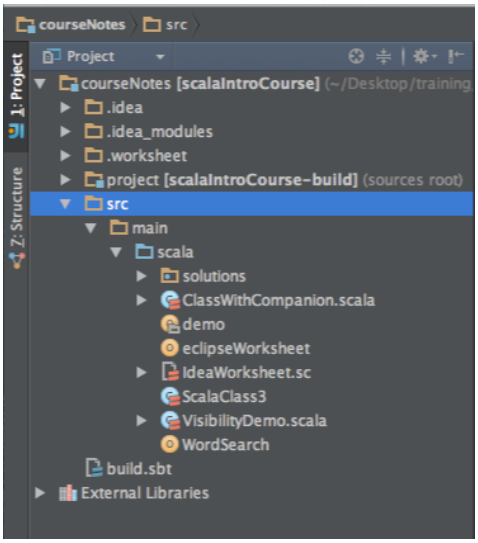
1. Start IDEA and click the **Open Project** menu item.
2. Select the directory called `courseNotes` and click `Choose`.
3. The Scala project should load into IDEA. We can now finish setting up IDEA. You will notice a small rotating icon at the bottom of the IDEA window, and the word **Indexing...** 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.
4. Click the tiny rectangle at the lower left of the IDEA screen to open the Project panel



5. Click on **Project**



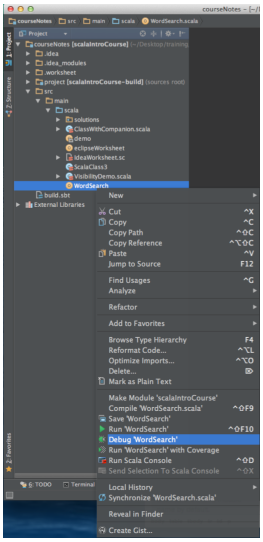
6. Double-click on the project name at the top of the **Project** pane to expand it. Also expand the `src` directory contents.



7. 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, one for the SBT build system and one to compile your code.

   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 JDK 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**.

8. Select a runnable application, such as `WordSearch`, and right-click.

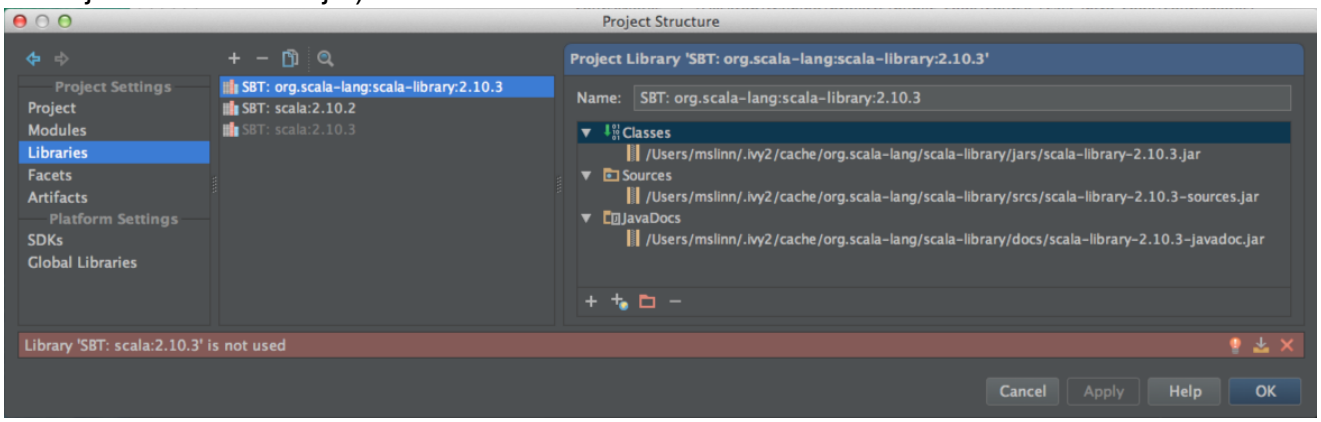


9. Select **Debug**. If all goes well, the project will compile and run. However, there is a really good chance that this will fail. We'll tackle that right now.

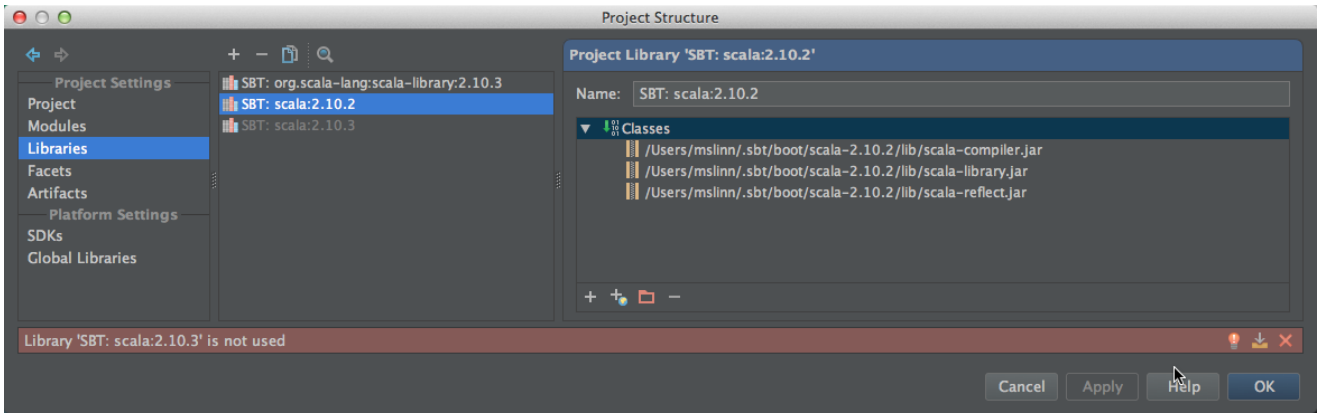
Fixing the No 'scala-library*.jar' in Scala compiler library error

The problem is that the Library definition for the scala compiler is missing some or all of its contents. IDEA 13 is supposed to make this problem go away. Until then, here is what you need to do:

1. Select the **File / Project Structure** menu item.
2. Select the **Libraries** menu on the left. You will see three definitions, and one will be empty. Let's look at each definition in turn:
    1. `scala-library` (this is the Scala run-time library, and it consists of the executable jar, source code jar and Javadoc jar)

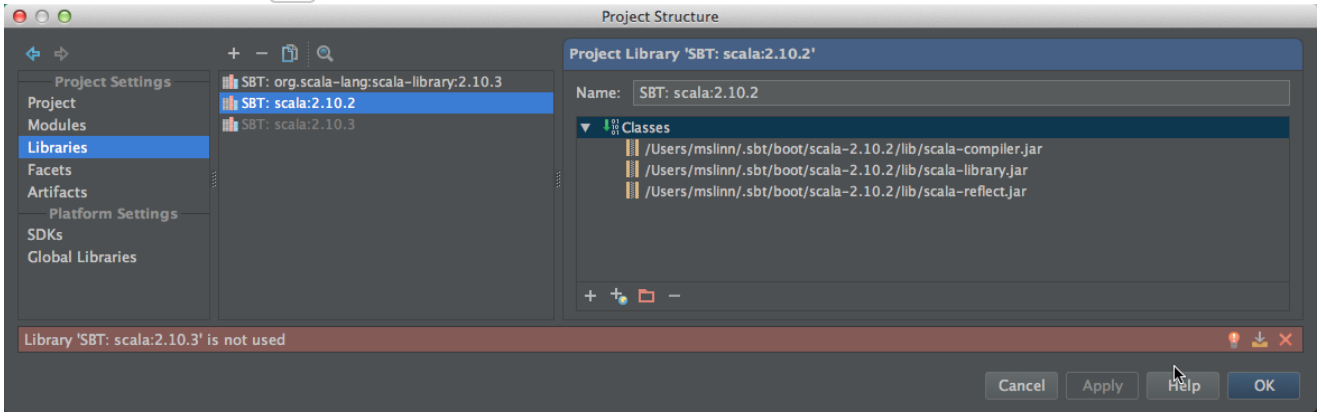


    2. `scala` 2.10.2 (this is the collection of jars for running the Scala compiler)

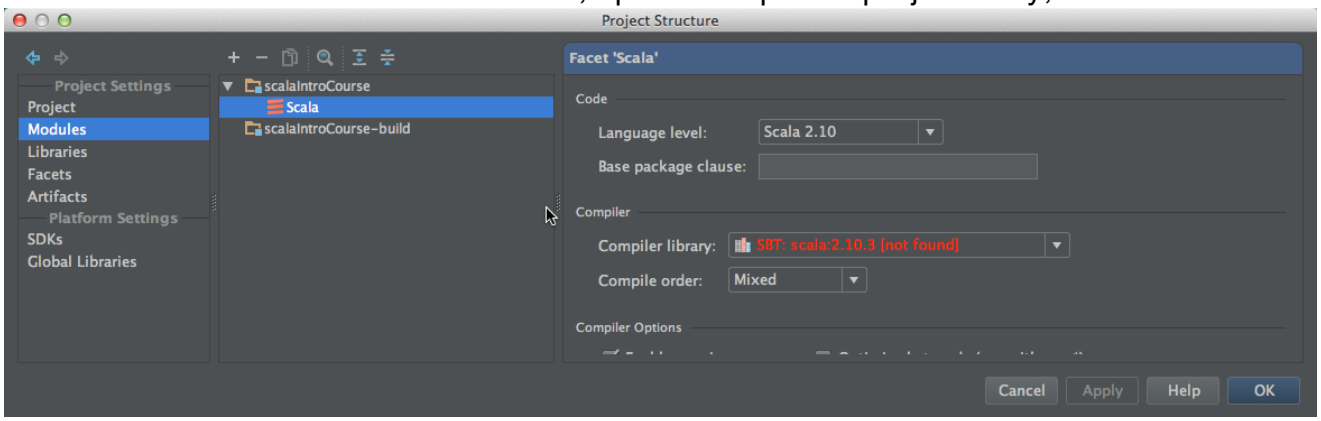


    3. scala 2.10.3 (empty!). This definition is the problem. Might as well delete it - that will make detecting where it is referenced easier. To do that, right-click on the entry and select

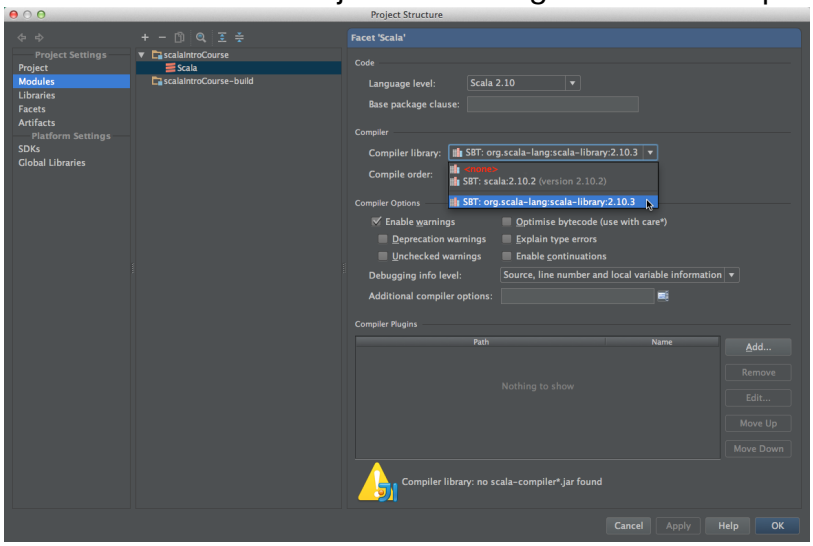
Delete, then click `OK`.



4. Now select the **Modules** item on the left, open the topmost project entry, and select **Scala**.



5. Select the collection of jars for running the Scala compiler.



6. Try debugging the project again. It should work this time!

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 `[_]`.

Worksheets

Scala worksheets are combination of a REPL with and IDE. They are very helpful since your project's classpath is provided to the worksheet. Check it out with **File / New / Scala Worksheet**.

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.

`Shift` `Shift` (twice in a row) - search everywhere, also shows recently edited files

`Alt` + `1` Toggle project view

`Ctrl` + `E` Open recently edited files

`Ctrl` + `Shift` + `E` Open recently changed files

`Alt` + `Home` Open navigation bar

`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` + `Shift` + `Enter` Correct syntax of code

`Ctrl` + `B` Rebuild project.

`Ctrl` + `Shift` + `A` Find Action (learn key bindings, or perform unbound actions)

`Ctrl` + `Alt` + `V` Create a variable from an expression.

`Ctrl` + `Shift` + `B` Toggle breakpoint on current line.

`Ctrl` + `Shift` + `J` Join lines.

`Ctrl` + `Shift` + `V` Paste 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 and optionally open one. Can also specify a filter.

1-10 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. Worksheets created by both IDEs have a `.sc` file type, and worksheets consist of the Scala code you entered. 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.

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`.

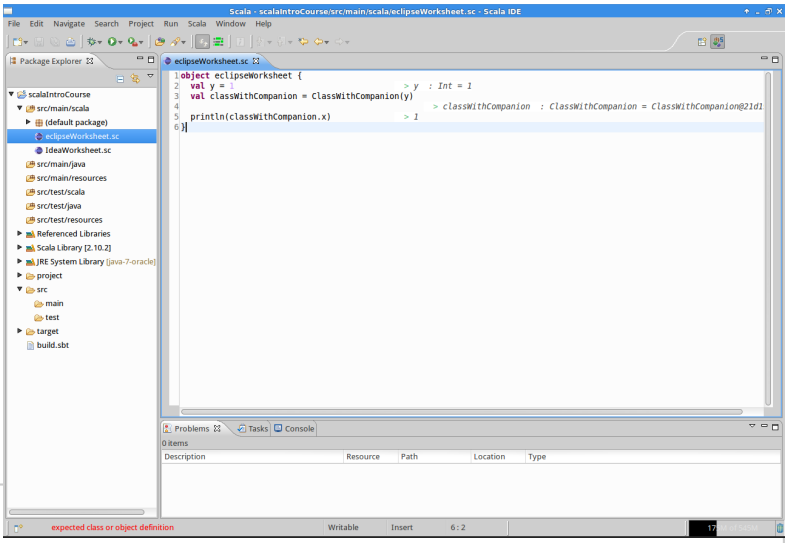
Although worksheets created by IDEA are not interoperable with Scala-IDE by default, it is simple to make them compatible, as we will see at the end of this lecture.

The `courseNotes` project contains a file called `ClassWithCompanion.scala` that defines a class called `ClassWithCompanion`, and a companion class. Companion objects and companion classes will be discussed in a later lecture in this course. 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.



Let's open the `courseNotes` project in Scala-IDE / Eclipse. As a reminder, the steps to do this are:

1. Make the SBT project Eclipse compatible by issuing the following command from the `courseNotes` directory:

```
$ sbt eclipse
Loading /usr/local/bin/sbt-launch-lib.bash
[info] Loading global plugins from /Users/mslinn/.sbt/0.13/plugins
[info] Updating {file:/Users/mslinn/.sbt/0.13/plugins/}global-plugins...
[info] Resolving org.fusesource.jansi#jansi;1.4 ...
[info] Done updating.
[info] Loading project definition from /Users/mslinn/work/training/projects/public_code/course_scala_intro_code/course
Notes/project
[info] Set current project to scalaIntroCourse (in build file:/Users/mslinn/work/training/projects/public_code/course_
scala_intro_code/courseNotes/)
[info] About to create Eclipse project files for your project(s).
[info] Successfully created Eclipse project files for project(s):
[info] scalaIntroCourse
```

2. Start Scala-IDE / Eclipse.
3. **File** / **Import** / **General** / **Existing Projects into Workspace** / `Next >`.
4. Click `Browse` and select the `courseNotes` directory.
5. Click `Open`.

