Scala School

From ∅ to Distributed Service

About

Scala school was started as a series of lectures at Twitter to prepare experienced engineers to be productive Scala programmers. Being a relatively new language, but also one that draws on many familiar concepts, we found this an effective way of getting new engineers up to speed quickly. This is the written material that accompanied those lectures. We have found that these are useful in their own right.

Approach

We think it makes the most sense to approach teaching Scala not as if it's an improved Java but as a new language. Experience in Java is not expected. Focus will be around the interpreter and the object-functional style as well as the style of programming we do here. An emphasis will be placed on maintainability, clarity of expression, and leveraging the type system.

Most of the lessons require no software other than a Scala REPL. The reader is encouraged to follow along,

Lessons

and go further! Use these lessons as a starting point to explore the language.

Built at @twitter by @stevej and @marius with much help from @evanm, @sprsquish, @kevino, @zuercher, @timtrueman, @wickman and @mccv

This lesson covers:

- values
- expressions
- functions
- classes
- methods
- basic inheritance
- try-catch-finally

About this class

The first few weeks will cover basic syntax and concepts. Then we'll start to open it up with more exercises.

Some examples will be given as if written in the interpreter, others as if written in a source file.

Having an interpreter available makes it easy to explore a problem space.

Why Scala?

- Expressive
  - First-class functions
  - Closures
- Concise
  - Type inference
  - Literal syntax for function creation
- Java interopability
  - Can reuse java libraries
  - Can reuse java tools
  - No performance penalty

How Scala?

- Compiles to java bytecode
- Works with any standard JVM
  - Or even some non-standard JVMs like Dalvik
  - Scala compiler written by author of Java compiler

Think Scala

Scala is not just a nicer Java. You should learn it with a fresh mind, you will get more out of these classes.

Start the Interpreter

Start the included `sbt console`.

```
$ sbt console

[...]

Welcome to Scala version 2.8.0.final (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_20).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Expressions

```
scala> 1 + 1
res0: Int = 2
```

res0 is an automatically created value name given by the interpreter to the result of your expression. It has the type Int and contains the Integer 2.

(Almost) everything in Scala is an expression.

Values

You can give the result of an expression a name.

```
scala> val two = 1 + 1
two: Int = 2
```

You cannot change the binding to a val.

Variables

If you need to change the binding, you can use a `var` instead

```
scala> var name = "steve"
name: java.lang.String = steve

scala> name = "marius"
name: java.lang.String = marius
```

Functions

You can create functions with def.

```
scala> def addOne(m: Int): Int = m + 1
addOne: (m: Int)Int
```

In Scala, you need to specify the type signature for function parameters. The interpreter happily repeats the type signature back to you.

```
scala> val three = addOne(2)
three: Int = 3
```

You can leave off parens on functions with no arguments

```
scala> def three() = 1 + 2
three: ()Int

scala> three()
res2: Int = 3

scala> three
res3: Int = 3
```

Anonymous Functions

You can create anonymous functions.

```
scala> (x: Int) => x + 1
res2: (Int) => Int = <function1>
```

This function adds 1 to an Int named x.

```
scala> res2(1)
res3: Int = 2
```

You can pass anonymous functions around or save them into vals.

```
scala> val addOne = (x: Int) => x + 1
addOne: (Int) => Int = <function1>

scala> addOne(1)
res4: Int = 2
```

If your function is made up of many expressions, you can use {} to give yourself some breathing room.

```
def timesTwo(i: Int): Int = {
  println("hello world")
  i * 2
}
```

This is also true of an anonymous function

```
scala> { i: Int =>
  println("hello world")
  i * 2
```

```
}
res0: (Int) => Int = <function1>
```

You will see this syntax often used when passing an anonymous function as an argument.

Partial application

You can partially apply a function with an underscore, which gives you another function.

```
scala> def adder(m: Int, n: Int) = m + n
adder: (m: Int,n: Int)Int
```

```
scala> val add2 = adder(2, _:Int)
add2: (Int) => Int = <function1>

scala> add2(3)
res50: Int = 5
```

You can partially apply any argument in the argument list, not just the last one.

Curried functions

Sometimes it makes sense to let people apply some arguments to your function now and others later.

Here's an example of a function that lets you build multipliers of two numbers together. At one call site, you'll decide which is the multiplier and at a later call site, you'll choose a multipicand.

```
scala> def multiply(m: Int)(n: Int): Int = m * n
multiply: (m: Int)(n: Int)Int
```

You can call it directly with both arguments.

```
scala> multiply(2)(3)
```

```
res0: Int = 6
```

You can fill in the first parameter and partially applying the second.

