Intermediate Scala

Core Scala

http://scalacourses.com

Intermediate Scala

Printed in the United States of America.

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

scalaIntermediate

Intermediate Scala

This course material can be taken online or delivered as a three-day in-person intermediate Scala course. Exercises are provided that build on the lecture material.

Section 1 scalaPracticalities **Practicalities** **10 lectures**

Lecture 1-3 scalaImplicitValues Implicit Values 00:21:09

- Managing Implicit Values
  - Accessing the Implicit Value in Scope
- @implicitNotFound
- Value Classes
  - Exercise: Implicit Single Parameter List
  - Solution
- 'With' Pattern Using Implicits
  - Challenge: Understanding Implicits by Reading Code
  - Solution

Lecture 1-4 scalaImplicitConversions Implicit Conversions 00:12:29

- Implicit Resolution Rules
- Implicit Scope Management
  - Exercise
  - Solution
- Default Values for Implicit Parameters
- Coercion by Implicit Conversions
  - Exercise - Convert Tuples to Complex
  - Solution
- Implicit Search Order
- Predef.scala
- REPL :implicits
- Scala 2.11 Improvement

Lecture 1-5 scalaImplicitClasses Implicit Classes 00:08:08

- Simple Example
- Implicit Value Class
  - Exercise - What does this print out?
  - Solution
- "Enhance My Library" / "Extension Method" Pattern
  - Exercise - Enhance Complex
  - Solution
  - Exercise - Separating Business Logic from the Domain Model
  - Solution

Lecture 1-7 scalaProcesses Process Control 00:12:16

- ProcessBuilder
- Helper Traits
  - URLBuilder
  - FileBuilder
- Composing ProcessBuilders

Useful ProcessBuilder operators
Challenge: Reading Process Output into Memory
Solution
Challenge: Managing Processes
Solution

Lecture 1-8 scalaIO Scala I/O 00:04:53

Console Input
Challenge: Prompt Loop
Solutions
io.Source
Writing to Files

Section 2 scalaTypesCollections **Types and Collections** **17 lectures**

Lecture 2-1 scalaHigher Higher-Order Functions 00:15:13

Higher-Order Functions
Higher Order Function Shorthand
Lambdas
Write Your Own Higher-Order Functions
Periodic Invocation
Timing a Task
Challenge: Using the Predefined WrappedString Implicit Class
Solution
Reading Binary Files Using Higher-Order Functions
Exercise: What does this program do?
Solution

Lecture 2-2 scalaParametric Parametric Types 00:11:07

Parametric Polymorphism
'With' Pattern Revisited
Challenge: What is output?
Solution
Example: Ternary Operator DSL
Parametric Traits as Rich Interfaces

Lecture 2-3 scalaCollectionIntro Introduction to Collections 00:14:34

Array
Traversable
Iterator
Collection Types
HashMap
Map
HashSet and LinkedHashSet
Set
Type Widening

Lecture 2-4 scalaCollectionImmutable Immutable Collections

List
Right-Associative Operators

Lecture 2-11 `view` Parametric Bounds and Variance

- Lower Bounds
- Upper Bounds
- Covariance
- Contravariance
- Example of Upper and Lower Bounds

Lecture 2-12 `scalaStructural` Structural Types

- Self Traits and Structural Types

Lecture 2-13 `scalaSeqMatch` Pattern Matching on Sequences

- Extracting members from a List

Lecture 2-14 `scalaTypeclasses` Typeclasses

- Implicit Conversions
- Typeclasses
- Context Bound Shorthand Notation
- Review of Bounds
- implicitly
- implicitNotFound
- Implicit Conversions vs. Typeclasses
- Another Example
  - View Bound Desugaring and Possible Deprecation

Lecture 2-16 `interSoFar1` Assignment

Lecture 2-17 `scalaPartiallyApplied` Partially Applied Functions

- Curried Functions
- Partially Applied Function with one or more Functors
  - Eta-expansion turns A* to Seq[A]
- The Loan Pattern
- Alternative Syntax
- Using Partially Applied Functions
- Wrapping Database Operations With the Loan Pattern

Section 3 `scalaMulti` **Concurrency and Parallelism** **10 lectures**

Lecture 3-1 `scalaConc` Concurrency

- Functional Style Pipelines
- Concurrency Definitions
  - java.util.concurrent
- j.u.c. Is Too Hard To Use
- Concurrency Hazards
- Typical Latencies
  - Standard Thread Pools
  - Daemon Threads
  - Fork / Join Framework
- JVM Command Line Settings
- Better Concurrency Options
- Akka Enhancements

- j.u.c. ExecutorService
- Akka Dispatcher and j.u.c.
- ActorSystem
  - Default Akka ActorSystem
    - Custom Akka Dispatcher With Fallback
  - ActorSystem Methods and Properties
    - Factories
    - Methods
    - Properties
- Shutdown Akka System
  - Sample Code
- Akka Configuration
  - Custom Akka Dispatcher

Lecture 3-2 `scalaParallel` Parallel Collections

- Use Case: Simple Simulation
- Regular Collections Can Be Transformed Into Parallel Collections
- Reducing Results

Lecture 3-3 `scalaPartialFns` Partial Functions

- Shorthand Notation
  - Composing Partial Functions
- Case Sequences Are Partial Functions
  - Example of Case Sequences and Pattern Matching In a For Comprehension
  - Parametric 'With' Pattern Revisited

Lecture 3-4 `scalaFutures` Futures

- Futures and Promises
- For Combinators
- Combinators
- Collections of Futures
  - Partial Functions and collect
- Failure Propagation
- Daemon Threads
- Bad: Accessing Shared Mutable State From A Future

Lecture 3-5 `akkaDataflow` Akka Dataflow

- Dataflow Compared to Future

Lecture 3-8 `scalaActorIntro` Introduction to Actors

- Actors
- Actor Wiring
- Queuing Theory
  - Actor Pooled Worker Pattern
    - Back Pressure / Metering
  - Queuing Example, multiple nodes
- Testing Actors
- Akka Pattern Package
  - akka.pattern.ask
  - akka.pattern.pipe

This course provides a deep, hands-on investigation into the real-world application of functional programming and concurrency techniques.

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 checkout the course project as a git repository.

This is a hands-on course. Please don't just read the notes, try every code example yourself. Type along with the REPL or a Scala Worksheet. 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.

Sample Code

Registered users will see a tab on the course overview page entitled **Course Details** that provides the URL for the sample code repository. As with the previous course, I show code throughout this course that can be pasted into the REPL, and that code is also provided in the the `courseNotes` directory. You can choose to run the sample code as standalone programs, or you can paste it into the REPL to play with, or you can paste it into a Scala worksheet.

Prerequisites

Introduction to Scala course, or equivalent.

Before Coming to Class

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 few 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, however any text editor will do.

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?

The instructor will attempt to deal with issues on the spot if you are taking this course in a live class. If you are taking this course online, and for unresolved problems and for suggestions, please use the Issue Tracker.

Course Evaluations

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

1 Practicalities

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaPracticalities

1-3 Implicit Values

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaPracticalities / scalaImplicitValues

The code for this lecture can be found in ImplicitValues.scala.Registered users will see a tab on the course overview page entitled **Course Details** that provides the URL for the sample code repository. As with the previous course, I show code throughout this course that can be pasted into the REPL, and that code is also provided in the the courseNotes directory. You can choose to run the sample code as standalone programs, or you can paste it into the REPL to play with, or you can paste it into a Scala worksheet.

Implicit values allow missing method parameters to be found in the runtime context. This means implicit values are useful for simplifying APIs. Both the object being supplied as an implicit parameter and the method signature must be decorated with the word implicit.

```
implicit val defaultMultiplier = 2

def multiply(value: Int)( implicit multiplier: Int): Int = value * multiplier
```

Note that multiply has two parameter lists, and that the second is prefaced with the word implicit. This is because if a parameter in a list is implicit, all parameters in that list become implicit. Only the last parameter list may be implicit. Let's see how to use the above definitions:

```
scala> multiply(2)(3)
res0: Int = 6

scala> multiply(5)
res1: Int = 10
```

The above code can be run like this:

```
sbt "run-main ImplicitValues"
```

Following the uniform access principle, vals, vars and defs may all be marked implicit, however implicit methods have a special use, as we shall learn in the next lecture.

If multiple implicitly defined objects are available, the first matching object found in the following search order will be used: local scope, enclosing class, parent class, and companion object.

Note that implicit values can not be top-level, they must be members of an object or class.

Managing Implicit Values

The above example shows an implicit of type Int being used in a code fragment. This works fine for small programs, however imagine a much larger program that has many implicits of type Int, used for various purposes. It would be difficult to identify which implicit that should be used because the Scala compiler matches implicits based on type. Let's rewrite the above so it uses a wrapper class to allow us to identify the Int values we wish to target to implicit parameters. This writing style is more appropriate for a large-scale program. To drive the point home, we introduce another

operation, division, which also requires an `implicit Int`. The `Divider` wrapper class clarifies the `Int` that should be used when resolving implicits. Using the REPL:

Define wrapper types when you wish to provide common value types as implicits

```
case class Multiplier(value: Int)

case class Divider(value: Int)

implicit val defaultMultiplier = Multiplier(2)

implicit val defaultDivider = Divider(3)

def multiply(value: Int)(implicit multiplier: Multiplier ): Int = value * multiplier.value

def divide(value: Int)(implicit divider: Divider ): Int = value / divider.value
```

Now we can use the above just the same as before:

```
scala> multiply(2)(Multiplier(3))
res2: Int = 6

scala> multiply(5)
res3: Int = 10

scala> divide(12)(Divider(4))
res4: Int = 3

scala> divide(9)
res5: Int = 3
```

The rules for where implicits may be defined and how they are resolved are bent a bit by the REPL. The only way to find out what is actually legal is to write a real Scala program. Let's recast the above as an actual Scala program.

```
object ImplicitValues extends App {
  implicit val defaultMultiplier = Multiplier(2)

  implicit val defaultDivider = Divider(3)

  def multiply(value: Int)(implicit multiplier: Multiplier): Int = value * multiplier.value

  def divide(value: Int)(implicit divider: Divider): Int = value / divider.value

  println(s"multiply(2)(Multiplier(3))=${multiply(2)(Multiplier(3))}")
  println(s"multiply(5)=${multiply(5)}")
  println(s"divide(12)(Divider(4))=${divide(12)(Divider(4))}")
  println(s"divide(9)=${divide(9)}")
}
```

You can run this code as follows:

```
sbt "run-main ImplicitValues2"
```

Output is:

```
multiply(2)(Multiplier(3))=6
multiply(5)=10
divide(12)(Divider(4))=3
divide(9)=3
```

The next lecture will show how we can rewrite the above so `Int`s get converted to `Multiplier`s and `Divider`s automatically, as required.

Accessing the Implicit Value in Scope

Sometimes you need to obtain the `implicit` value selected by the compiler for a given type. The `implicitly` function can provide that implicit value to you.

We can use `implicitly` to search for an implicit value of type `String` and assign it to a variable called `m`:

```
scala> val m = implicitly[Multiplier]
m: Multiplier = Multiplier(2)
```

Now lets look for an implicit value that does not exist. For example, let's search for an `implicit` value of type `Int` and attempt to assign it to `c`.

```
scala> val c = implicitly[Int]
<console>:6: error: could not find implicit value for parameter e: Int
       val c = implicitly[Int]
```

The error is confusing. In the code shown there is no parameter called `e`!

@implicitNotFound

This is another reason to use a strongly typed implicit: because you can specify a custom error message when an implicit is not found. Let's do that with `Multiplier2` and `Divider2`: by using the `@implicitNotFound` annotation:

```
import annotation.implicitNotFound

@implicitNotFound("Cannot find implicit of type Multiplier2 in scope")
case class Multiplier2(value: Int)

@implicitNotFound("Cannot find implicit of type Divider2 in scope")
case class Divider2(value: Int)
```

`@implicitNotFound` can be applied to all kinds of classes and traits. Here is how it works:

```
scala> implicitly[Multiplier2]
<console>:16: error: Cannot find implicit of type Multiplier2 in scope
              implicitly[Multiplier2]
                         ^
```

Value Classes

`Multiplier` and `Divider` require a new object allocation each time an instance is created; this is known as "boxing". Referencing the wrapped value requires "unboxing". The boxing and unboxing can be avoided by deriving the wrapper classes from `AnyVal`, which is the parent of all value classes. Recall that a value class simply contains a single value. Examples of value classes include `Int` and `String`. Value class constructors can only accept one `val` parameter, which is the underlying runtime representation. We can modify `Multiplier` and `Divider` to become value classes by having them extend `AnyVal`, like this:

```
case class Multiplier3(value: Int) extends AnyVal

case class Divider3(value: Int) extends AnyVal
```

Now `Int`s can be converted to `Multiplier3` and `Divider3` instances without boxing and unboxing, because the runtime representation of value classes is actually the primitive type (`Int`). Once the references to `Multiplier` and `Divider` are changed to `Multiplier3` and `Divider3`, no other changes are required to the rest of the program. You can run the version of the sample code which uses value classes like this:

```
sbt "run-main ImplicitValues3"
```

By the way, you can define value classes as regular classes, if you prefer. Just be sure to make the value a property by decorating it with `val`:

```
object ImplicitValues4 extends App {
  @implicitNotFound("Cannot find implicit of type Multiplier4 in scope")
  class Multiplier4( val value: Int) extends AnyVal

  @implicitNotFound("Cannot find implicit of type Divider4 in scope")
  class Divider4( val value: Int) extends AnyVal

  implicit val defaultMultiplier = new Multiplier4(2)

  implicit val defaultDivider = new Divider4(3)

  def multiply(value: Int)(implicit multiplier: Multiplier4): Int = value * multiplier.value

  def divide(value: Int)(implicit divider: Divider4): Int = value / divider.value

  println(s"multiply(2)(3)=${multiply(2)( new Multiplier4(3))}")
  println(s"multiply(5)=${multiply(5)}")
  println(s"divide(12)(4)=${divide(12)( new Divider4(4))}")
  println(s"divide(9)=${divide(9)}")
}
```

You can run the version of the sample code which uses value classes that are not case classes like this:

```
sbt "run-main ImplicitValues4"
```

Finally, value classes may not contain internally defined variables. For example, this generates a compiler error:

```
class Multiplier4(val value: Int) extends AnyVal {
  val x = 1
}
```

Value classes have numerous limitations, and may require allocation (leading to boxing/unboxing) in many circumstances. This means that benchmarking your application may reveal that value classes provide no benefit while imposing awkward limitations. The use case presented here, typesafe default values delivered via implicits, without any further manipulation of those values, is a good use of value classes.

Exercise: Implicit Single Parameter List

This exercise will show you an implementation quirk of Scala's implicit values. Modify the `ImplicitValues4` program above and add a new operation called `square`. This new method only accepts one parameter, which is implicit, that is the value to square. The method signature should be:

```
def square(implicit squarer: Squarer): Int
```

Hint

1. The quirk is this: to call the new `square` method such that it picks up the implicit value currently in scope, simply type the name of the method without using any parentheses, like this:

   ```
   square
   ```

Solution

```
package solutions

import scala.annotation.implicitNotFound

object ImplicitValues extends App {
  @implicitNotFound("Cannot find implicit of type Multiplier4 in scope")
  class Multiplier4(val value: Int) extends AnyVal

  @implicitNotFound("Cannot find implicit of type Divider4 in scope")
  class Divider4(val value: Int) extends AnyVal

   @implicitNotFound("Cannot find implicit of type Squarer in scope")
  class Squarer(val value: Int) extends AnyVal

  implicit val defaultMultiplier = new Multiplier4(2)

  implicit val defaultDivider = new Divider4(3)

   implicit val defaultSquarer = new Squarer(4)

  def multiply(value: Int)(implicit multiplier: Multiplier4): Int = value * multiplier.value

  def divide(value: Int)(implicit divider: Divider4): Int = value / divider.value

   def square(implicit squarer: Squarer): Int = squarer.value * squarer.value

  println(s"multiply(2)(3)=${multiply(2)(new Multiplier4(3))}")
  println(s"multiply(5)=${multiply(5)}")
  println(s"divide(12)(4)=${divide(12)(new Divider4(4))}")
  println(s"divide(9)=${divide(9)}")
  println(s"square(5)=${ square(new Squarer(5) )}")
  println(s"square=${ square }")
}
```

To run this solution, type:

```
sbt "runMain solutions.ImplicitValues"
```

'With' Pattern Using Implicits

As we saw in the Introduction to Scala course, the With Pattern is common used in Scala; again, this is my name for the pattern – that name is not widely recognized. Let's modify the example we used to include implicit values.

```
case class Blarg(i: Int, s: String)

def withBlarg(blarg: Blarg)(operation: Blarg => Unit):  Unit = operation(blarg)

def double(implicit blarg: Blarg): Blarg = blarg.copy(i=blarg.i*2, s=blarg.s*2)

def triple(implicit blarg: Blarg): Blarg = blarg.copy(i=blarg.i*3, s=blarg.s*3)
```

The above code looks very much like what we saw in the previous course. This code does not yet take advantage of the `implicit` parameters defined for the `double` and `triple` methods:

```
scala> withBlarg(Blarg(1, "asdf")) { blarg =>
  println(double(blarg))
  println(triple(blarg))
}

Blarg(2,asdf asdf)
Blarg(3,asdf asdf asdf)
```

If we decorate the `blarg` reference passed into the closure with `implicit`, we can rewrite as follows:

```
scala> withBlarg(Blarg(1, "qwer")) {  implicit  blarg =>
  println( double )
  println( triple )
}

Blarg(2,qwer qwer)
Blarg(3,qwer qwer qwer)
```

You can arbitrarily decorate the incoming parameter(s) with `implicit` any time you see a With Pattern being used. The author of `withBlarg` does not know or care if you decorate the incoming `Blarg` instance with `implicit`.

The above is provided in the same source file. To run:

```
sbt "run-main With2"
```

Output is:

```
Blarg(2,asdf asdf )
Blarg(3,asdf asdf asdf )
Blarg(2,qwer qwer )
Blarg(3,qwer qwer qwer )
```

Challenge: Understanding Implicits by Reading Code

The ability to read someone else's code is a skill that only comes with practice. What does this program do?

```
import java.util.{Date, Locale}
import java.text.{NumberFormat, DateFormat}

object WithLocale extends App {
  def formatDateTime(date: Date)(implicit dateFormat: DateFormat): String = dateFormat.format(date)

  def formatNumber(number: BigDecimal)(implicit numberFormat: NumberFormat): String = numberFormat.format(number)

  def withLocale(locale: Locale)(body: Locale => String): String = body(locale)

  def result(date: Date, number: BigDecimal)(implicit locale: Locale): String = {
    implicit val dateFormat = DateFormat.getDateTimeInstance(DateFormat.DEFAULT, DateFormat.DEFAULT, locale)
    implicit val numberFormat = NumberFormat.getNumberInstance(locale)
    val currencyCode = numberFormat.getCurrency.getCurrencyCode
    s"""${locale.getDisplayCountry}; currency: $currencyCode; """ +
      s"""date/time: ${formatDateTime(date)}; large number: ${formatNumber(number)}"""
  }

  val resultUS = withLocale(Locale.US) { implicit locale =>
    result(new Date, 123456789)
  }

  val resultFrance = withLocale(Locale.FRANCE) { implicit locale =>
    result(new Date, 987654321)
  }

  println(resultUS)
  println(resultFrance)
}
```

Solution

You can run the program like this:

```
sbt "runMain WithLocale"
```

The output is:

```
United States; currency: USD; date/time: Mar 1, 2014 3:04:24 PM; large number: 123,456,789
France; currency: EUR; date/time: 1 mars 2014 15:04:24; large number: 987 654 321
```

1-4 Implicit Conversions

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaPracticalities / scalaImplicitConversions

The source code for this lecture may be found in ImplicitConversions.scala.

Implicit methods allow an instance of a type to be viewed as if it were another type. Implicit methods provide implicit conversions.

Implicit conversion methods are automatically lifted into functions

Using the example from the previous lecture, we can write implicit converters between Int and Multiplier3, and Int and Divider3:

```
@implicitNotFound("Cannot find implicit of type Multiplier3 in scope")
case class Multiplier3(value: Int) extends AnyVal

@implicitNotFound("Cannot find implicit of type Divider3 in scope")
case class Divider3(value: Int) extends AnyVal

object ImplicitConversions extends App {
  implicit val defaultMultiplier = Multiplier3(2)

  implicit val defaultDivider = Divider3(3)

  def multiply(value: Int)(implicit multiplier: Multiplier3): Int = value * multiplier.value

  def divide(value: Int)(implicit divider: Divider3): Int = value / divider.value

   implicit def intToMultiplier(int: Int) = Multiplier3(int)

   implicit def intToDivider(int: Int) = Divider3(int)

  println(s"multiply(2)(3)=${multiply(2)(3)}")
  println(s"multiply(5)=${multiply(5)}")
  println(s"divide(12)(4)=${divide(12)(4)}")
  println(s"divide(9)=${divide(9)}")
}
```

intToMultiplier and intToDivider are implicitly called when the Int values highlighted in orange (3 and 4) need to be converted into Multiplier3 or Divider3 instances. Because these methods are called implicitly, their names do not appear anywhere in the program except where they are defined. Note that the original Int values (3 and 4) are not actually changed; instead, the implicit methods generate new Multiplier3 or Divider3 values from the original Int values, and the new values are passed as parameters to the multiply or divide methods.

Output is:

```
multiply(2)(3)=6
multiply(5)=10
divide(12)(4)=3
divide(9)=3
```

You can run this code as follows:

```
sbt "run-main ImplicitConversions"
```

Implicit methods can be defined in classes, objects and traits.

Implicit Resolution Rules

The previous lecture described where the Scala compiler looks for implicit values. The compiler uses the following rules to resolve implicit values and functions.

1. Decoration – Properties and methods must be decorated with `implicit` in order to be considered for resolution by the compiler.
2. Scope – The compiler only considers the implicits currently in scope as a single identifier for resolution. Properties of an implicit object are not considered, unless they are also marked as an `implicit`.
   ```
   implicit object Blah {
     implicit val yesMe: Boolean = true
    val notMe: Boolean = false
   }
   ```
3. One at a time – Only one implicit is tried at a time to resolve - implicits do not normally chain.
4. Explicits first – if a value or function is explicitly provided, then implicits are not used.

Implicit Scope Management

It is often helpful to define implicits in a package object. This makes accessing the implicit easier. The code below uses a trick to define the implicit converter `stringToYeller` in the `yeller` package object.

```
package yeller {
  case class Yeller(s: String) {
    def yell: String = s.toUpperCase + "!!"
  }

   object `package` {
    implicit def stringToYeller(s: String): Yeller = Yeller(s)
  }
}

object YellerMain extends App {
  import yeller._

  println("Look out".yell)
}
```

Notice the backticks that surround the word `` `package` ``. This causes the word `package` to be considered as an object name instead of a Scala keyword. You can place backticks around otherwise illegal package and variable names to force them to be evaluated as names. The result is that `stringToYeller` is defined in the `yeller` package object, which is imported into `YellerMain` by `import yeller._`. Because the implicit converter `stringToYeller` is now in scope, strings in the `YellerMain` object that have an otherwise illegal method invoked are automatically converted to Yeller instances as required. This allows us to write `"Look out".yell`.

You can run this program as follows:

```
sbt "runMain YellerMain"
```

Output is:

```
LOOK OUT!!
```

Exercise

1. Rewrite `YellerMain` so that it is more efficient by using value classes.
2. Add a `whisper` method that forces `String`s to lower case, and prefaces them with `"Shhh! "`

Solution

This solution is provided as `solutions.EfficientYeller`.

```
package solutions

case class Yeller(val s: String) extends AnyVal {
  def yell: String = s.toUpperCase + "!!"
   def whisper: String = "Shhh! " + s.toLowerCase
}

object `package` {
  implicit def stringToYeller(s: String): Yeller = Yeller(s)
}

object EfficientYeller extends App {
  println("Look out".yell)
   println("This is a secret".whisper)
}
```

You can run this solution by typing:

```
sbt "runMain solutions.EfficientYeller"
```

Output is:

```
LOOK OUT!!
Shhh! this is a secret
```

Default Values for Implicit Parameters

You can specify default values for implicit parameters. This can be a source of difficult-to-find bugs. I recommend that you do not do this!

```
scala> def asdf(implicit x: Int=3): Unit = println(x)
asdf: (implicit x: Int)Unit

scala> implicit val y=2
y: Int = 2

scala> asdf()
3
```

You can run this code as follows:

```
sbt "runMain ImplicitDefaultValues"
```

Coercion by Implicit Conversions

As you know, coercion is what a compiler of a typed language does when converting an argument or an operand to the type expected by a function or an operator. As we have already seen, Scala supports conversion through implicits. Here is an example:

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

  def -(that: Complex) =  new Complex(re - that.re, im - that.im)

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

implicit def doubleToComplex(d: Double) = Complex(d, 0)
```

Let's implicitly use the method `doubleToComplex` to convert a `Double` instance into a `Complex` instance, so we can add it to another `Complex` instance:

```
scala> Complex(2.0, 5.0) + 5.0
res8: Complex = 7.0 + 5.0i

scala> 5.0 + Complex(1.0, -2.0)
res9: Complex = 6.0 + -2.0i
```

You can run this code as follows:

```
sbt "runMain ImplicitCoercion"
```

Exercise - Convert Tuples to Complex

Extend the `ImplicitCoercion` program so `Tuple2`s of `Int`s, `Float`s, `Double`s and all 9 combinations thereof can be added and subtracted with `Complex` instances. Assume that the first property of a tuple is the real part, and the second property is the imaginary part.

Hint

As discussed in the previous course, you could use the special notation for `Tuple2`. However, your program will read better if you do not.

Solution

```
package solutions

object ImplicitCoercion extends App {
  case class Complex(re: Double, im: Double) {
    def + (that: Complex) = new Complex(re + that.re, im + that.im)

    def -(that: Complex) =  new Complex(this.re - that.re, this.im - that.im)

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

  implicit def doubleToComplex(d: Double) = Complex(d, 0)

   implicit def tupleToComplex1(t: (Int, Int))       = Complex(t._1, t._2)
  implicit def tupleToComplex2(t: (Int, Float))     = Complex(t._1, t._2)
  implicit def tupleToComplex3(t: (Float, Int))     = Complex(t._1, t._2)
  implicit def tupleToComplex4(t: (Float, Float))   = Complex(t._1, t._2)
  implicit def tupleToComplex5(t: (Int, Double))    = Complex(t._1, t._2)
  implicit def tupleToComplex6(t: (Double, Int))    = Complex(t._1, t._2)
  implicit def tupleToComplex7(t: (Double, Double)) = Complex(t._1, t._2)
  implicit def tupleToComplex8(t: (Float, Double))  = Complex(t._1, t._2)
  implicit def tupleToComplex9(t: (Double, Float))  = Complex(t._1, t._2)

  println(s"""Complex(2, 5) + 5.0 = ${Complex(2, 5) + 5.0}""")
  println(s"""5.0 + Complex(1, -2) = ${5.0 + Complex(1, -2)}""")

  println(s"""Complex(2, 5) + (2f, 5) = ${Complex(2, 5) + (2f, 5)}""")
  println(s"""Complex(2, 5) + (2, 5f) = ${Complex(2, 5) + (2, 5f)}""")
  println(s"""Complex(2, 5) + (2f, 5f) = ${Complex(2, 5) + (2f, 5f)}""")

  println(s"""Complex(2, 5) + (2d, 5) = ${Complex(2, 5) + (2d, 5)}""")
  println(s"""Complex(2, 5) + (2, 5d) = ${Complex(2, 5) + (2, 5d)}""")
  println(s"""Complex(2, 5) + (2d, 5d) = ${Complex(2, 5) + (2d, 5d)}""")

  println(s"""Complex(2, 5) + (2f, 5d) = ${Complex(2, 5) + (2f, 5d)}""")
  println(s"""Complex(2, 5) + (2d, 5f) = ${Complex(2, 5) + (2d, 5f)}""")
  println(s"""Complex(2, 5) + (2d, 5d) = ${Complex(2, 5) + (2d, 5d)}""")
```

```
}
```

You can run this solution by typing:

```
sbt "runMain solutions.ImplicitCoercion"
```

Output is:

```
Complex(2, 5) + 5.0 = 7.0 + 5.0i
5.0 + Complex(1, -2) = 6.0 + -2.0i
Complex(2, 5) + (2f, 5) = 4.0 + 10.0i
Complex(2, 5) + (2, 5f) = 4.0 + 10.0i
Complex(2, 5) + (2f, 5f) = 4.0 + 10.0i
Complex(2, 5) + (2d, 5) = 4.0 + 10.0i
Complex(2, 5) + (2, 5d) = 4.0 + 10.0i
Complex(2, 5) + (2d, 5d) = 4.0 + 10.0i
Complex(2, 5) + (2f, 5d) = 4.0 + 10.0i
Complex(2, 5) + (2d, 5f) = 4.0 + 10.0i
Complex(2, 5) + (2d, 5d) = 4.0 + 10.0i
```

Implicit Search Order

This is a more detailed explanation of where the Scala compiler looks for implicit values and converters.

1. Implicits defined in the current scope bind first. The package object is part of the current scope.

   ```
   object InnerScope {
     implicit val list2 = List(1, 2, 3)

     def res(implicit list: List[Int]): List[Int] = list
   }

   object OuterScope extends App {
     implicit val list = List(1, 2)

     println(InnerScope.res)
   }
   ```

   To run this program, type:

   ```
   sbt "runMain OuterScope"
   ```

   Output is:

   ```
   List(1, 2)
   ```

2. If an implicit of the required type is not found in the current scope or parent scope, the Scala compiler then searches imports for an implicit declaration of the required type.

   ```
   object InnerScope {
     implicit val list2 = List(1, 2, 3)

     def res(implicit list: List[Int]): List[Int] = list
   }

   object ImportedImplicit extends App {
     import InnerScope._

     println(res)
   }
   ```

   To run this program, type:

```
sbt "runMain ImportedImplicit"
```

Output is:

```
List(1, 2, 3)
```

3. The Scala compiler then looks in companion objects for implicits:

```
object CompanionScope extends App {
  class A(val n: Int) {
    def +(other: A) = new A(n + other.n)
  }

  object A {
    implicit def fromInt(n: Int) = new A(n)
  }

  val x = 1 + new A(1) // is converted into:
  val y = A.fromInt(1) + new A(1)

  println(s"x.n=${x.n}")
  println(s"y.n=${y.n}")
}
```

To run this, type:

```
sbt "runMain CompanionScope"
```

Output is:

```
x.n=2
y.n=2
```

Here is an excellent article on the subject. I provide a modified version of the code in `ImplicitFun.scala` in the `courseNotes` project.

Predef.scala

The source code for the Scala runtime library includes `Predef.scala`, which defines type aliases and many implicit conversion methods. These definitions are automatically imported into every Scala program. Predef provides many implicit conversions which convert from a value type, such as `Int` or `String`, to an enhancing type, such as `RichInt` and `WrappedString`.

`Predef.scala` also defines a useful method called `require`, which has the following signature:

```
final def require(requirement: Boolean, message: ⇒ Any): Unit
```

`require` tests an expression, throwing an `IllegalArgumentException` if `false`. This method is similar to `assert`, but blames the caller of the method for violating the condition.

I encourage you to read the source code for Predef.scala.

REPL :implicits

The Scala REPL's `:implicits` command displays all the implicits in scope. Unless you add the `-v` switch, it does not display the implicit conversions defined in `Predef.scala`.