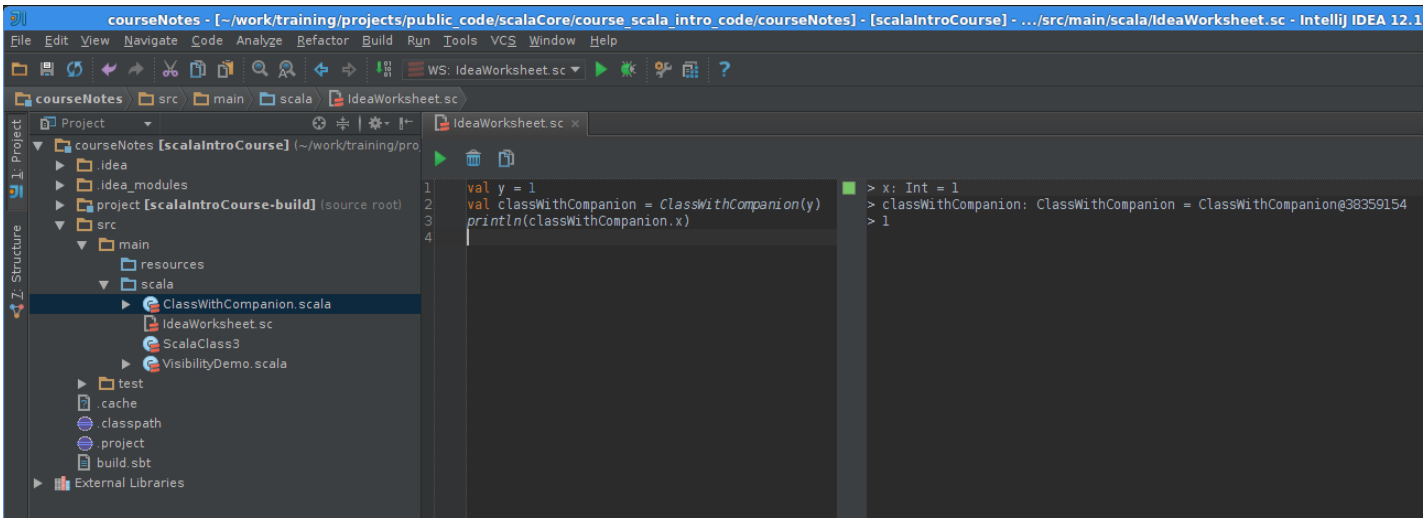
Now we can open an existing worksheet, stored as `src/main/scala/eclipseWorksheet.sc`.

1. Find the file in Package Explorer and double-click on it. The image above shows what you should see.
2. Try changing `y` to another value and then save with `Ctrl` / `Cmd` + `s`. Notice that the worksheet updates, rather like a spreadsheet.
3. Create a new line within the `eclipseWorksheet` scope, and enter an expression like `21 * 2`, then save. The result is shown to the right of the expression as `res0`, rather like how the REPL works.

The name of the enclosing object does not matter. You can rename it to almost any unique name. You will have to do this if you create more than one worksheet in the same package.

Scala-IDE compiles worksheet `.sc` files into Scala code prior to running. The intermediate files are stored in the `.worksheet` directory. You do not need to check in this directory into your source code repository, so I have added `.worksheet/` to `.gitignore`.

IDEA Worksheets



IDEA worksheets do not contain the last evaluation values as comments, and evaluation is relatively slow, compared with Scala-IDE / Eclipse worksheet evaluation. Let's play with the IDEA worksheet in the courseNotes directory.

1. Click on **Open Project**.
2. Navigate to the `courseNotes` directory and double-click on `build.sbt`.

At the time this lecture was written there was an IntelliJ IDEA bug that prevented this from working. Instead, we can open the project as follows:

1. From a bash prompt, navigate to the `courseNotes` directory.
2. Type `sbt gen-idea`
3. From IntelliJ IDEA, click on **Open Project**.
    1. Navigate to the `courseNotes` directory and select the directory (do not click on `build.sbt`)

4. Click on `Choose`.

To open and run the IDEA worksheet that I have prepared:

1. Reveal the **Project** panel tab if necessary by clicking on the small unmarked square at the bottom left of the IDEA window and selecting **Project**.
2. Open the **Project** panel by clicking on the **Project** tab.
3. Open the `courseNotes` project by double-clicking on it in the **Project** panel.
4. Open the file by double-clicking on `src` / `main` / `scala` / `IdeaWorksheet.sc`.
5. Run the worksheet by clicking on the green play button at the top of the worksheet. The worksheet pane will divide in half, with the Scala code on the left and the evaluation results on the right. Unfortunately, IDEA worksheets are not evaluated upon save. The key binding for evaluation is `Ctrl` / `Option` / `W` for all OSes.
6. Create a new line, and enter an expression like `21 * 2`, then run the worksheet. The result is shown to the right of the expression. Unlike the REPL and unlike Scala-IDE, no name for the value is shown.

IDEA worksheets do not need an enclosing object, although you can provide one for interoperability with Eclipse.

Worksheet Interoperability

Here is the worksheet we used with Scala-IDEA / Eclipse. Notice that three statements are wrapped inside an object called `eclipseWorksheet`. The name of the object is not important; give it any name you like.

```
object eclipseWorksheet {
    val y = 1
    val classWithCompanion = ClassWithCompanion(y)
                                                  //> classWithCompanion  : ClassWithCompanion = ClassWi
thCompanion@21d195ca
    println(classWithCompanion.x)             //> 1
}
```

IDEA worksheets and Scala-IDE (Eclipse) worksheets can interoperable if you wrap an IDEA worksheet in an `object { }` as shown. It does not matter if the comments are deleted or not. IDEA will not update the Scala-IDE / Eclipse comments, which can be confusing.

1-11 Scala Imports and Packages

ScalaCourses.com / scalaIntro / ScalaCore / scalaRunning / scalaImportsPackages

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 `TemporalSpatial.scala`. Two classes are defined: `com.micronautics.Temporal` and `com.micronautics.alternate.Spatial`. These classes are all provided in the `courseNotes` SBT project.

```
package com.micronautics

class Temporal

package object alternate {
  def doubler(x: Int): Int = x * 2
}

package alternate {
  class Spatial {
    println(s"Spatial's constructor says that doubling 21 gives ${doubler(21)}")
  }
}
```

Just to make a point, the file above is in the `a/b/c` source directory. The point I am making is that the organization on disk is quite unrelated to the `package` statements. You can put all your classes in the same directory, even if they declare different packages, or you can put them in completely random directories. I do not recommend this, but it is possible. The Scala compiler only looks at the `package` statements to determine the package structure of your code.

Package Objects

Scala also has `package objects`, which can contain package-level definitions – usually you just define methods and properties, as well as package-level documentation in `package objects`. Notice that the `doubler` method is defined within a `package object` for the package `com.micronautics.alternate`. You can invoke `doubler` from anywhere in the `com.micronautics.alternate` package without qualification because `package objects` are automatically imported into the package of the same name. This is actually done in the constructor for class Spatial, which evaluates doubler(21) and prints the result.

Here is an entry point for a console application that exercises the classes above, rather like how Java's `static main` method works:

```
object PackageDemo extends App {
  val ceeABC = new com.micronautics.Temporal
  val ceeXYZ = new com.micronautics.alternate.Spatial
}
```

We can run the above from the command line as follows:

```
$ sbt "run-main PackageDemo"
[info] Loading global plugins from /Users/mslinn/.sbt/0.13/plugins
[info] Loading project definition from /Volumes/Samples/work/training/projects/public_code/course_scala_intro_code/courseNotes/project
[info] Updating {file:/Volumes/Samples/work/training/projects/public_code/course_scala_intro_code/courseNotes/project/}coursenotes-build...
[info] Resolving org.fusesource.jansi#jansi;1.4 ...
[info] Done updating.
[info] Set current project to scalaIntroCourse (in build file:/Volumes/Samples/work/training/projects/public_code/course_scala_intro_code/courseNotes/)
 Spatial's constructor says that doubling 21 gives 42
[success] Total time: 7 s, completed Dec 17, 2013 10:17:19 PM
```

Import Statements

Import statements can reference specific classes:

```
import scala.collection.immutable.List
```

... or several classes:

```
import scala.collection.immutable.{List, Seq, Map}
```

.. or all classes in a package (the underscore is often used as a wildcard in Scala):

```
import scala.collection.immutable._
```

The `scala` package name can be omitted if there is no ambiguity with other packages of the same name:

```
import collection.immutable.List
```

Classes and packages can be renamed. Here we rename the Java `List` to `JList`, so the Scala `List` can also be imported without a naming conflict:

```
import java.util.{List=>JList}
import collection.{immutable=>cim}
import cim.List
```

You can hide some class names. As Yoda said, there is no `Try`, there is only `Do`, or `DoNot`:

```
import scala.util.{Try => _, Success => Do, Failure => DoNot}
```

Locally Scoped Imports

Unlike Java, where `import` statements are all placed at the top of source files, `import` statements in Scala should be located in the most local scope practical. This file is provided as `courseNotes/src/main/scala/TimedTask.scala`. The method `TimedTask.apply` accepts two parameters: the number of seconds between invocations of a closure (`intervalSeconds`), and a closure (`op`) to run periodically. This is an example of how Scala can be used to extend Java's capabilities. I am not going to explain the rest of this program right now, I just want you to see how the `import` statements have been scoped within `objects`.

```
object TimedTask {
  import java.util.{Timer, TimerTask}

  def apply(intervalSeconds: Int=1)(op: => Unit) {
    val task = new TimerTask {
      def run = op
    }
    val timer = new Timer
    timer.schedule(task, 0L, intervalSeconds*1000L)
  }
}

object TimedDemo extends App {
  import java.util.Date

  TimedTask(1)(println(s"Hello, world! ${new Date}"))
}
```

We can run the `TimedDemo` entry point at a console prompt with:

```
$ sbt ~"runMain TimedDemo"
Loading /usr/share/sbt/bin/sbt-launch-lib.bash
[info] Loading global plugins from /home/mslinn/.sbt/0.13/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/)
Hello, world! Sun Dec 29 10:30:51 PST 2013
Hello, world! Sun Dec 29 10:30:52 PST 2013
Hello, world! Sun Dec 29 10:30:53 PST 2013
Hello, world! Sun Dec 29 10:30:54 PST 2013
Hello, world! Sun Dec 29 10:30:55 PST 2013
Hello, world! Sun Dec 29 10:30:56 PST 2013
^C
```

Imports Can Enable Advanced Scala Language Features

The `scala.language` object controls the language features available to the programmer. Each of these features has to be explicitly imported into the current scope to become available. In this way an organization can control the nature of the Scala code that is written, by mandating or disallowing these imports.

```
import scala.language.postfixOps
List(1, 2, 3) reverse
```

To enable all language features:

```
import scala.language._
```

The language features that can be enabled are:

- `dynamics` adds developer driven dynamic binding (dynamic dispatch) and gives a subset of the features of dynamic typing to Scala, via the Dynamic trait. This feature is not described in this course.
- `postfixOps` enables postfix operators. This feature enables an alternative writing style, which may be more natural in some circumstances, and is described this course.

- `reflectiveCalls` enables structural types. This feature is described in detail in the Intermediate Scala course.
- `implicitConversions` enables defining implicit methods and members. This feature is described in detail in the Intermediate Scala course.
- `higherKinds` enables writing higher-kinded types.
- `existentials` enables writing existential types.
- `experimental` contains newer features that have not yet been tested in production and may be significantly changed or removed in future versions of Scala. One of the most promising features in the experimental category of Scala version 2.10 is the macro capability. This feature is not described in this course.

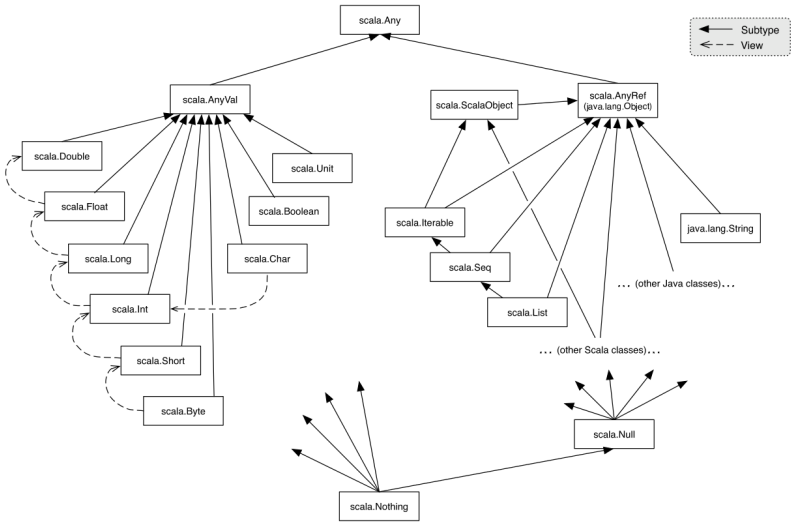
2 Object-Oriented Scala

ScalaCourses.com / scalaIntro / ScalaCore / scalaOO

2-1 Type Hierarchy and Equality

ScalaCourses.com / scalaIntro / ScalaCore / scalaOO / scalaTypes

The Scala inheritance hierarchy is shown in the diagram to the right. Notice that:



- All Scala values are objects, including numbers and strings.
- All objects are subclasses of `Any`.
  - `Any` is the base type of the Scala type system, and defines final methods like `==`, `!=` and `isInstanceOf`.
  - `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.
- There are two main classifications of Scala types:
  - Immutable value types, such as the primitive types `Byte`, `Short`, `Char`, `Int`, `Long`, `Float`, `Double`, `Boolean`, and `Unit`.
    - The base class of all value types is `AnyVal`.
    - `Byte`s can be viewed as / converted to `Short`s, `Short`s and `Char`s can be viewed as / converted to `Int`s, `Int`s can be viewed as / converted to `Long`s, `Long`s can be viewed as / converted to `Float`s, and `Float`s can be viewed as / converted to `Double`s.
    - `Unit` can also be written as `()`.
  - 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`.
    - All non-value objects (aka reference types) are subclasses of `AnyRef`.
    - Reference types are created using the `new` keyword. As we will learn shortly, companion objects often use the default method `apply` to hide the use of the `new` keyword, but it is there nonetheless.
    - All objects that you can define are subclasses of `ScalaObject`, including all types of `Function`s.
    - `Null` is a subtype of all reference types; its only instance is `null`.
- Since `Null` is not a subtype of value types, `null` cannot be assigned to a variable defined as a value type.
- `Nothing` is a subtype of all types. In other words, `Nothing` is a subtype of everything – very Zen-like!

Type Widening

An `if` statement without an `else` clause is subject to *type widening* to type `AnyVal`:

```
scala> def useless(x: Int, y: Int) = if (x>y) x
useless: (x: Int, y: Int)AnyVal

scala> useless(13, 14)
res2: AnyVal = ()

scala> useless(132, 14)
res3: AnyVal = 132
```

The nonexistent clause has value `Unit`, which means 'no value'. `Unit` is often written as `()`. The Scala compiler realizes that type `AnyVal` is the most specific type that can represent all possible returned values from this `if/then` clause. For this reason, you should always include an `else` clause if an if/then clause is intended to return a value from a block of code.

We can also experience type widening when an `if` or `match` statement does not return a consistent type. We will explore `match` statements in more detail later, but for now let's consider it as just a multi-way branch. The following tests the current milliseconds to see if it is an even number, or if today is a Tuesday, and concludes with a catch-all clause. The return values are highlighted in orange, and their types are different for each case: `Boolean`, `String` and `Long`. This file is provided as `courseNotes/src/main/scala/TypeHierarchy.scala`.

```
scala> val rightNow = System.currentTimeMillis
rightNow: Long = 1388348247540

scala> val x = if (rightNow % 2 == 0) true else if (new java.util.Date(rightNow).toString.contains("Tue ")) "Today is a T
uesday" else rightNow
x: Any = true

scala> println(s"x has type ${x.getClass.getName} and value $x")
x has type java.lang.Boolean and value true
```

Running the above in the REPL shows that `x` has type `Any` due to type widening. Of course, the runtime type of `x` might actually be `Boolean`, `Long` or `String`, which are all subclasses of `Any`.

Object Equality

== and != use value equality as defined by `equals`, not reference equality like Java or C#. As is the case with Java, if you define `equals` you must also define `hashCode`.