```
scala> val timesTwo = multiply(2)(_)
timesTwo: (Int) => Int = <function1>

scala> timesTwo(3)
res1: Int = 6
```

This sometimes lead to crazy pieces of code.

```
multiplyThenFilter { m: Int =>
  m * 2
} { n: Int =>
  n < 5
}
```

I promise you get used to this over time.

You can take any function of multiple arguments and curry it. Let's try with our earlier

```
adder
```

```
scala> (adder(_, _)).curried
res1: (Int) => (Int) => Int = <function1>
```

Variable length arguments

There is a special syntax for methods that can take parameters of a repeated type.

```
def capitalizeAll(args: String*) = {
  args.map { arg =>
    arg.capitalize
  }
}
```

Classes

```
scala> class Calculator {
     |   val brand: String = "HP"
     |   def add(m: Int, n: Int): Int = m + n
     | }
defined class Calculator

scala> val calc = new Calculator
calc: Calculator = Calculator@e75a11

scala> calc.add(1, 2)
res1: Int = 3

scala> calc.brand
res2: String = "HP"
```

Contained are examples are defining methods with def and fields with val. methods are just functions that can access the state of the class.

Constructor

Constructors aren't special methods, they are the code outside of method definitions in your class. Let's extend our Calculator example to take a constructor argument and use it to initialize internal state.

```
class Calculator(brand: String) {
  /**
   * A constructor.
   */
  val color: String = if (brand == "TI") {
    "blue"
```

```
  } else if (brand == "HP") {
    "black"
  } else {
    "white"
  }

  // An instance method.
  def add(m: Int, n: Int): Int = m + n
}
```

Note the two different styles of comments.

Expressions

Our BasicCalculator example gave an example of how Scala is expression-oriented. The value color was bound based on an if/else expression. Scala is highly expression-oriented, most things are expressions rather than statements.

Inheritance

```
class ScientificCalculator(brand: String) extends Calculator(brand) {
  def log(m: Double, base: Double) = math.log(m) / math.log(base)
}
```

Overloading methods

```
class EvenMoreScientificCalculator(brand: String) extends ScientificCalculator(brand) {
  def log(m: Int) = log(m, math.exp(1))
}
```

Traits

`traits` are collections of fields and behaviors that you can extend or mixin to your classes.

```
trait Car {
```

```
  val brand: String
}
```

```
class BMW extends Car {
  val brand = "BMW"
}
```

Types

Earlier, you saw that we defined a function that took an `Int` which is a type of Number. Functions can also be generic and work on any type. When that occurs, you'll see a

```
type parameter
```

introduced with the square bracket syntax:

You can introduce as many type parameters. Here's an example of a Cache of generic Keys and Values.

```
trait Cache[K, V] {
  def get(key: K): V
  def put(key: K, value: V)
  def delete(key: K)
}
```

Methods can also have type parameters introduced.

```
def remove[K](key: K)
```

This lesson covers:

- apply
- objects
- case classes
- update
- Functions are Objects
- packages
- pattern matching

apply methods

apply methods give you a nice syntatic sugar for when a class or object has one main use.

```
object FooMaker {
  def apply() = new Foo
}

scala> class Bar {
     |   def apply() = 0
     | }
defined class Bar

scala> val bar = new Bar
bar: Bar = Bar@47711479

scala> bar()
res8: Int = 0
```

Here our instance object looks like we're calling a method. More on that later!

Objects

Objects are used to hold single instances of a class. Often used for factories.

```
object Timer {
  var count = 0

  def currentCount(): Long = {
    count += 1
    count
  }
}
```

How to use

```
scala> Timer.currentCount()
res0: Long = 1
```

Classes and Objects can have the same name. The object is called a ‘Companion Object’. We commonly use Companion Objects for Factories.

Here is a trivial example that only serves to remove the need to use ‘new’ to create an instance.

```
class Bar(foo: String)

object Bar {
  def apply(foo: String) = new Bar(foo)
}
```

Functions are Objects

In Scala, we talk about object-functional programming often. What does that mean? What is a Function, really?

A Function is a set of traits. Speficially, a function that takes one argument is an instance of a Function1 trait. This trait defines the `apply()` syntactic sugar we learned earlier, allowing you to call an object like you would a function.

```
cala> object addOne extends Function1[Int, Int] {
     |   def apply(m: Int): Int = m + 1
     | }
defined module AddOne
```

```
scala> addOne(1)
res2: Int = 2
```

There is Function1 through 22. Why 22? It's an arbitrary magic number. I've never needed a function with more than 22 arguments so it seems to work out.

The syntatic sugar of apply helps unify the duality of object and functional programming. You can pass classes around and use them as functions and functions are just instances of classes under the covers.

Does this mean that everytime you define a method in your class, you're actually getting an instance of Function*? No, methods in classes are methods. methods defined standalone in the repl are Function* instances.

Classes can also extend Function and those instances can be called with ().