```
scala> :implicits
No implicits have been imported other than those in Predef.

scala> :implicits -v
/* 78 implicit members imported from scala.Predef */
  /* 48 inherited from scala.Predef */
  implicit def boolean2BooleanConflict(x: Boolean): Object
  implicit def byte2ByteConflict(x: Byte): Object
  implicit def char2CharacterConflict(x: Char): Object
  implicit def double2DoubleConflict(x: Double): Object
  implicit def float2FloatConflict(x: Float): Object
  implicit def int2IntegerConflict(x: Int): Object
  implicit def long2LongConflict(x: Long): Object
  implicit def short2ShortConflict(x: Short): Object

  implicit def Boolean2boolean(x: Boolean): Boolean
  implicit def Byte2byte(x: Byte): Byte
  implicit def Character2char(x: Character): Char
  implicit def Double2double(x: Double): Double
  implicit def Float2float(x: Float): Float
  implicit def Integer2int(x: Integer): Int
  implicit def Long2long(x: Long): Long
  implicit def Short2short(x: Short): Short
  implicit val StringCanBuildFrom: generic.CanBuildFrom[String,Char,String]
  implicit def any2ArrowAssoc[A](x: A): ArrowAssoc[A]
  implicit def any2Ensuring[A](x: A): Ensuring[A]
  implicit def any2stringadd(x: Any): runtime.StringAdd
  implicit def any2stringfmt(x: Any): runtime.StringFormat
  implicit def arrayToCharSequence(xs: Array[Char]): CharSequence
  implicit def augmentString(x: String): immutable.StringOps
  implicit def boolean2Boolean(x: Boolean): Boolean
  implicit def booleanArrayOps(xs: Array[Boolean]): mutable.ArrayOps[Boolean]
  implicit def byte2Byte(x: Byte): Byte
  implicit def byteArrayOps(xs: Array[Byte]): mutable.ArrayOps[Byte]
  implicit def char2Character(x: Char): Character
  implicit def charArrayOps(xs: Array[Char]): mutable.ArrayOps[Char]
  implicit def conforms[A]: <:<[A,A]
  implicit def double2Double(x: Double): Double
  implicit def doubleArrayOps(xs: Array[Double]): mutable.ArrayOps[Double]
  implicit def exceptionWrapper(exc: Throwable): runtime.RichException
  implicit def float2Float(x: Float): Float
  implicit def floatArrayOps(xs: Array[Float]): mutable.ArrayOps[Float]
  implicit def genericArrayOps[T](xs: Array[T]): mutable.ArrayOps[T]
  implicit def int2Integer(x: Int): Integer
  implicit def intArrayOps(xs: Array[Int]): mutable.ArrayOps[Int]
  implicit def long2Long(x: Long): Long
  implicit def longArrayOps(xs: Array[Long]): mutable.ArrayOps[Long]
  implicit def refArrayOps[T <: AnyRef](xs: Array[T]): mutable.ArrayOps[T]
  implicit def seqToCharSequence(xs: IndexedSeq[Char]): CharSequence
  implicit def short2Short(x: Short): Short
  implicit def shortArrayOps(xs: Array[Short]): mutable.ArrayOps[Short]
  implicit def tuple2ToZippedOps[T1, T2](x: (T1, T2)): runtime.Tuple2Zipped.Ops[T1,T2]
  implicit def tuple3ToZippedOps[T1, T2, T3](x: (T1, T2, T3)): runtime.Tuple3Zipped.Ops[T1,T2,T3]
  implicit def unaugmentString(x: immutable.StringOps): String
  implicit def unitArrayOps(xs: Array[Unit]): mutable.ArrayOps[Unit]

  /* 30 inherited from scala.LowPriorityImplicits */
  implicit def Boolean2booleanNullConflict(x: Null): Boolean
  implicit def Byte2byteNullConflict(x: Null): Byte
```

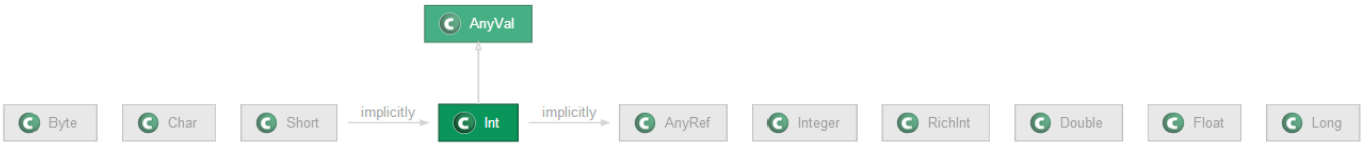
```
implicit def Character2charNullConflict(x: Null): Char
implicit def Double2doubleNullConflict(x: Null): Double
implicit def Float2floatNullConflict(x: Null): Float
implicit def Integer2intNullConflict(x: Null): Int
implicit def Long2longNullConflict(x: Null): Long
implicit def Short2shortNullConflict(x: Null): Short
implicit def booleanWrapper(x: Boolean): runtime.RichBoolean
implicit def byteWrapper(x: Byte): runtime.RichByte
implicit def charWrapper(c: Char): runtime.RichChar
implicit def doubleWrapper(x: Double): runtime.RichDouble
implicit def fallbackStringCanBuildFrom[T]: generic.CanBuildFrom[String,T,immutable.IndexedSeq[T]]
implicit def floatWrapper(x: Float): runtime.RichFloat
implicit def genericWrapArray[T](xs: Array[T]): mutable.WrappedArray[T]
implicit def intWrapper(x: Int): runtime.RichInt
implicit def longWrapper(x: Long): runtime.RichLong
implicit def shortWrapper(x: Short): runtime.RichShort
implicit def unwrapString(ws: immutable.WrappedString): String
implicit def wrapBooleanArray(xs: Array[Boolean]): mutable.WrappedArray[Boolean]
implicit def wrapByteArray(xs: Array[Byte]): mutable.WrappedArray[Byte]
implicit def wrapCharArray(xs: Array[Char]): mutable.WrappedArray[Char]
implicit def wrapDoubleArray(xs: Array[Double]): mutable.WrappedArray[Double]
implicit def wrapFloatArray(xs: Array[Float]): mutable.WrappedArray[Float]
implicit def wrapIntArray(xs: Array[Int]): mutable.WrappedArray[Int]
implicit def wrapLongArray(xs: Array[Long]): mutable.WrappedArray[Long]
implicit def wrapRefArray[T <: AnyRef](xs: Array[T]): mutable.WrappedArray[T]
implicit def wrapShortArray(xs: Array[Short]): mutable.WrappedArray[Short]
implicit def wrapString(s: String): immutable.WrappedString
implicit def wrapUnitArray(xs: Array[Unit]): mutable.WrappedArray[Unit]
```

Remember that all of the implicit conversions are defined as methods, but Scala lifts all of them into functions. (We discussed method 'lifting' in the previous course.) This is only important when you want to use `implicitly` to discover the implicit converter in scope.

Discovering the Current Implicit Conversion In Scope with Implicitly

Sometimes you need to access the implicit function that converts between two types. The `implicitly` method can check if an implicit converter of a given type is available and return it if so.

For example, `Int` can be implicitly converted to `Long`, as you can see in the class diagram for `Int` (click on "Type Hierarchy").



We can use the `implicitly` method in the REPL to discover the function that performs the implicit conversion from `Int` to `Long`.

```
scala> val i2l = implicitly[Function[Int, Long]]
i2l: Int => Long = <function1>

scala> i2l(3)
res10: Long = 3
```

We can use an alternative syntax to obtain the same implicit conversion function, and then its `apply` method is invoked while passing in 5, so the result `5L` can be computed:

```
scala> implicitly[Int => Long].apply(5)
res11: Long = 5
```

This code can be run as follows:

```
sbt "runMain ImplicitlyConversion"
```

Scala 2.11 Improvement

The implicit arguments of implicit conversions are now handled correctly with Scala 2.11.

1-5 Implicit Classes

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaPracticalities / scalaImplicitClasses

Implicit classes allow the addition of extra functionality onto an existing type without actually modifying the original type. Implicit classes allow business logic to be defined outside of value objects, and provides a form of dependency injection. Implicit classes are just like regular Scala classes, with the following additional rules:

1. Implicit classes are decorated with the `implicit` keyword.
2. Implicit classes must not be top-level objects. You must define them in another object or class.
3. Implicit class constructors must receive at least one parameter.
4. The only methods in implicit classes that will be considered for implicit conversions are those that do accept any parameters.

The sample code for this lecture is provided in `ImplicitClasses.scala`.

Simple Example

In the following simple example, an implicit class is defined which wraps an `Int`. Note that the name of this class is never directly referenced in your program because it is implicitly referenced. This means that the name of the class is completely arbitrary. The `length` method returns the length of the string representation of the `Int`. From the point of view of the programmer when this implicit class is in scope, it seems as if `Int`s now have a method called `length`! However, please note that the definition of `Int` has not changed. All that really happens is that the Scala compiler looks around for implicit classes whenever it cannot find a method - rather like Ruby's `method_missing` mechanism.

```
scala> implicit class randomName(int: Int) { def length: Int = int.toString.length }
defined class randomName

scala> 5.length
res4: Int = 1
```

Implicit Value Class

You can define an implicit class as a value class and possibly eliminate boxing and unboxing, if value class constraints are not a problem for your use case:

```
scala> implicit class randomName2( val  long: Long)  extends AnyVal  { def length: Int = long.toString.length }
defined class randomName2

scala> 5L.length
res4: Int = 1
```

Exercise - What does this print out?

```
object AppleFanBoi extends App {
  implicit class IosInt(val i: Int) extends AnyVal { def s: Int = i + 1 }

  println(s"I have an iPhone ${4.s}")
}
```

Solution

You can run this program by typing:

```
sbt "runMain solutions.AppleFanBoi"
```

Output is:

```
I have an iPhone 5
```

"Enhance My Library" / "Extension Method" Pattern

This pattern has been known by several names, including the original (and offensive) "Pimp my library" name.

The functionality defined by Scala's built-in value types are extended by Scala's implicit conversions. This is 'the Scala way', and you should consider doing this for your domain model classes. This keeps your model separate from your business logic. Let's see how this works by extending the `Dog` case class we used earlier. I defined `DogCommands` as an implicit value class because it only contains methods and no variables.

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

class Stick

case class Ball(color: String)

implicit class DogCommands(val dog: Dog) extends AnyVal {
  def call(me: String): String = s"Here, ${dog.name} come to $me"

  def fetch(stick: Stick): String = s"${dog.name}, fetch the stick!"

  def fetch(ball: Ball): String = s"${dog.name}, fetch the ${ball.color} ball!"
}
```

We can use the implicit class this way:

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

scala> dog.call("me")
res2: String = Here, Fido come to me

scala> dog.fetch(new Stick)
res3: String = Fido, fetch the stick!

scala> val ball = new Ball("green")
ball: Ball = Ball(green)

scala> dog.fetch(ball)
res4: String = Fido, fetch the green ball!
```

This is similar to how Scala's `RichInt` class extends the functionality of the `Int` value class using implicits.

You can also run this program like this:

```
sbt "runMain EnhanceMyLibrary"
```

Exercise - Enhance Complex

Use the Enhance My Library pattern to add an absolute value method to `Complex`. Compute this value by taking the square root of the square of the real part plus the square of the imaginary part.

Hint

Use `math.sqrt` to take the square root.

Solution

```
package solutions

object ImplicitClasses extends App {
  import solutions.ImplicitCoercion.Complex

  implicit class RichComplex(val complex: Complex) extends AnyVal {
    def abs: Double = math.sqrt(complex.re * complex.re + complex.im * complex.im)
  }

  println(s"Complex(3, 4).abs=${Complex(3, 4).abs}")
}
```

You can run this solution by typing:

```
sbt "runMain solutions.ImplicitClasses"
```

Output is:

```
Complex(3, 4).abs=5.0
```

Exercise - Separating Business Logic from the Domain Model

Define a domain class `Weather` that somehow describes the current state of the weather. Define an implicit class `RichWeather` that provides actions like `rain()`, `sunshine()` and `hurricane()`. Those methods should change the current state of the weather and print a suitable message. Run the solution as a console program.

Solution

```
package solutions

object WeatherMain extends App {
  case class Weather(var status: String)

  implicit class RichWeather(weather: Weather) {
    def genericOp(newStatus: String): Weather = {
      weather.status = newStatus
      println(s"It is now ${weather.status}")
      weather
    }

    def rain(): Weather = genericOp("raining")

    def hail(): Weather = genericOp("hailing")

    def sunshine(): Weather = genericOp("shining")
  }

  val weather = Weather("shining")
  weather.rain()
  weather.hail()
  weather.sunshine()
}
```

To run, type:

```
sbt "runMain solutions.WeatherMain"
```

Output is:

```
It is now raining
It is now hailing
It is now shining
```

1-7 Process Control

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaPracticalities / scalaProcesses

Let's start by having some fun with the REPL. `:sh` is a REPL command that runs a shell command and returns a value to an automatically assigned variable.

```
scala> :sh cat /etc/passwd
res0: scala.tools.nsc.interpreter.ProcessResult = `cat /etc/passwd` (43 lines, exit 0)
```

Let's print out the `Array[String]` contained in `res0`.

```
scala> res0 foreach println
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
...
```

These code examples are provided in `ProcessIO.scala`.

ProcessBuilder

You can run the code examples in this section by typing:

```
sbt "runMain ProcessIO"
```

Scala programs also have ways of running shell processes. The method I am about to show you does exactly the same thing as the REPL example above, but it does not rely on the REPL's `:sh` command and so can be incorporated into your program. Instead, it uses the Scala `ProcessBuilder` DSL:

```
scala> import sys.process._
import sys.process._
```

I imported `sys.process._`, which is the same as writing `scala.sys.process._`. We can now run an operating system command or another program and obtain `stdout`:

```
scala> val passwds = "cat /etc/passwd" !!
passwds: String =
"root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
...
```

`!!` is an implicit method defined in `scala.sys.process.Process` that executes the preceding `String` as a system command, and returns the output string to the variable `passwds`. `!!` was used with a postfix method invocation, and the Scala compiler sometimes has difficulty in figuring out if a line ending with a postfix method invocation should continue across newlines. This means it is often necessary to terminate lines ending with postfix method invocations with a semicolon. The `ProcessIO.scala` version of this code example shows this clearly.

We can capture file names from `ProcessBuilder` like this:

```
scala> val fileNames = "ls" !!
fileNames: String =
"build.sbt
project
src
target
test
"

scala> fileNames
res0: String =
"build.sbt
project
src
target
test
"
```

`ProcessBuilder` provides several ways to pipe data into processes. The following code executes the `bash printf` command, which creates a three-line string literal which looks like this:

```
xray
yankee
zulu
```

Those three lines are then piped into `grep`, which displays any lines containing `x`. The `%s` argument is the string format specifier for `printf`. The backslashes are doubled inside the String literal because Scala interprets them before invoking `printf`.

```
scala> "printf xray\\nyankee\\nzulu" #> "grep x" !!
res1: String =
"xray
"
```

Helper Traits

`ProcessBuilder` has some helper traits that can create `Processes`, such as `URLBuilder` and `FileBuilder`. Both of these helper traits have methods with similar signatures. For example, each defines a method called `#>`, which is overloaded to accept a `File` instance to write to. Explore the ScalaDoc to learn about the other methods available in these helper classes.

The `scala.sys.process` package object extends `ProcessImplicits` (defined in `Process.scala`) thereby bringing all the implicits that it defines into scope.

URLBuilder

Here is a little program that downloads a web page into a file called `scalaCourses.html`:

```
object URLBuilderDemo extends App {
  import java.io.File
  import java.net.URL
  import sys.process._

  new URL("http://scalacourses.com") #> new File("scalaCourses.html") !;
  println("cat scalaCourses.html" !!)
}
```

Output is the HTML source for this web site. It works because `ProcessImplicits` contains an implicit conversion from a `URL` instance to a `URLBuilder`.

```
implicit def urlToProcess(url: URL): URLBuilder
```

You can run this program by typing:

```
sbt "runMain URLBuilderDemo"
```

FileBuilder

This little program uses the `#>` operator to pipe the output of the bash `ls` command to a file called `dirContents.txt` and prints it out. It works because `ProcessImplicits` contains an implicit conversion from a `File` instance to a `URLBuilder`.

```
object FileBuilderDemo extends App {
  import java.io.File
  import sys.process._

  "ls" #> new File("dirContents.txt") !;
  println("cat dirContents.txt" !!)
}
```

You can run this program by typing:

```
sbt "runMain FileBuilderDemo"
```

Composing ProcessBuilders

`ProcessBuilders` can be composed, which means that you can chain sequences of `ProcessBuilders` that pipe data between themselves, and you can also specify `stderr` handling. The following method constructs a `ProcessBuilder` using the `cat` method.

```
def readUrlPBuilder(url: String, fileName: String): ProcessBuilder = new URL(url) #> new File(fileName) cat
```

We can call `readUrlPBuilder` like in the REPL this:

```
scala> readUrlPBuilder("http://scalacourses.com", "scalaCourses.html") !
res2: Int = 0

scala> :sh cat scalaCourses.html
res3: scala.tools.nsc.interpreter.ProcessResult = `cat scalacourses.html` (457 lines, exit 0)
```

`readUrlPBuilder` returns a `ProcessBuilder`, which is executed by the `!` method. We can also invoke `readUrlPBuilder` such that it encounters an error:

```
scala> readUrlPBuilder("http://n_o_s_u_c_h.com", "nosuch.html") !
java.net.UnknownHostException: n_o_s_u_c_h.com
    at java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:178)
    at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:392)
    at java.net.Socket.connect(Socket.java:579)
    at java.net.Socket.connect(Socket.java:528)
    at sun.net.NetworkClient.doConnect(NetworkClient.java:180)
    at sun.net.www.http.HttpClient.openServer(HttpClient.java:432)
    at sun.net.www.http.HttpClient.openServer(HttpClient.java:527)
    at sun.net.www.http.HttpClient.<init>(HttpClient.java:211)
    at sun.net.www.http.HttpClient.New(HttpClient.java:308)
    at sun.net.www.http.HttpClient.New(HttpClient.java:326)
    at sun.net.www.protocol.http.HttpURLConnection.getNewHttpClient(HttpURLConnection.java:996)
    at sun.net.www.protocol.http.HttpURLConnection.plainConnect(HttpURLConnection.java:932)
    at sun.net.www.protocol.http.HttpURLConnection.connect(HttpURLConnection.java:850)
    at sun.net.www.protocol.http.HttpURLConnection.getInputStream(HttpURLConnection.java:1300)
    at java.net.URL.openStream(URL.java:1037)
    at scala.sys.process.ProcessBuilderImpl$URLInput$$anonfun$$init$$1.apply(ProcessBuilderImpl.scala:30)
    at scala.sys.process.ProcessBuilderImpl$URLInput$$anonfun$$init$$1.apply(ProcessBuilderImpl.scala:30)
    at scala.sys.process.ProcessBuilderImpl$IStreamBuilder$$anonfun$$init$$5.apply(ProcessBuilderImpl.scala:44)
    at scala.sys.process.ProcessBuilderImpl$IStreamBuilder$$anonfun$$init$$5.apply(ProcessBuilderImpl.scala:44)
    at scala.sys.process.ProcessBuilderImpl$ThreadBuilder$$anonfun$1.apply$mcV$sp(ProcessBuilderImpl.scala:57)
    at scala.sys.process.ProcessImpl$Spawn$$anon$1.run(ProcessImpl.scala:22)
```

Let's compose two `ProcessBuilders` such that the second is only executed if the first succeeds:

```
scala> readUrlPBuilder("http://scalacourses.com", "scalacourses.html") #&& "echo yes" !!
yes
res4: String = yes
```

Now let's compose two `ProcessBuilders` such that the second is only executed if the first fails:

```
scala> readUrlPBuilder("http://n_o_s_u_c_h.com", "nosuch.html") #|| "echo no" !!
no
res5: String = no
```

Useful ProcessBuilder operators

The helper classes also define some of the same operators as `ProcessBuilder`.

`!` Execute the command and return the exit value (by convention, 0 means success, anything else means failure). This is a blocking call.

`!!` Like `!`, but also captures Scala process standard output.

`#|` Pipe output to Scala process standard output.

`###` Sequence two operations.

`#&&` Do the following operation if the previous operation succeeded.

`#||` Do the following operation if the previous operation failed.

`#<(b: ProcessBuilder): ProcessBuilder` Reads the output of the given `ProcessBuilder` into this process's input stream.

`#<(b: InputStream): ProcessBuilder` Reads the contents of the given `InputStream` into this process's input stream.

`#<(b: URL): ProcessBuilder` Reads the contents of the given `URL` into this process's input stream.

`#<(b: File): ProcessBuilder` Reads the contents of the given `File` into this process's input stream.

`#>``(b: ProcessBuilder): ProcessBuilder` Writes the output of this process to the given `ProcessBuilder`'s input stream..

`#>``(b: OutputStream): ProcessBuilder` Writes the output of this process to the given `OutputStream`.

`#>``(b: File): ProcessBuilder` Writes the output of this process to the given `ProcessBuilder`'s input stream..

`#>>` Create a new `ProcessBuilder` and connect its input to the output of the previous `ProcessBuilder`

There are many more operators!

Challenge: Reading Process Output into Memory

Write a short program that capitalizes every word from this text file and prints it out: `https://raw.github.com/mslinn/ExecutorBenchmark/master/README.md`

Hints

- You do not always need to invoke `ProcessBuilder` in order to create a `Process`; you can also create a `Process` using `Process.apply`, which is overloaded.
- A Java `ByteArrayOutputStream` is useful for collecting results. Look for a `ProcessBuilder` operator that accepts a `Stream` parameter.
- `ByteArrayOutputStream.toString` converts the accumulated value to a `String`.

Solution

```
import scala.sys.process._
import java.net.URL

val p = Process(new URL("https://raw.github.com/mslinn/ExecutorBenchmark/master/README.md"))
val os = new java.io.ByteArrayOutputStream
p #> os !;
os.toString.split(" ") foreach { w =>
  val word = w.length match {
    case 0 => ""
    case 1 => w.toUpperCase
    case _ => w.substring(0, 1).toUpperCase + w.substring(1)
  }
  print(s" $word")
}
```

The semicolon is required because Scala cannot tell if the next line should be considered as a continuation. It is unusual to need to specify a semicolon like this.

The solution is provided in `solutions/MemoryProcess.scala`. Run it like this:

```
sbt "runMain solutions.MemoryProcess"
```

Challenge: Managing Processes

Use the REPL or a Scala worksheet and `ProcessBuilder` to discover running Scala programs. Here is how you could write this entirely in `bash`.

```
ps aux | grep scala
```

If you are unsure what it does, try typing it into a bash shell.

Hint

- You cannot create a Scala `Process` that contains a pipe, so this will not work:

  ```
  "ps aux | grep scala" !!
  ```

- Instead, use the `Process` piping capability to chain two `Process`es.

Solution

Run the following in the REPL:

```
import scala.sys.process._
"ps aux" #> "grep scala" !!
```

1-8 Scala I/O

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaPracticalities / scalaIO

Console Input

Reading input from the user is often necessary for console applications. `Console.readLine` can accept an optional `prompt` parameter. In this example, I typed in `blah` and pressed carraige return, but the newline was not echoed by the REPL:

```
scala> val line = Console.readLine("prompt> ")
prompt> blah line: String = blah
```

Challenge: Prompt Loop

Write a small Scala program that repeatedly prompts the user for a number, then prints a running total. Handle `String` to `Int` conversion problems using `try` / `catch`. The dialog should look like this:

```
Total: 0; Input a number to add, Enter to stop> 2
Total: 2; Input a number to add, Enter to stop> 33
Total: 35; Input a number to add, Enter to stop> asdf
Invalid number ignored. Please try again.
Total: 35; Input a number to add, Enter to stop> 3
Total: 38; Input a number to add, Enter to stop>
```

Hints

1. You can convert an `Int` to a `String` like this:

   ```
   scala> "3".toInt
   res19: String  = 3

   scala> "3asdf".toInt
   java.lang.NumberFormatException: For input string: "3asdf"
           at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)

   scala> try { "3asdf".toInt } catch { case _: Throwable => 0 }
   res23: Int = 0
   ```

2. You could use a mutable variable to accumulate the result, or you could use recursion. Tail recursion is optimized by the Scala compiler so it does not use large amounts of stack memory. You can tell the compiler that a method is intended to implement tail recursion by annotating the method with `@tailrec`.
3. `sys.exit` can be used to terminate a program.
4. `sys.err` exits a program with an error condition and outputs an error message through `stderr`.

Solutions

Solution #1

This solution looks like a Java programmer wrote it. It works fine, runs efficiently and is easy to understand.

```
object PromptLoop extends App {
  var total = 0
  do {
    val line = Console.readLine(s"Total: $total; Input a number to add, Enter to stop> ").trim
    if (line.isEmpty) sys.exit()
    val number: Int = try {
      line.toInt
    } catch {
      case nfe: NumberFormatException =>
        println("Invalid number ignored. Please try again.")
        0

      case throwable: Throwable =>
        sys.error(throwable.getMessage)
    }
    total = total + number
  } while (true)
}
```

You can run the above solution like this:

```
sbt "runMain solutions.PromptLoop1"
```

Solution #2

This solution uses tail recursion, so it does not require a mutable variable to accumulate results. It is also efficient and easy to understand. Because no mutable variables are used, this style of programming is useful for multicore operations.

```
object PromptLoop2 extends App {
  @tailrec
  def getValue(increment: Int, subtotal: Int): Int = {
    val total = subtotal + increment
    val line = Console.readLine(s"Total: $total; Input a number to add, Enter to stop> ").trim
    if (line.isEmpty) 0 else {
      val userValue = try {
        line.toInt
      } catch {
        case nfe: NumberFormatException =>
          println("Invalid number ignored. Please try again.")
          0

        case throwable: Throwable =>
          sys.error(throwable.getMessage)
      }
      getValue(userValue, total)
    }
  }

  getValue(0, 0)
}
```

You can run the above solution like this:

```
sbt "runMain solutions.PromptLoop2"
```

io.Source

Scala has some easy ways to read in text files. This code is provided in `IoSource.scala`.

```
scala> io.Source.fromFile("/etc/passwd") foreach print
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
...

scala> io.Source.fromFile("/etc/passwd").mkString.length
res1: Int = 2200

scala> val rootLines = io.Source.fromFile("/etc/passwd").getLines()
rootLines: Iterator[String] = non-empty iterator
```

As you can see, `Source.getLines` returns an `Iterator` of `String`. If we did not convert the `Iterator` to a `List`, referencing the `Iterator` a second time would return nothing, like this:

Beware: iterators can only be traversed once; this is a potential source of bugs in your software!

```
s"${ rootLines .length} lines printed from /etc/p
asswd:\n  " +  rootLines .mkString("\n  ")
res0: String =
"28 lines printed from /etc/passwd:
  "
```

We can convert the `Iterator` to a `List` easily; notice that the contents of the file are now displayed:

```
scala> val rootLines = io.Source.fromFile("/etc/passwd").getLines() .toList
rootLines: List[String] = List(man:x:6:12:man:/var/cache/man:/bin/sh, lp:x:7:7:lp:/var/spool/lpd:/bin/sh, mail:x:8:8:mail:/
var/mail:/bin/sh, news:x:9:9:news:/var/spool/news:/bin/sh, uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh, www-data:x:33:33:www-
data:/var/www:/bin/sh, backup:x:34:34:backup:/var/backups:/bin/sh, list:x:38:38:Mailing List Manager:/var/list:/bin/sh, irc
:x:39:39:ircd:/var/run/ircd:/bin/sh, gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh, libuuid:x:100
:101::/var/lib/libuuid:/bin/sh, messagebus:x:102:105::/var/run/dbus:/bin/false, avahi-autoipd:x:103:106:Avahi autoip daemon
,,,:/var/lib/avahi-autoipd:/bin/false, speech-dispatcher:x:108:29:Speech Dispatcher,,,:/var/run/speech-dispatcher:/bin/sh,
colord:x:109:117:colord colour management daemon,,,:/var/li...

scala> s"${rootLines.length} lines printed from /etc/passwd:\n  " + rootLines.mkString("\n  ")
res2: String =
28 lines printed from /etc/passwd containing '/var/':
  man:x:6:12:man:/var/cache/man:/bin/sh
  lp:x:7:7:lp:/var/spool/lpd:/bin/sh
  mail:x:8:8:mail:/var/mail:/bin/sh
  news:x:9:9:news:/var/spool/news:/bin/sh
  uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
  www-data:x:33:33:www-data:/var/www:/bin/sh
  backup:x:34:34:backup:/var/backups:/bin/sh
  list:x:38:38:Mailing List Manager:/var/list:/bin/sh
  irc:x:39:39:ircd:/var/run/ircd:/bin/sh
  gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
  libuuid:x:100:101::/var/lib/libuuid:/bin/sh
  messagebus:x:102:105::/var/run/dbus:/bin/false
  avahi-autoipd:x:103:106:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/bin/false
  speech-dispatcher:x:108:29:Speech Dispatcher,,,:/var/run/speech-dispatcher:/bin/sh
  colo...
```

The Scala compiler converts the following into the same code as the above:

```
scala> val rootLines2 = (for {
    line <- io.Source.fromFile("/etc/passwd").getLines()
  } yield line). toList
rootLines2: List[String] = List(man:x:6:12:man:/var/cache/man:/bin/sh, lp:x:7:7:lp:/var/spool/lpd:/bin/sh, mail:x:8:8:mail:
/var/mail:/bin/sh, news:x:9:9:news:/var/spool/news:/bin/sh, uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh, www-data:x:33:33:www
-data:/var/www:/bin/sh, backup:x:34:34:backup:/var/backups:/bin/sh, list:x:38:38:Mailing List Manager:/var/list:/bin/sh, ir
c:x:39:39:ircd:/var/run/ircd:/bin/sh, gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh, libuuid:x:10
0:101::/var/lib/libuuid:/bin/sh, messagebus:x:102:105::/var/run/dbus:/bin/false, avahi-autoipd:x:103:106:Avahi autoip daemo
n,,,:/var/lib/avahi-autoipd:/bin/false, speech-dispatcher:x:108:29:Speech Dispatcher,,,:/var/run/speech-dispatcher:/bin/sh,
 colord:x:109:117:colord colour management daemon,,,:/var/li...

scala> s"${rootLines2.length} lines printed from /etc/passwd:\n  " + rootLines2.mkString("\n  ")
res3: String =
28 lines printed from /etc/passwd containing '/var/':
  man:x:6:12:man:/var/cache/man:/bin/sh
  lp:x:7:7:lp:/var/spool/lpd:/bin/sh
  mail:x:8:8:mail:/var/mail:/bin/sh
  news:x:9:9:news:/var/spool/news:/bin/sh
  uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
  www-data:x:33:33:www-data:/var/www:/bin/sh
  backup:x:34:34:backup:/var/backups:/bin/sh
  list:x:38:38:Mailing List Manager:/var/list:/bin/sh
  irc:x:39:39:ircd:/var/run/ircd:/bin/sh
  gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
  libuuid:x:100:101::/var/lib/libuuid:/bin/sh
  messagebus:x:102:105::/var/run/dbus:/bin/false
  avahi-autoipd:x:103:106:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/bin/false
  speech-dispatcher:x:108:29:Speech Dispatcher,,,:/var/run/speech-dispatcher:/bin/sh
  colo...
```

You can run this sample code by typing:

```
sbt "runMain IoSource"
```

We will learn how to read binary files in a the lecture on higher-order functions.

Writing to Files

Scala uses Java to write to files:

```
import java.io.{File, PrintWriter}

val writer = new PrintWriter(new File("test.txt"))
writer.write("Hello Scala")
writer.close()
```

A later lecture in this course will show how to handle I/O errors gracefully and efficiently.

2 Types and Collections

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaTypesCollections

2-1 Higher-Order Functions

If you are a bit fuzzy on how `Option` works, please review the previous course's lecture on `Option` before diving into this lecture. Essentially, an `Option` is a collection of zero or one elements. The `None` subclass of `Option` contains zero elements, and the `Some` subclass contains one element.

The code for this lecture is provided in `HigherFun.scala`.

Higher-Order Functions

Let's say we have an `Option` of `Int` that needs to have an operation performed upon it, such as squaring the element if present.

```
scala> def square(num: Int): Int = num * num
square: (num: Int)Int

scala> val maybeInt = Some(3)
maybeInt: Some[Int] = Some(3)

scala> maybeInt.map(x => square(x))
res1: Some[Int] = Some(9)
```

The `map` method iterates over each element of the `Option`, and passes each element to the `square` method. Another `Option` is created from the result, and because squaring an `Int` always results in another `Int`, the type of the function passed into the map method is `Int => Int`. In other words, the function passed in receives an `Int` parameter and returns another `Int`. `map` is called a *higher-order function* because it receives a function as a parameter. A higher-order function, also known as a *functor*, is a function that takes one or more functions as an input parameter and/or outputs a function.

Here is another example, using the standard Scala `toDouble` method which converts an `Int` to a `Double`; this `map` operation therefore transforms an `Option[Int]` into an `Option[Double]`.

```
scala> maybeInt.map( x => x.toDouble )
res2: Some[Double] = Option(3.0)
```

The type of the function argument is `Int => Double`. Let's try this same `map` operation with a different collection type. In this example, the input `Array[Int]` is transformed into the output `Array[Double]`.

```
scala> Array(1, 2, 3).map( x => x.toDouble )
res3: Array[Double] = Array(1.0, 2.0, 3.0)
```

map can convert a container of one type of element into a similar container of another type of element

Higher Order Function Shorthand

Scala allows us to write this same operation more succinctly. The collection item passed in by `map` to the anonymous function is only referenced once, so there is really no need to give it a name. The underscore (_) refers to the current list item as `map` iterates through the collection's items.

```
scala> maybeInt.map(_.toDouble)
res4: Some[Double] = Some(3.0)
```

Now let's rewrite `maybeInt.map(x => square(x))` using progressively terser shorthands. First we can elide the definition and single usage of the temporary variable to _:

```
scala> maybeInt.map(square _)
res5: Some[Int] = Some(9)
```

If only one parameter is passed to a higher-order function, the underscore can also be omitted:

```
scala> maybeInt.map(square)
res6: Some[Int] = Some(9)
```

Using infix notation we can even do away with the parentheses:

```
scala> maybeInt map square
res7: Some[Int] = Some(9)
```

Lambdas

The function passed into `map` could be formally defined, like `square` and `toDouble`, or it could be anonymous, as we shall now see. BTW, anonymous functions are often called *lambdas*. This lambda doubles whatever is passed into it:

```
scala> maybeInt.map(_ * 2)
res8: Some[Int] = Some(6)
```

We are not able to use the underscore shorthand in higher order functions if the bound variable is referenced more than once in the anonymous function.

```
scala> maybeInt.map(x => x * x)
res8: Some[Int] = Some(9)
```

Write Your Own Higher-Order Functions

Periodic Invocation

This higher-order function accepts an interval length in seconds, and a functor , then executes the functor periodically according to the specified interval. Notice that the Java `Timer` and `TimerTask` classes are imported inside the `TimedTask` object, and that the `run` method specified by the Java `TimerTask` interface is implemented in Scala code. The higher-order function `apply` was written to accept two parameter lists; this was not required, however the usage of the higher-order function is cleaner when one functor is provided per parameter list:

```
object TimedTask {
  import java.util.Timer
  import java.util.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)
  }
}
```

The functor is shown in yellow twice in the above code; the second time it is shown the functor is not invoked. Instead, this assignment connects the `run` method required by the Java `TimerTask` interface to the implementation passed via the `apply` method's second parameter list.

The code below uses the `TimedTask` object; you can find this code in `TimedTask.scala`, in the `courseNotes` directory. You can use curly braces to surround lambdas any time.

If a lambda has more than one statement, the lambda must be surrounded with curly braces instead of parentheses

```
object TimedFun extends App {
  var i = 0

  TimedTask(1)  {
    println(s"Tick #$i")
    i = i + 1
  }
}
```

You can run this program by typing:

```
sbt "runMain TimedFun"
```

Here is some sample output:

```
Tick #0
Tick #1
Tick #2
Tick #3
Tick #4
Tick #5
Tick #6
Tick #7
```

Timing a Task

This next higher-order function accepts a block of code and times how long it takes to execute. The method prints out the time it took and returns the result of the method execution.

The `block` parameter is lazily evaluated; it is a call by name parameter and is only evaluated if the body of the time method specifically invokes `block`. The `block` parameter is defined such that it could accept a method which does not require any parameters. We can write the type of a function that can be passed in as `() => Any`.

(Recall that `()` is a synonym for `Unit`, which is a singleton object that denotes no value.) The `block` parameter could also accept a variable or constant value of any type, because all types are subclasses of `Any`.

The return type of the time method is `Any`, so it can return any type of result. We will rewrite this method in the lecture on parametric types later in this course so that it returns a strongly typed result instead of `Any`.