Let's say we want dogs and hogs with the same name to be equivalent, but disallow the comparison of dogs with hogs. If regular Scala classes are used, you will need to use `isInstanceOf` to validate whether the comparison makes sense or not, like this:

```
object DogHog extends App {
  class Dog(val name: String) {
    override def equals(that: Any): Boolean = canEqual(that) && hashCode==that.hashCode

    override def hashCode = name.hashCode

    def canEqual(that: Any) : Boolean = that. isInstanceOf[Dog]
  }

  class Hog(val name: String) {
    override def equals(that: Any): Boolean = canEqual(that) && hashCode==that.hashCode

    override def hashCode = name.hashCode

    def canEqual(that: Any) : Boolean = that. isInstanceOf[Hog]
  }

  val dog = new Dog("Fido")
  val hog = new Hog("Porky")

  println(s"Should a dog be compared to a hog? ${dog.canEqual(hog)}")
  println(s"Comparing ${dog.name} with ${hog.name} gives: ${dog==hog}")
}
```

Output is:

```
Should a dog be compared to a hog? false
Comparing Fido with Porky gives: false
```

In a later lecture we will explore how Scala's case classes automatically define `canEqual`, which informs us if two objects can logically be compared.

Calling canEqual Prevents Bugs

A source of bugs that I personally encounter more often than I care to admit is when comparing a type with an `Option` of that type. Continuing our example above:

```
scala> val maybeDog: Option[Dog] = Some(dog)
scala> dog==maybeDog // returns false, but the comparison should not be made!
```

Here is one possible way of dealing with the problem. We enhance our `Dog` class with an strict equality operator `===`. More advanced Scala knowledge would allow a more comprehensive and elegant solution such that the comparison is done at compile time.

```
object MaybeDog2 extends App {
  class Dog2(val name: String) {
    override def equals(that: Any): Boolean = canEqual(that) && hashCode==that.hashCode

    override def hashCode = name.hashCode

    def canEqual(that: Any) : Boolean = that.isInstanceOf[Dog2]

    def ===(that: Any): Boolean =
      if (canEqual(that)) this==that else {
        println(s"ERROR: ${getClass.getName} should not be compared to a ${that.getClass.getName}")
        false
      }
  }

  val dog2 = new Dog2("Fido2")
  val maybeDog2 = Some(dog2)
  println(s"Comparing dog2 with maybeDog2: ${dog2===maybeDog2}")
}
```

When we use `===` to compare a `Dog2` with `Option[Dog2]` we get a warning that the comparison is invalid. This is certainly a bug in our code:

```
ERROR: MaybeDog2$Dog2 should not be compared to a scala.Some
Comparing dog2 with maybeDog2: false
```

This solution works at runtime for the class that it is written for. In the next course we introduce a generalized version that works at compile time using implicits and view bounds. It is surprisingly short!

REPL's :type

The Scala REPL has a `:type` command that fully explains the type of any expression. Here is an example of how it can be used:

```
scala> :type 1
Int

scala> :type Some("a")
Some[String]

scala> :type -v Some("a")
// Type signature
Some[String]

// Internal Type structure
TypeRef(
  TypeSymbol(
    final case class Some[+A] extends Option[A] with Product with Serializable

  )
  args = List(
    TypeRef(
      TypeSymbol(
        final class String extends Serializable with Comparable[String] with CharSequence

      )
    )
  )
)

scala> :type Option
Option.type

scala> :type -v Option
// Type signature
Option.type

// Internal Type structure
TypeRef(TypeSymbol(class Option extends Serializable))

scala> :type -v Some
// Type signature
Some.type

// Internal Type structure
TypeRef(TypeSymbol(class Some extends Serializable))
```

2-2 Classes

ScalaCourses.com / scalaIntro / ScalaCore / scalaOO / 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. The sample code for this section can be found in `courseNotes/src/main/scala/Animals.scala`.

```
$ scala
Welcome to Scala version 2.10.3 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_45).
Type in expressions to have them evaluated.
Type :help for more information.
```

Let's start by defining a class called `UselessAnimal` that accepts two parameters to the constructor.

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

defined class UselessAnimal
```

This class is rather odd in that the parameters (`numLegs` and `breathesAir`) are not used, and then they are forgotten. Actually, they are available within the class as private variables, but since no body is defined they serve no purpose.

In Scala, all of the code in a class that is not part of a method definition or an inner class definition is part of the primary constructor. The primary constructor is not enclosed in a single method, as is the case with Java. The following shows a bad practice: spreading primary constructor code throughout a class:

```
abstract class BadAnimal(numLegs: Int, breathesAir: Boolean) {
   val x = 3 // part of primary constructor
   println(breathesAir) // part of primary constructor

  def whatAmI = s"$canSwim;$numLegs legs"

  def anAbstractMethod(param: Int): String

   val y = 3 // part of primary constructor
}
```

We'll talk more about constructors in a later lecture.

Let's define an abstract class called `Animal` that uses the parameters to the constructor to compute an immutable public property called `msg`. By marking the class as abstract, only subclasses can be instantiated.

```
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
```

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 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. 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. 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
```

Named Parameters

Note that when reading the code the mapping between constructor parameters and their values is not obvious. 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. The sample code for this section can be found in `courseNotes/src/main/scala/Animals.scala`.

```
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, breathesAir: 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 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. Notice that the first parameter, `canSwim` is named. Because it is also defined as the first parameter and we have placed it in its declared position, we can name it without requiring all the other parameters to also be named.

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

We can change the value of the mutable property:

```
scala> frog3.numLegs=5
frog6.numLegs: Int = 5
```

... but not the immutable properties:

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

Optional Parameters

In Scala, parameters to methods can be designated as optional by providing default values. Let's create a new class `Frog4` which is like `Frog3`, but has default values for all parameters. The rule is that if a method parameter has a default value, then all following parameters must also have default values.

```
scala> class Frog4(val canSwim: Boolean=true, var numLegs: Int=4, breathesAir: Boolean=true) extends Animal(numLegs, breath
esAir) {
  override def toString() = s"canSwim: $canSwim; $numLegs legs; breathesAir:$breathesAir"
}
defined class Frog4

scala> val frog4a = new Frog4
frog4a: Frog4 = canSwim: true; 4 legs; breathesAir: true

scala> val frog4b = new Frog4(numLegs=2)
frog4b: Frog4 = canSwim: true; 2 legs; breathesAir: true

scala> val frog4c = new Frog4(false)
frog4c: Frog4 = canSwim: false; 4 legs; breathesAir: true

scala> val frog4d = new Frog4(false, breathesAir=false)
frog4d: Frog4 = canSwim: false; 4 legs; breathesAir: false
```

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.

A solution is provided in `src/main/scala/solutions/Bird1.scala`

Anonymous Classes

Classes that do not have a name can be defined, however their primary constructor cannot accept any parameters. Here is an example:

```
val adhoc = new {
  def test(predicate: Boolean): String = if (predicate) "Yes" else "No"
}
```

Let's use this class:

```
scala> adhoc.test(3>2)
res7: String = Yes
```

Anonymous classes can also be created by extending an existing class, trait, or Java interface. We will cover traits in a later lecture. If the definition being extended has any abstract members, they must be implemented in the body of the anonymous class. The syntax for is similar to how this is implemented in Java:

```
scala> import java.util.Date
import java.util.Date

scala> val myDate = new Date { def dayAfter = new Date(getTime + 1000L*60L*60L*24L) }
myDate: java.util.Date{def dayAfter: java.util.Date} = Sun Oct 27 22:13:22 PDT 2013

scala> myDate.dayAfter
res1: java.util.Date = Mon Oct 28 22:13:22 PDT 2013
```

Here is an example of extending an abstract class with an anonymous Scala class.

```
abstract class A {
  val a: Int
  def doubleA: Int
}

val a = new A {
  val a = 3
  def doubleA = a * 2
}
```

Let's use this anonymous class instance:

```
scala> a.doubleA
res5: Int = 6
```

We can define an instance of an anonymous class that extends an existing class, interface or trait using a syntax that is similar to Java's. Here we override the standard method `toString`.

```
scala> val frog3b = new Frog3(canSwim=true, 4, breathesAir=true) { override def toString = s"$canSwim; $numLegs legs; breathesAir=$breathesAir" }
frog3b: Frog3 = Frog5@6f97e3b2

scala> frog3b
res7: Frog3 = true; 4 legs; breathesAir=true
```

We can define a similar class instance as the anonymous class above, and this would be desirable to do if we needed more than one instance:

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

scala> val frog5 = new Frog5(canSwim=true, 4, breathesAir=true)
frog5: Frog5 = canSwim: true; 4 legs; breathesAir: true
```

Operator Overloading

Scala allows you to create multiple methods with the same name that accept different parameters. Let's define a complex number class that defines binary addition and a unary negation operators. Note how the unary minus sign operator is preceded with `unary_` so the compiler knows what we mean.

```
class Complex(val re: Double, val im: Double) {
  def +(another: Complex) = new Complex(re + another.re, im + another.im)

  def unary_- = new Complex(-re, -im)

  override def toString = s"${re} + ${im}i"
}
```

Scala allows methods to be defined called `unary_!`, `unary_~`, `unary_+` and `unary_-`.

Now let's use this definition in some complex arithmetic operations:

```
scala> new Complex(2, 5) + new Complex(1, -2)
res10: Complex = 3.0 + 3.0i

scala> -new Complex(1, -2)
res11: Complex = -1.0 + 2.0i
```

2-3 Setters, Getters and the Uniform Access Principle

ScalaCourses.com / scalaIntro / ScalaCore / scalaOO / scalaUniform

Quick Review

You can 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
         ^
```

You can 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. In the following example, echo returns `Unit`. This is a good practice, because it tells the reader that nothing is expected to be returned, and `println` returns `Unit`.

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

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

Setters and Getters

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, which can be found in `courseNotes/src/main/scala/GetSetDemo.scala`:

```
class GetSetExample1 {
  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 `GetSetDemo1` that can be invoked as a console application.

```
object GetSetDemo1 extends App {
  val x = new GetSetExample1
  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, which is simpler:

```
class GetSetExample2 {
  var name = ""
}

object GetSetDemo2 extends App {
  val x = new GetSetExample2
  x.name_$eq("mary")
  x.name = "lucy"
  println(x.name)
}
```

Let's decorate `Frog2`'s primary constructor arguments with `var` or `val` so they become properties. This sample code can be found in `courseNotes/src/main/scala/Animals.scala`.

```
scala> class Frog2(val canSwim: Boolean, val numLegs: Int, val breathesAir: Boolean) extends Animal(numLegs, breathesAir
) {
  override def toString = s"Frog2 canSwim=$canSwim, numLegs=$numLegs, breathesAir=$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
```

Uniform Access Principle

This code can be found in `courseNotes/src/main/scala/Uniform.scala`.

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` :

```
class Person4 extends AbstractPerson {  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`.

2-4 Deceptively familiar: access modifiers

ScalaCourses.com / scalaIntro / ScalaCore / scalaOO / scalaAccess

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

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.

Warning: the Scala 2.10 and 2.11 implementations of access visibility are buggy. Also, you cannot use the REPL to test access visibility, nor can you trust either IDE's warnings. The only way to know what really works is by compiling a Scala program.

	unadorned	**packageName**	**className**	**this**
protected	protected – When applied to a Scala class, 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.	protected[packageName] – protected and [scope] are additive, so protected[packageName] is accessible in subclasses as well as visible to all other classes in the specified package. When applied to a property or method, visibility is restricted to the defining Scala type and derived types in the specified package.	protected[className] – Spec is blurry but the 2.10 implementation is equivalent to private. Best not to use this.	protected[this] – Spec is blurry, but in the 2.10 implementation the member is visible to methods in the same instance plus the subclass instance.
private	private – Has same meaning in Scala and Java: class definitions and members	private[packageName] – This is known as package-private, and is the default visibility for Java. However, the implementation is	private[className] – "package-private access without inheritance". Spec is blurry. Class or member is visible in the scope of packageName, which must	private[this] – Only visible to methods in the same instance.

	are not visible from other classes.	different between Scala and Java, and they do not interoperate properly. Class or member is visible in the scope of `packageName`, which must enclose the class, either immediately or as a parent package. Subclasses from other packages cannot access subclasses or members whose superclass was marked `private[packageName]`.	enclose the class, either immediately or as a parent package. `private[innerClassName]` locks out the outer class.	

The `courseNotes` project has a demo program that exercises all types of access visibility. 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 many 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 or inner classes. The source code can be found in `courseNotes/src/main/scala/VisbilityDemo.scala`.

Primary constructor visibility

You can declare the default constructor as `private` or `protected` by inserting the appropriate keyword between the class name and the parameter list, like this:

```
class Foo protected (arg1: MyType1, arg2: MyType2) { /* class body here */ }
```

or

```
class Foo private (arg1: MyType1, arg2: MyType2) { /* class body here */ }
```

Syntax for creating inner classes

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
```

2-5 Companion Objects

ScalaCourses.com / scalaIntro / ScalaCore / scalaOO / scalaCompanion

The source code for this lecture is in courseNotes/src/main/scala/ClassWithCompanion.scala.

Companion objects are singletons in which you can define fields, methods, inner objects, inner classes and inner traits. In order for companion objects to have their magic activated, they must be defined in the same file as a class of the same name. The REPL's paste mode was designed to allow companion objects to be defined with companion classes. For example, let's redefine Frog5 and give it a companion object.

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

class Frog6(val canSwim: Boolean, numLegs: Int, breathesAir: Boolean) {
  override def toString = s"canSwim: $canSwim; $numLegs legs; breathesAir: $breathesAir"
}

object Frog6 {
  def apply(canSwim: Boolean=true, numLegs: Int=4, breathesAir: Boolean=true) = new Frog6(canSwim, numLegs, breathesAir)
}
^D
// Exiting paste mode, now interpreting.

defined class Frog6
defined module Frog6
```

If both of these definitions were placed in the same file then Frog6 would be a companion object of the Frog6 class, and the Frog6 class would be a companion class to the Frog6 object. In Scala, methods called apply are default methods, and are often used as factories. Because they are default methods, you don't have to use their name when invoking them. For example, you could create a new Frog6 instance with either of the following statements; both are equivalent:

```
scala> val frog6a = Frog6(canSwim=true)
frog6a: Frog6 =  canSwim: true; 4 legs; breathesAir: true

scala> val frog6b = Frog6.apply(canSwim=true)
frog6b: Frog6 = canSwim: true; 4 legs; breathesAir: true
```

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 (Frog6) refers to the singleton instance.