```
cala> class AddOne extends Function1[Int, Int] {
     |   def apply(m: Int): Int = m + 1
     | }
defined class AddOne

scala> val plusOne = new AddOne()
plusOne: AddOne = <function1>

scala> plusOne(1)
res0: Int = 2
```

A nice short-hand for `extends Function1[Int, Int]` is `extends (Int => Int)`

```
class AddOne extends (Int => Int) {
  def apply(m: Int): Int = m + 1
}
```

Packages

You can organize your code inside of packages.

```
package com.twitter.example
```

at the top of a file will declare everything in the file to be in that package.

Values and functions cannot be outside of a class or object. Objects are a useful tool for organizing static functions.

```
package com.twitter.example

object colorHolder {
  val BLUE = "Blue"
  val RED = "Red"
}
```

Now you can access the members directly

```
println("the color is: " + com.twitter.example.colorHolder.BLUE)
```

Notice what the scala repl says when you define this object:

```
scala> object colorHolder {
     |   val Blue = "Blue"
     |   val Red = "Red"
     | }
defined module colorHolder
```

This gives you a small hint that the designers of Scala designed objects to be part of Scala’s module system.

Pattern Matching

One of the most useful parts of Scala.

Matching on values

```
val times = 1
```

```
times match {
  case 1 => "one"
  case 2 => "two"
  case _ => "some other number"
}
```

Matching with guards

```
times match {
  case i if i == 1 => "one"
  case i if i == 2 => "two"
  case _ => "some other number"
}
```

Notice how we captured the value in the variable ‘i’.

The `_` in the last case statement is a wildcard, it ensures that we can handle any statement. Otherwise you will suffer a runtime error if you pass in a number that doesn’t match. We discuss this more later.

Matching on class members

Remember our calculator from earlier.

Let’s classify them according to type.

```
def calcType(calc: Calculator) = calc match {
  case calc.brand == "hp" && calc.model == "20B" => "financial"
  case calc.brand == "hp" && calc.model == "48G" => "scientific"
  case calc.brand == "hp" && calc.model == "30B" => "business"
  case _ => "unknown"
}
```

Wow, that’s painful. Thankfully Scala provides some nice tools specifically for this.

Case Classes

case classes are used to conveniently store and match on the contents of a class. You can construct them without using new.

```
scala> case class Calculator(brand: String, model: String)
defined class Calculator

scala> val hp20b = Calculator("hp", "20b")
hp20b: Calculator = Calculator(hp,20b)
```

case classes automatically have equality and nice toString methods based on the constructor arguments.

```
scala> val hp20b = Calculator("hp", "20b")
hp20b: Calculator = Calculator(hp,20b)

scala> val hp20B = Calculator("hp", "20b")
hp20B: Calculator = Calculator(hp,20b)

scala> hp20b == hp20B
res6: Boolean = true
```

case classes can have methods just like normal classes.

CASE CLASSES WITH PATTERN MATCHING

case classes are designed to be used with pattern matching. Let's simplify our calculator classifier example from earlier.

```
val hp20b = Calculator("hp", "20B")
val hp30b = Calculator("hp", "30B")

def calcType(calc: Calculator) = calc match {
  case Calculator("hp", "20B") => "financial"
  case Calculator("hp", "48G") => "scientific"
  case Calculator("hp", "30B") => "business"
  case Calculator(ourBrand, ourModel) => "Calculator: %s %s is of unknown type".format(ourBrand, ourModel)
```

```
}
```

Other alternatives for that last match

```
  case Calculator(_, _) => "Calculator of unknown type"
```

OR we could simply not specify that it’s a Calculator at all.

```
  case _ => "Calculator of unknown type"
```

OR we could re-bind the matched value with another name

```
  case c@Calculator(_, _) => "Calculator: %s of unknown type".format(c)
```

Exceptions

Exceptions are available in Scala via a try-catch-finally syntax that uses pattern matching.