```
def time( block: => Any ): Any = {
  val t0 = System.nanoTime()
  val result: Any = block
  val elapsedMs = (System.nanoTime() - t0) / 1000000
  println("Elapsed time: " + elapsedMs + "ms")
  result
}
```

Let's use the `time` method to see how long it takes to calculate `Pi` to 1,000,000 decimal places. Instead of writing a lambda, I have defined a method called `calculatePiFor`:

```
scala> def calculatePiFor(decimalPlaces: Int): Double = {
  var acc = 0.0
  for (i <- 0 until decimalPlaces)
    acc += 4.0 * (1 - (i % 2) * 2) / (2 * i + 1)
  acc
}

scala> time( calculatePiFor(100000) )
Elapsed time: 3ms
res24: Any = 3.1415826535897198
```

`calculatePiFor` is called with `decimalPlaces` set to 1,000,000. The entire invocation with the bound parameter is passed into the `time` method, not just the unbound `calculatePiFor` method like we saw earlier. This bound invocation satisfies the required signature, `() => Any`, and is passed to the `block` parameter of the `time` higher-order function.

You can run this code using SBT by typing:

```
sbt "runMain TimedPi"
```

Challenge: Using the Predefined WrappedString Implicit Class

Write a one line program using the REPL that reverses every word in a text file – any file you have access to on your computer's hard drive. `/etc/passwd` is fine.

Hints

1. At the REPL prompt, type a string, then dot (.), then `tab`. See any useful methods? Unfortunately, the REPL does not look for implicit methods when doing code completion – however both Scala IDE and IntelliJ IDEA do, so a Scala worksheet or a Scala source file provides extra capability beyond what the REPL offers.
2. Implicit methods that make the `WrappedString` class available for implicit conversions are predefined by `predef.scala`; using the IDE of your choice, can you see if it has any useful methods?
3. Since `WrappedString` is a predefined implicit class, will your final code actually need to

mention that class?

4. `String.split(" ")` is the dual of `Array.mkString(" ")`.
5. `String.reverse` reverses a `String`.

Solution

```
object StringReverser {
  import scala.sys.process._
  "cat /etc/passwd".!!.split(" ").map(_.reverse).mkString(" ")
}
```

Executing the above results in:

```
TN-U:445:91:*:ecivreSlacoL
::81-5-1-S,:445:81:*:METSYS TN-U:445:02:*:ecivreSkrowteN
::91-5-1-S,ecivreSlacoL\YTIROHTUA TN-U:4927694924:4927694924:*:rellatsnIdetsurT
::445-23-5-1-S,:445:445:*:srotartsinimdA
::02-5-1-S,ecivreSkrowteN\YTIROHTUA ekiM
hsab/nib/:tseuG/emoh/:105-6299776593-9239486513-5407444803-12-5-1-S,tseuG\raeB-U:315:105:desunu:tseuG
hsab/nib/:rotartsinimdA/emoh/:005-6299776593-9239486513-5407444803-12-5-1-S,rotartsinimdA\raeB-U:315:005:desunu:rotartsinimdA
::4648741722-1362923581-4408301381-9462258143-588800659-08-5-1-S,rellatsnIdetsurT\ECIVRES ekiM\raeB-U:315:0001:desunu:nnilS ekiM/emoh/:0001-6299776593-9239486513-5407444803-12-5-1-S,nnilS
hsab/nib/:resUsutadpU/emoh/:2001-6299776593-9239486513-5407444803-12-5-1-S,resUsutadpU\raeB-U,resUsutadpU:315:...
```

The solution can be found in the `solution.StringReverser.sc.` Scala worksheet.

Reading Binary Files Using Higher-Order Functions

Now that we know how higher-order functions work, we can use them to read binary files.

`io.Source.fromFile` uses an implicit to provide the value of a codec that specifies the character set used. You can discover the default codec like this:

```
scala> implicitly[io.Codec]
res0: scala.io.Codec = UTF-8
```

You can read a binary file into a `byte` array by explicitly specifying an 8-bit codec, such as `ISO-8859-1`:

```
import sys.process._
io.Source.fromFile("which scalac".!!.trim, "ISO-8859-1").map(_.toByte).toArray
```

Notice that I ran a bash command, "`which scalac`", using the Scala `Process` facility. I trimmed off the trailing newline returned from the process before passing the resulting fully qualified name to `io.Source.fromFile`. `fromFile` returns an `Iterator` of `Character`. The `map` method I just showed converts each

`Character` to a `Byte`. In other words, map converts the `Iterator[Character]` into an `Iterator[Byte]`. The resulting `Iterator[Byte]` is converted to an `Array[Byte]` by the `toArray` method.

Since we want to always ensure that the source file is closed, this incantation is preferred:

```
val source = scala.io.Source.fromFile("which scalac".!!.trim, "ISO-8859-1")
val byteArray = source.map(_.toByte).toArray
source.close()
```

Even better, `try/catch/finally` should be used to guarantee `source` is closed after it is read.

```
val source = scala.io.Source.fromFile("which scalac".!!.trim, "ISO-8859-1")
try { source.map(_.toByte).toArray } finally { source.close() }
```

This code fragment does a lot of character conversions, which is inefficient. A more efficient mechanism is shown in the lecture on collections.

Exercise: What does this program do?

```
import java.io.File
import java.sql.Date

object HigherFun extends App {
  def uploadForm(saec: SignedAndEncodedComplete)(selectBucketUrl: PublisherConfig => String): String = {
    s"""<form action="${selectBucketUrl(saec.pubConf)}" method="post" enctype="multipart/form-data">
  <input type="hidden" name="key" value="${saec.fileName}">
  <input type="hidden" name="AWSAccessKeyId" value="${saec.accessKey}">
  <input type="hidden" name="acl" value="${saec.acl}">
  <input type="hidden" name="policy" value="${saec.encodedPolicy}">
  <input type="hidden" name="signature" value="${saec.signedPolicy}">
  <input type="hidden" name="Content-Type" value="${saec.contentType}">
  Select <code>${saec.fileName}</code> for uploading to S3:
  <input name="file" type="file">
  <br>
  <input type="submit" value="Upload">
</form>""".stripMargin
  }

  val file = new File("/etc/passwd")
  val pubConf = PublisherConfig(
    name="blah",
    awsAccountName="blah",
    awsSecretKey="blah",
    awsAccessKey="blah",
    bucketName="bucket1")
  val saec = SignedAndEncodedComplete(
    fileName=file.getAbsolutePath,
    contentLength=file.length,
    accessKey="blahBlah",
    secretKey="blah Blah",
    acl="private",
    pubConf=pubConf,
    encodedPolicy="blah blah",
```

```
    signedPolicy="blah blah",
    contentType="blah blah")

  val uf1 = uploadForm(saec)(_.homeworkBucketUrl(file.getName))
  val uf2 = uploadForm(saec)(_.uploadBucketUrl(file.getName))
  println(uf1)
  println(uf2)

  case class PublisherConfig(
    name: String,
    awsAccountName: String,
    awsSecretKey: String,
    awsAccessKey: String,
    bucketName: String,
    created: Date = new Date(System.currentTimeMillis),
    active: Boolean = false,
    id: Option[Long] = None
  ) {
    def uploadBucketUrl(file: String): String = s"http://upload.$bucketName/$file"
    def homeworkBucketUrl(file: String): String = s"http://homework.$bucketName/$file"
  }

  case class SignedAndEncodedComplete(
    fileName: String,
    contentLength: Long,
    accessKey: String,
    secretKey: String,
    acl: String,
    encodedPolicy: String,
    signedPolicy: String,
    pubConf: PublisherConfig,
    contentType: String
  )
}
```

Solution

Output is:

```
<form action="http://homework.bucket1/passwd" method="post" enctype="multipart/form-data">
  <input type="hidden" name="key" value="/etc/passwd">
  <input type="hidden" name="AWSAccessKeyId" value="blahBlah">
  <input type="hidden" name="acl" value="private">
  <input type="hidden" name="policy" value="blah blah">
  <input type="hidden" name="signature" value="blah blah">
  <input type="hidden" name="Content-Type" value="blah blah">
  Select <code>/etc/passwd</code> for uploading to S3:
  <input name="file" type="file">
  <br>
  <input type="submit" value="Upload">
</form>
<form action="http://upload.bucket1/passwd" method="post" enctype="multipart/form-data">
  <input type="hidden" name="key" value="/etc/passwd">
  <input type="hidden" name="AWSAccessKeyId" value="blahBlah">
  <input type="hidden" name="acl" value="private">
  <input type="hidden" name="policy" value="blah blah">
  <input type="hidden" name="signature" value="blah blah">
  <input type="hidden" name="Content-Type" value="blah blah">
  Select <code>/etc/passwd</code> for uploading to S3:
  <input name="file" type="file">
  <br>
  <input type="submit" value="Upload">
</form>
```

You can play with this program in `courseNotes/solutions`; it is called `HigherFun.scala`. Run it like this:

```
sbt "runMain solutions.HigherFun"
```

2-2 Parametric Types

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaTypesCollections / scalaParametric

The code examples in this lecture are provided in ParametricFun.scala.

The following method below takes an Option of Int and returns an indication of whether the Option contains Some value or not:

```
def hasValue(option: Option[Int]): Boolean = option.isDefined
```

There is nothing in the definition above that is specific to integers, other than the declaration of Option's type. The code for checking to see if an Option of String has a value is identical:

```
def hasValue(option: Option[String]): Boolean = option.isDefined
```

We can generalize the two definitions above by parameterizing not just the input option but also its type:

```
def hasValue[T] (option: Option[T]): Boolean = option.isDefined
```

Type parameters are listed in square brackets after the method name and can be used in the type declarations within the method. In the definition above, T is the (only) type parameter, standing for any Scala type. This defines a *family* of length functions, one for each type T that the Scala compiler notices is used in the program: hasValue[Int], hasValue[String]...

```
scala> hasValue[Int](Some(1))
res0: Boolean = true

scala> hasValue[Int](None)
res1: Boolean = false
```

Recall from the previous course that None is a singleton object, so there is only one instance. This means that unlike Some, the None instance is identical for all parametric types. The following incantation therefore provides the same result as the previous incantation:

```
scala> hasValue(None)
res2: Boolean = false
```

Let's look at a few more scenarios:

```
scala> hasValue[String](Some("a"))
res3: Boolean = true

scala> hasValue[(Int, Int)](Option((1,2)))
res4: Boolean = true

scala> hasValue[Array[Int]](Some(Array(1,2)))
res5: Boolean = true
```

In most cases the type annotation for hasValue can be dropped because the Scala compiler will usually be able to figure out the right parametric type automatically:

```
scala> hasValue(Some(1))
res6: Boolean = true

scala> hasValue(Some("a"))
res7: Boolean = true

scala> hasValue(Some((1, 2)))
res8: Boolean = true

scala> hasValue(Some(Array(1, 2)))
res9: Boolean = true
```

The method `hasValue` exhibits *parametric polymorphism* (also known as *generics* or *templates*), which allows the method definition to represent many `hasValue` methods differing only for the type of the elements in the `Option` input. The type of `hasValue` is `Option[T] => Boolean`. This means for any type `T`, `hasValue` takes an `Option` of type `T` and returns a `Boolean`.

Classes, traits and methods can be parametric.

Parametric Polymorphism

Here is the `time` method from the previous lecture, rewritten to use parametric types to return a strongly typed result. We'll define it as part of a parametric class:

```
class Timeable[T] {
  def time(block: => T): T = {
    val t0 = System.nanoTime()
    val result: T = block
    val elapsedMs = (System.nanoTime() - t0) / 1000000
    println("Elapsed time: " + elapsedMs + "ms")
    result
  }
}
```

The square brackets introduce a parametric type to the `time` method. In this case we only define one type, called `T`. The type could equally well have been called `Result`, but it is common to use one letter names for parametric types. Anywhere in the method that you wish to refer to that type, merely write `T`. You can invoke the method with any value of `T`, and if you do not specify the value of `T` it will be inferred. In other words, if you wanted to time three methods that return an `Int`, a `String` and a `Double`, this single parametric method saves you from having to write the following methods, because `T` is a parametric type:

```
def time(block: => Int): Int = ???

def time(block: => String): String = ???

def time(block: => Double): Double = ???
```

Let's use this method to again time how long it takes to compute `Pi` to many decimal places.

```
def calculatePiFor(decimalPlaces: Int): Double = {
  var acc = 0.0
  for (i <- 0 until decimalPlaces)
    acc += 4.0 * (1 - (i % 2) * 2) / (2 * i + 1)
  acc
}
```

Now we can run `time` on `calculatePiFor`. Notice the the result is strongly typed as `Double`, instead of `Any` as was the case in the previous lecture:

```
scala> new Timeable.time[ Double ] { calculatePiFor(100000) }
Elapsed time: 0ms
res10: Double = 3.1415826535897198
```

Let's time a function that returns a `String` instead of a `Double`:

```
scala> new Timeable.time[ String ] { io.Source.fromURL("http://scalacourses.com").mkString.trim }
Elapsed time: 481ms
res11: String =
"<!DOCTYPE html>
<html lang="en">
<head>
  <title>Welcome to ScalaCourses.com</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <script src='/webjars/jquery/1.9.1/jquery.min.js' type='text/javascript'>
  <script src='/webjars/jquery-ui/1.10.2/ui/minified/jquery-ui.min.js' type='text/javascript'>
  <script src='/webjars/bootstrap/2.3.2/js/bootstrap.min.js' type='text/javascript'>
  <script src='/webjars/bootstrap-datepicker/1.0.1/js/bootstrap-datepicker.js' type='text/javascript'>
  <script src='/assets/javascripts/jwerty.js' type="text/javascript">
  <script src='/javascriptRoutes' type="text/javascript">
  <link href='/assets/images/favicon.ico' rel="shortcut icon" type="image/x-ic...
```

You can run this program by typing:

```
sbt "runMain TimeableT"
```

'With' Pattern Revisited

We can make a parametric polymorphic version of the With pattern we learned in the Introduction to Scala course. For such a powerful facility, it is surprising how simple and easy it is to define and use with Scala:

```
def withT[T](t: T)(operation: T => Unit): Unit = { operation(t) }
```

Now let's use it:

```
scala> withT(6) { i => println(i*2) }
6

scala> withT(new java.util.Date) { println }
Tue Oct 29 05:48:07 PDT 2013

scala> case class Blarg(i: Int, s: String)
defined class Blarg

scala> withT(Blarg(1, "hi")) { blarg => println(blarg) }
Blarg(1,hi)
```

You can run this code example by typing:

```
sbt "runMain ParametricWith"
```

Challenge: What is output?

```
package parametricSimulation {
  abstract class AbstractSimulation[X, Y, Z] {
    def simulate(x: X, y: Y, z: Z): String
  }

  object AbstractSimulation {
    implicit lazy val defaultSimulation = new AbstractSimulation[Int, Double, String] {
      def simulate(x: Int, y: Double, z: String) = s"Companion simulation: $x, $y, $z"
    }
  }

  trait Implicit {
    implicit lazy val defaultSimulation = new AbstractSimulation[Int, Double, String] {
      def simulate(x: Int, y: Double, z: String) = s"Implicit simulation: $x, $y, $z"
    }
  }

  object ParametricSimulation extends App {
    object CompanionSimulation {
      println(implicitly[AbstractSimulation[Int, Double, String]].simulate(1, 2, "three"))
    }

    object TraitSimulation extends Implicit {
      println(implicitly[AbstractSimulation[Int, Double, String]].simulate(10, 20, "thirty"))
    }

    CompanionSimulation
    TraitSimulation
  }
}
```

Solution

You can run `ParametricSimulation` like this:

```
sbt "run-main ParametricSimulation"
```

Output is:

```
Companion simulation: 1, 2.0, three
Implicit simulation: 10, 20.0, thirty
```

Example: Ternary Operator DSL

Let's use implicit conversions to define a DSL that provides a ternary operator similar to the ? operator in C and C++. We want to be able to write using the following syntax:

```
(4*4 > 14) ? "Yes" | "No"
```

If we rewrite the expression using dot notation it will be easier to understand how to define the ternary operator for Scala:

```
(4*4 > 14).?("Yes").|("No")
```

We can see that we will need to define a method called ? that can be invoked from the predicate, and a method called | that can be invoked from the then clause. Let's define a class called `TernaryThen` that holds the predicate, and have it define a method called ? that accepts the then clause. To trigger the invocation we will need an implicit conversion from a `Boolean` (the predicate) to `TernaryThen`. The `TernaryThen` class will also need to define an inner class, which we will call `TernaryEval`, to handle the else clause and to evaluate the entire expression. All of the parameters should be defined as being call by name, so they are not evaluated until their value is required. This means that neither the `TernaryEval` class nor the `TernaryThen` class can be defined as value objects.

```
class TernaryThen(predicate: => Boolean) {
  def ?[A](thenClause: => A) = new TernaryEval(thenClause)

  class TernaryEval[A](thenClause: => A) {
    def |(elseClause: => A): A = if (predicate) thenClause else elseClause
  }
}

implicit def toTernaryThen(predicate: => Boolean) = new TernaryThen(predicate)
```

The above code shows an implicit converter called `toTernaryThen` that converts from `Boolean` to `TernaryThen`. `TernaryThen` defines the ? method, which accepts the then clause and chains to an inner class called `TernaryEval` that defines the | method, which returns the result of evaluating the appropriate clause.

Let's try it:

```
scala> (4*4 > 14) ? "Yes" | "No"
res6: String = Yes
```

Here is an implementation that uses anonymous classes:

```
implicit def boolToOperator(predicate: Boolean) = new {
  def ?[A](trueClause: => A) = new {
    def |(falseClause: => A) = if (predicate) trueClause else falseClause
  }
}
```

Let's try it:

```
scala> (4*4 > 14) ? "Yes" | "No"
res8: String = Yes
```

The anonymous class implementation defines two new classes each time the Scala compiler resolves the implicit conversion. It is therefore better to use the `TernaryThen` implementation.

Parametric Traits as Rich Interfaces

This pattern is useful to enhance existing Scala and Java classes.

```
trait RichIterable[A] {
  def iterator: java.util.Iterator[A]

  def foreach(f: A => Unit): Unit = {
    val iter = iterator
    while (iter.hasNext) f(iter.next)
  }

  def foldLeft(seed: A)(f: (A, A) => A): A = {
    var result = seed
    foreach { item =>
      result = f(result, item)
    }
    result
  }
}
```

Let's use `RichIterable` to add Scala functionality to a Java runtime class:

```
scala> val richSet = new java.util.HashSet[Int] with RichIterable[Int]
richSet: java.util.HashSet[Int] with RichIterable[Int] = []

scala> richSet.add(1)
res13: Boolean = true

scala> richSet.add(2)
res14: Boolean = true

scala> richSet.foldLeft(0)(_ + _)
res15: Int = 3

scala> s"richSet = ${richSet.toArray.mkString(", ")}; total = $total"
richSet = 1, 2; total = 6
```

To run this program, type:

```
sbt "runMain RichInterface"
```

2-3 Introduction to Collections

`ScalaCourses.com` / `scalaIntermediate` / `ScalaCore` / `scalaTypesCollections` / `scalaCollectionIntro`

The sample code shown in this lecture is provided in `CollectionFun.scala`.

Array

Arrays in Scala are pretty much what you might expect: an ordered collection of items of similar type, accessible via an index. Scala arrays are implemented as a thin wrapper around Java `Arrays`, which means that they are mutable. It is easy to create a Scala array; in the following example the type of the array elements is detected by the Scala compiler as `Int`:

Caution: because arrays are mutable, they are not threadsafe

```
scala> val a1 = Array(1, 2, 3)
a1: Array[ Int ] = Array(1, 2, 3)
```

The REPL shows us that `a1` has type `Array[Int]`.

This is the syntax to create an empty array of type `Int`:

```
scala> val a2 = Array.empty[Int]
a2: Array[Int] = Array()
```

You can explicitly declare the type of an array variable:

```
scala> val a3:  Array[Int]  = Array(1, 2, 3)
a3: Array[Int] = Array(1, 2, 3)
```

You can modify an array:

```
scala> a1(0) = a1(0) + 1

scala> a1
res0: Array[Int] = Array(2, 2, 3)
```

The Scala compiler *desugars* this expression:

```
a1(0) = a1(0) + 1
```

...into:

```
a1.update(0, a1.apply(0) + 1)
```

The `update` method changes an element of a Scala array.

Scala provides predefined implicit conversions to enrich arrays with extra capabilities. The Array ScalaDoc Type Hierarchy diagram shows that the `ArrayOps` implicit class enriches `Array`; it also shows that Scala predefines an implicit converter from `Array` to Java's `CharSequence`. You can convert an `Array` to another Scala collection type

by using the appropriate conversion method, such as `toList`, `toSeq`, `toBuffer`, etc. We will learn about Scala collection types over the next several lectures.

Traversable

All collections can be traversed because they extend `Traversable`. This trait defines higher order functions such as `foreach`, `map`, `flatMap`, `filter`, `exists` and `forall`. We have already seen some of these in action, and we will cover all of them and more in this lecture and lectures that follow. You should get to know the `Traversable` methods, because all collection classes implement them. Unless you define your own collection types you won't define subclasses of `Traversable`.

Iterator

`Iterator` extends `Traversable` and allows iteration over the elements of a collection. The order of iteration is not defined. `Iterators` can only be traversed once, so you might want to get in the habit of transforming an `Iterator` to a `Vector` or a `List` by calling `toVector` or `toList` if you think might ever need to traverse it again. You can call `mkString` on an iterator, which will call `toString` on each item in the collection, then concatenate the results with an optional separator. For example, we have already seen how we can obtain an `Iterator` of all of the lines in a source file like this:

```
scala> val iterator = io.Source.fromFile("/etc/passwd")
iterator: scala.io.BufferedSource = non-empty iterator

scala> iterator.mkString
res1: String =
"root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
syslog:x:101:103::/...
```

You can specify the separator between `Iterator` items, like this:

```
scala> io.Source.fromFile("/etc/passwd").mkString(", ")
res2: String =
"r, o, o, t, :, x, :, 0, :, 0, :, r, o, o, t, :, /, r, o, o, t, :, /, b, i, n, /, b, a, s, h,
, d, a, e, m, o, n, :, x, :, 1, :, 1, :, d, a, e, m, o, n, :, /, u, s, r, /, s, b, i, n, :, /, b, i, n, /, s, h...
```

You can also specify a prefix and a suffix (the REPL cut off the suffix):

```
scala> io.Source.fromFile("/etc/passwd").mkString(">>>", ", ", "<<<")
re3: String =
>>>r, o, o, t, :, x, :, 0, :, 0, :, r, o, o, t, :, /, r, o, o, t, :, /, b, i, n, /, b, a, s, h,
, d, a, e, m, o, n, :, x, :, 1, :, 1, :, d, a, e, m, o, n, :, /, u, s, r, /, s, b, i, n, :, /, b, i, n, /, s, h, ...
```

Although Scala's `Iterator` is similar to Java's `Iterator` in that it provides the `hasNext` and `next` methods, you do not normally need to invoke those methods because the underlying `Traversable` class provide nicer ways to walk through `Iterator` instances.

Collection Types

Collections may be strict or lazy. Lazy collections have elements that do not consume memory or require computation until they are accessed, such as `Ranges`, introduced in the preceding Scala Introduction course and `Streams`, discussed in the next lecture. Collection traits such as `Seq`, `Map` and `Set` define common behavior for

concrete implementations. Both the `mutable` and `immutable` collection packages (`collection.mutable` and `collection.immutable`) have implementations of collection traits. You can profitably spend a lot of time studying the collection classes defined in those packages.

You must specify whether you want a mutable or an immutable version when working with a collection type. If your program just uses immutable data structures, then put this at the top of every `.scala` file that references collections:

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

Similarly, if your program just use mutable data structures, put this at the top of every `.scala` file that references collections:

```
import collection.mutable._
```

It is common to use both mutable and immutable data structures in the same program. For this scenario, use the following import and qualify the collection type you need. This course generally adopts that writing style.

```
import collection._
```

The next lecture focuses on immutable collections. A lecture dedicated to mutable collections follows. Many of the capabilities of immutable collections are also implemented by mutable collections. I will point this out in the upcoming lectures.

HashMap

Hash tables (sets and maps) are very fast so long as the objects placed in them have a good distribution of hash codes.

`HashMap` is very similar to its Java counterpart, with the Scala `Iterator` behavior mixed in, and is provided in mutable and immutable versions which implement the behavior defined in the `Map` trait. `HashMap` is parametric in two types: the key type and the value type. For example, the following defines an empty immutable `HashMap` with `Int` keys and `String` values:

```
scala> val immutableMap = immutable.HashMap.empty[Int, String]
immutableMap: scala.collection.immutable.HashMap[Int,String] = Map()
```

If you define a `HashMap` and provide key/value pairs as tuples then the Scala compiler will figure out the types:

```
scala> val mutableMap = mutable.HashMap( 1 -> "eh", 2 -> "bee", 3 -> "sea" )
mutableMap: scala.collection.mutable.HashMap [Int,String]  = Map(1 -> eh, 2 -> bee, 3 -> sea)
```

Map

A `Map` is an unordered collection of key-value pairs, which can be specified as tuples of arity 2. You can use this fact to construct a `Map` by passing in a sequence of tuples of arity 2.

`Map` is a collection trait from which concrete implementation classes such as `HashMap` are defined. You can of course define a variable or parameter to be of type `Map`:

```
scala> val map:  Map[Int, String]  = immutable.HashMap( 1 -> "eh", 2 -> "bee", 3 -> "sea" )
map: scala.collection.Map[Int,String] = Map(1 -> eh, 2 -> bee, 3 -> sea)
```

Notice the difference between `Map.get` and the default method `Map.apply`:

```
scala> map .get(1)
res27: Option[String] =  Some(eh)

scala> map (1)
res28: String = eh

scala> map. get(0 )
res29: Option[String] =  None

scala> map(0)
java.util.NoSuchElementException: key not found: 0
```

Depending on your application, you might want to provide a default value, which will be returned if a key is not found. Just for fun, I've created a mutable map :

```
scala> val map =  mutable .HashMap( 1 -> "eh", 2 -> "bee", 3 -> "sea") .withDefaultValue("eh")
map: scala.collection.mutable.Map[Int,String] = Map(1 -> eh, 2 -> bee, 3 -> sea)

scala> map(0)
res30: String = eh
```

From the previous course, recall that `Nothing` is a subtype of all other types. That is why the compiler widens the types if they are not specified when creating an empty `HashMap`. To mix things up this `Map` is immutable :

```
scala> val emptyMap =  immutable .HashMap.empty
emptyMap: scala.collection.mutable.Map[Nothing,Nothing] = Map()
```

You can create a new `Map` that contains the first `Map`, with a new tuple appended:

```
scala> map + (3 -> "c")
res31: scala.collection.immutable.Map[Int,String] = Map(1 -> eh, 2 -> bee, 3 -> c)
```

Of course, you can append multiple tuples at once:

```
scala> val map2 = map + (3 -> "c", 4 -> "d")
map32: scala.collection.immutable.Map[Int,String] = Map(2 -> bee, 4 -> d, 1 -> eh, 3 -> c)
```

Maps can be combined:

```
scala> val map3 = map ++ map2
map33: scala.collection.immutable.Map[Int,String] =  Map(2 -> bee, 4 -> d, 1 -> eh, 3 -> c)
```

You can obtain all of the keys from a map:

```
scala> map3.keys
res34: Iterable[Int] =  Set(2, 4, 1, 3)
```

You can update a specific key. For an immutable map, this means that a modified copy of the original map is returned:

```
scala> map3.updated(1, "q")
res35: scala.collection.immutable.Map[Int,String] =  Map(2 -> bee, 4 -> d, 1 -> q, 3 -> c)
```

You can test to see if a key exists for a map:

```
scala> map3.isDefinedAt(42)
res36: Boolean = false

scala> map3.isDefinedAt(2)
res37: Boolean = true
```

`Map.mapValues` transforms a `Map` by applying a function to every value in the collection.

```
scala> val map = Map(1 -> "apricot", 2 -> "banana", 3 -> "clementine", 4 -> "durian", 5 -> "fig", 6 -> "guava", 7 -> "jackf
ruit", 8 -> "kiwi", 9 -> "lime", 10 -> "mango")
map: scala.collection.immutable.Map[Int,java.lang.String] = Map(5 -> fig, 10 -> mango, 1 -> apricot, 6 -> guava, 9 -> lime,
 2 -> banana, 7 -> jackfruit, 3 -> clementine, 8 -> kiwi, 4 -> durian)

scala> map.mapValues(_ capitalize)
res10: scala.collection.immutable.Map[Int,String] = Map(5 -> Fig, 10 -> Mango, 1 -> Apricot, 6 -> Guava, 9 -> Lime, 2 -> Ba
nana, 7 -> Jackfruit, 3 -> Clementine, 8 -> Kiwi, 4 -> Durian)
```

There are lots more methods defined for `Map`. The lecture on combinators has a section on `Map`.

HashSet and LinkedHashSet

Both `HashSet` and `LinkedHashSet` are like `List` without duplicates. There is a mutable and an immutable version of `HashSet`, however `LinkedHashSet` is only available as a mutable collection. `LinkedHashSet` maintains FIFO ordering, while `HashSet` is unordered.

```
scala> immutable.HashSet(1, 1, 2)
res38: scala.collection.immutable.HashSet[Int] = HashSet(1, 2)

scala> mutable.HashSet(1, 1, 2)
res39: scala.collection.mutable.HashSet[Int] = Set(1, 2)

scala> mutable.LinkedHashSet(4, 4, 4)
res40: scala.collection.mutable.LinkedHashSet[Int] = Set(4)
```

Adding an element to a mutable hash table (sets and maps) can normally be done in constant time.

```
scala> val set = mutable.HashSet.empty[String]
map: scala.collection.mutable.HashSet[String] = Set()

scala> set += "Buy a dog"
res5: set.type = Set(Buy a dog)

scala> set += "Sell the cat"
res6: set.type = Set(Sell the cat, Buy a dog)
```

Set

Set is the trait that defines the interface for all set implementations like `HashSet`, `ListSet` and `LinkedHashSet`. You might want to declare method parameters and abstract members as being of type `Set`.

```
scala> def doSomething(set: Set[Int]): Unit = println(set.mkString(", "))
doSomething: (set: scala.collection.Set[Int])Unit

scala> doSomething(mutable.HashSet(1, 2, 3))
1, 2, 3

scala> doSomething(mutable.LinkedHashSet(1, 2, 3))
1, 2, 3
```

Type Widening

Collection items are subject to type widening. If you put an `Int` and a `Double` into a collection, the `Int`s will be converted to `Double`s and you will get a collection of `Double`, unless you explicitly tell the compiler you want the common supertype of all Java numbers, `Number`. There are several ways you can express your desires:

```
scala> immutable.HashSet(1.0, 2)
res40: scala.collection.immutable.HashSet[Double] = HashSet(1.0, 2.0)

scala> val set: Set[Number] = immutable.HashSet(1.0, 2)
set: Set[Number] = Set(1.0, 2)

scala> val set = immutable.HashSet[Number](1.0, 2)
set: scala.collection.immutable.HashSet[Number] = HashSet(1.0, 2)
```

2-4 Immutable Collections

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaTypesCollections / scalaCollectionImmutable

The sample code for this lecture is provided in CollectionImmutableFun.scala.

List

Scala's List is an immutable linked list, which means that List is an ordered collection. List is designed for linear (non-random) access. You can easily make a new List:

```
scala> val list1 = List(1, 2, 3)
list1: List[Int] = List(1, 2, 3)
```

Nil refers to the empty list. You can use the cons operator (::) or the prepend operator(+:) to prepend items to a list and thereby create a new list. Both of these operators are equivalent.

```
scala> val list2 = 4 :: 5 :: 6 :: Nil
list2: List[Int] = List(4, 5, 6)

scala> 7 +: 8 +: 9 +: Nil
res0: List[Int] = List(7, 8, 9)
```

You can concatenate two lists with the ::: and ++ combinators. They both do the same thing:

```
scala> list1 ::: list2
res1: List[Int] = List(1, 2, 3, 4, 5, 6)

scala> list1 ++ list2
res2: List[Int] = List(1, 2, 3, 4, 5, 6)
```

You can have a List that contains any type of object. For example, let's say we have a Thing case class:

```
scala> case class Thing(i: Int, s: String)
defined class Thing
```

We could make a collection of Thing:

```
scala> val thing1 = Thing(1, "z")
thing1: Thing = Thing(1,a)

scala> val thing2 = Thing(2, "y")
thing2: Thing = Thing(2,b)

scala> val thing3 = Thing(3, "x")
thing3: Thing = Thing(3,c)

scala> val things = List(thing3, thing1, thing2)
things: List[Thing] = List(Thing(3,x),Thing(1,z),Thing(2,y))
```

Like we saw for Array, there are several ways to create empty collections like List. The compiler needs to know the type of item contained in the collection, and there are several ways you can do that:

```
scala> List.empty[Double]
res3: List[Double] = Seq()

scala> val emptyList: List[Double] = List.empty
emptyList: List[Double] = List()

scala> val emptyList: List[Double] = List()
emptyList: List[Double] = List()
```

Right-Associative Operators

You may have noticed that the `::` and `+:` operators act unusually, in that the item to be prepended precedes the `List` instance. This is possible because Scala treats all operators that end in a colon character as right-associative when written using infix notation. In other words, this expression:

```
scala> 5 :: Nil
res4: List[Int] = List(5)
```

is equivalent to:

```
scala> Nil.::(5)
res5: List[Int] = List(5)
```

Also these expressions are equivalent:

```
scala> 5 +: Nil
res6: List[Int] = List(5)

scala> Nil.+:(5)
res7: List[Int] = List(5)
```

Vector

The `immutable.Vector` class is also an immutable ordered collection derived that provides a compromise between indexed and linear access. It has both effectively constant time indexing overhead and constant time linear access overhead. Because if this, `Vector` is a good foundation for mixed access patterns where both indexed and linear accesses are used. The `::` and `:::` `List` operations are not supported by `Vector`, but you can use the `+:` and `++:` equivalents.

```
scala> val vector1 = Vector(1, 2, 3)
res8: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)

scala> 0 +: vector1
res9: scala.collection.immutable.Vector[Int] = Vector(0, 1, 2, 3)

scala> Vector(thing1, thing2, thing3)
res10: scala.collection.immutable.Vector[Thing] = Vector(Thing(1,z), Thing(2,y), Thing(3,z))
```

If you cannot decide if you should use `List` or `Vector`, use `Vector` if possible.

```
scala> val vector2 = Vector(4, 5, 6)
res11: scala.collection.immutable.Vector[Int] = Vector(4, 5, 6)

scala> vector1 ++ vector2
res12: Vector[Int] = Vector(1, 2, 3, 4, 5, 6)
```

Seq

`Seq` is the base trait for ordered sequences and lists such as `List` and `Vector`. Unlike the `Iterator` trait, the Seq trait enables collections with a definate order. `Seq` is a generalization of sequences, so you should use `Seq` to define interfaces and use `List` or `Vector` for defining variables, like this:

```
scala> def doSomething(seq: Seq[Int]) = seq.foreach(println)
doSomething: (seq: Seq[Int])Unit

scala> doSomething(List(1, 2, 3))
1
2
3

scala> doSomething(Vector(4, 5, 6))
4
5
6
```

In other words, Scala's `Seq` is similar to Java's `List` interface, and Scala's `List` is similar to an immutable version of Java's `LinkedList` concrete class.

BTW, the `Seq` constructor actually returns a `List`, which implements the `Seq` trait. It is not good practice to use this constructor, even though many Scala programmers write code like this:

```
scala> val seq = Seq(1, 2, 3)
seq: Seq[Int] = List(1, 2, 3)
```

Sorting on Demand

You can sort on demand using one or more properties of an ordered collection's elements. Of course, for this to work all elements in the collection must have the required properties. This means that the declared type of the items in the collection will define sortable properties, and you would not sort collection types such as `HashMap` or `HashSet`:

```
scala> things.sortBy(_.i)
res13: List[Thing] = List(Thing(1,z), Thing(2,y), Thing(3,x))

scala> things.sortBy(_.s)
res14: List[Thing] = List(Thing(3,x), Thing(2,y), Thing(1,z))
```

In order to sort by more than one property, specify them as a tuple. The order of the sort elements defines the sort key order.