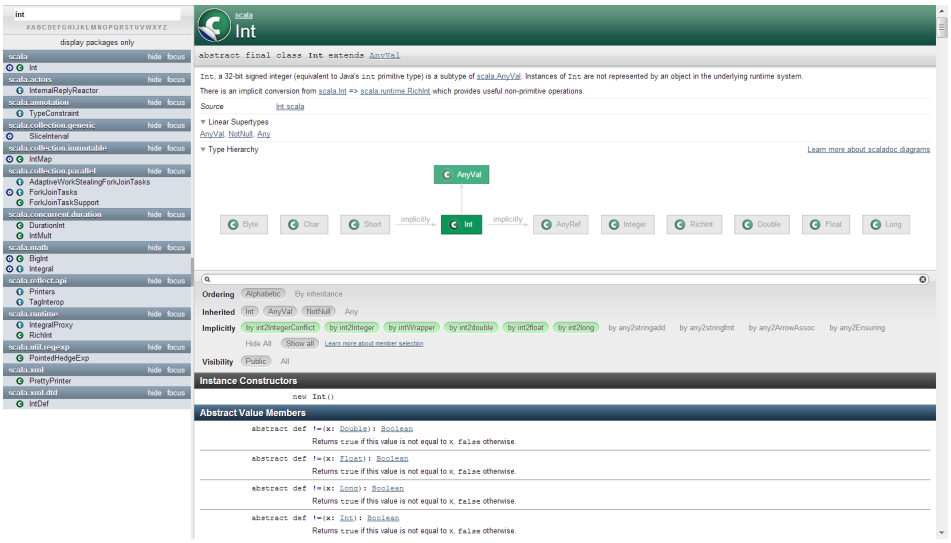
If you want the methods and properties defined in the companion object to be available in the companion class, you must fully qualify them or import them:

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

class Frog7(val canSwim: Boolean, numLegs: Int, breathesAir: Boolean) {
   import Frog7.croak

  def makeNoise = croak(3)

  override def toString = s"canSwim: $canSwim; $numLegs legs; breathesAir: $breathesAir"
}

object Frog7 {
  def apply(canSwim: Boolean=true, numLegs: Int=4, breathesAir: Boolean=true) = new Frog7(canSwim, numLegs, breathesAir)

  def croak(times: Int): String = ("Croak " * times).trim
}
// Exiting paste mode, now interpreting.

defined class Frog7
defined module Frog7

scala> Frog7(canSwim=true, 4, breathesAir=true).makeNoise
res0: String = "Croak Croak Croak"
```

2-6 Reading ScalaDoc

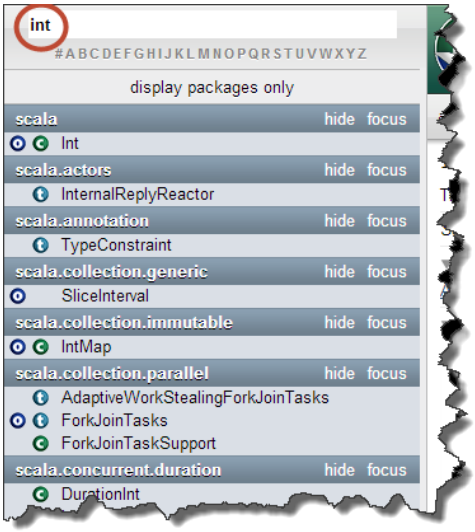
`ScalaCourses.com` / `scalaIntro` / `ScalaCore` / `scalaOO` / `scalaDocReading`

The ScalaDoc for the current version of the Scala language can be found at http://www.scala-lang.org/api/current. Please click on the link and we will explore it together. This is what you should see:

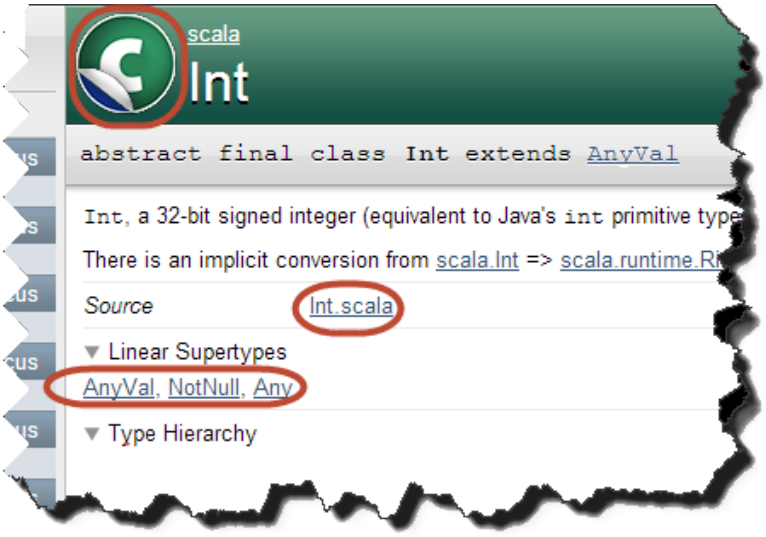


As you know, Scala source files are organized into packages. The front page of ScalaDoc for a project always displays the root package's package-level overview documentation.

1. The top left of the ScalaDoc page features a search field. Please type `int` into that field now.

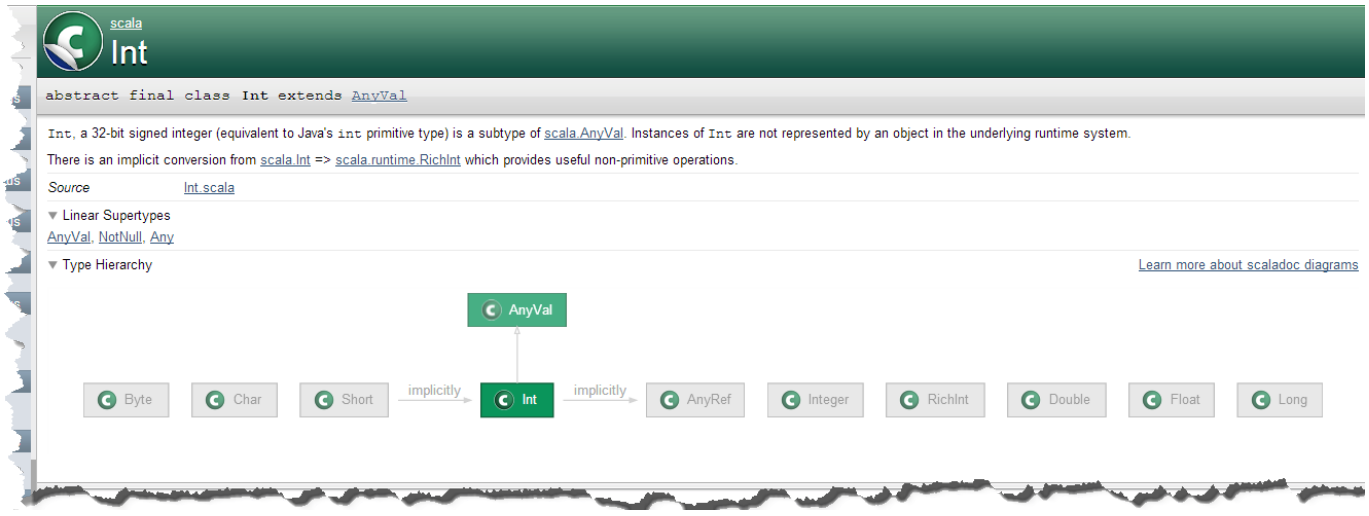


   Notice that the packages and classes shown on the left frame are now filtered such that only those files that contain the search term are now displayed.
2. The first displayed class is `Int`, and there are small **O** and **C** icons to the left of the class name. If you click on them you will be presented with the documentation for the `Int` companion object and the `Int` companion class, respectively.
3. The companion class is displayed in the main frame with dark green heading and highlights, which the companion object is displayed with dark blue heading and highlights.



About 10% of the population is color-blind to some extent, so those people will need to rely on the large **O** or **C** symbols to the left of the class name in the main frame.

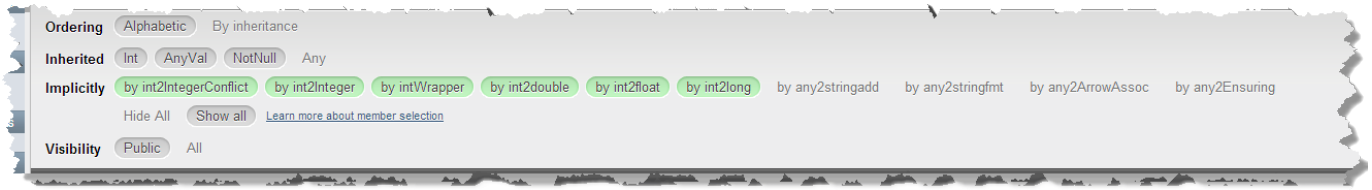
4. If you click on the large **O** and **C** symbols, you will be taken to the dual; in other words, if the companion object is displayed, and hence a large **O** is displayed next to the class name, clicking on the large **O** will cause the companion class to be displayed, and vice-versa.
5. Notice that the URL for the `Int` companion object ends with `#scala.Int$`. while the URL for the `Int` companion class ends with `#scala.Int`. This is consistent with the compiler's naming convention for companion objects and companion classes.
6. If source code is available, it can be viewed for the displayed class, object or trait by clicking on the file name to the right of the word **Source**.
7. The inheritance hierarchy may be viewed by clicking on the **Linear Supertypes** link.
8. The Type hierarchy section appears next when viewing classes. This section can be very informative, so be sure to open it up. Not only can you see the inheritance hierarchy visually, but you can also see implicit classes that reference the class you are viewing.



All of these classes are clickable.

9. The displayed properties and methods can be filtered by typing into the text field above the **Ordering** or **Value Members** heading.

10. By default, only the properties and methods defined in the artifact are displayed. You can instead view the inherited properties and values by clicking on the **By inheritance** ordering link.



You can return to viewing the artifact's properties and methods by clicking on the **Alphabetic** ordering link.

11. When viewing **By inheritance**, you can cause the inherited properties and values to be displayed by toggling the links in the **Inherited** section. You can even toggle the visibility of the displayed artifact's properties and methods here.
12. Two convenience links are provided: **Hide all** and **Show all**. **Hide all** deselects all inherited classes, traits and interfaces. **Show all** shows them all, except `Any` and `AnyVal`; if you want to include their properties and methods you must click on them.
13. **Visibility** controls whether only public methods and properties are displayed, or if all methods and properties are displayed.

2-7 Auxiliary Constructors

ScalaCourses.com / scalaIntro / ScalaCore / scalaOO / scalaAuxCons

Sample code is provided in courseNotes/src/main/scala/Auxiliary.scala.

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 defaults values defined. The primary constructor is also referred to as this(), but with a different method signature.

An auxiliary constructor must call another constructor on its first line.

Let's modify the class Animal from a previous lecture and add an auxiliary constructor . In the following code I defined two factory methods called apply in the companion object – recall that apply is a default method. Once again we use the REPL's paste mode in order to define a companion object interactively:

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

  class Animal3(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(numLegs=2, breathesAir=true)  // specifying return type is not allowed

    override def toString = s"numLegs: $numLegs, breathesAir: $breathesAir, msg: $msg"
  }

  object Animal3 {
     def apply(numLegs: Int, breathesAir: Boolean): Animal3  = new Animal3(numLegs, breathesAir)

     def apply(): Animal3  = new Animal3 // this method definition must have parentheses!
  }
^D
// Exiting paste mode, now interpreting.

defined module Animal3
defined class Animal3
```

Now let's try out the above definitions:

```
scala> val animal3a = new Animal3(4, true)
animal3a: Animal3 = numLegs: 4, breathesAir: true, msg: I have 4 legs and I breathe air

scala> val animal3b = Animal3(4, true)
animal3b: Animal3 = numLegs: 4, breathesAir: true, msg: I have 4 legs and I breathe air

scala> val animal3c = new Animal3
animal3c: Animal3 = numLegs: 2, breathesAir: true, msg: I have 2 legs and I breathe air

scala> val animal3d = Animal3() // without parentheses you get a reference to Animal3.apply(), instead of invoking it
animal3d: Animal3 = numLegs: 2, breathesAir: true, msg: I have 2 legs and I breathe air
```

The rules for when parentheses may or must be applied to method definitions and invocations seem a bit inconsistent. To summarize:

Return type rules

1. Constructors and auxiliary constructors may not define return types; the return type is implicit.
2. All other methods may define return types. It is good practice to do so.

Parentheses rules when defining methods

1. Defining a method with parentheses either suggests that the method has a side effect or it overrides a method that was defined that way.
2. Constructors must always be defined with parentheses.

Parentheses rules when invoking methods

1. Parentheses are always required if parameters are required when writing with the default infix notation.
2. If a constructor does not require parameters, creating an instance of that type with `new` does not need parentheses and they are often not supplied.
3. Parentheses should be supplied when invoking methods that are defined with parentheses, even if the methods do not require parameters.
4. Parentheses must always be supplied when invoking `apply()`.
5. As we will see in the next course, supplying parentheses can be used to force a function reference to be invoked.

Exercise

Add an auxiliary constructor to the `Bird` you defined in the exercise for the lecture on Scala classes (`src/main/scala/solutions/Bird1.scala`). Call this new class `Bird2` so it does not conflict with the previous `Bird` definition. The auxiliary constructor assumes that the bird can fly, so it only accepts the `topSpeed` parameter. If the `Bird2` instance 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)
```

A solution is provided as `src/main/scala/solutions/Bird2.scala`.

2-8 Traits / Mixins

ScalaCourses.com / scalaIntro / ScalaCore / scalaOO / scalaTraits

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

The source code for this lecture is provided in courseNotes/src/main/scala/Traits.scala.

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. We discussed why this is a good practice in the lecture on the Uniform Access Principle.

```
import java.sql.Date

trait Checkable {  def  preFlight: Boolean }

class Course(
  startDate: Date = new Date(System.currentTimeMillis),
   override val  preFlight: Boolean = false // override is unnecessary but legal
)  extends Checkable  {
   override val  toString = s"startDate=$startDate, preFlight=$preFlight"
}

def isReady(checkable:  Checkable ): Boolean = checkable.preFlight
```

It is up to you to decide if you want to decorate val preFlight with override or not when overriding an abstract method or property. I will not do so any further. Now let's create some instances of the class Course which mixes in the pure trait Checkable:

```
scala> val course1 = new Course(Date.valueOf("2014-01-01"), true)
course1: Course = Course(2014-01-01,true)

scala> val course2 = new Course()
course2: Course = Course(2014-01-05,false)
```

You can view an instance of a class as an instance of any of its traits. For example, lets pass a Course instance to isReady, which accepts a Checkable parameter:

```
scala> isReady(course1)
res0: Boolean = true
```

Trait with Implementation

A trait with an implementation is similar to an abstract class, however traits cannot have a constructor that takes arguments. Because HasId is not a pure trait in the following code, classes that mix in this trait must specify override if id is to be mixed in as a property instead of a method.

```
import java.sql.Date

trait Checkable {  def  preFlight: Boolean }

trait HasId {  def id: Long = 0L  }

class Lecture(
   override val  id: Long = 0L,
  startDate: Date = Date.valueOf("2014-01-01"),
   val  preFlight: Boolean = false
) extends Checkable with HasId {
  override val toString = s"id=$id, startDate=$startDate, preFlight=$preFlight"
}
```

Now let's create some instances:

```
scala> val lecture1 = new Lecture(1L, preFlight=true)
lecture1: Lecture = Lecture(1,2014-01-01,true)

scala> val lecture2 = new Lecture()
lecture2: Lecture = Lecture(0,2014-01-05,false)
```

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`.

Some solutions are provided in `courseNotes/src/main/scala/solutions/Emotions.scala`.

Extending Multiple Traits

Objects and classes can extend multiple traits. The first trait or class being extended is mixed in with the keyword `extends`. All other traits are mixed in with the keyword `with`. The order of the traits is 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`. Not only does the order of the mixin determine the type, it also determines the behavior of the resulting class.

There are many traits and a few classes that are combined to make spaceships of various types and capabilities. Many of these traits define a method called `speedUp`. The order of the mixin matters, and the implementation of the `speedUp` method will be provided by the rightmost trait with a concrete implementation of

that method.

Let's walk through the code.

- The enclosing object, `BoldlyGo`, creates an `Explorer`, which we will look at in a moment.
- There is an abstract class called `Spacecraft`, which defines a method called `engage` that accepts no parameters and does not return anything. Clearly `engage` is only useful for side effects, so it is written with empty parentheses, which is hint to the reader that this method is 'side-effecty'.
- The `Bridge` trait is not a pure trait because its `engage` method has an implementation. Its `speedUp` method is undefined, but because it is written with empty parentheses we expect that implementations will have side effects.
- The `Engine` trait also defines a `speedUp` method.
- The `PulseEngine` trait extends the `Engine` trait, and it:
  - Provides an implementation for the `speedUp` method declared in `Engine`, and the implementation's side effect is to set the value of `currentPulse`.
  - Defines a mutable variable of type `Int` called `currentPulse`.
  - Defines an abstract method called `maxPulse` that looks like a getter, because it does not receive any parameters and returns a value, and we don't expect side effects because it was not defined using paretheses.
- The `ControlCabin` trait provides an implementation for `engage` and declares an `increaseSpeed` method.
- The `Shuttle` class extends the `Spacecraft` abstract superclass, mixes in `ControlCabin` and `PulseEngine`, and provides a concrete implementation of `maxPulse` and `increaseSpeed`. Scala only allows one class or abstract class to be extended when defining a new class, and if mixing in one or more traits as well, the superclass must be mentioned first using the `extends` keyword. If a defined class does not inherit from a superclass and only extended traits, then the first trait mixed in would be prefaced with the keyword `extends` and the names of the other traits would follow, separated by the keyword `with`. `increaseSpeed` invokes the `speedUp` method from `PulseEngine`, not the method of the same name from `ControlCabin` because `PulseEngine` was mixed in to the right of `ControlCabin`.
- The `WarpEngine` trait subclasses `Engine`, adding the new abstract method `maxWarp`, which looks like a getter, the mutable `currentWarp` variable and a concrete implementation of the `speedUp` method from `Engine`.
- Class `Explorer` extends the abstract class `Spacecraft`, which must be mentioned before any traits are mixed in. The `Bridge` and `WarpEngine` traits are then mixed in. `Explorer` provides a concrete implementation of `maxWarp` declared by `WarpEngine` and a concrete implementation of `speedUp` declared by `Engine`.
- The singleton object `Defiant` is an instance of `Explorer` that also extends `Spacecraft` but mixes in the `ControlCabin` and `WarpEngine` traits instead of the `Bridge` and `WarpEngine` traits that the `Explorer` class mixes in.