```
try {
  remoteCalculatorService.add(1, 2)
} catch {
  case e: ServerIsDownException => log.error(e, "the remote calculator service is unavailble. should have kept your trustry HP.")
} finally {
  remoteCalculatorService.close()
}
```

`try`s are also expression-oriented

```
val result: Int = try {
  remoteCalculatorService.add(1, 2)
} catch {
  case e: ServerIsDownException => {
```

```
    log.error(e, "the remote calculator service is unavailble. should have kept your trustry HP.")
    0
  }
} finally {
  remoteCalculatorService.close()
}
```

This is not an example of excellent programming style, just an example of try-catch-finally resulting in expressions like most everything else in Scala.

Finally will be called after an exception has been handled and is not part of the expression.

This lesson covers:

- What are types?
- Parametric Polymorphism
- Type inference: Hindley-Milner vs. local type inference
- Variance, bounds & quantification

What are static types? Why are they useful?

According to Pierce: “A type system is a syntactic method for automatically checking the absence of certain erroneous behaviors by classifying program phrases according to the kinds of values they compute.”

Types allow you to denote function domain & codomains. For example, from mathematics, we are used to seeing:

```
f: R -> N
```

this tells us that function “f” maps values from the set of real numbers to values of the set of natural numbers.

In the abstract, this is exactly what *concrete* types are. Type systems give us some more powerful ways to express these sets.

Given these annotations, the compiler can now *statically* (at compile time) verify that the program is *sound*. That is, compilation will fail if values (at runtime) will not comply to the constraints imposed by the program.

Generally speaking, the typechecker can only guarantee that *unsound* programs do not compile. It cannot guarantee that every sound program *will* compile.

With increasing expressiveness in type systems, we can produce more reliable code because it allows us to prove invariants about our program before it even runs (modulo bugs in the types themselves, of course!). Academia is pushing the limits of expressiveness very hard, including value-dependent types!

Note that all type information is removed at compile time. It is no longer needed. This is called erasure.

Types in Scala

Scala has a very powerful type system, allowing for very rich expression. Some of its chief features are:

- parametric polymorphism
- (local) type inference

- existential quantification
- views

Parametric polymorphism

Polymorphism is used in order to write generic code (for values of different types) without compromising static typing richness.

For example, without parametric polymorphism, a generic list data structure would always look like this (and indeed it did look like this in Java prior to generics):

```
scala> 2 :: 1 :: "bar" :: "foo" :: Nil
res5: List[Any] = List(2, 1, bar, foo)
```

Now we cannot recover any type information about the individual members.

```
scala> res5.head
res6: Any = 2
```

And so our application would devolve into a series of casts ("asInstanceOf[]") and we would lack typesafety (because these are all dynamic).

Polymorphism is achieved through specifying *type variables*.

```
scala> def drop1[A](l: List[A]) = l.tail
drop1: [A](l: List[A])List[A]

scala> drop1(List(1,2,3))
res1: List[Int] = List(2, 3)
```

Scala has rank-1 polymorphism

Suppose you had some function

```
def toList[A](a: A) = List(a)
```

which you wished to use generically:

```
def foo[A, B](f: A => List[A], b: B, c: Int) = (f(b), f(c))
```

This does not compile, because all type variables have to be fixed at the invocation site.

Type inference

A traditional objection to static typing is that it has much syntactic overhead. Scala alleviates this by providing *type inference*.

The classic method for type inference in functional programming languages is *Hindley-Milner*, and it was first employed in ML.

Scala's type inference system works a little differently, but it's similar in spirit: infer constraints, and attempt to unify a type.

In Scala, for example, you cannot do the following:

```
scala> { x => x }
<console>:7: error: missing parameter type
       { x => x }
```

Whereas in OCaml, you can:

```
# fun x -> x;;
- : 'a -> 'a = <fun>
```

In scala all type inference is *local*. For example:

```
scala> def id[T](x: T) = x
id: [T](x: T)T

scala> val x = id(322)
x: Int = 322

scala> val x = id("hey")
x: java.lang.String = hey
```

```
scala> val x = id(Array(1,2,3,4))
x: Array[Int] = Array(1, 2, 3, 4)
```

Types are now preserved, The Scala compiler infers the type parameter for us. Note also how we did not have to specify the return type explicitly.

Variance

Scala's type system has to account for class hierarchies together with polymorphism. Class hierarchies allow the expression of subtype relationships. A central question that comes up when mixing OO with polymorphism is: if T' is a subclass of T, is Container[T'] considered a subclass of Container[T]? Variance annotations allow you to express the following relationships between class hierarchies & polymorphic types:

covariant	C[T'] is a subclass of C[T]
contravariant	C[T] is a subclass of C[T']
invariant	C[T] and C[T'] are not related

The subtype relationship really means: for a given type T, if T' is a subtype, can you substitute it?