```
scala> things.sortBy(t => (t.i, t.s))
res15: List[Thing] = List(Thing(1,z), Thing(2,y), Thing(3,x))
```

You will often want to define a natural ordering for your collection. We will discuss that topic in the lecture on Sorting and Ordered Collections.

You can sort tuples in a similar manner:

```
scala> val tuples = Vector((4, "z"), (2, "q"), (5, "b"))
tuples: scala.collection.immutable.Vector[(Int, String)] = List((4,z), (2,q), (5,b))

scala> tuples.sortBy(_._1)
res16: scala.collection.immutable.Vector[(Int, String)] = Vector((2,q), (4,z), (5,b))

scala> tuples.sortBy(_._2)
res17: scala.collection.immutable.Vector[(Int, String)] = Vector((5,b), (2,q), (4,z))

scala> tuples.sortBy(t => (t._1, t._2))
res18: scala.collection.immutable.Vector[(Int, String)] = Vector((2,q), (4,z), (5,b))
```

Head, Tail and the Remainder of a Collection

`head`, `headOption`, `init`, `tail`, `last` and `lastOption` all work together. `head` returns the first item in a collection, or throws an exception if the collection is empty. These combinators are defined in `IterableLike`, which means there are available throughout the Scala collection classes. They can be applied to ordered and unordered collections, and also to mutable and immutable collections.

```
scala> val seq: Seq[Int] = Vector(1, 2, 3)
seq: Seq[Int] = Vector(1, 2, 3)

scala> seq.head
res19: Int = 1

scala> Vector.empty.head
java.util.NoSuchElementException: head of empty list

scala> Nil.head
java.util.NoSuchElementException: head of empty list
```

`headOption` returns an `Option` (`Some` or `None`) and never throws an exception:

```
scala> seq.headOption
res20: Option[Int] = Some(1)

scala> Vector.empty.headOption
res21: Option[Nothing] = None

scala> Nil.headOption
res22: Option[Nothing] = None
```

`tail` returns the remainder of the collection after the head is removed. If the collection is empty an exception is thrown.

```
scala> seq.tail
res23: Seq[Int] = Vector(2, 3)

scala> Nil.tail
java.lang.UnsupportedOperationException: tail of empty list
```

`last` returns the last element of the collection, or throws an exception if the collection is empty

```
scala> seq.last
res24: Int = 3

scala> Nil.last
java.util.NoSuchElementException
```

`lastOption` returns an `Option` and never throws an exception:

```
scala> seq.lastOption
res25: Option[Int] = Some(3)

scala> Nil.lastOption
res26: Option[Nothing] = None
```

`init` returns all but the last element, or throws an exception if the collection is empty. If there is only one element in the collection, the empty collection is returned.

```
scala> seq.init
res27: Seq[Int] = Vector(1, 2)

scala> Vector(1).init
res28: Vector[Int] = List()

scala> Nil.init
java.lang.UnsupportedOperationException: empty.init
```

`take` returns up to the specified elements from the beginning of the collection. `drop` returns the remainder of the collection after the specified number of elements have been removed. If you try to `take` or `drop` more elements from the collection than actually exist, only the number of elements are processed an no exception is raised.

```
scala> seq.take(2)
res29: Seq[Int] = Vector(1, 2)

scala> seq.take(0)
res30: Seq[Int] = Vector()

scala> seq.take(4)
res31: Seq[Int] = Vector(1, 2, 3)

scala> seq.drop(2)
res32: Seq[Int] = Vector(3)

scala> seq.drop(0)
res33: Seq[Int] = Vector(1, 2, 3)

scala> seq.drop(-4)
res34: Seq[Int] = Vector(1, 2, 3)

scala> seq.drop(4)
res35: Seq[Int] = Vector()
```

`takeWhile` is a version of `take` that accepts a predicate instead of a constant value. We can use `takeWhile` in conjunction with `Iterator.continually` to obtain unique values until a user indicates they have enough.

```
scala> Iterator.continually(System.nanoTime).takeWhile(_ => !Console.readLine("\nMore? <Y/n>: ").toLowerCase.startsWith("n")).foreach(println)
More? <Y/n>: 23773991672470
More? <Y/n>: 23775269319438
More? <Y/n>:
```

Challenge: Compute SHA of a file

Git uses SHA-1 to obtain the digest of each file that it checks in. SHA-1 is not secure but is inexpensive to compute. Write a console application that reports the SHA-1 of `/etc/passwd`. We are not going to write this application for maximum efficiency; instead, we are going to use it as practice for processing a `Stream` of data. Even though the source of the data is a file, consider how flexible this would be

Hints

- You can create a buffer to hold the accumulated result by calling `new Array[Byte](1024)`.
- You can get the SHA-1 message digest algorithm by calling `java.security.MessageDigest.getInstance("SHA-1")`
- `java.io.FileInputStream.read` returns the number of bytes read, or -1 to signal no more data.
- Use `Iterator.continually` and `takeWhile` to read successive byte arrays from an `InputFileStream` until `FileInputStream.read` signals no more data.
- `foreach` iteration use `MessageDigest.update` to replace the data in the buffer with the current result. The third argument must be set to the number of bytes read for this iteration.

Output should be something like:

```
SHA-1 of /etc/passwd is 2112088146319-7-11555-1126870-43-105-63-21-5028-43-98
```

Solution

```
package solutions

import java.security.MessageDigest
import java.io.FileInputStream

object SHA1 extends App {
  def digest(fileName: String, algorithm: String="SHA-256"): Array[Byte] = {
    val md = MessageDigest.getInstance(algorithm)
    val input = new FileInputStream(fileName)
    val buffer = new Array[Byte](1024)
    Iterator.continually { input.read(buffer) }
      .takeWhile(_ != -1)
      .foreach { md.update(buffer, 0, _) }
    md.digest
  }

  println(s"""SHA-1 of /etc/passwd is ${digest("/etc/passwd", "SHA-1").mkString}""")
}
```

To run this solution, type:

```
sbt "runMain solutions.SHA1"
```

Streams

Scala streams are unrelated to Java streams. Instead, Scala `Streams` are subtypes of `Seq`. A `collection.immutable.Stream[T]` is a *lazy* collection whose elements of type `T` are obtained only upon request. In contrast, all other Scala collections are *strict* – that is, they are fully computed. BTW, the `view` method converts a strict collection to a lazy collection. `Predef` imports `collection.immutable.Stream` so it is always available.

Usages for `Stream` are for infinite sequences, or for sequences that are expensive to compute. `Stream` memoizes elements as they are computed.

`Stream.continually` creates a Scala `Stream`, which can be considered as a lazy `Seq` containing elements are only evaluated when they are needed. For example, here is how one might compute a `Stream` of unique `Longs`:

```
scala> val ids = Stream.continually(System.nanoTime)
ids: scala.collection.immutable.Stream[Long] = Stream(86653342711375, ?)
```

We can examine the first 5 elements of the stream this way. Note that `toVector` forces evaluation:

```
scala> ids.take(5).toVector
res36: Vector[Long] = Vector(86549694623903, 86592186252578, 86592186259051, 86592186261332, 86592186263348)
```

This demonstrates that `Stream` memoizes values that have been generated:

```
scala> ids.head
res37: Long = 72091815908327
```

`Stream.continually` can be used to write a similar program, but should only be used if the previous values need to be accessed as the program executes:

```
scala> Stream.continually(System.nanoTime).takeWhile(_ => !Console.readLine("\nMore? <Y/n>: ").toLowerCase.startsWith("n"))
.foreach(println)
More? <Y/n>: 24282594984003
More? <Y/n>: 24283649919676
More? <Y/n>:
```

Streaming Binary IO

Here is a more efficient way to read in a binary file into a byte array, than the sample code shown in the the earlier lecture on higher-order functions. This version does not do pointless character encoding.

```
import sys.process._
val fileName = ("which scalac" !!).trim
val bis = new java.io.BufferedInputStream(new java.io.FileInputStream(fileName))
val bArray = Stream.continually(bis.read).takeWhile(-1 !=).map(_.toByte).toArray
bis.close()
```

Several variations of this code are provided in `HigherFun.scala`. You can run this program by typing:

```
sbt "runMain BinaryIO"
```

Here is an explanation:

a. `BufferedInputStream.read` returns the next byte of data, or -1 if the end of the input stream is reached.
b. `Stream.continually` creates a Scala `Stream`, which is a lazy `Seq` where elements are only evaluated when they are needed. `Stream` memoizes elements as they are computed.
c. `takeWhile` returns the beginning elements of a `Seq` that satisfy a predicate.
d. `Stream.continually(bis.read).takeWhile(_.!=(-1))` returns a `Stream[Int]`
e. `Stream.continually(bis.read).takeWhile(-1 !=).map(_.toByte)` returns a `Stream[Byte]`
f. The functor `-1 !=` is extremely terse because shorthand is combined with infix notation. It is equivalent to each of these incantations:
    i. `item => item != -1`
    ii. `item => item.!=(-1)`
    iii. `_.!=(-1)`
    iv. `_ != -1` This is my preferred style because it is reasonably terse while remaining understandable to most programmers
    v. `-1 != _`

2-5 Mutable Collections

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaTypesCollections / scalaMutableCollections

Buffer is a mutable subtrait of Seq. There are many Buffer implementations, however invoking toBuffer on Seq, List and Vector returns an ArrayBuffer. For example, let's convert an immutable List to ArrayBuffer:

```
scala> import collection._
import collection._

scala> List(1, 2, 3).toBuffer
res32: scala.collection.mutable.Buffer[Int] = ArrayBuffer(1, 2, 3)
```

A ListBuffer is a mutable List with constant time prepend and append.

```
scala> val lb = mutable.ListBuffer(1, 2, 3)
lb: scala.collection.mutable.ListBuffer[Int] = ListBuffer(1, 2, 3)
```

Method update is not defined for List because lists are immutable, however ListBuffer does define update:

```
scala> lb(0) = lb(0) * 3

scala> lb
res33: scala.collection.mutable.ListBuffer[Int] = ListBuffer(3, 2, 3)
```

Let's convert this ListBuffer to various types of immutable collections:

```
scala> lb.toArray
res34: Array[Int] = Array(3, 2, 3)

scala> lb.toSeq
res35: Seq[Int] = List(3, 2, 3)

scala> lb.toIndexedSeq
res36: scala.collection.immutable.IndexedSeq[Int] = Vector(3, 2, 3)
```

Threadsafe Mutable Collections

Immutable collections are by definition threadsafe. Mutable collections are not threadsafe by default. You can add thread safety to mutable collections by mixing in a SynchronizedXXX trait (SynchronizedBuffer, SynchronizedMap, SynchronizedPriorityQueue, SynchronizedQueue, SynchronizedSet an SynchronizedStack).

```
val mb = new mutable.ArrayBuffer[String] with mutable.SynchronizedBuffer[String]
```

As always, it is better to create a new class definition that mixes in the desired trait, instead of inadvertently creating lots of anonymous classes:

```
class SynchronizedArrayBuffer[T] extends mutable.ArrayBuffer[T] with mutable.SynchronizedBuffer[T]
val mb = new SynchronizedArrayBuffer[String]
```

HashSet, HashMap and WeakHashMap

From the Scala documentation: "A weak hash map is a special kind of hash map where the garbage collector does not follow links from the map to the keys stored in it. This means that a key and its associated value will disappear from the map if there is no other reference to that key. Weak hash maps are useful for tasks such as caching, where you want to re-use an expensive function's result if the function is called again on the same key. If keys and function results are stored in a regular hash map, the map could grow without bounds, and no key would ever become garbage. Using a weak hash map avoids this problem. As soon as a key object becomes unreachable, its entry is removed from the weak hashmap."

```
scala> mutable.WeakHashMap(1 -> "one", 33 -> "thirty-three", 4 -> "four")
res9: scala.collection.mutable.WeakHashMap[Int,Thang] = Map(4 -> four, 33 -> thirty-three, 1 -> one)
```

Concurrent Maps

Concurrent maps are implemented by the Java runtime library and are wrapped by the Scala runtime library. They are synchronized on each key, thus multiple threads can update or retrieve different keys simultaneously. You can create a new concurrent map this way:

```
scala> import java.util.concurrent.ConcurrentHashMap
import java.util.concurrent.ConcurrentHashMap

scala> import collection.JavaConverters._
import scala.collection.JavaConverters._

scala> val cmap = new ConcurrentHashMap[String, String].asScala
cmap: scala.collection.concurrent.Map[String,String] = Map()
```

The `collection.concurrent.Map` trait can also be mixed into any `Map` to make it concurrent, if you are willing to write an implementation. The default concurrent map implementation is a `TrieMap`, and the only predefined concurrent mutable `Map` is also a `TrieMap`.

```
scala> import collection.concurrent.TrieMap
import collection.concurrent.TrieMap

scala> val tm = TrieMap.empty[String, String].withDefaultValue("default")
tm: scala.collection.mutable.Map[String,String] = Map()

scala> tm.put("key1", "value1")
res10: Option[String] = None

scala> tm.put("key2", "value2")
res11: Option[String] = None

scala> tm.get("key1")
res12: Option[String] =  Some(value1)

scala> tm("key1")
res13: String =  value1

scala> tm("x")
res14: String =  default
```

Queue, Stack and ArrayStack

Scala's mutable `Queue` exhibits first-in first-out (FIFO) behavior:

```
scala> val queue = mutable.Queue.empty[String]
queue: scala.collection.mutable.Queue[String] = Queue()

scala> queue += "asdf"
res15: queue.type = Queue(asdf)

scala> queue += "qwer"
res16: queue.type = Queue(asdf, qwer)

scala> queue.dequeue()
res17: String = asdf

scala> queue
res18: scala.collection.mutable.Queue[String] = Queue(qwer)
```

Scala's normal mutable `Stack` exhibits last-in first-out (LIFO) behavior:

```
scala> val stack1 = mutable.Stack("one", "two", "three")
res19: scala.collection.mutable.Stack[String] = Stack(one, two, three)
```

`ArrayStack` is and alternative implementation of a mutable stack which is backed by an `Array` that gets re-sized as needed. It provides fast indexing and is more efficient for most operations than a normal mutable stack.

```
scala> val stack2 = mutable.ArrayStack("four", "five", "six")
res20: scala.collection.mutable.ArrayStack[String] = ArrayStack(four, five, six)
```

LinkedHashMap, LinkedHashSet, LinkedList and DoubleLinkedList

Linked lists are mutable sequences of nodes linked by references. Scala's linked lists do not throw exceptions if the next reference is not directly manipulated.

```
scala> val lhset = mutable.LinkedHashSet(1)
lhset: scala.collection.mutable.LinkedHashSet[Int] = Set(1)

scala> val lhmap = mutable.LinkedHashMap(1 -> "one")
lhmap: scala.collection.mutable.LinkedHashMap[Int,String] = Map(1 -> one)

scala> val dllist = mutable.DoubleLinkedList(1 -> "two")
dllist: scala.collection.mutable.DoubleLinkedList[(Int, String)] = DoubleLinkedList((1,two))

scala> val llist = mutable.LinkedList(1)
llist: scala.collection.mutable.LinkedList[Int] = LinkedList(1)

scala> val lhset = mutable.LinkedHashSet(1)
lhset: scala.collection.mutable.LinkedHashSet[Int] = Set(1)

scala> val lhmap = mutable.LinkedHashMap(1 -> "one")
lhmap: scala.collection.mutable.LinkedHashMap[Int,String] = Map(1 -> one)

scala> val dllist = mutable.DoubleLinkedList(1 -> "two")
dllist: scala.collection.mutable.DoubleLinkedList[(Int, String)] = DoubleLinkedList((1,two))
```

2-6 Collection Converters

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaTypesCollections / scalaCollectionConverters

The sample code for this lecture is provided in CollectionConverters.scala.

You can convert one collection type to another because the following are defined in the Traversable trait. These same methods can be used on a mutable collection to obtain an immutable version of the desired collection, and vice-versa.

```
def toArray : Array[A]
def toArray [B >: A] (implicit arg0: ClassManifest[B]) : Array[B]
def toBuffer [B >: A] : Buffer[B]
def toIndexedSeq [B >: A] : IndexedSeq[B]
def toIterable : Iterable[A]
def toIterator : Iterator[A]
def toList : List[A]
def toMap [T, U] (implicit ev: <:<[A, (T, U)]) : Map[T, U]
def toSeq : Seq[A]
def toSet [B >: A] : Set[B]
def toStream : Stream[A]
def toString () : String
def toTraversable : Traversable[A]
```

Let's convert a List to an Array:

```
scala> List(1, 2, 3).toArray
res30: Array[Int] = Array(1, 2, 3)
```

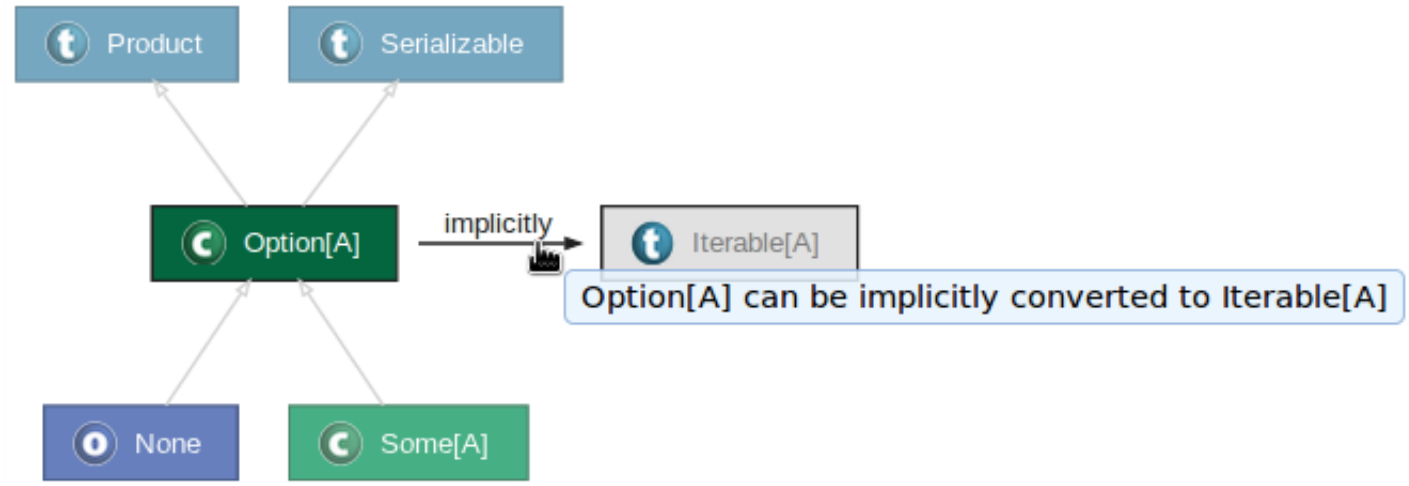
... and a Vector to a Stream:

```
scala> Vector(1, 2, 3).toStream
res0: scala.collection.immutable.Stream[Int] = Stream(1, ?)
```

Discovering the Current Implicit Conversion In Scope with Implicitly

As discussed in the lecture on implicit conversions, you can access the implicit method that converts between two types. The implicitly method can check if an implicit converter of a given type is available and return it if so.

For example, Option can be implicitly converted to Iterable, as you can see in the class diagram for Option.



We can use the `implicitly` method in the REPL to discover the method that performs the implicit conversion from `Option` to `Iterable`.

```
scala> implicitly[Option[Int] => Iterable[Int]]
res2: Some[Int] => scala.collection.Iterable[Int] = <function1>
```

We can also use the long form syntax to obtain the same implicit conversion method:

```
scala> implicitly[Function[Option[Int], Iterable[Int]]]
res1: Option[Int] => Iterable[Int] = <function1>
```

It doesn't matter if we try to convert from `Option[A]` to `Iterable[A]`, or from `Some[A]` to `Iterable[A]`. because `Some` is a subclass of `Option`.

```
scala> implicitly[Some[Int] => Iterable[Int]]
res0: Some[Int] => Iterable[Int] = <function1>
```

FYI, here is the source code for `option2Iterable`, which is the implicit conversion function that converts from `Option` to `Iterable`, as found in the source code for `Option`:

```
object Option {
  /** An implicit conversion that converts an option to an iterable value */
  implicit def option2Iterable[A](xo: Option[A]): Iterable[A] = xo.toList

  /** An Option factory which creates Some(x) if the argument is not null, and None if it is null.
   *  @param  x the value
   *  @return Some(value) if value != null, None if value == null */
  def apply[A](x: A): Option[A] = if (x == null) None else Some(x)

  /** An Option factory which returns `None` in a manner consistent with the collections hierarchy. */
  def empty[A] : Option[A] = None
}
```

Converting to and from Java Collections

`collection.JavaConverters` is useful for presenting a Java collection as a Scala collection, and vice versa. There is also a `JavaConversions` package, but it assumes that you want to wrap every collection and does not allow you to control what gets converted; this adds unnecessary overhead and so `JavaConversions` should not be used.

`JavaConverters` provides converters for commonly-used Java collections with `asScala` methods and Scala collections with `asJava` methods.

Bidirectional Converters

`JavaConverters` provides the following two-way conversions.

```
collection.Iterable <=> java.lang.Iterable
collection.Iterable <=> java.util.Collection
collection.Iterator <=> java.util.{ Iterator, Enumeration }
collection.mutable.Buffer <=> java.util.List
collection.mutable.Set <=> java.util.Set
collection.mutable.Map <=> java.util.{ Map, Dictionary }
collection.mutable.ConcurrentMap <=> java.util.concurrent.ConcurrentMap
```

If you do a round-trip conversion using these bidirectional converters you end up with the original collection. For example, we can convert from `collection.mutable.ListBuffer` to `java.util.List` and then back to `collection.mutable.ListBuffer`:

```
import collection.JavaConverters._
import collection._
val sl = mutable.ListBuffer[Int](1, 2, 3)
val jl: java.util.List[Int] = sl.asJava   // the type of j1 need not be declared, merely added for clarity
val s2: mutable.Buffer[Int] = jl.asScala  // the type of s2 need not be declared, merely added for clarity
assert(sl eq s2)
```

Here is another example:

```
val i1 = Iterable(1, 2, 3)
val i2 = i1.asJava.asScala
assert(i1 eq i2)
```

And another example:

```
val m1 = mutable.HashMap(1 -> "eh", 2 -> "bee", 3 -> "sea")
val m2 = m1.asJava.asScala
assert (m1 eq m2)
```

Unidirectional Converters

`JavaConverters` provides the following one-way conversions. Remember that `immutable.Seq`, `immutable.Set` and `immutable.Map` are imported by `Predef` as `Seq`, `Set` and `Map`.

```
collection.immutable.Seq => java.util.List
collection.mutable.Seq   => java.util.List
collection.immutable.Set => java.util.Set
collection.immutable.Map => java.util.Map
```

Exercise

`System.getenv` returns a `java.util.Map[String, String]`. Use a Scala worksheet to write a method that accepts a `String` to do a case-insensitive search of the values contained in the `Map`. The method should convert the Java `Map` to a `collection.mutable.Map[String,String]` and then use a filter to obtain a `List` of all the values that contain the given word.

Hint: you can use the placeholder syntax in a higher-order function in order to examine the values of a `Map`: `_._2`

Although `sys.env` returns a Scala `Map`, and would normally be recommended for production code, this exercise is intended to give you practice on converting a Java collection to a Scala collection, so use of `sys.env` is considered cheating for this exercise.

Solution

```
object EnvSearch {
  def showEnv(name: String): Seq[String] = {
    import collection.JavaConverters._
    System.getenv.asScala.filter(_._2.toLowerCase contains name.toLowerCase).values.toList
  }
  showEnv("java").mkString("\n")
}
```

This code is provided in `solutions/EnvSearch.sc`.

2-7 Sorting and Ordered Collections

`ScalaCourses.com` / `scalaIntermediate` / `ScalaCore` / `scalaTypesCollections` / `scalaSortCollections`

The sample code for this lecture is provided in `CollectionSorting.scala`.

We showed how higher-order functions can be written that accept one parameter earlier in this course. With collections it is useful to work with higher-order functions that accept more than one parameter. A common use case is for sorting a collection:

```
scala> List(3, 7, 5, 2).sortWith((x, y) => x < y)
res29: List[Int] = List(2, 3, 5, 7)
```

The `sortWith` higher order function compares a pair elements from the original list at a time, and repetitively sorts the intermediate results internally.

If multiple parameters are supplied by a higher-order function to a functor, multiple underscores can be used in the shorthand, and they are bound in order. For example, we can use shorthand to perform the same sort, like this:

```
scala> List(3, 7, 5, 2).sortWith(_ < _)
res9: List[Int] = List(2, 3, 5, 7)
```

Exercise - Sorting a List Using a Higher-Order Function

Use the REPL to sort `List(3, -7, 5, -2)` according to the absolute value of each of the elements.

Hint

- You can find the absolute value of a number using `math.abs`.

Solution

```
scala> List(3, -7, 5, -2).sortWith(math.abs(_)<math.abs(_))
res11: List[Int] = List(-2, 3, 5, -7)
```

To run this solution, type:

```
sbt "runMain solutions.AbsSort"
```

Only One Natural Ordering – Ordered

As we saw earlier, you can sort upon demand, however you will often want to define a natural ordering for your collection. Scala provides the `Ordered[T]` trait for classes that have only one natural ordering, or a default natural ordering. Classes that implement this trait can be sorted with `scala.util.Sorting` and can be compared with standard comparison operators like > and <. It is really easy to apply this trait: merely extend your class with the trait. Notice that the class name is a parameter for the abstract trait `Ordered`, which requires the `compare` method be implemented:

```
class Thing(val i: Int, val s: String)  extends Ordered[Thing]  {
   def compare(that: Thing) =  {
    val primaryKey = this.i - that.i
    if (primaryKey!=0) primaryKey else this.s compare that.s
  }

  override def equals(other: Any) = {
    val that = other.asInstanceOf[Thing]
    this.i == that.i && this.s==that.s
  }

  override def hashCode = super.hashCode

  override def toString = s"Thing($i, $s)"
}
```

The `Ordered` trait does not provide a default implementation of equality, so if you define compare I strongly suggest that you also redefine `equals`, which is the == method. Whenever you define equals you should also define `hashCode`.

Now let's define some instances and compare them:

The Ordered trait does not provide a default implementation of equality

```
scala> val thing1 = new Thing(1, "z")
thing1: Thing = Thing(1,a)

scala> val thing2 = new Thing(2, "y")
thing2: Thing = Thing(2,b)

scala> val thing3 = new Thing(3, "x")
thing3: Thing = Thing(3,c)

scala> thing1>thing2
res40: Boolean = false

scala> thing1<thing2
res41: Boolean = true

scala> thing1 == new Thing(1, "z")
res49: Boolean = true

scala> thing1==thing2
res42: Boolean = false

scala> thing1<=thing2
res43: Boolean = true

scala> thing1>=thing2
res44: Boolean = false
```

We can now sort the all the `Things`:

```
scala> val things = Array(thing2, thing1, thing3)
things: Array[Thing] = Array(Thing(2,y), Thing(1,z), Thing(3,x))

scala> things.sorted
res45: Array[Thing] = Array(Thing(1,z), Thing(2,y), Thing(3,x))
```

Ordering is Preferred to Ordered

Scala also provides the `Ordering` trait, for objects that need multiple sorting strategies. It is often better to use `Ordering` instead of `Ordered` because `Ordered` must be implemented by the type to compare, while with `Ordering` you can define this ordering elsewhere. To define the natural ordering, which is the default `Ordering` instance for your type, you just define an implicit ordering value in the companion object. If you want a different ordering in a certain context, define an implicit `Ordering` instance there, and because Scala's implicit resolution searches compantion objects after searching local scope, the local implicit will be used at the appropriate time.

Scala provides an implicit conversion that converts any Ordering instance into an Ordered value

```
class Thang(val i: Int, val s: String) extends Ordering[Thang] {
  def compare(a: Thang, b: Thang) =  {
    val primaryKey = a.i - b.i
    if (primaryKey!=0) primaryKey else a.s compare b.s
  }

  override def equals(other: Any) = {
    val that = other.asInstanceOf[Thang]
    this.i == that.i && this.s==that.s
  }

  override def hashCode = super.hashCode

  override def toString = s"Thang($i, $s)"
}

object Thang {
  implicit val ThangOrdering = Ordering.by { thang: Thang =>
    (thang.i, thang.s)
  }
}
```

Let's define two sort schemes:

```
scala> val orderByI = Ordering.by { thang: Thang => thang.i }
orderByI: scala.math.Ordering[Thang] = scala.math.Ordering$$anon$9@9083a57

scala> val orderByS = Ordering.by { thang: Thang => thang.s }
orderByS: scala.math.Ordering[Thang] = scala.math.Ordering$$anon$9@27daad50
```

Now we create some instances and sort them by each scheme:

```
scala> val thangs = Array(new Thang(1, "x"), new Thang (33, "b"), new Thang(4, "m"))
thangs: Array[Thang] = Array(Thang(1,x), Thang(33,b), Thang(4,m))

scala> thangs.sorted(orderByI)
res27: Array[Thang] = Array(Thang(1,x), Thang(4,m), Thang(33,b))

scala> thangs.sorted(orderByS)
res28: Array[Thang] = Array(Thang(33,b), Thang(4,m), Thang(1,x))
```

Discovering the Default Implicit Ordering with Implicitly

As we saw in the lecture on implicit conversions, the `implicitly` keyword returns the implicit value in scope for a given type. Let's use the default implicit `Ordering` (defined in the `scala package object`, so it is available to all programs) to compare two tuples:

```
scala> val ordering = implicitly[Ordering[(Int, String)]]
ordering: Ordering[(Int, String)] = scala.math.Ordering$$anon$11@12e6ca10

scala> ordering.compare( (1, "b"), (1, "a") )
res3: Int = 1

scala> ordering.compare( (1, "b"), (1, "b") )
res4: Int = 0

scala> ordering.compare( (1, "b"), (1, "c") )
res5: Int = -1
```

2-8 Combinators

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaTypesCollections / scalaCombinators

A combinator, like `map`, transforms input into output. Chaining combinators together is defines the data flow of a functional program. The data pipeline computes results in a predictable manner because combinators perform transformations that are idempotent. Scala collections define many more combinators, such as `flatMap` and `filter`. Let's look at both of these next.

Combinators have no bound variables, which is why they are idempotent

Let's transform a collection of `Int` into a new collection of `Int`, but transform the contents so they are divided by 2. We'll use a `Vector` to hold the collection of `Int`s.

```
scala> val vector = Vector(0, 1, 2, 3)
vector: Vector[Int] = Vector(0, 1, 2, 3)

scala> vector.map( _/2 )
res0: Vector[Int] = Vector(0, 0, 1, 1)
```

The `filter` in the next statement creates a new collection of the same type that only contains elements that pass the predicate; only even `Int`s are included in the resulting collection.

```
scala> vector.filter( _%2==0 )
res1: Vector[Int] = Vector(0, 2)
```

`filterNot` includes the elements that fail the predicate test:

```
scala> vector.filterNot( _%2==0 )
res2: Vector[Int] = Vector(1, 3)
```

If you wanted both lists to be generated at once, that is, a `Vector` of all items that pass the predicate and another `Vector` of all items that fail the predicate, you can use the `partition` combinator:

```
scala> vector. partition ( _%2==0 )
res3: (Vector[Int], Vector[Int]) = (Vector(0, 2),Vector(1, 3))
```

The above returned two lists, as you can see. These lists can be captured with one assignment, like this:

```
scala>  val (pass, fail) = vector.partition( _%2==0 )
pass: Vector[Int] = Vector(0, 2)
fail: Vector[Int] = Vector(1, 3)

scala> pass
res4: Vector[Int] = Vector(2)

scala> fail
res5: Vector[Int] = Vector(1, 3)
```

If you are concerned that readers of your code might be unclear as to the type of the assigned variables, you can explicitly declare their types:

```
scala> val (pass : Vector[Int] , fail : Vector[Int] ) = vector.partition( _%2==0 )
pass: Vector[Int] = Vector(0, 2)
fail: Vector[Int] = Vector(1, 3)
```

You can prune out duplicate items from any collection that mixes in `Seq` such as `Vector` and `List` by using the `distinct` method:

```
scala> Vector(1, 2, 2, 3).distinct
res25: Vector[Int] = Vector(1, 2, 3)
```

`exists`, `find` and `forall` are similar. `exists` returns `true` if a predicate is true for at least one element in a collection. `find` returns `Some`(first element that fulfills a predicate), or `None`. `forall` only returns true if a predicate is true for all members of a vector.

```
scala> vector.exists(_%2==0)
res8: Boolean = true

scala> vector.find(_%2==0)
res9: Option[Int] = Some(1)

scala> vector.forall(_%2==0)
res10: Boolean = false
```

Map

You can temporarily filter tuples from mutable and immutable `Maps`. The following two syntaxes are equivalent - they both return even keys:

```
scala> val map = Map(1 -> "eh", 2 -> "bee", 3 -> "sea", 4 -> "d")
map11: scala.collection.immutable.Map[Int,String] = Map(1-> "eh", 2 -> bee, 3 -> sea, 4 -> d)

scala> map.filter(_._1%2==0)
res12: scala.collection.immutable.Map[Int,String] = Map(2 -> b, 4 -> d)
```

`groupBy` is a powerful `Map` method. Here we partition a `Map` into a smaller map that contains only even keys, and another `Map` containing only odd keys. The predicate passed into `groupBy` determines how each tuple in the original map is assigned and transformed into the resulting maps.

```
scala> val group = map.groupBy(_._1%2==0)
group: scala.collection.immutable.Map[Boolean,scala.collection.immutable.Map[Int,String]] = Map(false -> Map(1 -> eh, 3 ->
sea), true -> Map(2 -> bee, 4 -> d))

scala> group(true)
res13: scala.collection.immutable.Map[Int,String] = Map(2 -> b, 4 -> d)

scala> group(false)
res14: scala.collection.immutable.Map[Int,String] = Map(1 -> eh, 3 -> sea)
```

Here is a better way of getting the tuples with even and odd keys:

```
scala> val (even, odd) = map.partition(_._1%2==0)
even: scala.collection.immutable.Map[Int,String] = Map(2 -> bee, 4 -> d)
odd: scala.collection.immutable.Map[Int,String] = Map(1 -> eh, 3 -> sea)
```

Chaining Combinators

2-9 map, flatMap and For-Comprehensions

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaTypesCollections / scalaFor

The code for this lecture is provided in MonadicFun.scala.

map

map transforms the contents of a container/collection but preserves the type of the collection. For example, you can convert a List[Int] to a List[String] like this:

```
scala> Vector(1, 2, 3).map(x => x.toString)
res0: Vector[String] = Vector(1, 2, 3)
```

Map accepts a function as a parameter, often referred to as a *functor.* The functor used above is a function that accepts an Int and returns a String; it has the following signature Int => String and the implementation is (x) => x.toString. The shorthand for higher-order functions means that you could rewrite the above as:

```
scala> Vector(1, 2, 3).map(_.toString)
res1: Vector[String] = Vector(1, 2, 3)
```

flatten

flatten removes the wrappings around a sequence of wrapped items and returns an new sequence with just the unwrapped values. Here is an example - given a sequence of Option, flatten will just return the elements with values (only the values wrapped by Somes, not the Nones ;flatten ignores sequence elements which are Nil, None or null.