```
object BoldlyGo extends App {
  val explorer = new Explorer
  println(explorer)

  abstract class Spacecraft { def engage(): Unit }

  trait Bridge {
    def speedUp(): Unit
    def engage(): Unit = { speedUp(); speedUp(); 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)
  }
}
```

Traits Should Not Extend Classes

Just because you *can* do something, does not mean that you *should* do it. Extending classes with traits would allow multiple inheritance, which has many problems. Here is one example of an error that results from attempting multiple inheritance.

```
scala> class Animal
defined class Animal

scala> trait Furry extends Animal
defined trait Furry

scala> class Plant
defined class Plant

scala> trait Leafy extends Plant
defined trait Leafy

scala> trait Chimera extends Leafy with Furry
<console>:11: error: illegal inheritance; superclass Plant
 is not a subclass of the superclass Animal
 of the mixin trait Furry
       trait Chimera extends Leafy with Furry
                                        ^
```

2-9 More Traits

ScalaCourses.com / scalaIntro / ScalaCore / scalaOO / scalaTraits2

The source code for this lecture is provided in courseNotes/src/main/scala/Traits2.scala.

Stackable Traits

In Scala, just as in Java, super for subclasses is determined statically. Scala traits are much more flexible, and this lecture will explore two ways to work with Scala traits that are beyond anything that Java can do.

Methods defined in traits can override concrete methods defined in classes or traits that they extend. To do this, the trait's overriding methods must be decorated with abstract override. This is possible because for traits, super is dynamically determined according to the order in which the trait was mixed in. When a method is called on a trait, the trait furthest right in the mix which has that method provides the implementation. If this method calls super then it invokes the method on the trait on its left, and so on. In the following example, the Color trait is not abstract, so the Red, White and Blue traits can superimpose their two abstract override methods: color and paint on concrete implementations. The paint methods are written in such a way that you can see how Scala applies the overrides.

```
object WhichTrait extends App {
  trait Color {
    def color: String = "Unknown"
    def paint(highlight: String): String = s"$color with $highlight highlights"
  }

  trait Red extends Color {
    abstract override val color: String = "red"
    abstract override def paint(highlight: String): String = super.paint(s"pink and $highlight")
  }

  trait White extends Color {
    abstract override val color: String = "white"
    abstract override def paint(highlight: String): String = super.paint(s"light yellow and $highlight")
  }

  trait Blue extends Color {
    abstract override val color: String = "blue"
    abstract override def paint(highlight: String): String = super.paint(s"bluegreen and $highlight")
  }

  val red =  new Red{}
  println(s"""red.paint=${red.paint("polkadot")}""")

  val redWhite =  new Red with White
  println(s"""redWhite.paint=${redWhite.paint("polkadot")}""")

  val redWhiteBlue =  new Red with White with Blue
  println(s"""redWhiteBlue.paint=${redWhiteBlue.paint("polkadot")}""")

   class Concrete extends Red with White with Blue
  val concrete = new Concrete
  println(s"""concrete.paint=${concrete.paint("polkadot")}""")
}
```

Here is the output from running the above:

```
red.paint=red with pink and polkadot highlights
redWhite.paint=white with pink and light yellow and polkadot highlights
redWhiteBlue.paint=blue with pink and light yellow and bluegreen and polkadot highlights
concrete.paint=blue with pink and light yellow and bluegreen and polkadot highlights
```

The `red` variable contains an instance of an anonymous subclass of the `Red` trait. Similarly, the `redWhite` variable contains an instance of an anonymous subclass of the `Red` and `White` traits mixed together. The `redWhiteBlue` variables contains an instance of an anonymous subclass of the `Red`, `White` and `Blue` traits. The `concrete` variable contains an instance of the `Concrete` class, which is also a mix of the `Red`, `White` and `Blue` traits. Although the behavior of `redWhiteBlue` is identical to that of `concrete`, there is only one class definition for all `Concrete` instances. In contrast, each time you write `Red with White with Blue` a new abstract class definition is created.

Notice the order of the colors that are printed out by `redWhiteBlue.paint` and `concrete.paint`.

1. The invocation passes `polkadot` to `concrete.paint`.
2. The rightmost trait, `Blue`, prepends `bluegreen` and invokes `super.paint`.
3. The next rightmost trait, `White`, prepends `light yellow` and invokes `super.paint`.
4. The next rightmost trait, `Red`, prepends `pink` and invokes `super.paint`.
5. The `Color` trait, which is the superclass of the `Red` trait, prepends the value of `color`. Unlike the `paint` methods, the `color` methods in each trait do not call `super`, so that property is consistent for all traits, and it set by the right-most trait that was mixed in.

Another Example

The following example defines a trait that extends the mutable `java.util.Set[String]`, which is a Java interface that uses generics. Scala also supports generics, and we will cover that topic in the next course. This code is somewhat ugly because Scala generics are not used.

The `IgnoredCaseSet` trait's overriding `add` method converts any incoming `Strings` to lower case before adding it to the set. Other types of objects are merely added to the set without modification.

This code uses `isInstanceOf` to test if the runtime type of `obj` is a `String`, and `asInstanceOf` to typecast `obj` to `String` if that is its runtime type. Later in this course we will learn how to use the `match` keyword and then we won't need to use `isInstanceOf` any more.

```
trait IgnoredCaseSet extends java.util.Set[Object] {
  abstract override def add(obj: Object): Boolean =
    if (obj.isInstanceOf[String]) super.add(obj.asInstanceOf[String].toLowerCase) else super.add(obj)

  abstract override def contains(obj: Object): Boolean =
    if (obj.isInstanceOf[String]) super.contains(obj.asInstanceOf[String].toLowerCase) else super.contains(obj)
}

class MySet extends java.util.HashSet[Object] with IgnoredCaseSet
```

Java `HashSet` implements `Set`, which is why we could subclass `HashSet` by mixing in `IgnoredCaseSet` above. Let's create a `MySet` instance that contains `Strings`, and add mixed case `Strings` to it. We will then use the trait's overriding `contains` method to determine if the set contains the equivalent lower case `String`:

```
scala> val set = new MySet()
set: MySet = []

scala> set.add("One")
res3: Boolean = true

scala> set.add("Two")
res4: Boolean = true

scala> set.add("Three")
res5: Boolean = true

scala> set.contains("hi there")
res6: Boolean = false

scala> mySet.contains("two")
res7: Boolean = true
```

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 subclassing 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`. To see the difference between a self type and subclassing a trait, let's define a `User` trait, then use twice, once with a `Tweeter` trait that subclasses `User` and once with a `Tweeter2` trait that uses self typing to merely require `User`:

```
trait User { def name: String }

trait Tweeter extends User {
  def tweet(msg: String) = println(s"$name: $msg")
}

trait Tweeter2 { self: User =>
  def tweet(msg: String) = println(s"${self.name}: $msg")
  def tweet2(msg: String) = println(s"$name:$msg")
}

class Blabber(val name: String) extends Tweeter
class Blabber2( override  val name: String) extends Tweeter2 with User
```

Some observations about this code:

- The `User` trait defined the `name` property as a `def`, in accordance with the uniform access principle.
- The `Tweeter` trait extends `User`, so a `Tweeter` *is a* `User` because a `Tweeter` is a `User` subclass. This means that the `User` implementation is completely visible within `Tweeter`.
- `Tweeter` does not define `User.name`, so any concrete subclass of the `Tweeter` trait must fulfill the `User` contract by defining `name`.
- The `Tweeter2` trait does not extend `User`, so the `Tweeter2` trait does not fulfill the `User` contract. In other words, a `Tweeter2` *is not a* `User`, however it requires that the object which provides the concrete implementation of `Tweeter2` must be a `User`. That means that the implementation details of `User` are not exposed in the `Tweeter2` trait.
- The the self type instance's name is `self`, which is a common convention. Feel free to use any

arbitrary name in your code if you prefer.

- `Blabber`'s `name` parameter to the class constructor is also an immutable public property, so it fulfills the contract of the `User` trait, in that a `name` property must be defined.
- Both `Blabber.name` and `Blabber2.name` parameter implement `User.name`, and they do not need to be marked `override`.

Now let's try it out:

```
scala> val blabber = new Blabber("Mr. Itoktumuch")
blabber: Blabber = Blabber@57cd2786

scala> blabber.tweet("tweet tweet tweet")
Mr. Itoktumuch: tweet tweet tweet

scala> class Blabber2(override val name: String) extends Tweeter2 with User
defined class Blabber2

scala> val blabber2 = new Blabber2("Ms. Nufsaid")
blabber2: Blabber2 = Blabber2@4da5181a

scala> blabber2.tweet2("Le tweet")
Ms. Nufsaid:Le tweet

scala> blabber2.tweet("Une autre tweet")
Ms. Nufsaid: Une autre tweet
```

2-10 Case Classes

ScalaCourses.com / scalaIntro / ScalaCore / scalaOO / scalaCaseClasses

The code examples for this lecture are provided in courseNotes/src/main/scala/CaseClasses.scala.

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 Frog9a(canSwim: Boolean, numLegs: Int, breathesAir: Boolean)
defined class Frog9a
```

This is equivalent to:

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

As with regular classes, if you want a property to be mutable you must preface it with the `var` keyword:

```
case class Frog9c(canSwim: Boolean,  var  numLegs: Int, breathesAir: Boolean)
```

The case class's automatically generated companion object defines the default method called `apply` so you don't have to use the `new` keyword in order to create an instance. Notice that case classes also define a `toString` method which prints out the values of all the properties passed to the default constructor.

```
scala> val frog9a = Frog9a(canSwim=true, 4, breathesAir=true)
frog9a: Frog9a = Frog9a(true,4,true)

scala> val frog9b = Frog9b(canSwim=true, 4, breathesAir=true)
frog9b: Frog9bb = Frog9b(true,4,true)

scala> val frog9c = Frog9c(canSwim=true, 4, breathesAir=true)
frog10: Frog9c = Frog9c(true,4,true)
```

Object Equality

Let's revisit the `Dog` and `Hog` comparison we saw a few lectures ago, using case classes. Case classes automatically define a method called `canEqual`, which informs us if two objects can be compared. You must pass an instance of the class to be compared into `canEqual`, not a class name.

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

scala> case class Hog(name: String)
defined class Hog

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

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

scala> dog1.canEqual(dog2)  // provide an instance to compare, not a class: dog1.canEqual(Dog) is wrong
res4: Boolean = true

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

Let's say we want dogs with the same value of `name` to be equivalent, using a case-insensitive match. 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.toLowerCase.hashCode
}
```

Comparisons work as expected:

```
scala> Dog("fido")==Dog("Fido")
res7: Boolean = true
```

Now let's define `Hog`:

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

Now let's try to compare dogs with hogs.

```
scala> val hog1 = Hog("Porky")
hog1: Hog = Hog(Porky)

scala> dog1.canEqual(hog1) // provide an instance to compare, not a class: dog1.canEqual(Hog) is wrong
res7: Boolean = false

scala> dog1==hog1
res9: Boolean = false
```

Arity 22 Limitation

The primary constructors for case classes can only accept a maximum of 22 properties. The implications of this are that database records with 23 or more fields cannot be persisted to a database, and that Play Framework forms are restricted to 18 fields because Play uses 4 properties for bookkeeping. The arity 22 limitation of case

classes is expected to be removed in Scala 2.11, which is due in the spring of 2014. I expect new versions of Play to follow shortly thereafter.

Subclassing Traits and Classes into Case Classes

This is straightforward. Case classes can extend regular classes, abstract classes and traits, in exactly the same way that normal classes can extend other classes and traits. Here is an example of how to subclass a regular class as a case class:

```
abstract class AbstractFrog(canSwim: Boolean, numLegs: Int, breathesAir: Boolean) {
  override def toString = s"canSwim: $canSwim, numLegs=$numLegs, breathesAir=$breathesAir"
}

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

Now lets create an instance:

```
scala> val frog10 = Frog10(canSwim=true, 4, breathesAir=true)
frog10: Frog10 = canSwim: true, numLegs=4, breathesAir=true
```

Do Not Subclass Case Classes

Case classes must not be subclassed. The compiler will prevent this to some extent, and you should not do so even when it lets you do so. Instead, express your type hierarchy using traits and normal classes, and consider case classes as the equivalent of Java's `final`. Here is the error you get if you attempt to subclass a case class:

```
scala> case class nope(override val canSwim: Boolean, override val numLegs: Int, override val breathesAir: Boolean) extends
 Frog10(canSwim, numLegs, breathesAir)
<console>:10: error: case class nope has case ancestor Frog10, but case-to-case inheritance is prohibited. To overcome this
 limitation, use extractors to pattern match on non-leaf nodes.
```

Operator Overloading Revisited

Lets rewrite the complex arithmetic example we saw in the lecture about Scala classes, and use case classes instead. Notice how the usage of the case class reads better because the `new` keyword is not required.

```
case class Complex(re: Double, im: Double) {
  def + (another : Complex) = Complex(re + another.re, im + another.im)

  def unary_- = Complex(-re, -im)

  override def toString = s"${re} + ${im}i"
}
```

Now let's use this definition in some complex arithmetic operations:

```
scala> Complex(2, 5) + Complex(1, -2)
res10: Complex = 3.0 + 3.0i

scala> -Complex(1, -2)
res11: Complex = -1.0 + 2.0i
```

Standard Methods

The example code for each method assumes the following definitions, which were defined earlier in this lecture:

```
case class Frog9a(canSwim: Boolean, numLegs: Int, breathesAir: Boolean)
case class Frog9c(canSwim: Boolean, var numLegs: Int, breathesAir: Boolean)

val frog9a = Frog9a(canSwim=false, numLegs=4, breathesAir=true)
val frog9c = Frog9c(canSwim=true, numLegs=4, breathesAir=true)
```

Companion 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 frog9a2 = frog9a.copy(canSwim=true)
  frog9a2: Frog9a = Frog(true,4,true)

  scala> val frog9a3 = frog9a.copy(canSwim=true, numLegs=2)
  frog9a3: Frog9a = Frog9a(true,2,true)
  ```

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

  ```
  scala> Frog9a(true, 4, true).canEqual(Frog9c(true,4,true))
  res1: Boolean = false

  scala> frog9a2 canEqual frog9a3
  res2: Boolean = true
  ```

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

  ```
  scala> frog9a.equals(frog9c)
  res3: Boolean = false

  scala> frog9a equals frog9c
  res4: Boolean = false

  scala> frog9a == frog9c
  res5: Boolean = false
  ```

- `hashCode` - A digest stores a hash of the data from an instance of the class into a single hash value. Hashcodes are the basis of equality tests - if two objects have the same `hashCode` then they are `equal`.

  ```
  scala> frog9a.hashCode
  res5: Int = 272580090
  ```

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

```
scala> frog9a.productArity
res6: Int = 3
```

- `productElement` - retrieve the untyped value of the n^{th} case class constructor property.

```
scala> frog9a.productElement(0)
res7: Any = true

scala> frog9a.productElement(1)
res8: Any = 4

scala> frog9a.productElement(2)
res9: Any = true
```

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

```
scala> frog9a.productIterator.foreach(println) // prints property values
false
4
true
```

- `productPrefix` - Simply the name of the case class.