```
scala> class Covariant[+A]
defined class Covariant

scala> val cv: Covariant[AnyRef] = new Covariant[String]
cv: Covariant[AnyRef] = Covariant@4035acf6

scala> val cv: Covariant[String] = new Covariant[AnyRef]
<console>:6: error: type mismatch;
 found   : Covariant[AnyRef]
 required: Covariant[String]
       val cv: Covariant[String] = new Covariant[AnyRef]
                                   ^
```

```
scala> class Contravariant[-A]
defined class Contravariant

scala> val cv: Contravariant[String] = new Contravariant[AnyRef]
```

```
cv: Contravariant[AnyRef] = Contravariant@49fa7ba

scala> val fail: Contravariant[AnyRef] = new Contravariant[String]
<console>:6: error: type mismatch;
 found   : Contravariant[String]
 required: Contravariant[AnyRef]
       val fail: Contravariant[AnyRef] = new Contravariant[String]
                                     ^
```

Contravariance seems strange. When is it used? Somewhat surprising!

```
trait Function1 [-T1, +R] extends AnyRef
```

If you think about this from the point of view of substitution, it makes a lot of sense. Let’s first define a simple class hierarchy:

```
scala> class A
defined class A

scala> class B extends A
defined class B

scala> class C extends B
defined class C

scala> class D extends C
defined class D
```

```
scala> val f: (B => C) = ((b: B) => new C)
f: (B) => C = <function1>

scala> val f: (B => C) = ((a: A) => new C)
f: (B) => C = <function1>
```

```
scala> val f: (B => C) = ((a: A) => new D)
f: (B) => C = <function1>

scala> val f: (B => C) = ((a: A) => new C)
f: (B) => C = <function1>

scala> val f: (B => C) = ((c: C) => new C)
<console>:8: error: type mismatch;
 found   : (C) => C
 required: (B) => C
       val f: (B => C) = ((c: C) => new C)
                                 ^
                                    ^
```

Bounds

Scala allows you to restrict polymorphic variables using *bounds*. These bounds express subtype relationships.

```
scala> class F { def foo = "F" }
defined class F

scala> class E extends F { override def foo = "E" }
defined class E

scala> def callFoo[T](foos: Seq[T]) = foos map (_.foo) foreach println
<console>:8: error: value foo is not a member of type parameter T
       def callFoo[T](foos: Seq[T]) = foos map (_.foo) foreach println

scala> def callFoo[T <: F](foos: Seq[T]) = foos map (_.foo) foreach println
callFoo: [T <: F](foos: scala.collection.mutable.Seq[T])Unit

scala> callFoo(Seq(new E, new F))
E
F
```

Lower type bounds are also supported. They go hand-in-hand with contravariance. Let's say we had some class Node:

```
scala> class Node[T](x: T) { def sub(v: T): Node[T] = new Node(v) }
```

But, we want to make it covariant in T:

```
scala> class Node[+T](x: T) { def sub(v: T): Node[T] = new Node(v) }
<console>:6: error: covariant type T occurs in contravariant position in type T of value v
       class Node[+T](x: T) { def sub(v: T): Node[T] = new Node(v) }
                                  ^
```

Recall that method arguments are contravariant, and so if we perform our substitution trick, using the same classes as before:

```
class Node[B](x: B) { def sub(v: B): Node[B] = new Node(v) }
```

is **not** a subtype of

```
class Node[A](x: A) { def sub(v: A): Node[A] = new Node(v) }
```

because A cannot be substituted for B in the argument of “sub”. However, we can use lower bounding to enforce correctness.

```
scala> class Node[+T](x: T) { def sub[U >: T](v: U): Node[U] = new Node(v) }
defined class Node

scala> (new Node(new B)).sub(new B)
res5: Node[B] = Node@4efade06

scala> ((new Node(new B)).sub(new B)).sub(new A)
res6: Node[A] = Node@1b2b2f7f

scala> ((new Node(new B)).sub(new B)).asInstanceOf[Node[C]]
res7: Node[C] = Node@6924181b

scala> (((new Node(new B)).sub(new B)).sub(new A)).sub(new C)
```

```
res8: Node[A] = Node@3088890d
```

Note also how the type changes with subsequent calls to “sub”.

Quantification

Sometimes you do not care to be able to name a type variable, for example:

```
scala> def count[A](l: List[A]) = l.size
count: [A](List[A])Int
```

Instead you can use “wildcards”:

```
scala> def count(l: List[_]) = l.size
count: (List[_])Int
```

This is shorthand for:

```
scala> def count(l: List[T forSome { type T }]) = l.size
count: (List[T forSome { type T }])Int
```

Note that quantification can get tricky:

```
scala> def drop1(l: List[_]) = l.tail
drop1: (List[_])List[Any]
```

Suddenly we lost type information! To see what’s going on, revert to the heavy-handed syntax:

```
scala> def drop1(l: List[T forSome { type T }]) = l.tail
drop1: (List[T forSome { type T }])List[T forSome { type T }]
```

We can't say anything about T because the type does not allow it.

You may also apply bounds to wildcard type variables:

```
scala> def hashcodes(l: Seq[_ <: AnyRef]) = l map (_.hashCode)
hashcodes: (Seq[_ <: AnyRef])Seq[Int]

scala> hashcodes(Seq(1,2,3))
<console>:7: error: type mismatch;
 found   : Int(1)
 required: AnyRef
Note: primitive types are not implicitly converted to AnyRef.
You can safely force boxing by casting x.asInstanceOf[AnyRef].
       hashcodes(Seq(1,2,3))
                     ^

scala> hashcodes(Seq("one", "two", "three"))
res1: Seq[Int] = List(110182, 115276, 110339486)
```

This lesson covers:

- View bounds ("type classes")
- Higher kinded types & ad-hoc polymorphism
- F-bounded polymorphism / recursive types
- Structural types
- Abstract types members
- Type erasures & manifests
- Case study: Finagle

View bounds ("type classes")

Implicit functions in scala allow for on-demand function application when this can help satisfy type inference. eg.:

```
scala> implicit def strToInt(x: String) = x.toInt
strToInt: (x: String)Int

scala> "123"
res0: java.lang.String = 123

scala> val y: Int = "123"
y: Int = 123

scala> math.max("123", 111)
res1: Int = 123
```

view bounds, like type bounds demand such a function exists for the given type. eg.

```
scala> class Container[A <% Int] { def addIt(x: A) = 123 + x }
defined class Container
```

This says that **A** has to be "viewable" as **Int**. Let's try it.

```
scala> (new Container[String]).addIt("123")
res11: Int = 246

scala> (new Container[Int]).addIt(123)
res12: Int = 246

scala> (new Container[Float]).addIt(123.2F)
<console>:8: error: could not find implicit value for evidence parameter of type (Float) => Int
       (new Container[Float]).addIt(123.2)
        ^
```

Other type bounds

Methods may ask for specific “evidence” for a type, namely:

A =:= B	A must be equal to B
A <:< B	A must be a subtype of B
A <%< B	A must be viewable as B

```
scala> class Container[A](value: A) { def addIt(implicit evidence: A =:= Int) = 123 + value }
defined class Container

scala> (new Container(123)).addIt
res11: Int = 246

scala> (new Container("123")).addIt
<console>:10: error: could not find implicit value for parameter evidence: =:=[java.lang.String,Int]
```

Similarly, given our previous implicit, we can relax the constraint to viewability:

```
scala> class Container[A](value: A) { def addIt(implicit evidence: A <%< Int) = 123 + value }
defined class Container
```

```
scala> (new Container("123")).addIt
res15: Int = 246
```

Generic programming with views

In the Scala standard library, views are primarily used to implement generic functions over collections. For example, the "min" function (on **Seq[]**), uses this technique:

```
def min[B >: A](implicit cmp: Ordering[B]): A = {
  if (isEmpty)
    throw new UnsupportedOperationException("empty.min")

  reduceLeft((x, y) => if (cmp.lteq(x, y)) x else y)
}
```

The main advantages of this are:

- Items in the collections aren't required to implement **Ordered**, but **Ordered** uses are still statically type checked.
- You can define your own orderings without any additional library support:

```
scala> List(1,2,3,4).min
res0: Int = 1

scala> List(1,2,3,4).min(new Ordering[Int] { def compare(a: Int, b: Int) = b compare a })
res3: Int = 4
```

As a sidenote, there are views in the standard library that translates **Ordered** into **Ordering** (and vice versa).

```
trait LowPriorityOrderingImplicits {
  implicit def ordered[A <: Ordered[A]]: Ordering[A] = new Ordering[A] {
    def compare(x: A, y: A) = x.compare(y)
  }
}
```

Context bounds & implicitly[]

Scala 2.8 introduced a shorthand for threading through & accessing implicit arguments.

```
scala> def foo[A](implicit x: Ordered[A]) {}
foo: [A](implicit x: Ordered[A])Unit

scala> def foo[A : Ordered] {}
foo: [A](implicit evidence$1: Ordered[A])Unit
```

Implicit values may be accessed via **implicitly**

```
scala> implicitly[Ordering[Int]]
res37: Ordering[Int] = scala.math.Ordering$Int$@3a9291cf
```

Combined, these often result in less code, especially when threading through views.

Higher-kinded types & ad-hoc polymorphism

Scala can abstract over “higher kinded” types. This is analagous to function currying. For example, whereas “unary types” have constructors like this:

```
List[A]
```

Meaning we have to satisfy one “level” of type variables in order to produce a concrete types (just like an uncurried function needs to be supplied by only one argument list to be invoked), a higher-kinded type needs more:

```
scala> trait Container[M[_]] { def put[A](x: A): M[A]; def get[A](m: M[A]): A }

scala> new Container[List] { def put[A](x: A) = List(x); def get[A](m: List[A]) = m.head }
res23: java.lang.Object with Container[List] = $anon$1@7c8e3f75

scala> res23.put("hey")
res24: List[java.lang.String] = List(hey)
```

```
scala> res23.put(123)
res25: List[Int] = List(123)
```

Note that **Container** is polymorphic in a parameterized type (“container type”).

If we combine using containers with implicits, we get “ad-hoc” polymorphism: the ability to write generic functions over containers.