```
scala> Vector(Some(1), None, Some(3), Some(4)).flatten
res2: Vector[Int] = Vector(1, 3, 4)
```

flatMap

flatMap also flattens a sequence, but in addition flatMap accepts a functor which is applied to each sequence item, thereby transforming the sequence while flattening it. Note that the wrapped items (Some(1), Some(3) and Some(4)), are passed into the functor, and the unwrapping is done on the functor's returned value. This means that the None items would also be passed in if they had not been filtered out. Here we filter out the undesirable None values, so the functor's logic is simplified:

```
scala> Vector(Some(1), None, Some(3), Some(4)). filter (_.isDefined). flatMap (x => Some(x.get*2))
res3: Vector[Int] = Vector(2, 6, 8)
```

Compare the result of flatMap with that of map. As you can see, flatMap 'unwraps' each element of the collection. Some(2) becomes 2.

```
scala> Vector(Some(1), None, Some(3), Some(4)).filter(_.isDefined). map (x => Some(x.get*2))
res4: Vector[Some[Int]] = Vector(Some(2), Some(6), Some(8))
```

If the filter were not present, an error would occur when the functor tried to unwrap the None:

```
scala> Vector(Some(1), None, Some(3), Some(4)).flatMap(x => Some(x.get*2))
java.util.NoSuchElementException: None.get
    at scala.None$.get(Option.scala:313)
```

We can **not** write the flatMap's lambda function using shorthand, because the underscore is not used at the outermost context of the functor. Used in this context, the underscore tries to match against a *partially applied function*, discussed in an upcoming lecture.

```
scala> Vector(Some(1), None, Some(3), Some(4)).filter(_.isDefined).flatMap( Some(_.get*2) )
<console>:8: error: missing parameter type for expanded function ((x$2) => x$2.get.$times(2))
              Vector(Some(1), None, Some(3), Some(4)).filter(_.isDefined).flatMap(Some(_.get*2))
```

For Loops

Scala's for keyword fulfils several roles. There are subtle differences in how each flavor of `for` is written, and there are huge differences in how they work. Let's start off slowly, by printing out something 3 times:

```
scala> for ( i <- 1 to 3 ) println("Hello, world!")
Hello, world!
Hello, world!
Hello, world!
```

We saw `Ranges` in one of the first lectures of this course. In this case, the `Range 1 to 3` is called a generator, because it is written to the right of the `<-` symbol. The Scala compiler causes a temporary variable to be instantiated for each value that the generator emits, and then the body of the `for` expression is executed. The body can consist of a code block, wrapped in parenthesis:

```
import java.util.Calendar

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

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

for ( i <- 1 to 3 ) {
  println(s"Only $secsUntilXmas seconds until Christmas!")
  Thread.sleep(5000)
}
```

Outputs:

```
Only 10544398 seconds until Christmas!
Only 10544393 seconds until Christmas!
Only 10544388 seconds until Christmas!
```

Exercise

In the above program, how would you rewrite it so it works exactly the same, but by using another iteration mechanism instead of a `for` statement?

Solution

```
package solutions

import java.util.Calendar

object Loopy extends App {
  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

  1 to 3 foreach { i =>
    println(s"Only $secsUntilXmas seconds until Christmas!")
    Thread.sleep(5000)
  }
}
```

To run this solution, type:

```
sbt "runMain solutions.Loopy"
```

Multiple generators

You can index into a 2D array easily using multiple generators:

```
scala> val array = Array.ofDim[Int](4, 4)
array: Array[Array[Int]] = Array(Array(0, 0, 0, 0), Array(0, 0, 0, 0), Array(0, 0, 0, 0), Array(0, 0, 0, 0))

scala> for {
 i <- 0 until array(0).length
 j <- 0 until array(1).length
} array(i)(j) = (i+1) * 2*(j+1)

scala> array.foreach(row => println(row.mkString(", ")))
2, 4, 6, 8
4, 8, 12, 16
6, 12, 18, 24
8, 16, 24, 32
```

For Comprehensions

Scala offers a much more convenient way to express `map`, `flatMap` and `filter`: for comprehensions. Here is how we could print out a list of even numbers up to 10:

```
scala> for (i <- 1 to 10 if i % 2 == 0) println(i)
2
4
6
8
10
```

Compare that to:

```
scala> 1 to 10 filter( _ % 2 == 0) foreach { i => println(i) }
2
4
6
8
10
```

OK, so there does not seem to be much difference for this simple case. Let's try something more complex - picking apart a list of maps, similar to what you might have to contend with when receiving a JSON response from a web service.

```
scala> val selectedKeys = Map("selectedKeys"->Seq("one", "two", "three"))
selectedKeys: scala.collection.immutable.Map[String,Seq[String]] = Map(selectedKeys -> List(one, two, three))

scala> val otherKeys = Map("otherKeys"->Seq("four", "five"))
otherKeys: scala.collection.immutable.Map[String,Seq[String]] = Map(otherKeys -> List(four, five))

scala> val list: List[Map[String, Seq[String]]] = List(selectedKeys, otherKeys)
list: List[Map[String,Seq[String]]] = List(Map(selectedKeys -> List(one, two, three)), Map(otherKeys -> List(four, five)))
```

The test data is now ready in `list`. Curly braces are required if the for comprehension contains more than one statement:

```
scala> val result = for {
  data <- list
  selectedKeysSeq <- data.get("selectedKeys").toList
  id <- selectedKeysSeq.toList
} yield id
res6: List[String] = List(one, two, three)
```

Lets' rewrite this, showing types, to clarify what is happening:

```
val result2: List[String] = for {
  data: Map[String, Seq[String]] <- list
  selectedKeysSeq: Seq[String] <- data.get("selectedKeys").toList
  id: String <- selectedKeysSeq.toList
} yield id
```

Let's break this down.

1. The line `data <- list` causes a looping action to be invoked on each element of `list`. A temporary variable called `data` of type `Map[String, Seq[String]]` is created for each loop iteration, and it receives the current element of `list`.
2. The next line (`selectedKeysSeq <- data.get("selectedKeys").toList`) pulls out the value associated with the key `"selectedKeys"` from the `data` map, which is `Seq("one", "two", "three")` and starts an inner loop. Each element of the `Seq` is assigned to a new temporary variable called `selectedKeysSeq`.

3. The next line (`id <- selectedKeysSeq.toList`) iterates through each item in `selectedKeysSeq` and assigns it to a temporary variable called `id`.
4. The last line forms a new collection consisting of each of the values of that `id` had assumed. The collection type is determined by the collection type of each of the previous statements, which must be compatible. In this case, they are all `List`s.

So, this means that equivalent code is:

```
list.flatMap { data: Map[String, Seq[String]] =>
  data.get("selectedKeys").toList.flatMap { selectedKeysSeq: Seq[String] =>
    selectedKeysSeq
  }
}
```

Without types it would look like this:

```
list.flatMap { data =>
  data.get("selectedKeys").toList.flatMap { selectedKeysSeq =>
    selectedKeysSeq
  }
}
```

Which form would you rather write?

Exercise

The sample code below defines a `Map` created by a `groupBy`.

```
scala> val map = Map( "selectedKeys"->Seq("one", "two", "three"), "otherKeys"->Seq("two", "one") )
map: scala.collection.immutable.Map[String,Seq[String]] = Map(selectedKeys -> List(one, two, three), otherKeys -> Li
st(four, five))

scala> val inversion = map.groupBy(_._1)
inversion: scala.collection.immutable.Map[String,scala.collection.immutable.Map[String,Seq[String]]] = Map(otherKeys
 -> Map(otherKeys -> List(two, one)), selectedKeys -> Map(selectedKeys -> List(one, two, three)))
```

How can you transform `inversion` into the following?

```
Map[String,Seq[String]]] = Map(otherKeys -> List(two, one), selectedKeys -> List(one, two, three))
```

Hints

1. If you are stuck and want to debug your for comprehension, or you would like to invoke a method in an outer loop that only has side effects, you can assign to a placeholder:

```
scala> val result = for {
  x <- List(1, 2, 3)
  _ <- List(println(x))
  y <- List("a", "b")
} yield ((x, y))

1
2
3
result: List[(Int, String)] = List((1,a), (1,b), (2,a), (2,b), (3,a), (3,b))
```

2. Notice that the `println` is wrapped in a `List`; this is necessary because the for-comprehension consists of three nested loops, and should be read as follows:
   a. For each item in `List(1, 2, 3)`, assign the current item to `x`
   b. For each item in `List(println(x))`, perform `println(x)` and discard the result (which is `Unit`)
   c. For each item in `List("a", "b")`, assign the current item to `y`
   d. Combine the current value of `x` and `y` into a tuple and add to the accumulated result
   e. When the entire looping construct is complete, assign the result to `result`

Solutions

This solution is clumsy:

```
inversion.map { x =>
  val value = x._2(x._1)
  x._1 -> value
}
```

This solution is much easier to understand:

```
inversion.values.flatten.toMap
```

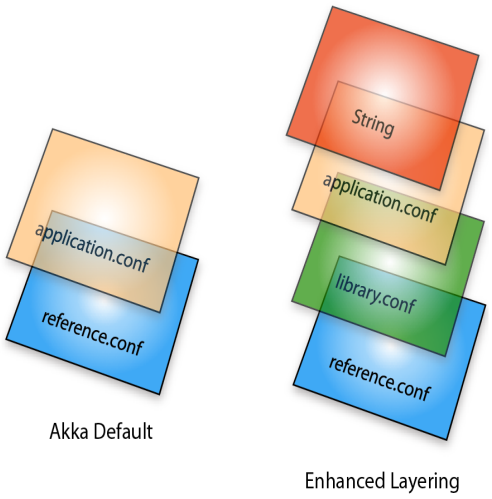
These solutions are provided in `solutions.GroupBy`. You can run them by typing:

```
sbt "runMain solutions.GroupBy"
```

2-10 Typesafe Config

`ScalaCourses.com` / `scalaIntermediate` / `ScalaCore` / `scalaTypesCollections` / `config`

The Typesafe Config utility is a flexible and standards-compliant way to provide configuration data to any Java or Scala application. The utility classes are written in Java and do not currently have a standardized Scala wrapper, although some people have created their own. This Java-based configuration utility is the de facto standard for specifying and parsing Scala configuration data. Typesafe Config can read Java properties, YAML, JSON, and a human-friendly JSON superset called HOCON. Subtrees of configuration data can be overlaid and automatically merged. Configuration data can be stored locally, over a network or loaded from the classpath. Upcasting of types is automatically supported. Overrides from environment variables allow sensitive data to be provided at runtime instead of committed into a code base.



It is easy to load configuration values from the default location, `application.conf`

```
val conf = ConfigFactory.load()
val value1 = conf.getString("keyName1")
val value2 = conf.getBoolean("keyName2")
val value3 = conf.getInt("nested.key.name3")
val value4 = conf.getConfig("nested.key.name4")
```

`application.conf` need only be on the `classpath`. For an SBT project, that means placing it in `src/main/resources` or `test/main/resources`. Using HOCON, the `.conf` file could be hierarchical. Let's imagine that `application.conf` contains the following for a moment:

```
keyName1 : "Value 1"
keyName2 = true
nested.key.name3 : 5
nested {
  key {
    name4 = {
      "one" : 10,
      "two" : 12
    }
  }
}
```

You can also read configuration values from a string. This is useful for specifying defaults, or for testing. The above could be expressed as the following (paste this into the REPL):

```
import com.typesafe.config.{Config, ConfigFactory}
val string = """
keyName1 : "Value 1"
keyName2 = true
nested.key.name3 : 5
nested { key { name4 = { "one" : 10, "two" : 12 } } }
array : [ 1, 2, 3]
""".stripMargin
val config = ConfigFactory.parseString(string)
```

Because Config was written in Java, the collections it provides don't have the handy Scala behavior. You can enhance the behavior by importing `JavaConverters` and using the `asScala` method:

```
scala> import collection.JavaConverters._
import collection.JavaConverters._

scala> val keys = config.entrySet.asScala.map(_.getKey)
keys: scala.collection.mutable.Set[String] = Set(array, nested.key.name4.one, keyName1, keyName2, nested.key.name4.two, nes
ted.key.name3)

scala> config.getString("keyName1")
res8: String = Value 1

scala> val array = config.getIntList("array")
array: java.util.List[Integer] = [1, 2, 3]

scala> val configNested = config.getConfig("nested")
configNested: com.typesafe.config.Config = Config(SimpleConfigObject({"key":{"name3":5,"name4":{"two":12,"one":10}}}))

scala> configNested.entrySet.asScala.map(_.getKey)
res9: scala.collection.mutable.Set[String] = Set(key.name4.two, key.name4.one, key.name3)

scala> configNested.getString("key.name3")
res10: String = 5
```

Extended Config Example

Here is a practical example: your application requires an AWS user ID and password; there are a default values for development, but production values must be supplied via environment variables. The following fragment of an `application.conf` file is all that is needed to specify these values. Although I originally drew this diagram to describe Akka configuration, it can equally well apply to your program. Let's set up a similar layering of configuration files:

```
import com.typesafe.config.{Config, ConfigFactory}

val defaultStr = """aws {
  accessKey = "stringAccessKey"
  secretKey = "stringSecretKey"
}""".stripMargin
val strConf = ConfigFactory.parseString(defaultStr) // experiment by commenting this line out
//val strConf = ConfigFactory.parseString("")       // ... and uncommenting this one
val appConf = ConfigFactory.load("application.conf")
val libConf = ConfigFactory.load("library.conf")
val defConf = ConfigFactory.load
val config: Config = ConfigFactory.load(
  strConf
    .withFallback(appConf)
    .withFallback(libConf)
    .withFallback(defConf))
```

The configuration file, named `application.conf`, might contain the following:

```
aws {
  accessKey = "applicationAccessKey"
  accessKey = ${?ACCESS_KEY}

  secretKey = "applicationSecretKey"
  secretKey = ${?SECRET_KEY}
}
```

The above syntax defines default values for `aws.accessKey` and `aws.secretKey`, which will be overridden if environment variables called `ACCESS_KEY` or `SECRET_KEY` are defined.

The environment variables necessary to override the default values are in `bin/launchConfig`:

```
#!/bin/bash

export ACCESS_KEY=AXWAJ8K7JK3JDRL2JOQ
export SECRET_KEY=K9LSJDKEJ738373LSLS923821

sbt "run-main ConfigDemo"
```

Note that bash does not allow environment variables to contain a dot (period), so the convention is to name them in all capital letters, using underscores (_). The following code uses fallback to overlay the definitions from each of the layers.

The `ConfigDemo` class in the `courseNotes` contains the above code in a runnable program. In order to compile, `build.sbt` must specify the appropriate dependency. The `mvnrepository.com` web site shows the available versions and the syntax for specifying them in `build.sbt`. If the environment variables are not set, then the values specified in `application.conf` are used.

We need to be able to display the configuration values, so we'll write a method that accepts a message and a `Config` object, then checks to see if there is a key/value pair for `aws.accessKey` and `aws.secretKey` in the `Config` object. The `Config` class does not provide a direct way to obtain the available keys in an instance, so we use the `asScala` method provided by `collection.JavaConverters` to wrap the `entrySet` of key/value pairs into a Scala equivalent collection, and then use `map` to transform the key/value pairs into a collection of keys. We then use the `contains` method to detect if the desired keys are defined.

```
import collection.JavaConverters._

def showValues(msg: String, config: Config): Unit = {
  val keys: Set[String] = config.entrySet.asScala.map(_.getKey).toSet // mutable by default

  if (keys.contains("aws.accessKey")) {
    val accessKey = config.getString("aws.accessKey")
    println(s"$msg accessKey=$accessKey")
  } else
    println(s"$msg does not define aws.accessKey")

  if (keys.contains("aws.secretKey")) {
    val secretKey = config.getString("aws.secretKey")
    println(s"$msg secretKey=$secretKey")
  } else
    println(s"$msg does not define aws.secretKey")

  println()
}
```

We will use the `showValues` method to display each `Config` instance:

```
showValues("defaultStr", strConf)
showValues("library.conf", libConf)
showValues("application.conf", appConf)
showValues("Default", defConf)
showValues("Combined", config)
```

Once the above code is wrapped in an App instance called `ConfigDemo` we can run it, without setting environment variables. You will find the completed program in the `courseNotes` directory.

```
$ sbt "run-main ConfigDemo"
[info] Loading global plugins from /home/mslinn/.sbt/plugins
[info] Loading project definition from /home/mslinn/work/training/projects/public_code/coreScala/course_scala_intermediate_
code/courseNotes/project
[info] Set current project to IntermediateScala Course (in build file:/home/mslinn/work/training/projects/public_code/coreS
cala/course_scala_intermediate_code/courseNotes/)
defaultStr does not define aws.accessKey
defaultStr does not define aws.secretKey

library.conf does not define aws.accessKey
library.conf does not define aws.secretKey

application.conf accessKey= applicationAccessKey
application.conf secretKey= applicationSecretKey

Default accessKey=applicationAccessKey
Default secretKey=applicationSecretKey

Combined accessKey=applicationAccessKey
Combined secretKey=applicationSecretKey

[success] Total time: 1 s, completed Sep 12, 2013 9:19:44 PM
```

If you comment out the line that creates `strConf`, and uncomment the one following it, the yellow highlighted values above change to:

```
application.conf accessKey= stringAccessKey
application.conf secretKey= stringSecretKey
```

If you use `bin/launchConfig` to define the environment variables and run the program, the output changes to:

```
$ bin/launchConfig
[info] Loading global plugins from /home/mslinn/.sbt/plugins
[info] Loading project definition from /home/mslinn/work/training/projects/public_code/coreScala/course_scala_intermediate_
code/courseNotes/project
[info] Set current project to IntermediateScala Course (in build file:/home/mslinn/work/training/projects/public_code/coreS
cala/course_scala_intermediate_code/courseNotes/)
defaultStr does not define aws.accessKey
defaultStr does not define aws.secretKey

library.conf does not define aws.accessKey
library.conf does not define aws.secretKey

application.conf accessKey= AXWAJ8K7JK3JDRL2JOQ
application.conf secretKey= K9LSJDKEJ738373LSLS923821

Default accessKey= AXWAJ8K7JK3JDRL2JOQ
Default secretKey= K9LSJDKEJ738373LSLS923821

Combined accessKey= AXWAJ8K7JK3JDRL2JOQ
Combined secretKey= K9LSJDKEJ738373LSLS923821

[success] Total time: 6 s, completed Sep 12, 2013 9:18:36 PM
```

Exercise

Write helper classes which enrich `Config` by adding Scala behavior. Define an implicit class that converts between a `String` and a `Config`, and another implicit class that enriches `Config` with a method called `keys` that returns a `Set[String]` of the available keys.

```
val config: Config = """
keyName1 : "Value 1"
keyName2 = true
nested.key.name3 : 5
nested { key { name4 = { "one" : 10, "two" : 12 } } }
array : [ 1, 2, 3]
""".stripMargin.toConf
config.keys
```

When you are done, compare your solution to `ConfigDemo` in the `assignment` project.

2-11 Parametric Bounds and Variance

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaTypesCollections / view

Lower Bounds

You might want to guarantee that a parametric type is a particular type, or a subtype. You can do that using lower bounds. As an example, let's define a class that performs operations on `Thang` and its subclasses. Let's extend our `Thang` example:

```
class Thung(override val i: Int, override val s: String) extends Thang(i, s)

class Th_ngMunger[A <: Thang](th_ng: A) {
  def doYourThing(count: Int) = s"${th_ng.i * count}: ${th_ng.s * count}"
}
```

Note that the type parameter `A` is constrained to be a `Thang` or a subclass of `Thang`, like `Thung`:

```
scala> new Th_ngMunger(thang1).doYourThing(3)
res1: String = 12: xxx

scala> new Th_ngMunger(thung1).doYourThing(5)
res2: String = 25: abababab
```

Upper Bounds

You can also specify upper bounds, which means that a parametric type must be a particular type or a supertype. The notation is `>:` and is usually used with collections, especially sequences and lists.

You can specify both lower and upper bounds, like this:

```
def put[U >: T <: Thang](item: U): Unit = ml += item
```

The above method is parametric in `T`, where `T` must be a `Thang` or a subtype in `Thang`, and a local type definition is created which must be a `T` or a supertype of `T`. In other words, you can put `Thungs` or `Thangs` in a `Bag`. We will see an example of that in a moment.

Covariance

Covariant types are immutable; a mutable container may not be designated as being covariant. Invariant types cannot be converted to subtypes or supertypes.

If a parametric type is *covariant*, types can be converted from a wider type (`Double`) to a narrower type (`Float`). A covariant collection of subtypes is a subtype of the collection type. For example, `List[+T]` is a subtype of

Contravariant types are used as arguments and covariant types are used as return types

`List[T]`. Covariant parameters are denoted
by preceding them with a plus sign, like this: `[+T]`. You can combine that with upper or lower bounds, like this: `[+T <: Thang]`. This means that `T` must be `Thang` or a subtype (like `Thung`), and collections of `T` subclasses are also subclasses of the collection.

If `Bag` is covariant, and if `Thung` is a subtype of `Thang`, then `Bag[Thung]` is a subtype of `Bag[Thang]`.

As an example, lets define a custom collection called `Bag`:

```
class Bag[+T] {
  val ml = collection.mutable.MutableList.empty[T]

  def put(item: T) = ml += item
}
```

`Array[Thang]` is a subtype of `Array[Thing]`.

You can combine covariance with lower bounds. For example, the following class (`Thoon`) is covariant in `T`, and `T` must be a `Thang` or a `Thang` subclass. We will see an example of that in a moment.

```
class [+T <: Thang] Thoon { /* etc */ }
```

Contravariance

Contravariance, denoted with a negative sign `[-T]`, means that the parametric collection is a supertype of the original collection. A conversion from a narrower type (`Float`) to a wider type (`Double`) is possible. If `Bag` is contravariant then `Bag[Thang]` is a supertype of `Bag[Thang]`.

Example of Upper and Lower Bounds

This example shows a quirk of the Scala compiler: a local upper bound.

```
import collection.mutable

class Thang(val i: Int, val s: String) {
  override def toString() = s"Thang $i: $s"
}

class Thung(override val i: Int, override val s: String) extends Thang(i, s) {
  override def toString() = s"Thung $i: $s"
}

class Bag [+T <: Thang]  {
  val ml = mutable.MutableList.empty[ Thang ]
  // this would be threadsafe, but slower:
  //val mb = new ArrayBuffer[Thang] with SynchronizedBuffer[Thang]

  def put [U >: T <: Thang] (item: U): Unit = ml += item

  def findByI(i: Int): List[Thang] = ml.filter(_.i==i).toList

  def findByS(s: String): List[Thang] = ml.filter(_.s==s).toList
}

object Main extends App {
  val bag = new Bag[ Thung ]
  bag.put(new  Thang (2, "abc"))
  bag.put(new  Thung (3, "def"))
  bag.put(new  Thang (4, "xyz"))
  println(bag.findByI(4))
  println(bag.findByS("def"))
}
```

`MutableList` is declared invariant and so we cannot specify that it contains items of the type covariant type parameter `T`. This only means that a `MutableList[Thung]` is not a subclass of `MutableList[Thang]`. However, `ml` can accept items that are subclasses of `Thang`, such as `Thung`.

The `put` method accepts items of type `Thang` or a subtype. The local type `U` is a Scala compiler implementation quirk. The explanation is long and not very interesting - suffiice it to say that when you put an item into a covariant collection you will need to specify a local supertype like `U`.

Two `Thang`s and one `Thung` are put in the `bag`. and are retrieved by the finders.

We done put our `Thang`s in th' `Bag`, podner!

2-12 Structural Types

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaTypesCollections / scalaStructural

Structural types are sometimes called 'duck typing'. Structural types are often most useful when used in conjunction with parametric types, and are easier to work with defined as type aliases. Here is a useful example that shows how to use a structural type. Note that `ByteArrayInputStream` has a `close` method, so it conforms to the `Closeable` structural type signature:

```
object Thing {
  type Closeable = { def close(): Unit }

  def using[A <: Closeable, B](closeable: A)(f: A => B): B = {
    try {
      f(closeable)
    } finally {
      try {
        closeable.close()
      } catch { case _: Throwable => () }
    }
  }

  val byteStream = new java.io.ByteArrayInputStream("hello world".getBytes)
  using(byteStream){ in =>
    val str = io.Source.fromInputStream(in).mkString
    println(s"'$str' has ${str.length} characters")
  }
}
```

Now let's run it:

```
scala> Thing
'hello world' has 11 characters
res1: Thing.type = Thing$@4669ed84
```

BTW, we can improve this program by use call by name instead of call by value for the `closeable` parameter:

```
object Thing {
  type Closeable = { def close(): Unit }

  def using[A <: Closeable, B](closeable: => A)(f: A => B): B = {
     val closeableRef = closeable // only reference closeable once
    try {
      f(closeableRef)
    } finally {
      try {
        closeableRef.close()
      } catch { case _: Throwable => () }
    }
  }

  val byteStream = new java.io.ByteArrayInputStream("hello world".getBytes)
  using(byteStream){ in =>
    val str = io.Source.fromInputStream(in).mkString
    println(s"'$str' has ${str.length} characters")
  }
}
```

Self Traits and Structural Types

Self types can extend structural types:

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

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

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

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

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

Let's play with this in the REPL:

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

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

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

2-13 Pattern Matching on Sequences

`ScalaCourses.com` / `scalaIntermediate` / `ScalaCore` / `scalaTypesCollections` / `scalaSeqMatch`

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

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

A better way to do this would be:

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

Putting aliases and sequence element matching together:

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

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

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

This is a simpler way of writing the same thing:

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

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

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

Extracting members from a List

This is quite elegant and powerful. Here is how we could print out the first three tokens of a string, or "Nope" if the string had less than three tokens:

```
"one two three blah blah".split(" ").toList match {
  case x1 :: x2 :: x3 ::  rest  => println(s"x1=$x1, x2=$x2, x3=$x3")
  case _ => println("Nope")
}
```

In the above code snippet, rest is a `List` containing the rest of the tokens, so it has the value `List("blah", "blah")`.

Here is how we could print out the tokens of a string if it has exactly three tokens, or "Nope" otherwise:

```
"one two thre".split(" ").toList match {
  case x1 :: x2 :: x3 :: Nil => println(s"x1=$x1, x2=$x2, x3=$x3")
  case _ => println("Nope")
}
```

2-14 Typeclasses

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaTypesCollections / scalaTypeclasses

We can use at least two approaches for enhancing existing types in Scala. Lets look at two ways of expressing that something can be quantified using an `Int`. We will first use implicit conversion, then we will do the same thing with typeclasses.

Implicit Conversions

We have seen this technique before, called the "enrich my library" pattern.

```
trait Quantifiable { def quantify: Int }
```

We can use implicit conversions to quantify, for example. `Strings` and `Lists`.

```
implicit def string2quant(s: String) = new Quantifiable{ def quantify: Int = s.length }

implicit def list2quantifiable[A](list: List[A]) = new Quantifiable { def quantify: Int = list.size }
```

After importing these implicit methods, we the method `quantify` can be called on `Strings` and `Lists`. Note that on every call to `quantify` the `list2quantifiable` implicit method gets triggered, reinstantiating `Quantifiable` and recalculating the `quantify` property.

Typeclasses

The *typeclass pattern* implements an alternative to inheritance by making functionality available through an implicit adapter. The typeclass pattern is an example of *ad-hoc polymorphism*, wherein a different algorithm may be used for each type of argument.

Continuing the above example, an alternative to the above implicit conversion would be to define an adapter method called `Quantified[A]`, which is the *typeclass*. An instance of an adapter method is often called a *witness,* and it indicates type `A` can be `Quantified`. In Scala, a type trait and a type class are both referred to as a typeclass.

```
trait Quantified[A] { def quantify(a: A): Int }
```

We then provide implicit instances of the typeclass for `String` and `List`, which implement different algorithms:

```
implicit val stringQuantifiable = new Quantified[String] { def quantify(s: String): Int =  s.length  }

implicit val intQuantifiable = new Quantified[Int] { def quantify(i: Int): Int =  i  }
```

Here is a method that quantifies its arguments using the typeclass:

```
def sumQuantities[A](as: List[A])(implicit witness: Quantified[A]) = as.map(witness.quantify).sum
```

Using the *context bound* syntax:

```
def sumQuantities [A : Quantified] (as: List[A]) = as.map( implicitly[Quantified[A]] .quantify).sum
```

Context Bound Shorthand Notation

Methods that looks like this:

```
def sum[T](list: List[T])( implicit integral: Integral[T] ): T = ???
```

...can be rewritten as:

```
def sum[T : Integral ](list: List[T]): T = ???
```

A context bound is of the form `[T: Bound]`; it is expanded to the plain type `T` together with an implicit parameter of type `Bound[T]`.

The typeclass methods can be invoked like this:

```
scala> val stringQuantities = sumQuantities(List("abc", "defghi"))
stringQuantities: Int = 9

scala> val intQuantities = sumQuantities(List(1, 2, 3))
intQuantities: Int = 6
```

The `typeclassFun.scala` program in the `courseNotes` project contains the above code.

The context bound syntax cannot be used if the typeclass has multiple type parameters. For example, if I want to quantify things not only with integers but with a list of tuples like `List[(A, B)]`, To do this I need to create a typeclass`Quantified[A, B]`. That is not possible with the context bound syntax, so we must use the longer form. In the following example, the name `evidence` is used for the witness parameter, following a popular convention:

```
implicit val abQuantifiable = new Quantified[Tuple2[A, B]] { def quantify(tuple: Tuple2[A, B]): Int = tuple._1.size + tuple
._2.size }

def sumQuantities[A](as: List[Tuple[(A, B)]]) (implicit evidence: Quantified[A, B]) = as.map(evidence.quantify).sum
```

Review of Bounds

View bound `A <% B` **(deprecated as of Scala 2.11)**

means 'can be viewed as', as in "`A` can be viewed as a `B`". In other words, there should be an **implicit conversion** from `A` to `B` in scope, so that one can call the methods of `B` on an object of type `A`.

Upper bound `A <: B`

means 'is a, or is a subtype of', as in "`A` is a `B`, or is a subtype of `B`".

Lower bound `A >: B`

means 'is a, or is a supertype of', as in "`A` is a `B`, or is a supertype of `B`".

Context bound `A : B`

means 'has a', as in "`A` has a `B`". In other words, a context bound means that an **implicit value** of a particular parameterized type (a typeclass) must exist in scope.

Here is a good comparison of view bounds and context bounds.

implicitly

Use of a context bound requires that the implementation of the method that uses the context bound must employ `implicitly` to obtain the implicit value of the typeclass. For example:

```
scala> implicitly[Numeric[Int]]
res2: Numeric[Int] = scala.math.Numeric$IntIsIntegral$@5ad1e727
```

`implicitly` is part of `predef`:

```
def implicitly[A](implicit a: A) = a
```

implicitNotFound

You should annotate traits that might be used as implicits for better error messages. Here's an example REPL session:

```
scala> import annotation.implicitNotFound
import annotation.implicitNotFound

scala> @implicitNotFound(msg = "Cannot find Serializable typeclass for ${T}") trait Serializable[T]
defined trait Serializable

scala> def foo[X : Serializable](x : X) = x
foo: [X](x: X)(implicit evidence$1: Serializable[X])X

scala> foo(5)
:11: error: Cannot find Serializable typeclass for Int
       foo(5)
          ^
```

The annotation has very little documentation, but supports `${TypeName}` templates to inject the names of types the compiler is looking for. The error message is rather mechanical to write; when decorating type `T` with `implicitNotFound`, the enriching typeclass `U` should have a message of this form: `"Cannot find U typeclass for ${T}"`. Seems like Scala compiler should automatically annotate all classes and traits this way.

Implicit Conversions vs. Typeclasses

When should you use implicit conversions, and when should you use typeclasses?

- Each method has a different idiom, which dictates how the code is expressed. Implicit conversions make it seem as if the new functionality is just another method or property of the class being enriched:

  ```
  "my string".newFeature
  ```

  Typeclasses are used as functions:

  ```
  newFeature("my string")
  ```

- Typeclasses do not create a new wrapper object for each use, so they perform better.
- Typeclasses allow properties to be added to a type, rather than to an instance of a type. You

can then access these properties even when you do not have an instance of the type available. For example:

```
trait Default[T] { def value : T }

implicit object DefaultInt extends Default[Int] {
  def value = 42
}

implicit def listsHaveDefault[T : Default] = new Default[List[T]] {
  def value = implicitly[Default[T]].value :: Nil
}

scala> default[List[List[Int]]]
resN: List[List[Int]] = List(List(42))
```

Another Example

Scala's `Numeric` trait defines many operations such as `plus`, `minus`, `times` and `divide`, which are handy when writing a domain-specific language (DSL). Importing `Numeric._` also imports many implicit conversions so the operations become available to the enriched typeclass:

```
scala> import Numeric._
import Numeric._

scala> def add[A](x: A, y: A)(implicit numeric: Numeric[A]): A = numeric.plus(x, y)
add: [A](x: A,y: A)(implicit numeric: Numeric[A])A

scala> add(1, 2) # implicit object is defined in the import
res0: Int = 3

scala> add(1.1, 2.2)
res1: Double = 3.3000000000000003
```

Using context bounds, the previous example can be rewritten as:

```
scala> import Numeric._
import Numeric._

scala> def add[A  : Numeric ](x: A, y: A): A =  implicitly[Numeric[A]] .plus(x, y)
dd: [A](x: A, y: A)(implicit evidence$1: Numeric[A])A

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

scala> add(1.1, 2.2)
res3: Double = 3.3000000000000003
```

View Bound Desugaring and Possible Deprecation

Recall that a view bound means "`A` can be viewed as a `B`":

```
def f[A <% B](a: A) = a.bMethod
```

The Scala compiler desugars the above to the following:

```
def f[A](a: A)(implicit evidence: A => B) = a.bMethod
```

There is talk about deprecating view bounds. If that happens, the desugared syntax would be used instead.

2-16 Assignment

`ScalaCourses.com` / `scalaIntermediate` / `ScalaCore` / `scalaTypesCollections` / `interSoFar1`

You have two hours for this assignment.

1. Create a new source file in the `courseNotes` project called `RichFile.scala`.
2. Enrich the `java.io.File` class such that the following methods are available, and implement them. Use `try`/`catch`/`finally` constructs to catch errors.