```
scala> frog9a.productPrefix
res10: String = Frog9a
```

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

```
scala> frog9a.toString
res11: String = Frog9a(true,4,true)

scala> frog9a
res12: Frog9a = Frog9a(true,4,true)
```

Companion Objects

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. An entire lecture is coming up that is devoted to this topic.

If you define a companion object for a case class, the methods and properties you define in the companion object will augment the default methods and properties in the automatically generated companion object.

2-11 Tuples Part 1

ScalaCourses.com / scalaIntro / ScalaCore / scalaOO / scalaTuples1

Source code for this lecture may be found in src/main/scala/TupleDemo.scala.

Tuples can be created by wrapping a comma-delimited list of objects in parentheses:

```
scala> (1, 2, 3)
res1: (Int, Int, Int) = (1,2,3)
```

Here is an alternative syntax:

```
scala> Tuple3(1, 2, 3)
res2: (Int, Int, Int) = (1,2,3)
```

The type of these tuples is Tuple3[Int, Int, Int].

Tuples do not need to contain homogeneous types:

```
scala> val t3a = (1, 2.0, "abc")
t3a: (Int, Double, String) = (1,2.0,abc)

scala> val t3b = Tuple3(1, 2.0, "abc")
t3b: (Int, Double, String) = (1,2.0,abc)
```

As you can see from the REPL output, the type of t3a and t3b are Tuple3[Int, Double, String].

Tuples with arity 2, or Tuple2 are probably the most commonly used arity of tuple. You can optionally write instances of Tuple2 using a special syntax, suggesting of an associative key/value pair:

```
scala> "key1" -> "value1"
res4: (String, String) = (key1,value1)

scala> ("key2", "value2")
res5: (String, String) = (key2,value2)
```

Tuple2s are used by Scala Maps, as we will see in the next course.

You can reference the n^{th} member of a tuple using a getter method called _n, where n is the one-based index of the element:

```
scala> t3._1
res6: Int = 1

scala> t3._2
res7: Double = 2.0

scala> t3._3
res8: String = abc
```

Because tuples are implemented as case classes, all of the standard case class methods are available. This also means that tuples suffer from the arity 22 limitation.

```
scala> t3.copy(_2=4.2)
res9: (Int, Double, String) = (1,4.2,abc)

scala> t3.copy(_1=99, _3="aardvark")
res10: (Int, Double, String) = (99, 4.2, aardvark)
```

There are 22 ScalaDoc entries for Scala 2.10 tuples, from Tuple1, Tuple2 through Tuple22. While `Tuples` may not ever support higher arity, `HLists` are expected to be used in the near future to overcome Tuple's arity 22 limitation. The ScalaCourses.com course material will be updated when `HLists` are provided in shipping product.

3 A Small Taste of Functional Programming

ScalaCourses.com / scalaIntro / ScalaCore / scalaFunctional

3-1 Option, Some and None

ScalaCourses.com / scalaIntro / ScalaCore / scalaFunctional / scalaOption

The source code for this lecture is in courseNotes/src/main/scala/OptionDemo.scala

Java methods often return null when there is no value to be returned. This is a leading source of runtime errors. Scala offers the Option container, which allows you to handle the two cases: Some value is returned, or the value is None. The Option construct has been cleverly implemented, and it is actually a collection of zero or one items. That means you can iterate over its contents to work with any value that might be within. Because Option is a container, when you declare a variable to be of type Option you must also declare the type that may be contained by the Option. For example, here is how we could declare a variable called maybeAnswer, which might contain an Int. The value stored in the variable is wrapped in a subclass of Option called Some, which contains the value we are interested in.

```
scala> val maybeAnswer: Option[Int] = Some(42)
maybeAnswer: Option[Int] = Some(42)
```

We can obtain the value of an Option that has Some value with get:

```
scala> maybeAnswer.get
res1: Int = 42
```

Similarly, we can declare a variable that might contain a String containing the name of our favorite type of chocolate, or None if we don't like chocolate:

```
scala> val maybeFavorite: Option[String] = None
maybeFavorite: Option[String] = None

scala> maybeFavorite.getOrElse("Bleah!")
res1: String = Bleah!
```

However, if we attempt to obtain the value of our favorite chocolate, we will get an error if we use get. Instead, we use getOrElse, which provides a default value if the Option contains None:

```
scala> maybeFavorite.get
java.util.NoSuchElementException: None.get

scala> maybeFavorite.getOrElse("Nope")
res6: String = Nope
```

We can also perform an action if an Option contains Some value:

```
scala> maybeAnswer.foreach { x => println(3 * x) }   // prints the value of 42 times 3
126

scala> maybeFavorite.foreach { println }             // does not print anything
```

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
```

Wrapping Nulls with Option

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 zero or one items. This means that collection operations work on `Option` instances. This leads to useful Scala idioms. We will discuss that in the next course. It is also helpful to think of `Option` as a container.

Exercise:

Write a Scala console app that checks the value of an environment variable. If it is defined, print out its value, otherwise print out "`undefined`".

Solution

This solution can be found in `courseNotes/src/main/scala/solutions/TupleAnswer.scala`

```
object TupleAnswer extends App {
  def osProp(name: String): String =  Option(System.getProperty(name)).getOrElse("undefined")

  if (args.length>=1)
    println(s"""Value of '${args(0)}' system property is '${osProp(args(0))}'""")
}
```

To run it, type:

```
sbt "runMain solutions.TupleAnswer  os.name "
```

or

```
sbt "runMain solutions.TupleAnswer  user.home "
```

3-2 Pattern Matching

ScalaCourses.com / scalaIntro / ScalaCore / scalaFunctional / scalaMatch

Scala can match on values, types, and a combination of types and values with conditional tests. Source code for this lecture is provided in courseNotes/src/main/scala/PatMatch.scala.

Multiway Branch/Match on Value

Scala has a multi-way branch, just like many other languages. Scala's match statement is similar to a switch statement in other languages. The cases are tested in the order written.

```
scala> def matchOnValue(x: String): Int =
  x match {
    case "a" => 1
    case _   => 0 // default case matches everyt
hing
  }
matchOnValue: (x: String)Int
```

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

Notice that the last match value was a wild card, specified with an underscore , which means that it matches every possible value and discards the matched value. It is a good practice to ensure that every possible match is provided for. Let's try this:

```
scala> matchOnValue("a")
res0: Int = 1

scala> matchOnValue("q")
res2: Int = 0
```

Instead of discarding the unmatched value, you could capture it by specifying a variable name, which I have named y :

```
scala> def matchOnValue2(x: String): Int =
  x match {
    case "a" => 1
    case y   => if (y.isEmpty) 0 else y.charAt(0).toInt // default case matches every possible value
  }
matchOnValue2: (x: String)Int
```

Notice that the default case, which defines y, contains a conditional test based on the value of y. This is not the best writing style, and we will see a better way of writing this using a guard next. Let's run this code:

```
scala> matchOnValue2("x")
res3: Int = 120

scala> matchOnValue2("a")
res4: Int = 1

scala> matchOnValue2("")
res5: Int = 0
```

Providing a guard

Scala can test against a condition with an `if` clause (notice that the test predicate is not enclosed in parentheses). This test predicate is often referred to as a guard. Let's rework `matchOnValue2` so it uses a guard:

```
def matchOnValue3(x: String): Int =
  x match {
    case "a"             => 1
    case y  if y.isEmpty  => 0
    case y               => y.charAt(0).toInt // if the guard predicate fails then this is the catch-all default case
  }
```

The results are the same as before with `matchOnValue2`. I think a writing style that uses guards is easier to read.

Matching On Type

Here is example of matching on type. Lets define a method called `whatever` that can return type `Any`. It is actually written to return an `Int` or a `String`:, but the declared return type `Any` tells the compiler what the range of expected types are for th.

```
scala> def whatever: Any = if (System.currentTimeMillis % 2 == 0) 1 else "blah"
whatever: Any

scala> whatever match {
      case a: Int    => println("no")
      case b: String => println(b)
    }
no
```

The `match` above does not match every possibility. For example, what if whatever returned a different type, such as a `Boolean`? For example, consider the following code, which does not compile:

```
 false  match {
  case a: Int    => println(s"Whatever: Int with value $a")
  case b: String => println(s"Whatever: String with value '$b'")
}
```

The compiler error message for the above is: `scrutinee is incompatible with pattern type`. We can make it compile by widening the type being compared by writing it this way:

```
 (false: Any)  match {
  case a: Int if a<3 => println(s"$a is an integer less than 3")
  case b: Int        => println(s"$b is an integer greater or equal to 3")
}
```

However, when you run the above you will get `scala.MatchError: false (of class java.lang.Boolean)`. Here is how we could provide a default case that handles unexpected parameter types:

```
(false: Any) match {
  case a: Int    => println("no")
  case b: String => println(b)
  case c         => println(s"$c has an unexpected type") // default case matches on all values and all types
}
```

Providing a default case for match is important, unless we know that all possible cases have been provided. We will discuss `sealed` classes and traits in a later lecture, which allows us to ensure that all possible cases have been considered. Let's add a default case to the previous match on `whatever`'s return value.

```
whatever match {
  case a: Int if x<3 => println(s"$b is an integer less than 3")
  case b: Int        => println(s"$a is an integer greater or equal to 3")
  case x             => println(s"$x is not an integer")
}
```

Example: UnWrapping Java nulls Wrapped with Option

We saw in the last lecture that we could use `get`, `getOrElse` and `orElse` to pull out the value that might be contained within an `Option`. You can also use `match` to obtain a value from `Option`.

```
def maybeSystemProperty(name: String): String =
  Option(System.getProperty(name)) match {
    case Some(value) => s"Property '$name' value='$value'" // value is extracted from the Option
    case None        => s"Property '$name' is not defined"
  }
```

If the system property is defined, the value is extracted from the `Some` instance. The next lecture will talk about how this works – hint: it uses the `unapply` method in the companion object for `Some`.

```
scala> maybeSystemProperty("os.name")
res3: String = Property 'os.name' value='Linux'

scala> maybeSystemProperty("a")
res4: String = Property 'a' is not defined
```

Matching On Type With a Guard

Let's modify the above to use a guard in a match on type.

```
whatever match {
  case x: Int  if x<3  => println(s"$x is an integer less than 3")
  case x: Int        => println(s"$x is an integer greater or equal to 3")
  case x             => println(s"$x is not an integer")
}
```

We can package the above into a method called `guardedmatch`. Notice that `guardedMatch` accepts `Any` type:

```
def guardedMatch(value: Any): String = value match {
  case x: Int  if x<3  => s"$x is an integer less than 3"
  case x: Int          => s"$x is an integer greater or equal to 3"
  case _               => "Did not get an integer" // catch-all case
}
```

Let's try it out:

```
scala> guardedMatch(0)
res3: String = 0 is an integer less than 3

scala> guardedMatch(99)
res4: String = 99 is an integer greater or equal to 3

scala> guardedMatch("blah")
res5: String = Did not get an integer
```

Another Example

This example demonstrates guards with a match on type.

```
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 the match on type was successful. Also notice that the last match (`x`) has no type specified – this means it will match against all types. You can think of each `case` as a filter.

```
def classify(animal: Animal): String = animal match {
  case frog: Frog if frog.numLegs>0 =>
    s"Got a Frog with ${frog.numLegs} legs; canSwim=${frog.canSwim} and breathesAir=${frog.breathesAir}"

  case tadpole: Frog =>
    s"Got a tadpole without legs; breathesAir=${tadpole.breathesAir}"

  case dog: Dog if dog.barksTooMuch =>
    s"Got a Dog that barks too much"

  case dog: Dog =>
    s"Got a quiet Dog"

  case x =>
    s"Got an unexpected animal $x"
}
```

The `classify` method simply prints the appropriate message. Let's try it out:

```
scala> val frog1 = Frog(canSwim=true, 4, breathesAir=true)
frog1: Frog = Frog(true,4,true)

scala> classify(frog1)
res1: String = 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)
res2: String = Got a Frog with 0 legs; canSwim=true and breathesAir=false

scala> val bigDog = Dog(barksTooMuch=false)
bigDog: Dog = Dog(false)

scala> classify(bigDog)
res3: String = Got a Dog and barksTooMuch=false

scala> val yapper = Dog(barksTooMuch=true)
yapper: Dog = Dog(true)

scala> classify(yapper)
res4: String = Got a Dog and barksTooMuch=true
```

Exercise

Given an array of various types of values:

1. Match against each type.
2. Each match case should return a string that states the matched value and the matching type
3. Print out the string

Hints:

1. You can declare an `Array` of `Any` like this:

   ```
   Array("blah", 1, 1.0)
   ```

2. You can loop through an array like this:

   ```
   Array("blah", 1, 1.0) foreach { valueToMatch => println(s"valueToMatch=$valueToMatch") }
   ```

3. You can use the Java `getClass.getName` method to obtain the fully qualified name of an object's class.

A solution is provided in `courseNotes/src/main/scala/solutions/PatMatch101a.scala`. You can run it like this:

```
sbt "runMain solutions.PatMatch101a"
```

3-3 Unapply

ScalaCourses.com / scalaIntro / ScalaCore / scalaFunctional / scalaUnapply

The source code for this lecture is in courseNotes/src/main/scala/Extractors.scala.

Unapply is a method that can be defined in a companion object. If it exists it will automatically be called when the Scala compiler needs to extract values from a Scala class or case class. Unapply is therefore useful for pattern matching. The value returned by unapply is often an Option of a tuple.

Let's define an unapply method for Frog6 and save it as Frog8. The unapply method will merely accept an instance of the companion class and return an Option of a tuple containing the primary constructor properties. In order for this to work, all parameters provided to the primary constructor must be public properties. Recall that for regular classes this means that each primary constructor parameter must be prefaced with the val or var keywords.

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

class Frog8( val  canSwim: Boolean,  val  numLegs: Int,  val  breathesAir: Boolean) {
  override def toString() = s"canSwim: $canSwim; $numLegs legs; breathesAir: $breathesAir"
}

object Frog8 {
  def apply(canSwim: Boolean=true, numLegs: Int=4, breathesAir: Boolean=true) = new Frog8(canSwim, numLegs, breathesAir)

  def unapply(frog: Frog8):Option[(Boolean, Int, Boolean)] = Some((frog.canSwim, frog.numLegs, frog.breathesAir))
}
^D
// Exiting paste mode, now interpreting.

defined class Frog8
defined module Frog8
```

In the code above I wrote Some ((frog.canSwim, frog.numLegs, frog.breathesAir)) , which has type Option[Tuple3[Boolean, Int, Boolean]]. You may see this written sometimes as Some (frog.canSwim, frog.numLegs, frog.breathesAir) – note that there is only one set of parentheses, instead of two nested parentheses. This is possible because when the Scala compiler expects to see an Option[TupleN[...]] but instead only finds TupleN[...] it will automatically wrap the TupleN in an Option. If compiler verbosity is set to WARN then a message will be displayed letting you know that your code was modified. I suggest you write two nested parentheses, because that is what you actually mean.

```
scala> val frog8 = Frog8(canSwim=true, 4, breathesAir=false)
frog8: Frog8 = canSwim: true; 4 legs; breathesAir: false

scala>  val Frog8(a1, b1, c1) =  frog8  // implicitly calls unapply
a1: Boolean = true
b1: Int = 4
c1: Boolean = false

scala>  val Frog8(a2, b2, c2) =  Frog8(canSwim=false, 2, breathesAir=false) // implicitly calls unapply
a2: Boolean = false
b2: Int = 2
c2: Boolean = false
```

Unapply Return Types

Define the return type of an `unapply` method as follows:

- ~~If it is just a test, return a `Boolean`.~~ Don't do this, it is of limited use and will go away in Scala 2.11.
- If it returns a single property of type `T`, return an `Option[T]`.
- If you want to return several properties $T_1, \ldots, T_n$, group them in an option of tuple `Option[(`T_1`, ..., `T_n`)]`.

Exercise

The following class definition has an incomplete `unapply` method in the companion object. Can you write a Scala program that parses the `input String` such that an instance of `Some[Tuple2[String, String]]` containing the first and last name is returned in the event of a successful parse, or `None` is returned if the parse fails?

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 an instance of `Some[Tuple2]` this way: `Some(("first", "last"))`. If you try this in the REPL you should see the following:

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

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.

Solutions are provided in `courseNotes/src/main/scala/solutions/Unapply.scala`. Run them like this:

```
sbt ~"runMain solutions.Unapply"
```

... and:

```
sbt ~"runMain solutions.Unapply2"
```

Overloading Unapply

When writing regular classes (not case classes) you should always define an `unapply` method in the companion object that accepts one parameter, and the type of that parameter should be the type of the companion class. Case classes do this for you automatically. You can also overload `unapply` by defining methods called `unapply` which accept parameters of any type that you wish to parse into instances of the companion class. For example, if you wish to create instances of a `Frog8` from information contained in a `String`, the method signature you need to implement is:

```
object Frog8 {
  def unapply(string: String): Option[(Boolean, Int, Boolean)] = ???
}
```

Exercise

Implement the above `unapply` such that it can parse the following `String`: `"true 4 false"`. Don't worry about error handling.