```
scala> trait Container[M[_]] { def put[A](x: A): M[A]; def get[A](m: M[A]): A }

scala> implicit val listContainer = new Container[List] { def put[A](x: A) = List(x); def get[A](m: List[A]) = m.head }

scala> implicit val optionContainer = new Container[Some] { def put[A](x: A) = Some(x); def get[A](m: Some[A]) = m.get }

scala> def tupleize[M[_]: Container, A, B](fst: M[A], snd: M[B]) = {
     | val c = implicitly[Container[M]]
     | c.put(c.get(fst), c.get(snd))
     | }
tupleize: [M[_],A,B](fst: M[A],snd: M[B])(implicit evidence$1: Container[M])M[(A, B)]

scala> tupleize(Some(1), Some(2))
res33: Some[(Int, Int)] = Some((1,2))

scala> tupleize(List(1), List(2))
res34: List[(Int, Int)] = List((1,2))
```

F-bounded polymorphism

Often it’s necessary to access a concrete subclass in a (generic) trait. For example, imagine you had some trait that is generic, but can be compared to a particular subclass of that trait.

```
trait Container extends Ordered[Container]
```

However, this now necessitates the compare method

```
def compare(that: Container): Int
```

And so we cannot access the concrete subtype, eg.:

```
class MyContainer extends Container {
  def compare(that: MyContainer): Int
}
```

fails to compile, since we are specifying Ordered for **Container**, not the particular subtype.

To reconcile this, we use F-bounded polymorphism.

```
trait Container[A <: Container[A]] extends Ordered[A]
```

Strange type! But note now how Ordered is parameterized on **A**, which itself is **Container[A]**

So, now

```
class MyContainer extends Container[MyContainer] {
  def compare(that: MyContainer) = 0
}
```

They are now ordered:

```
scala> List(new MyContainer, new MyContainer, new MyContainer)
res3: List[MyContainer] = List(MyContainer@30f02a6d, MyContainer@67717334, MyContainer@49428ffa)

scala> List(new MyContainer, new MyContainer, new MyContainer).min
res4: MyContainer = MyContainer@33dfeb30
```

Given that they are all subtypes of **Container[_]**, we can define another subclass & create a mixed list of **Container[_]**:

```
scala> class YourContainer extends Container[YourContainer] { def compare(that: YourContainer) = 0 }
defined class YourContainer

scala> List(new MyContainer, new MyContainer, new MyContainer, new YourContainer)
res2: List[Container[_ >: YourContainer with MyContainer <: Container[_ >: YourContainer with MyContainer <: ScalaObject]]]
  = List(MyContainer@3be5d207, MyContainer@6d3fe849, MyContainer@7eab48a7, YourContainer@1f2f0ce9)
```

Note how the resulting type is now lower-bound by **YourContainer with MyContainer**. This is the work of the type inferencer. Interestingly- this type doesn't even need to make sense, it only provides a logical greatest lower bound for the unified type of the list. What happens if we try to use **Ordered** now?

```
(new MyContainer, new MyContainer, new MyContainer, new YourContainer).min
<console>:9: error: could not find implicit value for parameter cmp:
  Ordering[Container[_ >: YourContainer with MyContainer <: Container[_ >: YourContainer with MyContainer <: ScalaObject]]]
```

No **Ordered[]** exists for the unified type. Too bad.

Structural types

Scala has support for **structural types** — type requirements are expressed by interface *structure* instead of a concrete type.

```
scala> def foo(x: { def get: Int }) = 123 + x.get
foo: (x: AnyRef{def get: Int})Int

scala> foo(new { def get = 10 })
res0: Int = 133
```

This can be quite nice in many situations, but the implementation uses reflection, so be performance-aware!

Abstract type members

In a trait, you can leave type members abstract.