```
/** @return None if the File does not exist.
  * If file is a directory, return the listing as a [List[File]] on the right side of the Either,
  * otherwise return the contents of the File as Array[Byte]] */
def contents: Option[Either[Array[Byte], List[File]]] = ???

/** @return directory listing of the File. If the File does not exist, or it is not a directory return an empty list.
*/
def directory: List[File] = ???

/** @return contents of the File as Option[String]. If the File does not exist, or it is a directory, or cannot be rep
resented as a String, return None. */
def contentsAsString: Option[String] = ???

/** Copy file to newFile; return true if successful.  */
def copy(newFile: File): Boolean = ???

/** Copy file to newFileName; return true if successful */
def copy(newFileName: String): Boolean = ???
```

3. Use `Stream.continually`, `Console.readLine` and `takeWhile` to rewrite the integer adding exercise we did earlier; prompt the user for an integer, display the cumulative total, stop prompting if the user enters an empty string.

2-17 Partially Applied Functions

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaTypesCollections / scalaPartiallyApplied

Functions that have multiple parameter lists are commonly used in Scala. Here is an example:

```
scala> object R1 { def repeat(a: String)(b: Int) = a * b }
defined module R1

scala> R1.repeat("ab ")(3)
res0: String = "ab ab ab "
```

You can think of functions with multiple parameter lists as being composed of a chain of methods, with the result of the first method passed into the second method, and so on.

```
scala> object R2 {
  def repeat1(a: String) = a
  def repeat2(b: Int) = { repeat1("ab ") * b }
}
defined module R2

scala> R2.repeat2(3)
res1: String = "ab ab ab "
```

The Scala syntax for defining and using partially applied functions is just the same as for lifting a method into a function, except that at least one parameter is supplied, or bound. Lets first define a partially applied function based on the `repeat` function we just saw.

```
scala> val xyz = R1.repeat("xyz_") _
xyz: Int => String = <function1>
```

Remember that in Scala, underscores perform many purposes. A common purpose is to act as a placeholder, and that is what it is used for in the above. The underscore tells the compiler to make a new function that just has the first parameter list applied (we say the first parameter list is *bound*). The second parameter list will be supplied later (it is *unbound*).

We named this new function `xyz`, and it has type `Int => String`. This can also be written as `(Int) -> String`; both incantations mean "function that accepts an `Int` and returns a `String`".

When we invoke `xyz` by binding to the missing parameter list, the `repeat` method is run with all parameters supplied:

```
scala> xyz(4)
res3: String = xyz_xyz_xyz_xyz_
```

Here is a function literal that accepts multiple parameter lists. I have highlighted the function type in yellow and the implementation in blue :

```
scala> val f4: (String) => (Int) => String = (arg1) => (arg2) => arg1 * arg2
f4: String => (Int => String) = <function1>

scala> f4("a")(4)
res12: String = aaaa
```

We can dispense with the parentheses around the single arguments. The return type , which is part of the `FunctionN` type, is shown in green:

```
scala> val f5: String => Int => String = arg1 => arg2 => arg1 * arg2
f5: String => (Int => String) = <function1>

scala> f5("a")(4)
res13: String = aaaa
```

Curried Functions

Functions that are defined with multiple parameter lists, each with only one parameter, are *curried functions*. You can convert a regular function to a curried function by using Scala's built-in `curried` method. Let's curry the repeat method we defined earlier:

```
scala> val curry2 = f2.curried
curry2: String => (Int => String) = <function1>

scala> val curry3 = f3.curried
curry3: String => (Int => String) = <function1>
```

The parentheses indicate that a function with of the type `Int => String` is returned when only the first parameter is specified.

We can also *uncurry* a function, thereby creating an equivalent function with one parameter list:

```
scala> val function = repeat _
function: String => (Int => String) = <function1>

scala> val uncurried2 = Function.uncurried(curry2)
uncurried2: (String, Int) => String = <function2>

scala> function("a")(3)
res4: String = aaa

scala> uncurried2("a", 3)
res5: String = aaa
```

Note that currying and uncurrying only works on functions, not methods.

```
scala> R2.repeat2.curried
<console>:9: error: missing arguments for method repeat2 in object R2;
follow this method with `_' if you want to treat it as a partially applied function
              R2.repeat2.curried
                 ^

scala> R2.repeat2.uncurried
<console>:9: error: missing arguments for method repeat2 in object R2;
follow this method with `_' if you want to treat it as a partially applied function
              R2.repeat2.curry
                 ^
```

Partially Applied Function with one or more Functors

If the trailing parameter lists of a method are functors which accept a single parameter of the same type as returned by the partially applied function created from the previous parameter lists, then partially applying those methods can become quite powerful. That sounds very confusing. Let's modify the previous example slightly, so the last parameter list accepts a functor. In the following example, note that the type of the `abc` method is `(String -> String) => String`. The portion of the return type inside the parentheses is the type of the bound method, and it means `abc` accepts a `String` parameter, and returns a `String`. The remainder of the type, outside the parentheses are the unbound types of this partially applied function. Note that `repeat2`'s last parameter list is a functor that accepts a `String` (from the partially applied function) and returns a `String`.

```
scala> def repeat2(a: String)(f: String => String) = f(a)
repeat: (a: String)(f: String => String)String

scala> def abc = repeat2("abc") _
abc: (String => String) => String
```

The line above ending with the underscore is an example of *eta expansion*, which causes the partially applied function to be created.

This next line is important, yet it seems familiar enough that you might miss the significance. **The partially applied function** `abc` **provides the bound parameters defined to the left of the parameter that the function as a parameter to the body of the functor.** In this case, the only parameter that is supplied is the `String a`. In the following code, we arbitrarily named the parameter reference `a2`, but that name could be anything; it is a placeholder. We will see this pattern used again later in this lecture, and if you find yourself asking "Where did that value in the body of the function come from?", remember this simple example.

Notice that curly braces are used instead of parentheses. Scala allows this if a parameter list has only one parameter. Using curly braces around a block of code is easier to read.

```
scala> abc { a2 => a2 * 4 }
res6: String = abcabcabcabc
```

As you should know by now, the previous line could be written as:

```
scala> abc { _ * 4 }
res7: String = abcabcabcabc
```

Let's try that again, but with an extra parameter list:

```
scala> def repeat3(a: String)(b: String)(f: (String, String) => String) = f(a, b)
repeat3: (a: String)(b: String)(f: (String, String) => String)String

scala> val abcdef = repeat3("abc")("def") _
abcdef: ((String, String) => String) => String = <function1>

scala> abcdef { (s1, s2) => s1 + s2 }
res8: String = abcdef
```

And again! First we define a method that references the bound parameters `a`, `b` and `f` in the implementation:

```
scala> def repeat3b(a: String)(b: String)(f: (String, String) => String) = a + b + f(a,b)
repeat3b: (a: String)(b: String)(f: (String, String) => String)String

scala> repeat3b("x")("y") { (u, v) => (u.length + v.length).toString }
res9: String = xy2
```

Now we partially apply the method, leaving `f` unbound:

```
scala> val r3b = repeat3b("x")("y") _
r3b: ((String, String) => String) => String = <function1>
```

Now let's apply an anonymous function to `f`:

```
scala> r3b { (u, v) => (u.length + v.length).toString }
res10: String = xy2
```

Eta-expansion turns A* to Seq[A]

Repeated arguments (also known as *varargs*) can be denoted by an asterisk. The following code defines a method `foo` that accepts one or more integers.

```
scala> def foo(ns:Int*) = ns.sum
foo: (ns: Int*)Int

scala> foo(1,2,3)
res11: Int = 6

scala> val foov = foo _
foov: Seq[Int] => Int = <function1>

scala> foov(Seq(1,2,3))
res12: Int = 6
```

The Loan Pattern

The Loan Pattern loans a resource to a higher-order function. Implementations of this pattern do the following:

- Create or access a resource
- Loan the resource to a higher-order function
- The higher-order function would use the resource
- Dispose of the resource reliably, efficiently and safely

The advantages of the loan pattern are:

- The loan pattern implementation is completely separate from the higher-order function
- Various higher-order function can be provided
- Higher-order functions are not concerned about the creation, destruction of the resource because the loan pattern implementation handles that

Here are a few examples of partially applied functions that use the loan pattern, and they work together. The first example, `withCloseable`, accepts two functors and returns a `Try`. The first functor is a factory method, which constructs an instance of a `java.io.Closeable`.

The second functor is an operation to be performed on the `Closeable` instance. The `Closeable` is closed after `operation` completes. A `Try` is returned, which wraps the result of running `operation`.

```
import java.io._
import scala.util.{Failure, Success, Try}

def withCloseable[C <: Closeable, T](factory: => C)(operation: C => T): Try[T] = {
  val closeable = factory
  try {
    val result: T = operation(closeable)
    closeable.close()
    Success(result)
  } catch {
    case throwable: Throwable =>
      try { closeable.close() } catch { case _: Throwable => }
      Failure(throwable)
  }
}
```

This factory method above does not require any input, so the functor could be written to accept an empty parameter list (also known as *Unit*). If we did that, the first parameter list would look like this:

```
(factory: () => C)
```

That is weird, but accurate. It says that `factory` does not require any input, since `()` is a synonym for `Unit`, and it returns an instance of `C`. Instead, we defined `factory` as a *call by name* parameter, which also means that it does not require any input.

```
(factory: => C)
```

Many Scala tutorials simply state that call by name parameters are not evaluated until it is referenced in the body of the method, and that is true, but that is also true of the empty parameter list that we did not employ. The key point about this particular call by name parameter is that it contains a reference to a function definition, because `C` is a function type, not a value type.

You cannot tell by simply reading the invocation of a function or method if it has any call by name parameters.

The easiest way to use `withCloseable` would be to invoke it with both parameters, like this:

```
def read(inputStream: InputStream): String = // we will explain this method in a later lecture
  Stream.continually(inputStream.read).takeWhile(_ != -1).map(_.toChar).mkString

withCloseable(new FileInputStream(new File("/etc/passwd"))) (fis => read(fis))
```

The above causes the contents of `/etc/passwd` to be displayed. The first parameter list constructs a `FileInputStream`, and the second parameter list receives the `FileInputStream` and passes it to the `read` method we just defined. You could also write it like this:

```
withCloseable(new FileInputStream(new File("/etc/passwd"))) (read(_))
```

Or like this:

```
withCloseable(new FileInputStream(new File("/etc/passwd"))) (read _)
```

You could even write it like this:

```
withCloseable(new FileInputStream(new File("/etc/passwd")))(read)
```

Now for the exciting part. You could *partially apply* the `withCloseable` function by only providing it the first parameter list, like this:

```
scala> withCloseable(new FileInputStream(new File("/etc/passwd"))) _
res13: (java.io.FileInputStream => Nothing) => scala.util.Try[Nothing] = <function1>
```

Because `withCloseable` is parametric, you must let the compiler know the type of the unbound method parameters. The easiest way to do this is to provide the types of all of the method parameters, like this:

```
scala> val opened = withCloseable[FileInputStream, String](new FileInputStream(new File("/etc/passwd"))) _
opened: (java.io.FileInputStream => String) => scala.util.Try[String] = <function1>
```

Note that the type of the new `opened` function is `(java.io.FileInputStream => String) => scala.util.Try[String]`.

You invoke `opened` by providing the missing parameter list, which merely consists of the operation to be performed:

```
scala> opened(read)
res14: scala.util.Try[String] =
Success(root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
...
```

Alternative Syntax

Here is a first attempt at another partially applied function. Notice the trailing underscore which lets us know that the second parameter list is unbound.

```
/** Partially applied function; the withCloseable functor that returns Try[T] is unbound */
def withBufferedInputStream[T](input: File) =
  withCloseable[BufferedInputStream, T](new BufferedInputStream(new FileInputStream(input))) _
```

The above has some redundancy; `BufferedInputStream` is indicated as a type parameter and in the body of the factory functor, and the unbound type is referenced as `T`. We can rewrite this without any redundancy by providing the return type as follows:

```
/** Partially applied function; the withCloseable functor that returns Try[T] is unbound */
def withBufferedInputStream[T](input: File): (BufferedInputStream => T) => Try[T] =
  withCloseable(new BufferedInputStream(new FileInputStream(input))) _
```

As you can see, the return type is `(BufferedInputStream => T) => Try[T]`. You could also write the above without the trailing underscore:

```
/** Partially applied function; the withCloseable functor that returns Try[T] is unbound */
def withBufferedInputStream[T](input: File): (BufferedInputStream => T) => Try[T] =
  withCloseable(new BufferedInputStream(new FileInputStream(input)))
```

With the above syntax, is less obvious that the first parameter list, of type `BufferedInputStream => T`, is bound, and the second parameter list, of type `T => Try[T]`, is unbound. You could only know that by matching up the parameter lists and noticing that the right-most one was missing. In this case the comment is helpful.

```
/** Partially applied function; the withCloseable functor that returns Try[T] is unbound */
def withBufferedOutputStream[T](output: File): (BufferedOutputStream => T) => Try[T] =
  withCloseable(new BufferedOutputStream(new FileOutputStream(output)))
```

Using Partially Applied Functions

Notice how the two partially applied functions (`withBufferedInputStream` and `withBufferedOutputStream`) provide the result of executing their bound parameters to their respective bodies as `inputStream` and `outputStream`. We again use the `read` method shown earlier:

```
withBufferedInputStream(new File("/etc/passwd")) { inputStream =>
  withBufferedOutputStream(new File("/tmp/blah")) { outputStream =>
    read(inputStream).foreach(outputStream.write(_))
  }
}
```

You will frequently see this pattern in the Akka library and in Scala `Futures`.

Exercise

Wrapping Database Operations With the Loan Pattern

Let's use the loan pattern to write a database wrapper. JDBC connections implement `AutoCloseable`, not `Closeable`. We can modify `withCloseable` to work with JDBC connections two ways:

1. Change the signature to accept the type `AutoCloseable` instead of `Closeable`
   ```
   def withCloseable[C <: AutoCloseable, T](factory: => C)(operation: C => T): Try[T]
   ```
2. Change the signature to accept a structural type that references a method called `close` which does not accept parameters and returns nothing. Structural types are a little slower than regular types because they are implemented with introspection.
   ```
   def withCloseable[C <: { def close(): Unit }, T](factory: => C)(operation: C => T): Try[T]
   ```

It is up to you to decide how you want to modify `withCloseable`. In either case, the body of the method would remain unchanged.

We'll use the SQLite database, which just uses a single file to store the database and doesn't have a concept of username or password. You will need to add the appropriate dependency in build.sbt. Here is the SQLite GitHub project. You need to find the most recent version of that dependency by using `http://mvnrepository.com` and insert it into `build.sbt`. If you use an IDE, don't forget to update the IDE project by typing `sbt gen-idea` or `sbt eclipse`.

Your program only needs this single import statement to make the SQLite functionality available:

```
import java.sql.{Connection, DriverManager}
```

You can get a connection to a SQLite database using the following:

```
def connection(url: String, userName: String="", password: String=""): Connection = {
  Class.forName("org.sqlite.JDBC")
  DriverManager.getConnection(url, userName, password)
}
```

The following will get a connection to a SQLite database in the current directory called `person.db` (remember that `userName` and `password` are ignored by SQLite):

```
val conn = connection("jdbc:sqlite:person.db")
```

You can create a table with the following method:

```
def createTable(connection: Connection, tableName: String, creationStatement: String): Connection = {
  val statement = connection.createStatement()
  statement.setQueryTimeout(30)
  statement.executeUpdate(s"drop table if exists $tableName")
  statement.executeUpdate(s"create table $tableName ($creationStatement)")
  connection
}
```

Given a database connection called `conn`, you can create a table called `person` with columns `id`, `name` and `age` by writing:

```
createTable(conn, "person", "id integer, name string, age integer")
```

The following will insert a new record into the database. Yes, Fred is 40,001 years old!

```
val stmt = conn.prepareStatement("insert into person (id, name, age) values (?, ?, ?)")
stmt.setInt(1, 1)
stmt.setString(2, "Fred Flintstone")
stmt.setInt(3, 400001) // Cromagnon man!
stmt.executeUpdate()
```

Your task is to write a console application that uses the loan pattern to obtain a database connection and insert a few records, then select them and print them out.

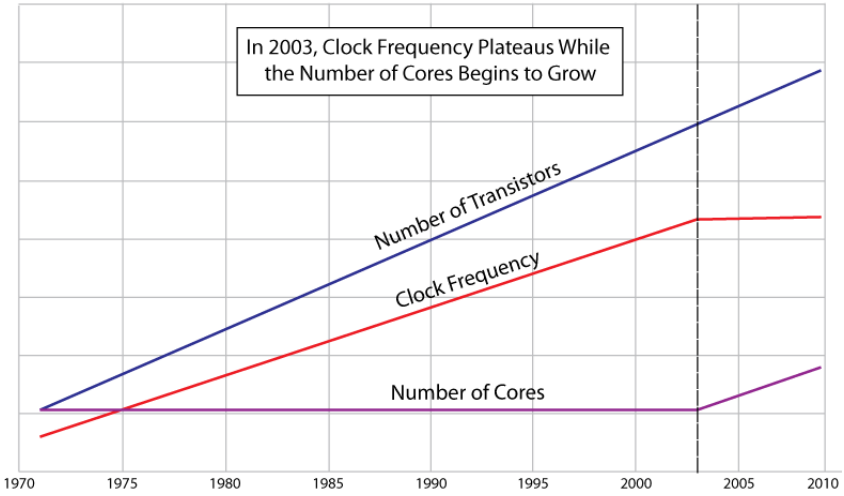
Hints:

1. You can select all records in a database like this:
   ```
   val stmt2 = conn.prepareStatement("select * from person")
   ```
2. If you consider `stmt2.executeQuery` as a factory returning a `resultSet`, you could pass it to `withCloseable` and obtain each successive record by calling `resultSet.next`. `ResultSet`s are mutable objects, which means they are not threadsafe, but they are convenient to work with in a loop.
3. You might want to use a `do while` loop to walk through the entire `resultSet`.
4. The number of columns in the `resultSet` can be obtained by calling `resultSet.getMetaData.getColumnCount`.
5. The `i`th column in a `resultSet` can be obtained by calling `resultSet.getObject(i)`.

3 Concurrency and Parallelism

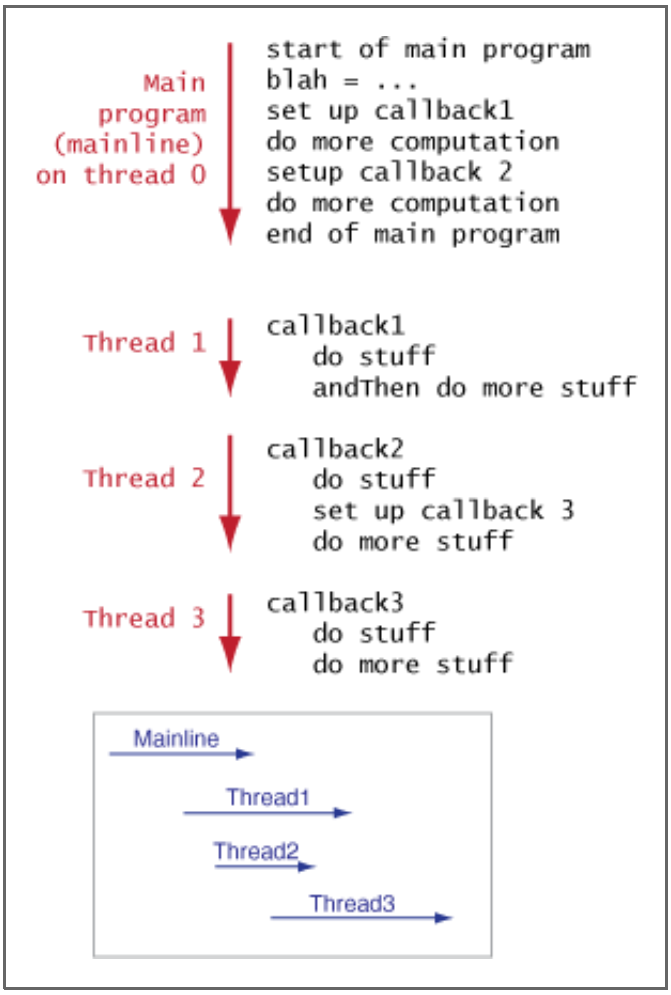
3-1 Concurrency

Until 2003, computers were able to run faster simply by increasing clock speed.



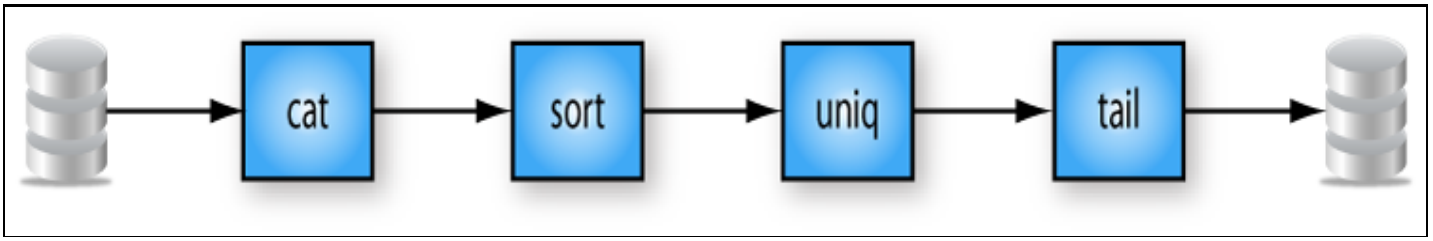
We have since hit the limits of physics and now must increase the speed of computation by doing many things at once, using multiple threads. Each thread requires hardware resources. Threads are independent, heap-sharing execution contexts, scheduled by the operating system. Creation is expensive, so thread pools are used to minimize thread creation. However, managing thread pools introduces additional complexity. Worse, multiple threads introduce cache coherency issues.

You could use multithreading and callbacks, but this is a very complex way to program:

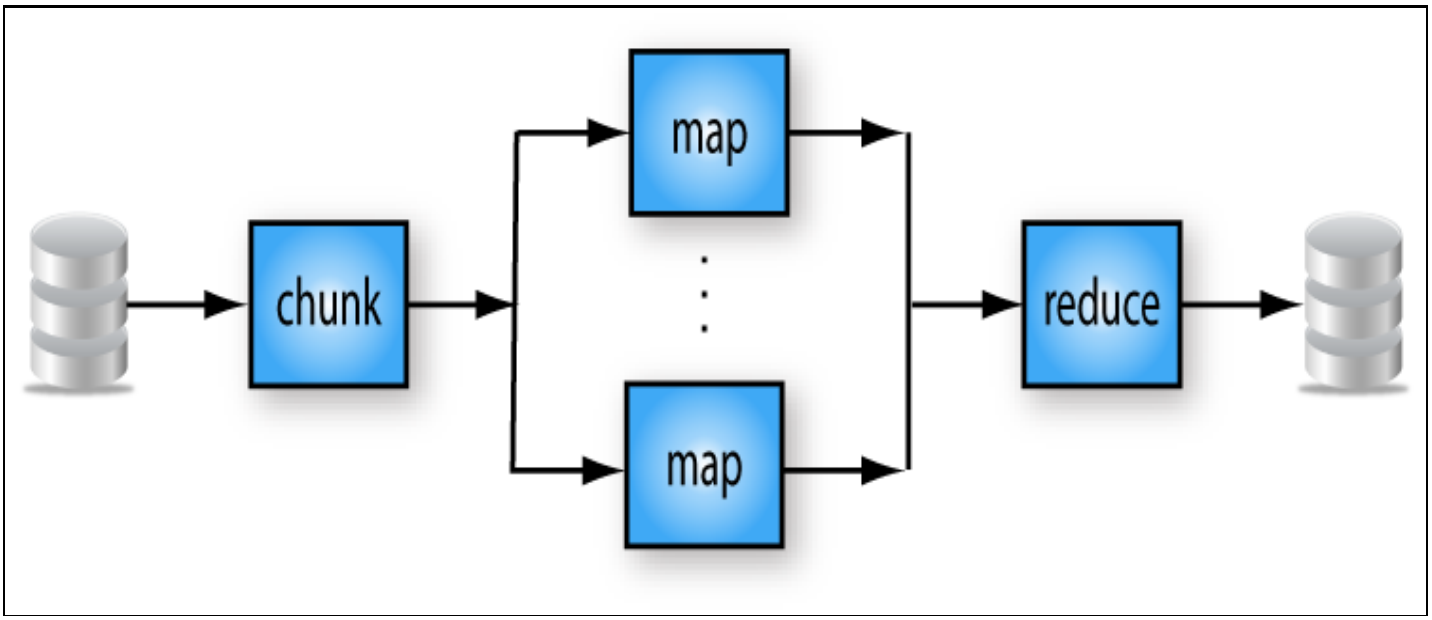


Functional Style Pipelines

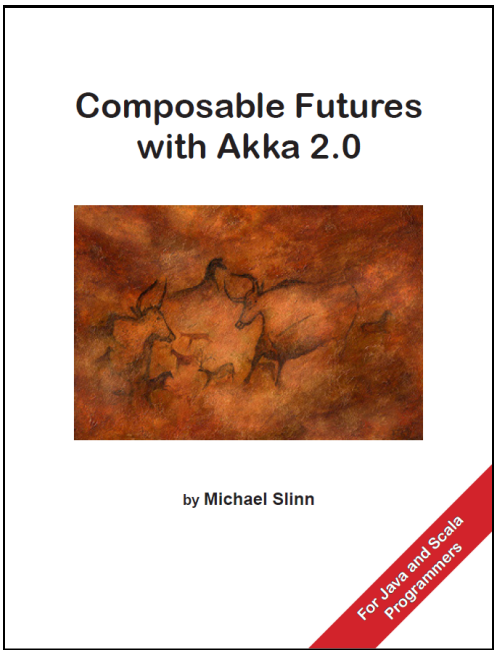
*nix pipes (linear)



Scala parallel collections & Akka composable futures (parallel)



Concurrency Definitions



These definitions are not universal, but they are the definitions that I use.

- Concurrent tasks are stateful, often with complex interactions
- Parallizable tasks are stateless, do not interact, and are composable

java.util.concurrent

- Scala and Akka use java.util.concurrent (j.u.c.)
- Scala parallel collections and futures are designed to be used with a `Callable`, which returns a result
- Executors contain factories and utility methods for concurrent classes

Provided in the j.u.c. package:

- Callable, Runnable
- `Thread`s, thread pools and `Executor`s
- `CompletionService`, `CountDownLatch`, `CyclicBarrier`, `Phaser`, `Semaphone`, `TimeUnit`
- `java.util.concurrent.atomic`
- `java.util.concurrent.locks`
- Concurrent collections
- Primitive `Future<T>`

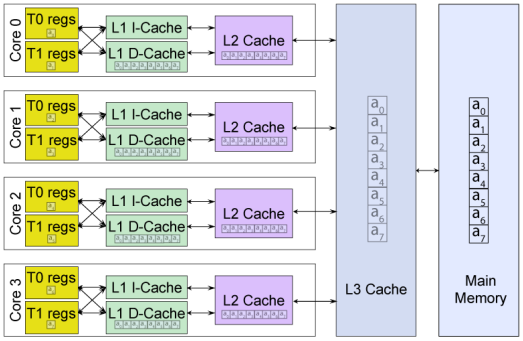
j.u.c. Is Too Hard To Use

- Multithreaded programs tend to be buggy
- Non-determinism caused by concurrent threads accessing shared mutable state.
- Actors and transactions manage state and are therefore not the first solution you should

consider.
- To get deterministic processing, avoid shared mutable state.

Concurrency Hazards

- CPU registers
- L1, L2 and L3 caches
- L1 cache for each core
- L2 cache shared by pairs of cores
- L3 cache shared by all cores
- Processors are different, so a program may run fine on one system but fail on another. AMD Opteron and Intel i7 caches are quite different, for example.



Typical Latencies

(Thanks to Peter Norvig, "Teach Yourself Programming in 10 Years")

- L1 cache reference 0.5 ns
- Execute instruction 1 ns (electricity travels 1')
- Branch mispredict 5 ns
- L2 cache reference 7 ns
- Mutex lock/unlock 25 ns
- L3 cache reference 40 ns
- Main memory reference 100-300 ns
- Electricity travels 984' 1,000 ns
- Send msg to Akka actor within JVM 1,000ns
- Compress 1K bytes with Zippy 3,000 ns
- Send 2K bytes over 1 Gbps net: 20,000 ns
- SSD disk seek: 100,000 ns
- Read 1 MB RAM sequentially 250,000 ns
- Create first instance of a medium-sized value class: 350,000 – 480,000 ns
- Disk seek 6,000,000 ns
- Read 1 MB seq. from disk 20,000,000 ns
- Send packet CA->Europe->CA 150,000,000 ns

Standard Thread Pools

`j.u.c` provides a variety of `Executors`, and you can create more. `Executor` contain factories and utility methods for concurrent classes such as:

`Executor`, `ExecutorService` and `Callable`. All of these classes are important when working with `Futures`. One important use of `Executors` is to create threadpools with various characteristics. Scala uses Java threadpools exactly like any other Java library would. Proper selection of the type of threadpool that your application requires is important in order to obtain the desired behavior. Threadpools are of type `j.u.c.ExecutorService`. Java provides several built-in types of threadpools, all of which use regular threads, as shown in the table below. You can define more threadpools classes to suit your needs.

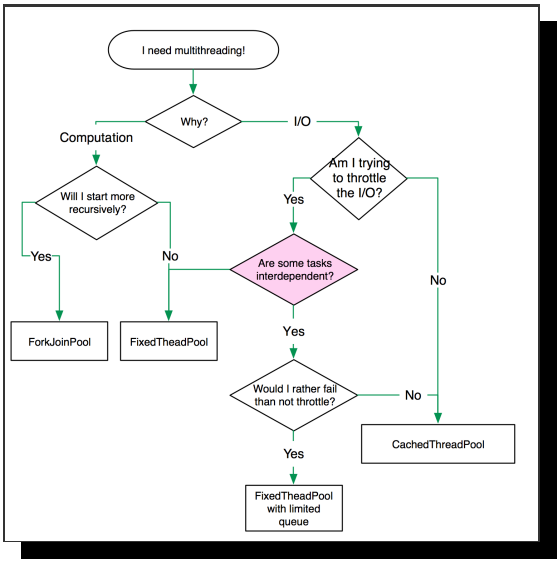
Name	Description
cached	`Executors.newCachedThreadPool` – Useful for CPU-intensive tasks with a varying load. Creates new threads as needed, but will reuse previously constructed threads when they are available.
fixed	`Executors.newFixedThreadPool(n)` – Useful for CPU- intensive tasks with a constant load. Reuses `n` threads operating off a shared unbounded queue. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. If any thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.
scheduled	`Executors.newScheduledThreadPool(corePoolSize,threadFactory)` – Can schedule commands to run after a given delay, or to execute periodically.
single	`Executors.newSingleThreadExecutor` – A single worker thread operating off an unbounded queue. If this single thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks. Tasks are guaranteed to execute sequentially, and no more than one task will be active at any given time.
single scheduled	`Executors.newSingleThreadScheduledExecutor` – A single-threaded executor that can schedule commands to run after a given delay, or to execute periodically. If this single thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks. Tasks are guaranteed to execute sequentially, and no more than one task will be active at any given time.
fork / join	`akka.jsr166y.ForkJoinPool` – It is suitable for IO-bound and recursive tasks. A improved version of the Java 7 `java.util.concurrent.ForkJoinPool`, will be provided with Java 8. This is the most complex of the standard `Executors`, and incurs slightly more overhead. It creates daemon threads, unlike all the other executors.

Parallelism is the number of threads assigned to a task. *Parallelism factor* is the inverse ratio of busy threads to total assigned threads; the other threads are blocked. For example, if 10 threads are assigned to a task, and only two are busy while 8 are blocked, the parallelism factor is 5.

Parallelism factor is an important metric to know when selecting and configuring a thread pool. It is useful to be able to target an appropriate parallelism factor at deployment time; Akka's configuration file makes this easy for actors and other concurrency schemes that it supports.

The threads in all these standard pools, except fork/join, will continue to exist until they are explicitly >shut down because they use regular threads. The Javadoc comment for *ExecutorService.shutdown()* reads “Initiates an orderly shutdown i which previously submitted tasks are executed, but no new tasks will be accepted. Invocation has no additional effect if already shut down.” The class-level Javadoc goes further and says “An unused `ExecutorService` should be shut down to allow reclamation of its resources.” `shutdown` can fail, and those failures can be caught with an instance of `RejectedExecutionHandler`.

The following flow chart is taken from "Choosing an ExecutorService" on the Abstractivate blog:



Daemon Threads

Scala concurrency is normally multi-threaded. Your Scala or Java program executes from beginning to end, but if it creates regular threads, your program will not exit until all of those threads are stopped. `ExecutorServices` create threads but does not stop them unless `shutdown` is explicitly called. Your program will not exit unless those threads are terminated, or they are set to run as daemon threads.

he class-level documentation for `j.u.c.ThreadPoolExecutor` says “New threads are created using a `ThreadFactory`. If not otherwise specified, a `Executors.defaultThreadFactory()` is used, that creates threads to all be in the same `ThreadGroup` and with the same `NORM_PRIORITY` priority and non-daemon status. By supplying a different `ThreadFactory`, you can alter the thread’s name, thread group, priority, daemon status, etc.”

Fork / Join Framework

Java 6 works fine with Scala, but Java 7 has additional features and improved performance. An improved version of one of those features, the fork/join framework, is included with Scala and automatically works with Java 6 and 7.

The fork/join framework is an implementation of the `ExecutorService` interface that helps programmers take advantage of multiple cores and processors. It is intended to be used with programs that can be recursively broken into smaller pieces. The result is that all the available processing power is efficiently used when running the application.

Because ScalaDoc does not parse Javadoc comments, the excellent documentation provided with ForkJoinPool does not appear in the Akka ScalaDoc.

All `ExecutorServices` manage a thread pool and distribute tasks to worker threads. The fork/join framework uses a work-stealing algorithm, which means that when worker threads complete their task, they can steal tasks from the inbound queues of other busy threads. By default, Scala parallelism and the Akka `ActorSystem` uses this

`Executor`. You can control the Akka `ActorSystem` parameters by setting values in an Akka configuration file. You can also create an instance of `ForkJoinPool` without involving `ActorSystem`.

You can also create an instance of this class by using any of the following constructors. The specified parallelism specifies the initial number of threads. As execution continues, `ForkJoinPool` may increase the number of threads to maintain the desired parallelism. The `j.u.c.` version does not provide an upper limit on the number of threads, however the Scala version does.

JVM Command Line Settings

Dave Dice at Oracle suggests that if you're doing large-scale concurrency on Java 7+, you should be using `-XX:+UseCondCardMark`. Of course, other settings such as `-server`, `-Xmx`, `-Xms`, `-XX:PermSize` and `-XX:MaxPermSize` should also be considered.

Better Concurrency Options

Ordered by increasing complexity & overhead:

1. **java.util.concurrent (`j.u.c.`)**
2. **Scala parallel collections†**
3. **Scala futures†***
4. **Akka/Scala dataflow***
5. Software Transactional Memory (STM)
6. Love child: Akka Agents
7. **Akka actors (Java & Scala)**

You can mix & match the above. We will only discuss the options in bold above.

* Covered in my book

† Covered in my April 23 InfoQ article Benchmarking JVM Concurrency Options for Java, Scala and Akka.

Akka Enhancements

Akka provides a configurable factory for the Akka `Executor` implementation. It is easy to work with, flexible and miserly with resource consumption.

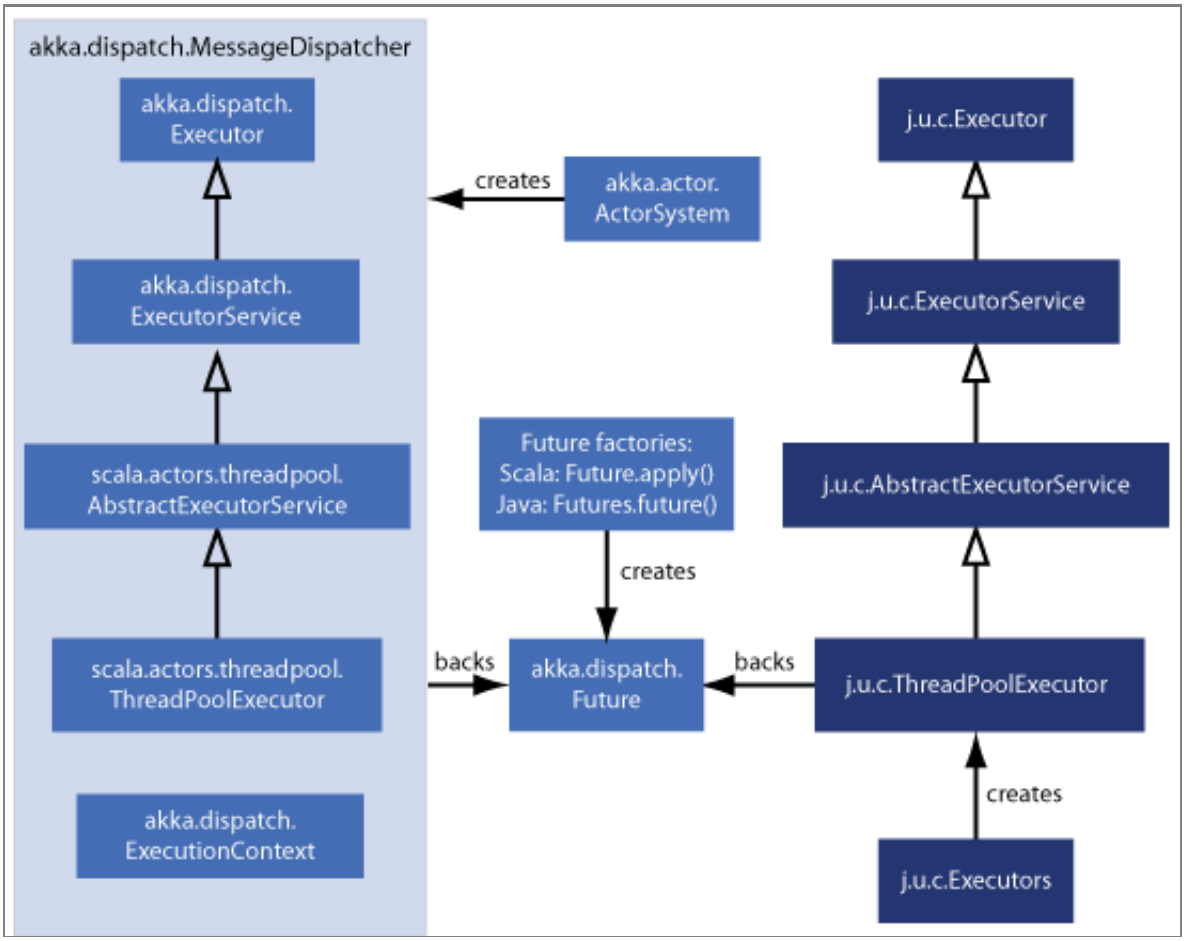
Akka's `ActorSystem.shutdown()` that shuts down all aspects of the Akka system, including the dispatcher. `Actors` and `Futures` become inoperative after calling this method. Akka's `Executor` manages threading resources for minimum impact. It does not manage threadpools created directly by `j.u.c.` calls, only threadpools configured and managed by Akka.

j.u.c. ExecutorService

`akka.dispatch.ExecutionContext` is a thin wrapper around a `j.u.c.Executor` implementation. `j.u.c.ExecutorService` is a concrete implementation of the `Executor` interface, and an `ExecutorService` instance is required in order to create a `Future`. Each `Future` is managed by an `Executor`.

Akka Dispatcher and j.u.c.

This is an artistic rendering of the Akka dispatcher mechanism and `j.u.c`. Some liberties were taken.



ActorSystem

- Akka run-time library
- Supports all of Akka, not just actors
- Heavyweight
- Flexible, configurable

Default Akka ActorSystem

If your program uses `Actors`, or if you want to easily configure the behavior of the threadpool, Akka provides you an easy way of configuring its default dispatcher. The default dispatcher is an instance of `akka.dispatch.MessageDispatcher` that has the `ExecutionContext` trait mixed in.