Hints

- A companion object and companion class must be defined in the same scope. Be sure to create a file that contains a copy of the `Frog8` source code.
- You can split a string into an array of space-delimited tokens by calling `"my string".split(" ")`
- You can access the n^{th} item in an array by using parentheses to supply the index, like this: `myArray(2)`
- You can convert an object to its `Boolean` equivalent by calling `any.toBoolean`. For example, `"true".toBoolean`
- Similarly, you can convert an object to its Int equivalent by calling `toInt`. For example, `"123".toInt`

A solution is provided in `courseNotes/src/main/scala/solutions.Unapply.scala`. You can run it like this:

```
sbt "runMain solutions.Unapply3"
```

Partial match

If you only want to retrieve some of the values and not others, use an underscore in place of the properties that you are not interested in extracting:

```
scala> val Frog8(a3, _, c3) = frog8 // implicitly calls unapply
a3: Boolean = true
c3: Boolean = false
```

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.

Case Classes

Case classes automatically define `apply` and `unapply`.

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

scala> val bigDog = Dog("Fido", barksTooMuch=false)
bigDog: Dog = Dog(Fido,false)

scala>  val Dog(x, y) = bigDog  // implicitly calls unapply
x: String = Fido
y: Boolean = false
```

Notice that two values were returned by the computation and stored into `x` and `y`. `Unapply` deconstructed the `Dog` returned by the computation and extracted the two parameters.

Overloading unapply

You can define overloaded versions of `apply` and `unapply` by augmenting the definition of a case class's automatically generated companion object. Lets do this for a case class called `Fraction` which models fractions.

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

case 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 { // this augments the automatically generated companion object instead of replacing it
    def unapply(string: String): Option[(Int, Int)] = {
      val tokens = string.split("/")
      if (tokens.length!=2)
        None
      else
        Some(tokens(0).toInt, tokens(1).toInt)
    }
}
^D

// Exiting paste mode, now interpreting.

defined class Fraction
defined module Fraction

scala> val fraction = Fraction(3,4) * Fraction(2,4)
fraction: String = 6/16

scala>  val Fraction(numer, denom) =  "3/4"  // implicitly calls unapply
numer: Int = 3
denom: Int = 4
```

Again, notice that two values were returned by the computation and stored into `numer` and `denom`. `Unapply` parsed the `String` `"3/4"` and returned an `Option[Tuple2[Int, Int]]` containing the two parsed parameters.

Exercise

You can augment a case class's companion object simply by defining new methods and overloading existing methods. Your task is to write a Scala program that defines 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[(Boolean, Int, Boolean)] = ??? // TODO write me

  def unapply(input: Array[String]): Option[(Boolean, Int, Boolean)] = ??? // TODO write me
}
```

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(swimmer3, legCount3, airBreather3) = Array("true", "4", "true")
val Frog9(swimmer4, legCount4, airBreather4) = Array("true", "0", "false")
```

The program should read the arguments from the command line. If you put your program in `courseNotes/src/main/scala` and call it `Extractor.scala` then you should be able to run it and display the parsed values like this:

```
$ sbt ~"runMain Extractor true 4 true"
swimmer = true
legCount = 4
airBreather = true

$ sbt "runMain Extractor true 0 false"
swimmer = true
legCount = 0
airBreather = false
```

Hints

1. To read arguments from the command line in a console app, reference the `args` variable. You can discover the number of variables by reading `args.size`, and you can obtain the first argument as `args(0)`.
2. Do not worry about error handling.

A solution is provided in `courseNotes/src/main/scala/solutions/Extractor.scala`. You can run it like this:

```
sbt "runMain solutions.Extractor true 4 false"
```

3-4 Sealed Classes and Extractors

ScalaCourses.com / scalaIntro / ScalaCore / scalaFunctional / scalaExtract

Source code for this lecture is provided in courseNotes/src/main/scala/SealedDemo.scala.

Sealed Classes and Traits Provide Exhaustive Matching

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. This can ensure that a match statement is exhaustive. Classes and traits can be decorated with the sealed attribute. Here is an example of a sealed Scala class.

```
object SealedDemo extends App {
  sealed abstract class Color
  class White extends Color
  class Yellow extends Color
  class Blue extends Color
  class Green extends Color
  class Black extends Color

  def colorSwatch(color: Color): Unit = color match {
    case color: White => println("White")
    case color: Yellow => println("Yellow")
    case color: Blue => println("Blue")
    case color: Green => println("Green")
    //case color: Black => println("Black")
  } // compiler will complain that Black was not tested for

  println(colorSwatch(new Blue))
}
```

The compiler's warning looks like this:

```
[warn] /home/mslinn/work/course_scala_intro_code/courseNotes/src/main/scala/SealedDemo.scala:151: match may not be exhausti
ve.
[warn] It would fail on the following input: Black()
[warn]   def colorSwatch(color: Color): Unit = color match {
```

Extracting Values

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.

```
case class Frog11(canSwim: Boolean, numLegs: Int, breathesAir: Boolean)

case class Dog3(name: String, barksTooMuch: Boolean)

case class Horse(name: String)

def extract(animal: Any): String = animal match {
  case Frog11(canSwim, numLegs, breathesAir) if numLegs>0 =>
    s"Got a Frog11 with $numLegs legs; canSwim=$canSwim and breathesAir=$breathesAir"

  case Frog11(canSwim, numLegs, breathesAir) =>
    s"Got a tadpole without legs; breathesAir=$breathesAir"

  case Dog3(name, barksTooMuch) =>
    s"Got a Dog3 called $name and barksTooMuch=$barksTooMuch"

  case x =>
    s"Got an unexpected animal: $x"
}
```

Output is:

```
scala> extract(new Dog3("Fido", barksTooMuch=false))
res0: String = Got a Dog3 called Fido and barksTooMuch=false

scala> extract(new Dog3("Fifi", barksTooMuch = true))
res1: String = Got a Dog3 called Fifi and barksTooMuch=true

scala> extract(new Frog11(canSwim=true, 0, breathesAir=false))
res2: String = Got a tadpole without legs; breathesAir=false

scala> extract(new Frog11(canSwim=true, 4, breathesAir=true))
res3: String = Got a Frog11 with 4 legs; canSwim=true and breathesAir=true

scala> extract(new Horse("Silver"))
res4: String = Got an unexpected animal: Horse(Silver)
```

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

Quiz: Extractors vs. Matching types or values

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?

Solution

Matching a variable against various types does not invoke `unapply`. `Unapply` is useful when you want to extract properties from an incoming object when a match occurs, or to parse an object into another type. The following code example contrasts the two approaches:

```
object PMQuiz extends App {
  case class Frog12(canSwim: Boolean, numLegs: Int, breathesAir: Boolean)

  val frog12 = Frog12(canSwim=true, numLegs=4, breathesAir=true)

  frog12 match { // match by type only, unapply is not invoked
    case kermit: Frog12 => println(s"kermit=$kermit")

    case other => println(other)
  }

  frog12 match { // match by type and invoke unapply implicitly to extract properties as separate variables
    case Frog12(a, b, c) => println(s"Extracted properties are: canSwim=$a, numLegs=$b, breathesAir=$c")

    case other => println(other)
  }
}
```

You can run this example as follows:

```
sbt "runMain PMQuiz"
```

Output is:

```
kermit=Frog12(true,4,true)
Extracted properties are: canSwim=true, numLegs=4, breathesAir=true
```

Aliases

Aliases allow you to match on type, as well as against property values if desired, and to obtain the entire object.

Aliases can be specified on patterns or on parts of a pattern. The alias is put before the pattern, separated by `@`. For the alias `address` below, we define a filter that finds `Addresses` in Paris, France and returns the result as 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 _ =>
  }
}
```

Output is:

```
543 Toulouse
```

3-5 Try and try/catch/finally

ScalaCourses.com / scalaIntro / ScalaCore / scalaFunctional / scalaTry

Source code for this lecture is provided in courseNotes/src/main/scala/TryDemo.scala.

Try is a utility class provided by the Scala runtime library. It either holds the value of a successful computation or it holds an Exception. There are two subclasses: Success and Failure. Here is an example:

```
scala> import scala.util.{Try, Success, Failure}
import scala.util.{Try, Success, Failure}

scala> def divide(dividend: Int, divisor: Int): Try[Int] = {
  Try(dividend/divisor) match {
    case Success(v) =>
      println(s"Result of $dividend/$divisor is: $v")
      Success(v)
    case Failure(e) =>
      println("You must have divided by zero or entered something that's not an Int.")
      println(e.getMessage)
      Failure(e)
  }
}
divide: (dividend: Int, divisor: Int)scala.util.Try[Int]

scala> divide(4, 2)
Result of 4/2 is: 2
res0: scala.util.Try[Int] = Success(2)

scala> divide(2, 0)
You must have divided by zero or entered something that's not an Int.
/ by zero
res1: scala.util.Try[Int] = Failure(java.lang.ArithmeticException: / by zero)
```

Exercise: Yoda He Is

"There is no try. There is only do, or do not."

For a bit of fun, rewrite the above using the following import:

```
import scala.util.{Try => _, Success => Do, Failure => DoNot}
```

This exercise is a review of how import aliases work.

Solution

```
object YodaHeis extends App {
  import scala.util.{Try => _, Success => Do, Failure => DoNot}

  def divide(dividend: Int, divisor: Int):  util.Try [Int] = {
     util.Try (dividend/divisor) match {
      case  Do (v) =>
        println(s"Result of $dividend/$divisor is: $v")
         Do (v)
      case  DoNot (e) =>
        println("You must have divided by zero or entered something that's not an Int.")
        println(s"Error: ${e.getMessage}")
         DoNot (e)
    }
  }

  println(s"divide(4, 2)=${divide(4, 2)}")
  println(s"divide(2, 0)=${divide(2, 0)}")
}
```

NoStackTrace

It is somewhat expensive to fill in a new `Exception`'s stack trace. Scala offers a way to create a new `Exception` that does not contain a stack trace. An `Exception` without a stack trace should not be thrown because it might be difficult to determine the answers to questions like "who created this `Throwable`?" when debugging. However, this lighter-weight `Exception` can be useful to pass around, as we shall see in a moment.

```
scala> import util.control.NoStackTrace
import util.control.NoStackTrace

scala> val e = new Exception("Help!") with NoStackTrace
e: Exception with scala.util.control.NoStackTrace = $anon$1: Help!
```

The above creates an anonymous class, which means an extra class file is created every time the code is executed. These class files will pile up throughout the program's life cycle in the JVM.

NoStackTrace Improved

We could improve the code by defining a class that mixes `NoStackTrace` into `Exception`. We discussed how mixing in traits could change the behavior of a Java or Scala class in a previous lecture.

```
scala> class ExceptTrace(msg: String) extends Exception(msg) with NoStackTrace
defined class ExceptTrace

scala> val e2 = new ExceptTrace("Boom!")
e2: ExceptTrace = ExceptTrace: Boom!
```

The above is efficient. If your exceptions are invariant in all their properties each time, you could define a singleton `object` instead of a `class` instance for even greater efficiency:

```
scala> object TheExceptTrace extends Exception("Boom!") with NoStackTrace
defined module TheExceptTrace
```

try / catch / finally

`try` / `catch` / `finally` is a Scala language control structure. Let's see it in action:

```
def divide2(dividend: Int, divisor: Int): Int = {
   try {
    dividend/divisor
  } catch {
    case e: ArithmeticException =>
      println(e.getMessage)
      0

    case e: Exception =>
      println(s"Did not see this one coming! $e")
      0
  }
}

scala> divide2(4, 2)
res23: Int = 2

scala> divide2(2, 0)
/ by zero
res24: Int = 0
```

As you can see, two problem cases are handled: `ArithmeticException` and all other `Exceptions`. The order that the exceptions are listed is significant. Each exception `case` can return a different value, and can contain nested `try` / `catch` clauses and optionally a `finally` clause. Unlike Java, the `finally` clause does affect the returned value, and is only useful for side effects:

```
def divide3(dividend: Int, divisor: Int): Int = {
   try {
    dividend/divisor
  } catch {
    case e: ArithmeticException =>
      println(e.getMessage)
      0

    case e: Exception =>
      println(s"Did not see this one coming! $e")
      0
  } finally {
      println("The end")
  }
}

scala> divide3(4, 2)
res23: Int = 2

scala> divide3(2, 0)
/ by zero
res24: Int = 0
```

The problem with the code above is that the return value does not indicate if a problem happened. Instead of returning a 0, it would be better to return the exception. Let's create a better program by combining `try` / `catch` / `finally` with `Try`:

```
def divide4(dividend: Int, divisor: Int): Try[Int] = {
  try {
    Success(dividend/divisor)
  } catch {
    case e: Exception =>
      println(e.getMessage)
      Failure(e)
  }
}

scala> divide4(4, 2)
res20: scala.util.Try[Int] = Success(2)

scala> divide4(2, 0)
/ by zero
res21: scala.util.Try[Int] = Failure(java.lang.ArithmeticException: / by zero)
```

We can express this more succinctly if we are willing to forego the `println`s:

```
def divide5(dividend: Int, divisor: Int): Try[Int] = Try(dividend/divisor)

scala> divide5(4, 2)
res20: scala.util.Try[Int] = Success(2)

scala> divide5(2, 0)
/ by zero
res21: scala.util.Try[Int] = Failure(java.lang.ArithmeticException: / by zero)
```

Simple, short, robust code!

3-6 Either, Left and Right

ScalaCourses.com / scalaIntro / ScalaCore / scalaFunctional / scalaEither

The source code for this lecture is in courseNotes/src/main/scala/EitherDemo.scala.

Many methods might return a different type of value when an error occurs. Either wraps both possibilities, and can either return the success value ("on the right") or it can return the failure value ("on the left"). That means Either is parametric in two types, one for Either.left and the other for Either.right. We will discuss parametric types in detail in the Intermediate Scala course. Since Scala 2.10 we have Try, described in the previous lecture, so only older code should have Either[Exception, SuccessType]. If you find yourself needing to return an Either, with an Exception on the left side for the failure case, you should return a scala.util.Try instead.

You can type each line of the following bold code 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[String, String] = Right("yay!")
result1: Either[String,String] = Right(yay!)
```

Storing a value into Either.left is similarly simple:

```
scala> val result2: Either[String, String] = Left("Oops, I did it again!")
result2: Either[String,String] = Left(java.lang.String)
```

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  // this is not the way to write production code!
res9: String = yay!

scala> result2.isRight
res12: Boolean = false

scala> result2.isLeft
res13: Boolean = true

scala> result2.left.get  // this is not the way to write production code!
res14: String = String: Oops, I did it again!
```

Either can only hold one value; either a value on the left or a value on the right. Either is also immutable, so once it has been assigned a value the Either instance cannot be changed.

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.

```
def parse(in: String): Either[String, Int] = try {
    Right(in.toInt)
  } catch {
    case e: Exception =>
      Left(in)
  }

def show(either: Either[String, Int]): 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
```

Type Alias

We can define a type alias using the `type` keyword. It is usually better to define a type alias with a descriptive name than to write out a long type like `Either[String, Int]`.

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

Let's use the type alias to shorten the above code:

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

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

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

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 a 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.

To create a console application, we will clone the `sbtTemplate` project on GitHub that we learned about in the sbt lecture. We will clone into a directory called `eitherExercise`.

```
$ git clone https://github.com/mslinn/sbtTemplate.git eitherExercise
$ cd eitherExercise
$ sbt gen-idea
```

Now lets edit a new file in `eitherExercise` called `src/main/scala/WordSearch.scala` using IntelliJ IDEA or 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 StringOrInt = 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 StringOrInt = Either[String, Int]

  def search(list: List[String], word: String): StringOrInt = ???
}
```

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 StringOrInt = Either[String, Int]

  def search(list: List[String], word: String): StringOrInt = ???

  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

Some solutions are provided in `src/main/scala/solutions/WordSearch.scala`. Examine each one – you will learn something with each solution.

```
package solutions

/** Two solutions are shown. Solution 2 is best. */
object WordSearch extends App {
  val list = List("word", "another", "yet another")

  type StringOrInt = Either[String, Int]

  /** Efficient but clumsy */
  def search(list: List[String], word: String): StringOrInt = {
    val index = list.indexOf(word)
    if (index == -1)
      Left(word)
    else
      Right(index)
  }

  /** Efficient, simple, elegant.
    * Here we see Scala's match keyword used as a multi-way branch.
    * Each match is tried in order until one succeeds.
    * In the second case, notice that the variable index is defined. It will match every value. */
  def search2(list: List[String], word: String): StringOrInt =
    list.indexOf(word) match {
      case -1 => Left(word)
      case index => Right(index)
  }

  List("word", "oink", "blahbla", "another").foreach { w =>
    println(s"Method 1 $w: ${search(list, w)}")
    println(s"Method 2 $w: ${search2(list, w)}")
  }
}
```

You can run them like this:

```
sbt ~"runMain solutions.WordSearch"
```

3-7 Functions are First Class

`ScalaCourses.com` / `scalaIntro` / `ScalaCore` / `scalaFunctional` / `scalaFunctions`

Source code for this lecture is provided in `courseNotes/src/main/scala/Fun.scala`.

Scala is a unique blend of object-oriented programming and functional programming. In Scala, functions are first-class objects. We will make a distinction between methods and functions in a moment.

A function literal is one possible syntax for defining a Scala function, and is an example of *syntactic sugar*. This syntax is useful for passing a function as an argument to a method. Here is an example of a function literal:

```
scala> (a: Int, b: Double) => (a * b).toString
res0: (Int, Double) => String = <function2>

scala> res0(2, 3)
res1: String = 6.0
```

As you can see, function literals are defined by indicating the arguments that they accept, followed by a right arrow, followed by the implementation. The type of the above is `(Int, Int) => Int`. We can define a type alias to assist us when referencing that type:

```
scala> type IntDblToStr = (Int, Double) => String
defined type alias IntDblToStr
```

Binding to a variable

You can bind a function literal to a variable. Now you can reference the function and pass it parameters when you invoke it.

```
scala> val mulStr = (a: Int, b: Double) => (a * b).toString
mulStr: (Int, Double) => String = <function2>

scala> mulStr(1, 2)
res3: String = 2.0
```

We could declare the type of the variable that references the function if we like:

```
scala> val mulStr2:  IntDblToStr  = (a: Int, b: Double) => (a * b).toString
add: (Int, Double) => String = <function2>

scala> mulStr2(1, 2)
res3: String = 2.0
```

Placeholder Syntax

If you write an expression with an underscore in it, Scala will assume that you are defining a function literal and the underscore is a variable whose value will be supplied by a function parameter. You must provide the type of the variable unless it is somehow obvious, as shown:

```
scala> val addFour = (_: Int) + 4
addFour: Int => Int = <function1>

scala> addFour(10)
res18: Int = 14

scala> val multiplyThree = (_: Int) * 3
multiplyThree: Int => Int = <function1>

scala> multiplyThree(20)
res19: Int = 60
```

Desugared Syntax

The Scala compiler implements function literals by creating an instance of an anonymous class. The above is equivalent to, or *desugared as*:

```
scala> val mulStr3: IntDblToStr = new Function2[Int, Double, String] { def apply(a: Int, b: Double): String = (a * b).toString }
mulStr3: (Int, Double) => String = <function2>
```

The desugared function literal type is `Function2[Int, Int, String]`, which is equivalent to `(Int, Int) => String`. ScalaDoc uses the latter convention. The return type is shown last.

Methods vs. Functions

Scala distinguishes between methods and functions. Methods are part of an object definition, and do not stand alone. Functions are first-class objects, which means they can be passed around.

You can think of a Scala object (ultimately derived from `Any`) as a container for holding properties and methods. `Functions` are a subset of Scala objects, which can be passed around as parameters and manipulated. Functions can have arity 1 through arity 22. (Note that the ScalaDoc only shows `Function1` and `Function2`, but the ScalaDoc for the other 20 `FunctionN` definitions also exist.)

Methods can accept functions as arguments, but methods cannot take methods as arguments. However, you can convert a method to a function through a process called *lifting*. We will discuss lifting in a moment.

No-Argument Function Literals

The syntax for a no-argument function literal is somewhat intuitive. Here is the definition for a function that accepts no arguments, expressed as `()`, which is a synonym for `Unit`. Although its return type is not specified, the Scala compiler knows that `System.getProperty` returns `String`, so it assigns the same return type to the function.

```
scala> val ud = () => System.getProperty("user.dir")
ud: () => String = <function0>

scala> ud()
res4: String = /home/mslinn
```

The desugared syntax for the above uses `Function0` because no arguments are accepted by the anonymous function:

```
scala> val ud2 = new Function0[ String ] { def apply(): String = System.getProperty("user.dir") }
ud2: () =>  String  = <function0>

scala> ud2()
res5:  String  = /home/mslinn
```

Notice that because the function does not accept arguments, only the return type `String` is shown as a parametric type .

Lifting a Method to a Function

You can also create a `FunctionN` by using the same syntax as was displayed in the REPL after you lifted a method into a function. For example, here is how to lift a method into an equivalent `FunctionN` that accepts the same arguments. First let's define a singleton object that has a method repeat which repeats a string a given number of times:

```
scala> object R1 { def repeat(string: String, times: Int): String = string * times }
defined module R1

scala> R1.repeat("asdf ", 3)
res6: String = asdf asdf asdf
```

Writing an underscore after a method name *lifts* the method from it original object container into a `FunctionN` container, which is why it can be stored into `val`s and not require `def`s. Lets lift `R1.repeat` to a function and then invoke it:

```
scala> val liftedFunction = R1.repeat _
function: String => (Int => String) = <function1>

scala> liftedFunction("asdf ", 3)
res7: String = asdf asdf asdf
```

As you can see, the results of running a method and its equivalent `FunctionN` are identical.

To define a `FunctionN` without having to lift it from an object:

1. Write the function type in terms of the arguments types and returned type, without variable names
2. Repeat the pattern with with the variable names that correspond to each type. The `FunctionN` type is highlighted in yellow (it includes the return type, `String`), and the function implementation is highlighted in blue . Notice that the `FunctionN` type is on the left side of the equals sign and the implementation is on the right of the equals sign :

```
scala> val  f2 :  (String, Int) => String  =  (arg1, arg2) => arg1 * arg2
f2: (String, Int) => String = <function2>

scala> f2("asdf ", 4)
res10: String = asdf asdf asdf asdf
```

The above definition for `f2` is somewhat redundant; we can use placeholder syntax to simplify the code to:

```
val f2: (String, Int) => String = _ * _
```

The REPL displays the type of `f2` using syntactic sugar. Another way to write the type is `Function2[String, Int, String]` as shown below. Using this syntax, the last parametric type is the return type , and the preceding types are the argument types .

```
scala> val f3 : Function2[ String, Int,  String ] = (arg1, arg2) => arg1 * arg2
f3: (String, Int) => String = <function2>

scala> f3("asdf ", 4)
res11: String = asdf asdf asdf asdf
```

The above definition for `f3` is somewhat redundant; we can use placeholder syntax to simplify the code to:

```
val f3: Function2[String, Int, String] = _ * _
```

The next course discusses currying and partially applied functions, which use lifting extensively.

Quiz: Passing Functions as parameters

Given the following program:

```
object FunSel extends App {
  type StringOp = (String, Int) => String

  def blackBox(f: StringOp, string: String, n: Int): String = f(string, n)

  val fn1: StringOp = _ substring _
  val fn2: StringOp = _ * _

  println(s"""fn1 supplied with "bad/good dog" and 4 gives: YourAnswerHere""")
  println(s"""fn2 supplied with "arf " and 3 gives: YourAnswerHere""")
}
```

1. What does the program do?
2. How can you modify it so the string YourAnswerHere is replaced with some code that invokes fn1 and fn2?

Solution

`StringOp` defines the signature for the functions passed to `blackBox`; they accept a `String` and an `Int` parameter and return a `String`. `blackBox` executes whatever method that it receives and passes in two parameters: a `String` and an `Int`. `blackBox` was written to work with any function that accepts the proper signature.

Two functions are defined, both written with a combination of infix and placeholder syntax. Infix syntax should only be used for purely functional methods (methods with no side-effects).

1. `fn1` takes a substring of the first parameter it is passed using the Java `String.substring` method, which must be a `String` (remember that `StringOp` defines the parameters). The

second parameter specifies where to start the substring. The substring continues to the end of the string. `fn1` could be written using postfix notation like this:

```
val fn1: StringOp = _.substring(_)
```

2. `fn2` repeats the first parameter it is passed, which must be a `String`. The second parameter, and `Int`, specifies the number of times to repeat the string.

The two `println` statements should be rewritten as follows:

```
println(s"""fn1 supplied with "good/bad dog" and 5 gives: ${fn1("bad/good dog", 4)}""")
println(s"""fn2 supplied with "arf " and 3 gives: ${fn2("arf ", 3)}""")
```

You can run the solution as follows:

```
sbt "runMain solutions.FunSel"
```

Output is:

```
fn1 supplied with "good/bad dog" and 5 gives: good dog
fn2 supplied with "arf " and 3 gives: arf arf arf
```

Composition

Here we see a trait that extends `Function1[Int, Int]`, expressed with syntactic sugar (`Int => Int`). The trait defines two methods: `apply`, which is a default method and so is implicitly called if no method name is provided, and tilde(~), commonly used in Scala as a synonym for "and then". The use of `andThen` is a form of *method composition*, also known as method chaining. We will see how that works in a second:

```
trait Fn extends (Int => Int) {
    def apply(x: Int): Int

    def ~(f: => Fn) = this andThen f
}

val addOne: Fn = new Fn { def apply(x: Int) = 1 + x }

val multiplyTwo: Fn = new Fn { def apply(x: Int) = 2 * x }
```

The trait `Fn` is abstract because the `apply` method is not implemented. `addOne` and `multiplyTwo` are instances of anonymous classes which implement the trait's abstract `apply` method.

Let's use these definitions. First, let's call each method separately:

```
scala> addOne(2)
res14: Int = 3

scala> multiplyTwo(4)
res15: Int = 8
```

Next we can compose the two functions together, thereby defining a new instance of a new anonymous class, also with type `Int => Int`.

```
scala> val compute = multiplyTwo ~ addOne
compute: Int => Int = <function1>
```

Now we can invoke this new function by passing it an `Int`. Because the definition of ~ is for `this` (which is `multiplyTwo`) to be performed, `andThen f` to be performed (which is `addOne`), the argument is first doubled then one is added.

```
scala> compute(2)
res16: Int = 5
```

We can define another anonymous class, which defines a function that performs the operations in the opposite order, and invoke it all together, like this:

```
scala> (addOne ~ multiplyTwo)(6)
res17: Int = 14
```

4 Useful Stuff

ScalaCourses.com / scalaIntro / ScalaCore / scalaIntroMisc

4-1 Unit testing With ScalaTest and Specs2

ScalaCourses.com / scalaIntro / ScalaCore / scalaIntroMisc / scalaUnit

ScalaTest and Specs2 are both comprehensive test frameworks. This lecture is merely intended to give you an introduction to common aspects of these test frameworks. Both of these unit testing frameworks are much more powerful than `JUnit`, however they interoperate with `JUnit` and can use the `JUnit` test runner. A nice feature of both frameworks is the ability to write unit tests using a DSL. One way that projects gradually introduce Scala into an all-Java code base is to write unit tests using these frameworks.

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`, `testOnly` and `testQuick` tasks. Scala-IDE and IntelliJ IDEA both offer integrated unit test support. You cannot use the REPL, Scala scripts or worksheets to write unit tests.

This lecture was prepared using IntelliJ IDEA 13 and Scala-IDE v3.0.2 64 bit on Linux and Mac OS X.

Project Setup

Specs2

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

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

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

ScalaTest

ScalaTest can be included into a `build.sbt` file like this:

```
libraryDependencies ++= Seq (
  "org.scalatest" %% "scalatest" % "2.0"   % "test",
  "junit"         %  "junit"     % "4.8.1" % "test" // Scala IDE requires junit; IntelliJ IDEA does not
)
```

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

ScalaTest and Specs2 In the Same Project

You can add dependencies for Specs2 and ScalaTest in the same `build.sbt` file. If Scala IDE support is also included, the complete `libraryDependencies` are:

```
libraryDependencies ++= Seq(
  "org.specs2"    %% "specs2"    % "2.1.1" % "test",
  "org.scalatest" %% "scalatest" % "2.0"   % "test",
  "junit"         %  "junit"     % "4.8.1" % "test"  // Scala IDE requires junit; IntelliJ IDEA does not
)
```

The `junit` dependency required by Scala IDE is compatible with IntelliJ IDEA. I suggest you always add it to `build.sbt` as shown for every project. Because all of these dependencies are only required for tests, they do not add overhead to your deployed project.

Quick and Easy Test Specifications

Both test frameworks provide several flavors of DSL for writing unit tests. Statements usually contain `should` or `must` statements which themselves contain one or more `in` statements. Tests written with the DSL read like

Be sure to place your unit tests in the test/scala or test/java directories

`"The MumbleFratz should obliviate the FramminJammin in"`. If a problem occurs the error messages are therefore quite helpful.

Specs2

This file is provided in `courseNotes/src/test/scala/Specs2Demo.scala`. The test class extends `Specification`, which provides a DSL for writing unit tests.

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

@RunWith(classOf[JUnitRunner])
class Specs2Demo 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.

ScalaTest

ScalaTest is extremely flexible in how unit tests can be structured. This example was set up to resemble the Specs2 unit test above as closely as possible. The test class extends `WordSpec`, which provides a DSL for writing unit tests. This file is provided as `courseNotes/src/test/scala/ScalaTestDemo.scala`:

```
import org.junit.runner.RunWith
import org.scalatest.junit.JUnitRunner
import org.scalatest._

@RunWith(classOf[JUnitRunner])
class ScalaTestDemo extends WordSpec {
  "The 'Hello world' string" should {
    "contain 11 characters" in {
      assert("Hello world".length == 11)
    }

    "start with 'Hello'" in {
      assert("Hello world".startsWith("Hello"))
    }

    "end with 'world'" in {
      assert("Hello world".endsWith("world"))
    }
  }
}
```

The ScalaTest documentation discusses additional test types. For more information.

SBT

SBT has a good test runner. You can run all tests by using the `sbt test` command, as shown below. Recall that the leading tilde (~) causes the tests to be rerun any time a source file is changed.

```
$ 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/courseNote
s/)
[success] Total time: 33 s, completed Sep 2, 2013 5:14:36 PM
1. Waiting for source changes... (press enter to interrupt)
```

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

```
$ sbt ~"testOnly ScalaTestDemo"
[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/courseNote
s/)
[success] Total time: 2 s, completed Sep 2, 2013 5:16:42 PM
1. Waiting for source changes... (press enter to interrupt)
```

A leading tilde in combination with `testQuick` causes only those tests affected by the most recently modified change to be recompiled and tested. This is probably the most useful incantation of the three:

```
$ sbt ~testQuick
[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/courseNote
s/)
[success] Total time: 2 s, completed Sep 2, 2013 5:16:42 PM
1. Waiting for source changes... (press enter to interrupt)
```

IntelliJ IDEA

Running ScalaTest and Specs 2 unit tests 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

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

1. Create a new SBT project.
2. Write a unit test using Specs2 that fetches the contents of `http://www.scalacourses.com` and does a case-insensitive search that verifies the word `scala` is present.
3. Write the test in ScalaTest as well.
4. Use any IDE, or no IDE.

Hints

1. You can read the contents of a URL as follows:
   ```
   io.Source.fromURL("http://scalacourses.com").getLines().mkString
   ```
2. You can convert a `String` to lower case via the `String.toLowerCase` method.

Solutions are provided in `courseNotes/test/scala/solutions/ScalaTestSolution` and `courseNotes/test/scala/solutions/Specs2Solution`. You could run them this way:

```
sbt "testOnly solutions.Specs2Solution"
sbt "testOnly solutions.ScalaTestSolution"
```