```
import akka.actor.ActorSystem;
implicit val system = ActorSystem("MySystem")
```

Akka's default dispatcher is highly optimized, and is based on an `ExecutorService` that has regular threads which are returned to the pool when idle. The default Akka threadpool shuts down after a short period of inactivity, and restarts when required. It also has an implementation of `RejectedExecutionHandler` that throws an exception if `shutdown` is attempted twice; you might like or dislike this feature.

For more information on how to configure the Akka system, read the Akka Configuration documentation.

```
val configString = """
akka {
  // dumps out configuration onto console when enabled and
  // loglevel >= "INFO"
  logConfigOnStart=on
  stdout-loglevel = "WARNING" # startup log level
  loglevel = "INFO" # loglevel once ActorSystem is started
  actor {
    my-dispatcher {
      type = Dispatcher,
      max-pool-size-max = 64, # default
      throughput = 5, # default
      core-pool-size-max = 64 # default
    }
  }
}"""
implicit val system = ActorSystem("actorSystem", ConfigFactory.parseString(configString))
```

If `application.conf` is present on the `classpath`, you can cause it to be ignored and use the default configuration. Remember that the contents of the sbt `src/resources` and `test/resources` directories are automatically placed on the classpath.

```
val system = ActorSystem("default", ConfigFactory.defaultReference)
```

You can also specify changes to the default configuration:

```
val str = "akka { daemonic = on }"
val system = ActorSystem("default", ConfigFactory.parseString(str))
```

Custom Akka Dispatcher With Fallback

You can specify a configuration file, and provide overrides of default values if the file is not present this way:

```
val system = ActorSystem("default", ConfigFactory.parseFile("blah.conf").withFallback(ConfigFactory.parseString(configStrin
g)))
```

ActorSystem Methods and Properties

The `Executor` provided by Akka's `ActorSystem` provides extra functionality beyond that provided by the `Executor` created with `j.u.c`.

Factories

`ActorSystem()` – Created with name "`default`" and default properties.

`ActorSystem(name: String)` – Created with specified name and default properties.

`ActorSystem(name: String, config: Config)` – Created with specified name and default properties that are selectively overridden by `config`.

Methods

`awaitTermination()` – Blocks the current thread until all other threads are idle and the `ActorSystem` terminates; `isTerminated` will be `true` after this method returns. This method typically takes 50ms to return after all threads have stopped.

`logConfiguration()` – outputs Akka system configuration to console (hundreds of lines).

`shutdown()` – Requests `ActorSystem` termination.

Properties

`isTerminated` – Boolean indicating if this `ActorSystem` has terminated.

`log` – Provides access to the `akka.event.LoggingAdapter`. See the Akka documentation for Scala and Java for more information.

`name` – Name of this `ActorSystem`.

Shutdown Akka System

Shut down the Akka system this way:

```
system.shutdown()
```

Sample Code

Here is sample code that demonstrates most of the above methods and properties in action.

```
import akka.actor.ActorSystem
import akka.event.Logging
implicit val system = ActorSystem()
system.logConfiguration() // outputs hundreds of lines
println(s"ActorSystem name=${system.name}")
println(s"Before shutdown: isTerminated=${system.isTerminated}")
system.shutdown() // allow System.exit()
println(s"After shutdown: isTerminated=${system.isTerminated}")
Thread.sleep(500)
println("After isTerminated=" + system.isTerminated)
```

After the Akka configuration is displayed, the remaining output is:

```
ActorSystem name=default
Before shutdown: isTerminated=false
After shutdown: isTerminated=false
After awaitTermination(): isTerminated=true
```

Akka Configuration

If you use the Executor provided by Akka's ActorSystem instead of a plain Executor created with j.u.c., then you may need to configure the Akka Executor behavior. This section details the settings that can be specified in application.conf. Only the default values of settings that pertain to futures are shown here. This information is also described in the Akka Configuration documentation.
See `src/main/resources/application.conf` in the `courseNotes` project for a commented example of an Akka configuration file that shows the default value of every parameter.

Custom Akka Dispatcher

Default values for Akka dispatcher configuration properties are defined in reference.conf, provided in akka-actor.jar. You can obtain a customized instance of the default Akka dispatcher by overriding selected properties. In the following code examples, the overridden properties are specified in configString. If you want to see all properties and their values when the system starts, uncomment `logConfigOnStart=on`.

3-2 Parallel Collections

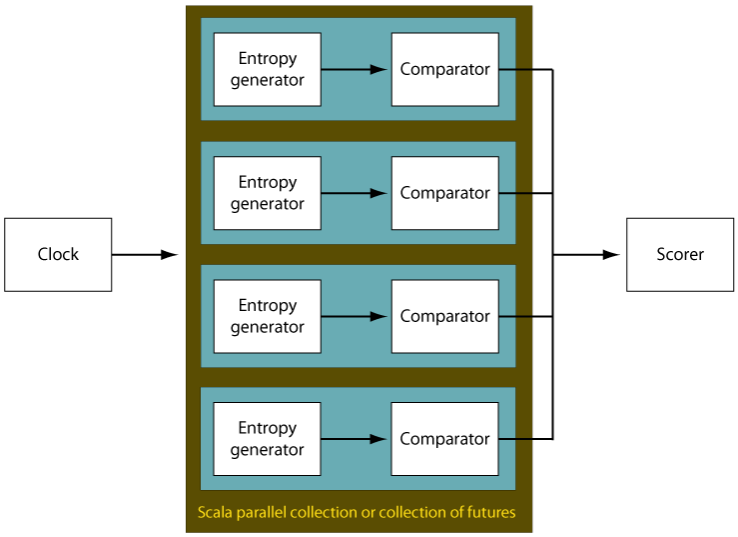
`ScalaCourses.com` / `scalaIntermediate` / `ScalaCore` / `scalaMulti` / `scalaParallel`

Scala parallel collections are:

- Very easy to use
- Not configurable
- Blocking
- Composable
- Easy to reason with
- Benefits from ongoing improvements to `ForkJoinPool` (Scala uses `jsr166y`, so `ForkJoinPool` is available even if you use JDK 6.)

Important: the tasks that you execute must be idempotent (have no side effects).

Use Case: Simple Simulation



Regular Collections Can Be Transformed Into Parallel Collections

The following two computations are characterizations of the type of tasks which need to be performed simultaneously as many times as possible: one is computationally intensive (CPU bound), and the other is I/O intensive (I/O bound). I also show the `time` higher-order function that was introduced earlier.

```
def calculatePiFor(decimals: Int): Double = {
  var acc = 0.0
  for (i <- 0 until decimals)
    acc += 4.0 * (1 - (i % 2) * 2) / (2 * i + 1)
  acc
}

/** Simulate a CPU-bound task */
def cpuIntensive(intensity: Int): Any = { calculatePiFor(intensity) }

/** @param minDelay minimum time (ms) to sleep per invocation
  * @param maxDelay maximum time (ms) to sleep per invocation
  * @param nrOfFetches number of times to repeatedly sleep then run a short computation per invocation */
private def simulateSpider(minDelay: Int, maxDelay: Int,  nrOfFetches: Int) {
  for (i <- 0 until nrOfFetches) {
    // simulate from minDelay to maxDelay ms latency
    Thread.sleep(random.nextInt(maxDelay-minDelay) + minDelay)
    calculatePiFor(0, 50) // simulate a tiny amount of computation
  }
}

/** Simulate an IO-bound task (web spider) */
def ioBound(): Any = simulateSpider(5, 30, fetchCount)

def time(block: => Any): Any = {
  val t0 = System.nanoTime()
  val result: Any = block
  val elapsedMs = (System.nanoTime() - t0) / 1000000
  println("Elapsed time: " + elapsedMs + "ms")
  result
}
```

Lets use the REPL to invoke these computations using a regular map:

```
scala> time((1 to 10000).toList.map(n => cpuIntensive(n)))
 Elapsed time: 4453ms
res27: Any = List(4.0, 2.666666666666667, 3.466666666666667, 2.8952380952380956, 3.3396825396825403, 2.9760461760461765, 3.
2837384837384844, 3.017071817071818, 3.2523659347188767, 3.0418396189294032, 3.232315809405594, 3.058402765927333, 3.218402
7659273333, 3.0702546177791854, 3.208185652261944, 3.079153394197428, 3.200365515409549, 3.0860798011238346, 3.194187909231
9425, 3.09162380666784, 3.189184782277596, 3.0961615264636424, 3.1850504153525314, 3.099944032373808, 3.1815766854350325, 3
.1031453128860127, 3.1786170109992202, 3.1058897382719475, 3.1760651768684385, 3.108268566698947, 3.1738423371907505, 3.110
350273698687, 3.1718887352371485, 3.112187242699835, 3.1701582571925884, 3.1138202290235744, 3.1686147495715193, 3.11528141
6238186, 3.167229468186238, 3.116596556793833, 3.1659792728...
```

Lets use the REPL to invoke these computations using a parallel collection:

```
scala> time((1 to 10000).toList. par .map(n => cpuIntensive(n)))
 Elapsed time: 139ms
res20: Any = ParVector(4.0, 2.666666666666667, 3.466666666666667, 2.8952380952380956, 3.3396825396825403, 2.976046176046176
5, 3.2837384837384844, 3.017071817071818, 3.2523659347188767, 3.0418396189294032, 3.232315809405594, 3.058402765927333, 3.2
184027659273333, 3.0702546177791854, 3.208185652261944, 3.079153394197428, 3.200365515409549, 3.0860798011238346, 3.1941879
092319425, 3.09162380666784, 3.189184782277596, 3.0961615264636424, 3.1850504153525314, 3.099944032373808, 3.18157668543503
25, 3.1031453128860127, 3.1786170109992202, 3.1058897382719475, 3.1760651768684385, 3.108268566698947, 3.1738423371907505,
3.110350273698687, 3.1718887352371485, 3.112187242699835, 3.1701582571925884, 3.1138202290235744, 3.1686147495715193, 3.115
281416238186, 3.167229468186238, 3.116596556793833, 3.16597...
```

As you can see, parallel collections are much faster, and code changes are minimal.

Scala provides many types of parallel collections:

- `ParArray`
- `ParVector`
- `mutable.ParHashMap`
- `mutable.ParHashSet`
- `immutable.ParHashMap`
- `immutable.ParHashSet`
- `ParRange`
- `ParTrieMap`

Reducing Results

Parallel collections will result in a collection of results, and it is common to require a reduction step (the 'reduce' of 'map/reduce'). Common reductions include maximum, minimum, average, etc. Scala collections give you several choices: `fold` (also known as `foldLeft`, `foldRight`. This example merely counts the number of results that have a six in the answer. Here is the first attempt

```
scala> (1 to 10000).toList.par.map { n => cpuIntensive(n) }.filter(_.toString.contains("6")).getClass.getName
res44: String = scala.collection.parallel.immutable.ParVector

scala> (1 to 10000).toList.par.map { n => cpuIntensive(n) }.filter(_.toString.contains("6")).toVector.getClass.getName
res51: String = scala.collection.immutable.Vector
```

The following error occurs because we declared the result returned by `cpuIntensive` to be `Any`:

```
scala> (1 to 10000).toList.par.map { n => cpuIntensive(n) }.filter(_.toString.contains("6")).toVector.reduce { (acc, n) =>
acc + 1 }
<console>:10: error: type mismatch;
 found   : Int(1)
 required: String
             (1 to 10000).toList.par.map { n => cpuIntensive(n) }.filter(_.toString.contains("6")).toVector.reduce { (acc,
 n) => acc + 1 }
```

We can coerce the compiler into viewing the result as a `Vector[Double]`, which is actually what the truth is:

```
scala> (1 to 10000).toList.par.map { n => cpuIntensive(n) }.filter(_.toString.contains("6")).toVector.asInstanceOf[Vector[D
ouble]].reduce { (acc, n) => acc + 1 }
res48: Double = 7368.666666666667
```

That is a long line of code! Let's break it into chunks:

```
scala> val parList = (1 to 10000).toList.par
parList: scala.collection.parallel.immutable.ParSeq[Int] = ParVector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47
, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 7
8, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 1
07, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131,
 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 15
6, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 1...

scala> val filteredResult = parList.map { n => cpuIntensive(n) }.filter(_.toString.contains("6"))
filteredResult: scala.collection.parallel.immutable.ParSeq[Any] = ParVector(2.666666666666667, 3.466666666666667, 2.8952380
952380956, 3.3396825396825403, 2.9760461760461765, 3.2523659347188767, 3.0418396189294032, 3.058402765927333, 3.21840276592
73333, 3.0702546177791854, 3.208185652261944, 3.200365515409549, 3.0860798011238346, 3.09162380666784, 3.189184782277596, 3
.0961615264636424, 3.1815766854350325, 3.1031453128860127, 3.1786170109992202, 3.1760651768684385, 3.108268566698947, 3.110
350273698687, 3.112187242699835, 3.1686147495715193, 3.115281416238186, 3.167229468186238, 3.116596556793833, 3.16597927284
32157, 3.117786501758878, 3.1648453252882898, 3.118868313794037, 3.163812134018756, 3.1198560900627124, 3.1628668427508844,
 3.1207615795929895, 3.161998692995051, 3.121594652591011, ...

scala> val answer = filteredResult.toVector.asInstanceOf[Vector[Double]].reduce { (acc, n) => acc + 1 }
answer: Double = 7368.666666666667
```

Exercise

This exercise should take at least an hour.

Given a large number of 'monkeys', how many characters of the following text could they match by randomly generating 1000 characters each?
`"I thought I saw a lolcat! I did, I did see a lolcat!"`

Write a little Scala program that uses Scala parallel collections to run a simulation that generates random characters. The simulation should loop through 10,000 iterations, and compare the result of each iteration with the desired result. Display the longest match.

Hints:

1. This is a map/reduce pattern.
2. Here is a method that returns the longest common substring of two strings, starting from the beginning of the string. FYI, here is an explanation of what a `view` is, and here is an explanation of what `tupled` does. The `takeWhile` combinator iterates through a list until it finds one element that doesn't satisfy the predicate, and returns the elements that did satisfy the predicate. In other words, it returns the longest prefix such that every element satisfies the predicate. If you would like to understand this method better, play with portions of it in the REPL or a Scala worksheet.

   ```
   def matchSubstring(str1: String, str2: String): String =
     str1.view.zip(str2).takeWhile(Function.tupled(_ == _)).map(_._1).mkString
   ```

3. Here is a method that generates a random string for a restricted alphabet:

```
val random = util.Random

val allowableChars = """ !.,;'""" + (('a' to 'z').toList ::: ('A' to 'Z').toList ::: (0 to 9).toList).mkString

def randomString(n: Int) = (1 to n).map { _ =>
  val i = random.nextInt(allowableChars.length-1)
  allowableChars.substring(i, i+1)
}.mkString
```

3-3 Partial Functions

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaMulti / scalaPartialFns

A *partial function* is a `Function` which is defined only for a subset of the possible values of its arguments.

To set up for an example, let's see what keys are defined in the system properties and environment variables.

```
scala> System.getProperties.keySet
res1: java.util.Set[Object] = [java.runtime.name, sun.boot.library.path, java.vm.version, java.vm.vendor, java.vendor.url,
path.separator, java.vm.name, file.encoding.pkg, user.country, sun.java.launcher, sun.os.patch.level, java.vm.specification
.name, user.dir, java.runtime.version, java.awt.graphicsenv, java.endorsed.dirs, os.arch, java.io.tmpdir, line.separator, j
ava.vm.specification.vendor, os.name, sun.jnu.encoding, java.library.path, java.specification.name, java.class.version, sun
.management.compiler, os.version, user.home, user.timezone, scala.home, java.awt.printerjob, file.encoding, java.specificat
ion.version, scala.usejavacp, java.class.path, user.name, java.vm.specification.version, sun.java.command, com.amazonaws.sd
k.disableCertChecking, java.home, sun.arch.data.model, use...

scala> System.getenv.keySet
res2: java.util.Set[String] = [SSH_AGENT_PID, JAVA_HOME, LESSCLOSE, SESSION_MANAGER, XDG_SESSION_COOKIE, GDMSESSION, PWD, E
XINIT, LS_OPTIONS, futures, GOOGLE_CLIENT_EMAIL, NLSPATH, XDG_MENU_PREFIX, TEXTDOMAIN, books, SMTP_SSL, PATH, XDG_CONFIG_DI
RS, GOOGLE_APP_KEY, PAYPAL_RECEIVER_EMAIL, XAUTHORITY, GDM_LANG, WINDOWPATH, GNOME_KEYRING_CONTROL, GTK_MODULES, meetup, bo
okish, secure, TEXTDOMAINDIR, COLORTERM, XDG_DATA_DIRS, hanuman, DATABASE_MAX_CONNS_PER_PARTITION, mirror, training, LOGNAM
E, experiments, GPG_AGENT_INFO, sbtest, sbusers, domain, _, GLADE_MODULE_PATH, SSH_AUTH_SOCK, PLAY21_HOME, PUB_CONFIG, ew,
DBUS_SESSION_BUS_ADDRESS, GLADE_CATALOG_PATH, GNOME_KEYRING_PID, LANGUAGE, JMETER_HOME, DESKTOP_SESSION, mocc, DISPLAY, USE
R, slick, GLADE_PIXMAP_PATH, new, ej, LS_COLORS, SUPERUSER...

scala> val home = Option(System.getenv("JAVA_HOME"))
home: Option[String] = Some(/usr/lib/jvm/java-7-oracle)
```

Now we can define a partial function that returns the value of an environment variable if defined:

```
val env = new PartialFunction[String, String] {
  private def value(name: String) = Option(System.getenv(name))

  def apply(name: String) = value(name).get

  def isDefinedAt(name: String) = value(name).isDefined
}
```

Partial functions are not capable of returning results for all possible values on input, and partial functions formalize this contract such that the validity of input values can be queried without running the partial function. Because it is uncertain if a partial function will be able to return a value for arbitrary input, the returned value is expresses as an `Option`. Note that the definition of `env` above is parametric - the first parametric type (`String`) pertains to the input. The second parametric type (`String`) pertains to the returned value - but this really means that the returned value is `Option[String]`. Partial functions always have only two types: input and output. If several values must be passed into a partial function, tuples, case classes or even regular classes can be used. If you use a regular class, be sure to define `unapply` in the companion object.

Exercise

Why must `unapply` be defined in a companion class passed into a partial function?

Now let's use the `env` partial function:

```
scala> env.isDefinedAt("JAVA_HOME")
res3: Boolean = true

scala> env("JAVA_HOME")
res4: String = /usr/lib/jvm/java-7-oracle

scala> env.isDefinedAt("x")
res5: Boolean = false

scala> env("x")
java.util.NoSuchElementException: None.get
```

Shorthand Notation

There is a shorthand notation (aka *syntactic sugar*) for Scala partial functions, which is simply to use one or more `case` statements. Let's rewrite the partial function introduced above using this syntactic sugar:

```
scala> var env: PartialFunction[String, String] = { case name: String => System.getenv(name) }
env: PartialFunction[String,String] = <function1>
```

The type declaration is required only if the partial function is not defined as an anonymous function, where the higher order function would have been able to provide hints to the compiler about the acceptable type signature of the lambda.

Now let's introduce a scenario where partial functions simplify a problem. Imagine a software product can be configured to use multiple JVMs, and it looks through environment variables for the most desired JVM. Here are two ways of writing this, using different syntaxes:

```
scala> List("JAVA_HOME_8", "JAVA_HOME_7", "JAVA_HOME_6", "JAVA_HOME_5", "JAVA_HOME").collect(env)
res6: List[String] = List(/usr/lib/jvm/java-7-oracle)

scala> List("JAVA_HOME_8", "JAVA_HOME_7", "JAVA_HOME_6", "JAVA_HOME_5", "JAVA_HOME") collect env
res7: List[String] = List(/usr/lib/jvm/java-7-oracle)
```

`collect` builds a new collection by applying a partial function to all elements of this list on which the function is defined. The partial function filters and maps the list. `Collect` returns the new list resulting from applying the partial function to each element on which it is defined. The order of the elements is preserved.

In other words, collect is like map, but it applies a partial function to each element of a list, instead of a normal function. If the partial function is not defined for an element input, that element is ignored. You could accomplish the same thing using `for` comprehensions or `map`/`flatMap` and `filter`, or even map with match, but partial functions are cleaner and more efficient.

Scala's future, actor and the Akka library make extensive use of partial functions.

Composing Partial Functions

`xs.collect(pf)` is like `xs.filter(pf.isDefinedAt(_)).map(pf)`, so it selects items matching a pattern and returns some function of those items.

```
scala> val sample = 1 to 10
sample: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val isEven: PartialFunction[Int, String] = { case x if x % 2 == 0 => x+" is even" }
isEven: PartialFunction[Int,String] = <function1>

scala> val evenNumbers = sample collect isEven
evenNumbers: scala.collection.immutable.IndexedSeq[String] = Vector(2 is even, 4 is even, 6 is even, 8 is even, 10 is even)

scala> val isOdd: PartialFunction[Int, String] = { case x if x % 2 == 1 => x+" is odd" }
isOdd: PartialFunction[Int,String] = <function1>

scala> val oddNumbers = sample collect isOdd
oddNumbers: scala.collection.immutable.IndexedSeq[String] = Vector(1 is odd, 3 is odd, 5 is odd, 7 is odd, 9 is odd)
```

The `orElse` method allows another partial function to be chained (composed) to handle input outside the declared domain of a partial function.

```
scala> val numbers = sample map (isEven orElse isOdd) // map is interchangeable here
numbers: scala.collection.immutable.IndexedSeq[String] = Vector(1 is odd, 2 is even, 3 is odd, 4 is even, 5 is odd, 6 is ev
en, 7 is odd, 8 is even, 9 is odd, 10 is even)
```

Case Sequences Are Partial Functions

Partial functions are of type `PartialFunction`, which is a subclass of `Function1`. If you read the ScalaDoc, this might not be obvious unless you know that another way of writing `Function1` is `(A) ⇒ B`. Since `Function1` is a function of 1 parameter, then so is `PartialFunction`.

Case sequence is a synonym for partial function

This means that a sequence of case statements is implemented as a subclass of `Function1`. In other words, whenever a `Function1` is accepted, you can supply one or more case statements. You can use this knowledge to write more succinct code. For example, here is some verbose code:

```
val list = List(Some(1), None, Some(3))
list map { item =>
  item match {
    case Some(x) => x
    case None => default
  }
}
```

Instead, write this more succinct code:

```
list map {
  case Some(x) => x
  case None => default
}
```

Example of Case Sequences and Pattern Matching In a For Comprehension

This code is provided in the `courseNotes` directory as `partialFun.scala`

```
case class Attendee(name: String, id: Long, knowledge: Option[String])

object PartialFun extends App {
  val attendees = List(
    Attendee("Fred", 1, None),
    Attendee("Lisa", 2, Some("Akka")),
    Attendee("You",  3, Some("Scala"))
  )

  for {
    you <- attendees.collect { case attendee if attendee.name=="You" => attendee }
    yourKnowledge <- you.knowledge orElse Some("Nothing")
  } yield {
    println(s"You know $yourKnowledge")
  }
}
```

Let's compare this to similar code using a guard instead of `collect`. This code is simpler:

```
  for {
    attendee <- attendees if attendee.name=="You"
    yourKnowledge <- attendee.knowledge orElse Some("Nothing")
  } yield {
    println(s"You know $yourKnowledge")
  }
```

The moral of this example is that you should realize that Scala provides many ways of doing things, and normally simplest is best.

Parametric 'With' Pattern Revisited

Let's use case sequences (aka partial functions) with the `With` pattern we learned in the Parametric Types lecture.

```
def withT(t: T)(operation: T => Unit):  Unit = { operation(t) }

case class Blarg(i: Int, s: String)

def handle(blarg: Blarg): Unit = withT(blarg) {
  case Blarg(0, s) => println("i is 0")
  case Blarg(i, "triple") => println("s is triple")
  case whatever => println(whatever)
}
```

Let's try out this code:

```
scala> handle(Blarg(1, "blarg"))
Blarg(1,blarg)

scala> handle(Blarg(1, "triple"))
s is triple

scala> handle(Blarg(0, "triple"))
i is 0
```

Unfortunately, there is no way to use implicits with case sequences, instead, you must resort to the more verbose `match` construct.

```
withT(Blarg(1, "blarg ")) {  implicit  blarg =>
  match {
    case Blarg(0, s) => println("i is 0")
    case Blarg(i, "triple") => println("s is triple")
    case whatever => println(whatever)
  }
}
```

Exercise

This exercise should take at least an hour. The body mass index (BMI), or Quetelet index, is a measure for human body shape based on an individual's weight and height. It was devised between 1830 and 1850 by Adolphe Quetelet of Belgium. A simplistic body mass index definition is the individual's body mass divided by the square of their height. Better computations of BMI involve statistical lookups.

This Github project consists of a single PHP web page that accepts some variables and computes BMI: `https://github.com/pimteam/BMI-Calculator`. As you can see from examining the source code, you can specify English units or metric. The script is live at `http://calendarscripts.info/overweight-calculator.html`

Your task is to define a partial function that accepts three values: units, age, height and weight. If units is `"English"`, then you will need to obtain the values as a string, and parse out feet and inches. Because these calculations assume an adult, the value of the age parameter must be 18 or more in order for the partial function to be defined; the computation also becomes meaningless for people over 100 years of age. Write a console app that performs a POST to the web page and parse the result, then displays the BMI on the console.

Hint:

1. Pass a tuple into a partial function like this:

```
scala> val myPartialFunction: PartialFunction[Tuple3[Int, Int, Int], String] = { case (a, b, c) => s"a=$a; b=$
b; c=$c" }
Some((1, 2, 3)) foreach { (a, b, c) => collect((a, b, c) => myPartialFunction(a, b, c)) }

scala> Some((1, 2, 3)).collect(myPartialFunction)
res1: Option[String] = Some(a=1; b=2; c=3)
```

   If you find that hard to read, consider this equivalent code:

```
scala> type pf3 = PartialFunction[Tuple3[Int, Int, Int], String]
defined type alias pf3

scala> val myPartialFunction: pf3 = { case (a, b, c) => s"a=$a; b=$b; c=$c" }
myPartialFunction: pf3 = <function1>

scala> Some((1, 2, 3)) collect myPartialFunction
res2: Option[String] = Some(a=1; b=2; c=3)
```

2. If you want to operate on a single variable, instead of a collection, you can use the partial function's `isDefinedAt` method, then call the method, like this:

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

scala> myPartialFunction.isDefinedAt(tuple)
res3: Boolean = true

scala> myPartialFunction(tuple)
res4: String = a=1; b=2; c=3
```

Once your console application prints the BMI for a single person, modify your program to read in data for several people from a file.

3-4 Futures

`ScalaCourses.com` / `scalaIntermediate` / `ScalaCore` / `scalaMulti` / `scalaFutures`

A `Future` is a container which will assume a value when a computation completes or a result becomes available. This lecture pertains to the Future class and companion object in the `scala.concurrent` package. We will not discuss the `Future` trait or `Futures` object in the `scala.actors` package in this lecture. Java 7's futures are neither efficient nor composable.

Do not confuse the Future class & companion object with the Actor Future trait & Futures object, or the deprecated parallel.Future trait

Futures:

- Manage concurrency
- Higher-level than j.u.c. primitives
- Immutable
- Inexpensive
- Provide fast, asynchronous, non-blocking parallel code, if you write your code properly.
- Can be preset to a value or exception when created.
- Blocking and non-blocking operations are supported

Futures and Promises

A `Promise[T]` is writable once. A `Future[T]` is readable many times. `Promises` can be created already completed, or can be completed at another time, thus the two states of a `Promise` are *pending* or *completed*. A `Promise`'s state is represented by an instance of `Try[T]`. Invoking `Promise.complete` triggers a transition from state `Pending` to `Completed`.

A Promise completes a Future

A `Future` starts an asynchronous computation and can be read many times. You can think of a `Promise` as being the write-side of a `Future`, and of a `Future` as being the read-side of a `Promise`.

`Futures` require an `ExecutionContext`, which is backed by a threadpool. Scala provides a default implementation, available by importing `concurrent.ExecutionContext.Implicits.global`.

```
scala> import concurrent._
import concurrent._

scala> import concurrent.ExecutionContext.Implicits.global
import concurrent.ExecutionContext.Implicits.global

scala> val promise1, promise2, promise3, promise4 = Promise[String]()
promise1: scala.concurrent.Promise[String] = scala.concurrent.impl.Promise$DefaultPromise@64eb29e1
promise2: scala.concurrent.Promise[String] = scala.concurrent.impl.Promise$DefaultPromise@b0aa68a
promise3: scala.concurrent.Promise[String] = scala.concurrent.impl.Promise$DefaultPromise@295e3857
promise4: scala.concurrent.Promise[String] = scala.concurrent.impl.Promise$DefaultPromise@68c4ecf8

scala> promise1.isCompleted
res1: Boolean = false

scala> promise3.isCompleted
res2: Boolean = true

scala> promise1.future.onSuccess { case value => println(s"Promise1 value=$value") }

scala> promise1.success("Hi there")
res3: promise1.type = scala.concurrent.impl.Promise$DefaultPromise@64eb29e1
 Promise1 value=Hi there
```

The `onComplete` callback will be executed if the future has already been completed. Similarly, one of `onSuccess` or `onFailure` will also be executed.

```
scala> promise1.isCompleted
res4: Boolean = true
```

Successful completion of a `Promise` to a preset value is easy to establish:

```
scala> val promise2 = Promise.successful(123)
promise2: scala.concurrent.Promise[Int] = scala.concurrent.impl.Promise$KeptPromise@d1ff12e
```

s we discussed in the previous course, `NoStackTrace` can increase efficiency if it is used to define an object or a class, but you lose stack traces which might be useful for debugging errors later on, leading to questions like "what created this `Throwable`?" If you are using `Futures` and `Promises` and therefore your program switches threads at lot, the cost of populating the stack trace might not be noticeable. John Rose of Oracle wrote an interesting article that discusses this topic.

```
scala> import util.control.NoStackTrace
import util.control.NoStackTrace

scala> class ExceptTrace(msg: String) extends Exception(msg) with NoStackTrace
defined class ExceptTrace

scala> promise3.failure(new ExceptTrace("Boom!"))
res5: promise3.type = scala.concurrent.impl.Promise$DefaultPromise@b0aa68a
```

The above is efficient; only one class is defined, an instance will be created each time one is required. If your exceptions are invariant in all their properties, 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

scala> val promise4 = Promise.failed(TheExceptTrace)
promise4: scala.concurrent.Promise[Nothing] = scala.concurrent.impl.Promise$KeptPromise@5f2adaa8
```

Let's examine the value of the completed `Future` that is associated with each `Promise`:

```
scala> promise1.future.value
res6: Option[scala.util.Try[String]] = Some(Success(Hi there))

scala> promise4.future.value
res7: Option[scala.utilTry[Nothing]] = Some(Failure(TheExceptTrace$: Boom!))
```

Let's make a `Future` now, which will contain the contents of a web page once the fetch is complete:

```
scala> import io.Source
import scala.io.Source

scala> val f = Future(Source.fromURL("http://scalacourses.com"))
f: scala.concurrent.Future[scala.io.BufferedSource] = scala.concurrent.impl.Promise$DefaultPromise@71127eac

scala> f.isCompleted
res8: Boolean = true
```

Futures have two states: *completed* and *not completed*. The `isCompleted` method lets you know the state. Once the future has completed, you can retrieve its value several ways. Lets start by feeling out way forward using the REPL:

```
scala> f.value
res9: Option[scala.util.Try[scala.io.BufferedSource]] = Some(Success(non-empty iterator))

scala> f.value.get
res10: scala.util.Try[scala.io.BufferedSource] = Success(non-empty iterator)

scala> f.value.get.get
res11: scala.io.BufferedSource = non-empty iterator

scala> f.value.get.get.mkString
res12: String =
"

<!DOCTYPE html>
<html lang="en">
<head>
  <title>Welcome to ScalaCourses.com</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <script src='/webjars/jquery/1.9.1/jquery.min.js' type='text/javascript'></script>
  <script src='/webjars/jquery-ui/1.10.2/ui/minified/jquery-ui.min.js' type='text/javascript'></script>
  <script src='/webjars/bootstrap/2.3.2/js/bootstrap.min.js' type='text/javascript'></script>
  <script src='/webjars/bootstrap-datepicker/1.0.1/js/bootstrap-datepicker.js' type='text/javascript'></script>
  <script src='/assets/javascripts/jwerty.js' type="text/javascript"></script>
  <script src='/javascriptRoutes' type="text/javascript"></script>
  <link href='/assets/images/favicon.ico' rel="shortcut icon" type="image/x-ic...
```

Obviously this is not how you should retrieve the value from a future, but it does show you some useful information.

- `f.value` returns `None` until the future has completed.
- `f.value.get` returns a `Try` object which can hold one of two types of results: a `Success` or a `Failure`. This is rather like how `Either` works, except `Try` is specialized such that the left side can only hold a `Throwable`.
- `f.value.get.get` unpacks the `Success` and returns an `iterator` (actually, a `BufferedSource`).
- `f.value.get.get.mkString` iterates through the `BufferedSource` and returns a String containing the contents of the URL.

There are several better ways of retrieving the value from the future. For example, we could use callbacks, however I have laid a trap for you:

```
scala> import scala.util.{Success,Failure}
import scala.util.{Success, Failure}

scala> val notContents = f onComplete {
  case Success(iterator) =>
    iterator.mkString

  case Failure(throwable) =>
    println(throwable.getMessage)
    ""
}
 notContents: Unit = ()
```

For Combinators

Here is an example of a Scala for-comprehension that uses `Futures`; the complete program is provided in `FutureFun.scala` as `ForComp1`.

```
val future4: Future[Int] = for {
  x <- Future(1+2)
  y <- Future(2+3)
  z <- Future(4+5)
} yield x + y + z
future4 onComplete {
  case Success(r) =>
    println(s"Success: $r")
    System.exit(0)
  case Failure(ex) =>
    println(s"Failure: ${ex.getMessage}")
    System.exit(0)
}
```

The result of the for-comprehension is a new `Future`, containing the result of adding the values of the other `Futures` together after they run to completion. The expression to the right of the <- symbol is known as a generator that creates bindings for the variable on its left. This for-comprehension has 3 generators, one for each of x, y and z. Note that if the `Future` that generates the `x` value fails, the `Future` that would have generated the `y` value will not be invoked. Similarly, if the `Future` that generates the `y` value fails, the `Future` that would have generated the `z` value will not be invoked.

Here is another way to write the above, without the intermediate `future4` variable. See `ForComp2`:

```
( for {
    x <- Future(1+2)
    y <- Future(2+3)
    z <- Future(4+5)
  } yield x + y + z )  onComplete {
    case Success(r) =>
      println(s"Success: $r")
      System.exit(0)
    case Failure(ex) =>
      println(s"Failure: ${ex.getMessage}")
      System.exit(0)
  }
```

The Scala compiler rewrites the above as follows; the complete program is provided as `ForComp3`.

```
Future(1+2).flatMap { x =>
  Future(2+3).flatMap { y =>
    Future(4+5).map { z => x + y + z }
  }
} onComplete {
  case Success(r) =>
    println(s"Success: $r")
    System.exit(0)
  case Failure(ex) =>
    println(s"Failure: ${ex.getMessage}")
    System.exit(0)
}
```

As you can see, the body of the `yield` statement in the for-comprehension is used as the body of a functor passed to map, and the preceding assignments are used to form enclosing `flatMaps`.

If you add a conditional expression (known as a guard) to the for-comprehension, it is implemented via `withFilter`. Older versions of Scala used `filter` instead of `withFilter`. Here is a for-comprehension with a guard:

```
val sky = "blue"
val future4 = for {
  x <- Future(1+2)
  y <- Future(2+3)
  z <- Future(4+5)
  if sky == blue
} yield x + y + z
```

If `sky` is not `blue`, the for-comprehension will yield a `Future` containing a `NoSuchElementException` on the left. All futures will run to completion regardless of the result of the guard, however. The Scala compiler rewrites the above as:

```
val future4 = Future(1+2).withFilter {
  x => sky=="blue"
}.flatMap { x =>
  Future(2+3).flatMap{ y =>
    Future(3+4).map { z => x + y + z }
  }
}
```

Combinators

Futures are composable:

- Operations on a future or a collection of futures can be chained together without blocking.
- Transformations such as `map` and `flatMap` can be applied to individual futures.
- Collections of futures can be manipulated (for example, reduced).

Combinators for individual futures:

- `collect`, `flatMap`, `filter`, `foreach`, `map`, `mapTo`, `zip`
- `fallbackTo`, `recover`, `recoverWith`
- `failed`, `success`

Operations on collections of futures:

- `find`, `fold`, `reduce`, `sequence`, `traverse`
- `firstCompletedOf`

Callbacks - only useful for side effects:

- `andThen`
- `onComplete`, `onFailure`, `onSuccess`

Note that each `case` returns a `String`, so the type of `notContents` would be a `String` except that Future's callbacks do not return values. That means callbacks can only perform side effects, and they cannot be composed. If you want to get the results of a future, use a combinator like `map` and `flatMap`, or a `for` comprehension. Let's first try just printing out the first 200 characters of a URL using a for comprehension and a future:

```
scala> for {
 value <- Future(Source.fromURL("http://scalacourses.com"))
 contents = value.mkString
} println(contents.substring(0, math.min(200, contents.length)))

scala>

<!DOCTYPE html>
<html lang="en">
<head>
  <title>Welcome to ScalaCourses.com</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <script src='/webjars/jquery/1.9.1/jqu
```

Notice that the REPL prompt appeared, then a pause, then the output of the future appeared. This is not a bug.

What if we simply yield the contents and assign the value produced by the for to a variable?

```
scala> val xx = for {
  something <- Future(Source.fromURL("http://scalacourses.com"))
} yield something.mkString
xx: scala.concurrent.Future[String] = scala.concurrent.impl.Promise$DefaultPromise@5523cc24
```

Turns out we get a completed future back from the for comprehension, containing the contents, implemented as a `Promise` subclass known as `DefaultPromise`. That is how for comprehensions and map work by default: you get the same collection type out that was put in. Note that "inside" the `for` comprehension the future's value will be

available, however back in the mainline the future is difficult to work with.

Finally, we could block until the future has completed, and obtain the value with `Await.result`.

```
scala> import scala.concurrent.duration._
import scala.concurrent.duration._

scala> import java.util.concurrent.TimeUnit._
import java.util.concurrent.TimeUnit._

scala> Await.result(xx, Duration.Inf)
```

We can perform multiple side effecting actions:

```
xx andThen { // returned values are ignored, so side effects are the only reason for using andThen!
  case Success(value) => println(value.mkString)
  case Failure(throwable) => println(throwable.getMessage)
}
```

Collections of Futures

If you are writing a web crawler your program will need to fetch many URLs at once. We can have a collection of `Future`, like this:

```
scala> val urls = List("http://scalacourses.com", "http://micronauticsresearch.com", "http://mslinn.com")
urls: List[String] = List(http://scalacourse.com, http://micronauticsresearch.com, http://mslinn.com)

scala> val futures = urls.map(u => Future(Source.fromURL(u).mkString))
futures: List[scala.concurrent.Future[String]] = List(scala.concurrent.impl.Promise$DefaultPromise@76909d25, scala.concurre
nt.impl.Promise$DefaultPromise@103d24f3, scala.concurrent.impl.Promise$DefaultPromise@1e44d5d9)
```

Now let's identify web pages that have the word *free* in them:

```
scala> val fs = Future.sequence(futures)
fs: scala.concurrent.Future[List[String]] = scala.concurrent.impl.Promise$DefaultPromise@a6e1f4c

scala> for {
  allDone <- fs
  (url, future) <- urls zip futures
  contents <- future if contents.toLowerCase.contains("free")
} println(url)
http://scalacourses.com
```

This takes some explaining. There are two issues: we need to wait until all the futures have completed, so we make a sequence and simply wait for it to complete. Then we need to relate each of the original futures to the URL that they worked from - and zip builds a list of tuples of URL to `Future`. We then examine the value of the completed futures to see if they contain the word that we are looking for. If so, we print out the URL.

Partial Functions and collect

We could also use partial functions with collections of futures. Recall that `collect` is like `map`, but it accepts a partial function instead of a functor.

```
scala> futures
res13: List[scala.concurrent.Future[String]] = List(scala.concurrent.impl.Promise$DefaultPromise@6689ef79, scala.concurrent
.impl.Promise$DefaultPromise@69c92586, scala.concurrent.impl.Promise$DefaultPromise@14beb6ac)

scala> futures.collect { case x  => x.getClass.getName }
res14: List[String] = List(scala.concurrent.impl.Promise$DefaultPromise, scala.concurrent.impl.Promise$DefaultPromise, scal
a.concurrent.impl.Promise$DefaultPromise)

scala> futures.collect { case x  => x.value.getClass.getName }
res15: List[String] = List(scala.Some, scala.Some, scala.Some)

scala> futures.collect { case x  => x.value.get.getClass.getName }
res16: List[String] = List(scala.util.Success, scala.util.Success, scala.util.Success)

scala> futures.collect { case x  => x.value.get.get.getClass.getName }
res17: List[String] = List(java.lang.String, java.lang.String, java.lang.String)

scala> futures.collect { case x if x.value.get.get.toLowerCase.contains("free") => x.value.get.get }
```

Failure Propagation

For comprehensions and the `map`/`flatMap` combinators that they are built from silently discard errors, leaving you with valid results. Most of the time this is what you would like. Before we get into error handling, let's take a look at a naive optimization of the previous for comprehension. Note that the second URL references a non-existent domain.

```
scala> val urls2 = List("http://scalacourses.com",  "http://not_really_here.com" , "http://micronauticsresearch.com")
urls2: List[String] = List(http://not_really_here.com, http://micronauticsresearch.com, http://mslinn.com)
```

The future created to load the non-existant URL will fail, and the exception will be stored in the future - the exception will not be thrown:

```
scala> val futures2 = urls2.map(u => Future(Source.fromURL(u).mkString))
futures: List[scala.concurrent.Future[String]] = List(scala.concurrent.impl.Promise$DefaultPromise@7e3da0ac, scala.concurre
nt.impl.Promise$DefaultPromise@7fffbcf3, scala.concurrent.impl.Promise$DefaultPromise@6ece372b)
```

Now for the naive optimization. Let's say we want to define a method to do the search. The method accepts the word to search for, and a list of URLs to fetch. The logic is similar to what we had before.

```
scala> def urlSearch(word: String, urls: List[String]): Unit = {
  val futures2: List[Future[String]] = urls.map(u => Future(Source.fromURL(u).mkString))
  for {
     allDone <- Future.sequence(futures2)
    (url, future) <- urls2 zip futures2
    contents <- future if contents.toLowerCase.contains(word)
  } println(url)
}

scala> urlSearch("free", urls2)

scala> urlSearch("scala", urls2)
```

However, when we run this, no results are returned. The reason is because `Future.sequence` transforms the `List[Future[String]]` to a `Future[List[String]]`, and if any of the incoming futures fails, the resultant future (`allDone`) also fails, causing the for comprehension to terminate without printing anything. Here is a better way of

writing the `urlSearch` method:

```
scala> import concurrent.duration.Duration
import concurrent.duration.Duration

scala> def urlSearch(word: String, urls: List[String]): Unit = {
  val futures2: List[Future[String]] = urls.map(u => Future(Source.fromURL(u).mkString))
   Await.ready(Future.sequence(futures2), Duration.Inf) // block until all futures complete
  for {
    (url, future) <- urls2 zip futures2
    contents <- future if contents.toLowerCase.contains(word)
  } println(url)
}
```

This is an appropriate usage of `Await` because the sequence is created merely to allow us to know when all of the futures have completed. If one future fails, the others continue to run. The 2nd parameter to `Await.ready`, of type `Duration`, specifies that the main thread will wait forever for the sequence to complete. The `for` comprehension then combines the URLs with the contents of their web pages into tuples as before, and *for each of the futures that succeeded* their contents are examined to see if the desired word is present.

Scala futures always run to completion

```
scala> urlSearch("free", urls2)

scala> urlSearch("scala", urls2)

scala> http://mslinn.com
http://micronauticsresearch.com
```

BTW, you should probably not wait forever, and instead wait a finite maximum time, like 30 seconds. The following code only works if all URLs resolve without error. `Future.sequence` will return a failure as soon as any future passed to it fails. This means we need to catch exceptions and handle them. The easiest way to do that is to return a `Future[String]` containing empty string whenever a read error is encountered:

```
scala> import scala.concurrent.duration._
import scala.concurrent.duration._

scala> def readUrl(url: String): String = Source.fromURL(url).mkString
readUrl: (url: String)String

scala> def urlSearch(word: String, urls: List[String]): Unit = {
    val futures2: List[Future[String]] = urls.map { url =>
      Future(readUrl(url)).recoverWith {
        case e: Exception => Future.successful("") // catches all Exceptions
      }
    }
    val sequence: Future[List[String]] = Future.sequence(futures2)
    Await.ready(sequence, 30 seconds) // block until all futures complete, timeout occurs, or a future fails
    println("sequence completed: " + sequence.isCompleted) // false if timeout occurred
    for {
      (url, future) <- urls zip futures2
      contents <- future if contents.toLowerCase.contains(word)
    } println(s"'$word' was found in $url")
  }
urlSearch: (word: String, urls: List[String])Unit

scala> urlSearch("scala", urls2)
scala> 'scala' was found in http://mslinn.com
'scala' was found in http://micronauticsresearch.com
```

A word of warning: `Futures` were designed such that for comprehensions assign a value from a generator of a future until the future has completed. The `Future` combinators will also not execute their body until the future has completed. This means we could write:

```
def urlSearch(word: String, urls: List[String]): Unit = {
  val futures2: List[Future[String]] = urls.map(u => Future(Source.fromURL(u).mkString))
  for {
    (url, future) <- urls2 zip futures2
    contents <- future if contents.toLowerCase.contains(word)
  } println(url)
}
```

We could write the code this way, but we probably should not. That is because each future will wait forever, as if `Duration.Inf` had been specified. The version we wrote immediately above is therefore preferred.

Daemon Threads

If you run the program several times you will notice that it does not always print out all the output that it should. The reason for this is that Scala's global threadpool (this is what you get when you `import concurrent.ExecutionContext.Implicits.global`) uses `ForkJoinPool` with daemon threads. Daemon threads are terminated once the program's main thread has completed. If you want to ensure that the threads assigned to each Future run to completion you must either configure another threadpool, or ensure that the program's main thread does not terminate, perhaps with `Thread.sleep(1000)`. Java 7 has much a better threadpool available: `ForkJoinPool`. Here is how you can configure it:

```
val pool: ExecutorService = new java.util.concurrent.ForkJoinPool()
implicit val executionContext: ExecutionContext = ExecutionContext.fromExecutor(pool)
```

Each time a future is created it will use the implicit `ExecutionContext`. If you have JDK 6, then you must use another threadpool:

```
val pool: ExecutorService = Executors.newFixedThreadPool(8)
implicit val executionContext: ExecutionContext = ExecutionContext.fromExecutor(pool)
```

The `courseNotes` project contains the completed program in `FutureFun.scala`.

Bad: Accessing Shared Mutable State From A Future

Futures normally run on their own thread, and are intended for parallelism, not concurrency. This means that they should not share mutable state. The following code sets up a race condition, wherein you are not sure what the value of the shared variable should be when it is evaluated by the future. In this example, the arithmetic computations `2 + 3 + offset` and `2 + 3 + accessor` run on its own threads. As you can see, the value of offset was changed to 42 after the `Future` was created, when it had the value 6.

Futures must not casually access shared mutable state in another scope. Akka dataflow and actors provide a mechanism for concurrency with mutable state. Futures are not intended for this purpose.

```
import akka.actor.ActorSystem
import com.typesafe.config.ConfigFactory
import concurrent.{ExecutionContext, Future}
import scala.util.{Success, Failure}

object Race extends App {
  private val configString: String = "akka { logConfigOnStart=off }"
  private val system: ActorSystem = ActorSystem.apply("actorSystem", ConfigFactory.parseString(configString))
  implicit private var dispatcher: ExecutionContext = system.dispatcher

  @volatile var offset = 6 // @volatile makes no difference because it does not solve race conditions
  def accessor = offset

  val f1 = Future {
    2 + 3 + offset // will be executed asynchronously
  } andThen {
    case Success(result)   => println("Race Scala result1 : " + result)
    case Failure(exception) => println("Race Scala exception 1: " + exception.asInstanceOf[Exception].getMessage)
  }

  val f2 = Future {
    2 + 3 + accessor // will be executed asyn
  } andThen {
    case Success(result)   => println("Race Scala result 2: " + result)
    case Failure(exception) => println("Race Scala exception 2: " + exception.asInstanceOf[Exception].getMessage)
  }
  offset = 42
  system.log.info("End of mainline, offset = " + offset)

  Future.sequence(List(f1, f2)) andThen {
    case _ => system.shutdown() // terminates this thread, and the program if no other threads are active
  }
}
```

What do you think that the printed value should be? 2 + 3 + 6 = 11, or 2+ 3 + 42 = 47? The result printed depends on whether the computer the example runs on has more than one core available, the `Executor` used, and possibly the workload of other threads managed by the `Executor`. Try running the code several times.

This problem arises because `Futures` normally execute on a different thread than the invoking thread, and the thread with the `Future` started with the offset set to 6. Similar problems will ensue even if you call a function from the scope that defines the `Future`; or if you use Software Transactional Memory (STM). For the full program, see `Race.scala` in the `courseNotes` project.

The `Future` is computed in a closure that runs as an anonymous class on another thread. The value of offset when the `Future` is first computed is passed to the `Future`, but when the `Future` evaluates in another stack frame, the value of offset might not have been not be updated to the current value on the original stack frame.

Memory consistency issues can arise when unsafely passing objects between threads; these code examples use integers, so there cannot be any partial writes that would lead to internal inconsistency. If a code example were to use an object containing non-final fields, and the calling thread were to attempt to update a field on the `Future`'s thread, then the code running in the `Future`'s context could see the object in an inconsistent state - that is, partially updated. These problems can be hard to detect, and examining variable values on the heap or the main program's stack can be misleading.

Interaction with shared mutable state or other external resources outside the `Future` is allowed if the code is threadsafe and uses mechanisms such as transactors or j.u.c. coding practices such as synchronized, volatile, latches and locks for obeying ordinary concurrency rules. The simple rule is to only access immutable data when

defining a `Future`. This rule is why futures can support parallelism and composition. You can overcome this restriction somewhat by using transactors and normal `j.u.c.` programming practices for multi-threaded programs.

`j.u.c.` has some handy and efficient mechanisms for producing immutable and concurrent versions of mutable collections, such as `CopyOnWriteArrayList`, `ConcurrentMap`, and other immutable and concurrent members of the Java collections framework.

`j.u.c.` mechanisms only work within a single Java virtual machine; they does not work across multiple JVMs. For explicitly created `Futures` such as found in this course, this restriction is not a problem. Distributed Akka Actors can create `Futures` on remote JVMs, however, and transactors are necessary to handle that scenario. Exploring transactors is beyond the scope of this course.

Exercise

This exercise should take at least an hour. Modify the monkey simulator program you wrote for the previous lecture so it uses `Futures`.

Exercise

This exercise should take at least an hour. Modify the database loan pattern code and you wrote in an earlier exercise the same SQLite database to define methods that return futures for the following database operations:

- `insert`
- `select`
- `delete`

Use these methods to write a short program that inserts some records, selects them and deletes them. All operations should be asynchronous and non-blocking.

3-5 Akka Dataflow

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaMulti / akkaDataflow

- Callable from Scala only
- Requires delimited continuations enabled
- Has peculiar operators
- Easy to reason with (deterministic)
- Designed for shared mutable state - hence dataflow can act as concurrent 'glue'
- Dataflow blocks are composable
- Inside the flow method you can use Promises as dataflow variables

Akka Dataflow documentation is here.

This sample program is provided in the courseNotes project as DataflowFun.scala.

```
import akka.dataflow._
import scala.concurrent.{ExecutionContext, Promise}
import java.util.concurrent.{Executors, ExecutorService}

object DataflowFun extends App {
  val pool: ExecutorService = Executors.newFixedThreadPool(8)
  implicit val executionContext: ExecutionContext = ExecutionContext.fromExecutor(pool)

  val x, y, z = Promise[Int]()

  flow { // new thread
    z << x() + y()
    println("z = " + z())
  }

  flow { // new thread
    x << 40
    x() // return value
  } onComplete { _.map { x => println(s"x=${x}") } }

  flow { // new thread
    y << 2
    y() // return value
  } onComplete { y => println(s"y=${y}") }
}
```

Output is:

```
z = 42
y=Success(2)
x=40
```

Exercise

- Notice DataflowFun's ExecutorService is a FixedThreadPool. What happens if a fork / join pool is used instead? How could you modify the program to make it work properly with a

fork / join pool?

- What happens if you modify the last `flow` to look like this?

```
flow { // new thread
    y << z() + 2
    y() // return value
  } onComplete { y => println(s"y=${y}") }
```

Dataflow Compared to Future

- Dataflow has a smaller code footprint and is easier to reason about.
- Futures in for-comprehensions are more general than dataflow, and can operate on a wide array of types.
- Scala delimited continuations have a reputation for being problematic, although I have not encounted any problems

3-8 Introduction to Actors

ScalaCourses.com / scalaIntermediate / ScalaCore / scalaMulti / scalaActorIntro

Actors

Actors encapsulate state in a threadsafe manner, provided that the state is not shared. Actors are event-driven and use message passing to communicate with each other. Messages can be any type of object. This decoupling of `Actors` supports interaction while keeping code bases separate. `Actors` are:

You can use Actors to integrate two different code bases without coupling them

- Callable from Java & Scala
- Cheap to spawn (300 bytes per instance)
- Scalable beyond one motherboard / network node, enabling distributed concurrency. Messages sent to remote actors must be serializable.
- Not composable, but can return futures for obtaining final results
- Difficult to get intermediate results from
- 40+ years old
- The most complex Scala concurrency option, so `Actor`s should be your last concurrency option to consider
- Recently over-hyped (remember EJB 2?)

`Actors` 'run in packs'; one `Actor` is useless. Top-level `Actors` should be designed to supervise a hierarchy of child actors. The network of decoupled `Actors`, all running in different thread contexts, and restartable in the event of failure, enables the "Let it crash" philosophy. In order to design networks of `Actors` properly, a knowledge of queuing theory can be helpful.

In order to use Akka `Actors`, add the following to `build.sbt` or `Build.scala`:

```
"com.typesafe.akka" %% "akka-actor" % "2.2.1" withSources
```

Each file that uses actors will need some or more of the following imports, and to define a default `ExecutionContext` (discussed in the `Future` lecture).

```
import akka.actor._
import akka.pattern.ask
import akka.util.Timeout
import concurrent.duration._
import concurrent.ExecutionContext.Implicits.global
import concurrent.{Await, Future}
import scala.util.{Success, Failure}
```

An `ActorSystem` provides runtime support for `Actors`. They are quite heavyweight, so you should only create them very occasionally. `ActorSystems` have names, which are used to reference them. Their constructor accepts other parameters, including a `Config` object (discussed in the `Config` lecture).

```
val system = ActorSystem("myActorSystem")
```

`Actors` are normally defined as classes or traits. Because `Actors` are driven by messages arriving at the `Actor`'s mailbox, each `Actor` must define a method called `receive` that accepts a `PartialFunction` for message processing. Within the `receive` method, `sender` is defined to hold a reference to the `Actor` that send the most recent message. An `Actor` can send an asynchronous message (fire and forget) using the `tell` method, also provided as the exclamation mark method.

```
class MyActor extends Actor {
  def receive = {
    case msg =>
      println(s"MyActor got '$msg'")
      sender ! "Got the message" // reply to sender
  }
}
```

Create an instance of a top-level actor using `ActorSystem.actorOf`, passing in `Props` and an option `name`. `Props` is a container that indicates the type of `Actor` to create and might provide constructor parameters.

```
val myActor1 = system.actorOf(Props[MyActor], name="myActor1")
val aMessage = "This is a message"
implicit val timeout = Timeout(60 seconds)
```

Asynchronous message sends to `Actors` return a `Future`. When a response is expected, use the `ask method` (same as a question mark operator):

```
val response: Future[Any] = myActor1 ? aMessage
```

As you can see, the resulting `Future` is untyped. You can convert the `Future` to contain another type with `mapTo`, but as with all type coercion this could raise an exception. As with any `Future`, you can block for a result with `Await`, use a combinator, or use a callback:

```
scala> Await.result(myActor1 ? aMessage, 5 seconds)
MyActor got 'This is a message'
res0: Any = Got the message

scala> Await.result((myActor1 ? aMessage).mapTo[String], 5 seconds)
MyActor got 'This is a message'
res1: Any = Got the message

scala> myActor1 ? aMessage map ( println )
MyActor got 'This is a message'
Got the message
res2: scala.concurrent.Future[Unit] = scala.concurrent.impl.Promise$DefaultPromise@4a261b29

scala> myActor1 ? aMessage onComplete ( println )
MyActor got 'This is a message'
Success(Got the message)
```

Actor Wiring

- Wiring is done in code, not declaratively as with Spring
- Examine `receive` method to discover messages accepted
- Must comb through entire code base to discover where messages are sent from. Typesafe has

not indicated they intend to address this. This is a third-party enhancement opportunity.

Within an `Actor`, several properties are available, including `self` and `context`. Remember that top-level actors can be created using `ActorSystem.actorOf`. Any `Actor` can create child actors that it supervises, using `context.actorOf`.

```
class AnotherActor extends Actor {
  val childActor = context.actorOf(Props[MyActor])
  childActor ! "Do my bidding"

  def receive = {
    case msg =>
      println(s"AnotherActor got '$msg'")
  }
}
```

An `Actor` can get a reference to the `ActorSystem` it runs under using `context.system`. `context` is of type `ActorContext`, and is worth taking a look at.

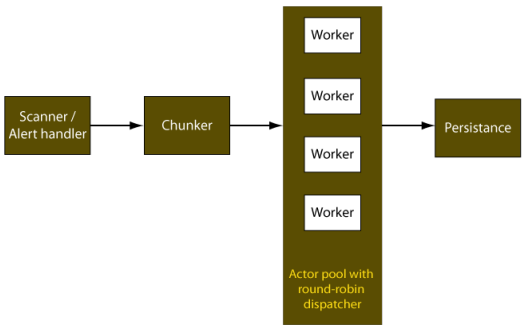
Queuing Theory

Can you name a famous queuing theorist?

- Agner Krarup Erlang (1878 –1929)
- Developed queuing theory and traffic engineering for telephone network analysis
- Ericsson named the Erlang language after him
- Erlang language makes extensive use of actors
- Erlang's Error Kernel Pattern: spawn new actor per request

Actor Pooled Worker Pattern

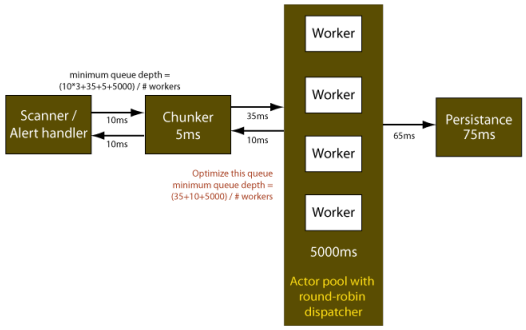
The `ActorFun.scala` file in the `courseNotes` shows a complete example of this pattern.



Back Pressure / Metering

- Bounded mailbox with appropriate push timeout can be difficult to set up without dropping messages
- Using link congestion for flow control is a bad idea; all communication between those actors backs up
- Pulling messages is inefficient
- Best to define messages for actor flow control; someone should define a standard trait for this

Queuing Example, multiple nodes



Testing Actors

- Can be challenging
- ScalaTest & Akka TestKit can test intra-actor behavior
- Specs2 and Akka TestKit do not play well together
- Can test Actor interactions
- Should test FSMs
- Use standard simulation techniques for debugging complex network protocols
- TestKit configuration can be specified in `application.conf`

Add the following to `libraryDependencies` in `build.sbt` or `Build.scala`:

```
"com.typesafe.akka" %% "akka-testkit"   % "2.2.1" "test" withSources,
"org.scalatest"     %  "scalatest_2.10" % "2.0.M7" % "test" withSources
```

For Scala IDE, also add the following:

```
"junit" % "junit" % "4.8.1"  % "test"
```

Exercise

This exercise should take at least an hour.

1. Enhance `ActorTest.scala` in `courseNotes` such that it sends a `ChunkerMsg` and a `PersistenceMessage` message to the `persistenceActor` that has been created for you. Verify that `targetString`, `bestMatchString` and `workers` get set to appropriate values.
2. Modify the program so the main program receives a message each time a new best match is calculated, and have it print the new matching string.
3. Modify the unit tests (and possibly the actors) so you can verify that the messages sent to the actors are properly processed

Akka Pattern Package

- Collection of patterns involving actors, futures and optionally dataflow

- Very helpful!
- Remember:
  - Dataflow requires Scala delimited continuations to be enabled
  - Not available from Java

akka.pattern.ask

`ask`: create temporary actor to interact once with an actor via a future. Recommended usage involves dataflow:

```
val f= ask(worker, request)(timeout) flow { // CPS block
  // f() returns future result
  EnrichedRequest(request, f())
} pipeTo nextActor
```

Other usages do not require dataflow

akka.pattern.pipe

Import this implicit conversion to gain the `pipeTo` method on `Future`:

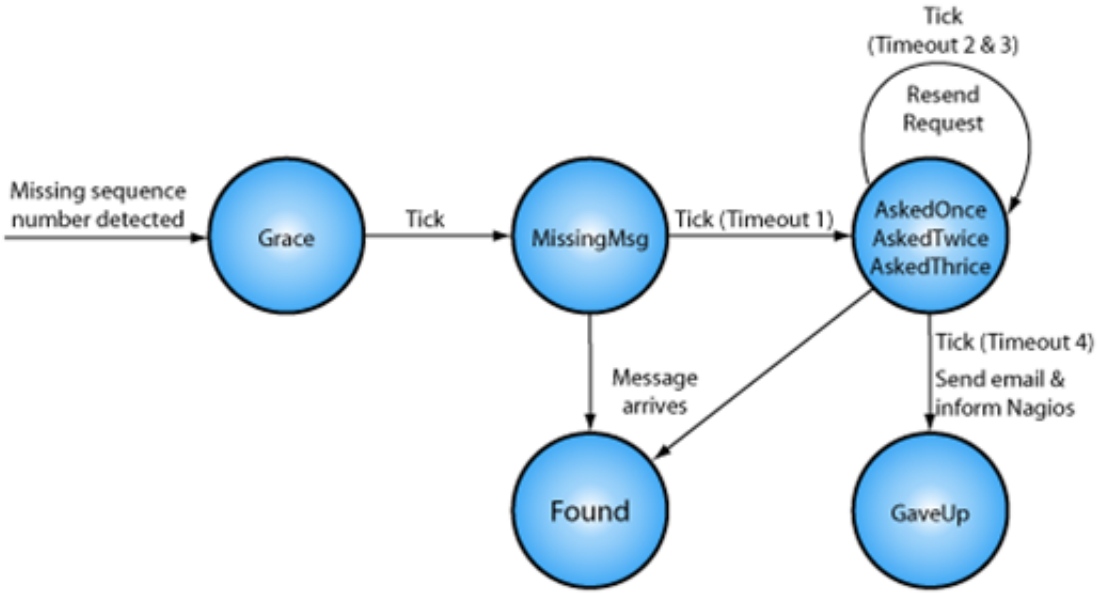
```
import akka.pattern.pipe
Future {
  doExpensiveCalc()
} pipeTo nextActor
```

or :

```
pipe(someFuture) to nextActor
```

Actors and Finite State Machines

- Interaction between actors is complex
- FSMs are a natural tool to define protocols
- Akka provides a special type of actor for FSM support
- Non-Akka FSMs are also very useful
- State diagrams are a really good idea


Tick
(Timeout 2 & 3)
Resend
Request
Missing sequence
number detected
Grace
Tick
MissingMsg
Tick (Timeout 1)
AskedOnce
AskedTwice
AskedThrice
Tick (Timeout 4)
Send email &
inform Nagios
Message
arrives
Found
GaveUp