```
scala> trait Foo { type A; val x: A; def getX: A = x }
defined trait Foo
```

```
scala> (new Foo { type A = Int; val x = 123 }).getX
res3: Int = 123

scala> (new Foo { type A = String; val x = "hey" }).getX
res4: java.lang.String = hey
```

This is often a useful trick when doing dependency injection, etc.

You can refer to an abstract type variable using the hash-operator:

```
scala> trait Foo[M[_]] { type t[A] = M[A] }
defined trait Foo

scala> val x: Foo[List]#t[Int] = List(1)
x: List[Int] = List(1)
```

Type erasures & manifests

As we know, type information is lost at compile time due to *erasure*. Scala features **Manifests**, allowing us to selectively recover type information. Manifests are provided as an implicit value, generated by the compiler as needed.

```
scala> class MakeFoo[A](implicit manifest: Manifest[A]) { def make: A = manifest.erasure.newInstance.asInstanceOf[A] }

scala> (new MakeFoo[String]).make
res10: String =
```

Case study: Finagle

See: https://github.com/twitter/finagle

```
trait Service[-Req, +Rep] extends (Req => Future[Rep])

trait Filter[-ReqIn, +RepOut, +ReqOut, -RepIn]
  extends ((ReqIn, Service[ReqOut, RepIn]) => Future[RepOut])
```

```
{
  def andThen[Req2, Rep2](next: Filter[ReqOut, RepIn, Req2, Rep2]) =
    new Filter[ReqIn, RepOut, Req2, Rep2] {
      def apply(request: ReqIn, service: Service[Req2, Rep2]) = {
        Filter.this.apply(request, new Service[ReqOut, RepIn] {
          def apply(request: ReqOut): Future[RepIn] = next(request, service)
          override def release() = service.release()
          override def isAvailable = service.isAvailable
        })
      }
    }

  def andThen(service: Service[ReqOut, RepIn]) = new Service[ReqIn, RepOut] {
    private[this] val refcounted = new RefcountedService(service)

    def apply(request: ReqIn) = Filter.this.apply(request, refcounted)
    override def release() = refcounted.release()
    override def isAvailable = refcounted.isAvailable
  }
}
```

An may authenticate requests with a filter.

```
trait RequestWithCredentials extends Request {
  def credentials: Credentials
}

class CredentialsFilter(credentialsParser: CredentialsParser)
  extends Filter[Request, Response, RequestWithCredentials, Response]
{
  def apply(request: Request, service: Service[RequestWithCredentials, Response]): Future[Response] = {
    val requestWithCredentials = new RequestWrapper with RequestWithCredentials {
      val underlying = request
      val credentials = credentialsParser(request) getOrElse NullCredentials
    }

    service(requestWithCredentials)
```

```
  }
}
```

Note how the underlying service requires an authenticated request, and that this is statically verified. Filters can thus be thought of as service transformers.

Many filters can be composed together:

```
val upFilter =
  logTransaction     andThen
  handleExceptions   andThen
  extractCredentials andThen
  homeUser           andThen
  authenticate       andThen
  route
```

Type safely!

This lesson covers SBT! Specific topics include:

- creating an sbt project
- basic commands
- the sbt console
- continuous command execution
- customizing your project
- custom commands
- quick tour of sbt source (if time)

About SBT

SBT is a modern build tool. While it is written in Scala and provides many Scala conveniences, it is a general purpose build tool.

Why SBT?

- Sane(ish) dependency management
  - Ivy for dependency management
  - Only-update-on-request model
- Full Scala language support for creating tasks
- Continuous command execution
- Launch REPL in project context

Getting Started

- Download the jar:http://code.google.com/p/simple-build-tool/downloads/list
- Create an sbt shell script that calls the jar, e.g.

```
java -Xmx512M -jar sbt-launch.jar "$@"
```

- make sure it's executable and in your path
- run sbt to create your project

```
[local ~/projects]$ sbt
Project does not exist, create new project? (y/N/s) y
Name: sample
Organization: com.twitter
Version [1.0]: 1.0-SNAPSHOT
Scala version [2.7.7]: 2.8.1
sbt version [0.7.4]:
Getting Scala 2.7.7 ...
:: retrieving :: org.scala-tools.sbt#boot-scala
 confs: [default]
 2 artifacts copied, 0 already retrieved (9911kB/221ms)
Getting org.scala-tools.sbt sbt_2.7.7 0.7.4 ...
:: retrieving :: org.scala-tools.sbt#boot-app
 confs: [default]
 15 artifacts copied, 0 already retrieved (4096kB/167ms)
[success] Successfully initialized directory structure.
Getting Scala 2.8.1 ...
:: retrieving :: org.scala-tools.sbt#boot-scala
 confs: [default]
 2 artifacts copied, 0 already retrieved (15118kB/386ms)
[info] Building project sample 1.0-SNAPSHOT against Scala 2.8.1
[info]    using sbt.DefaultProject with sbt 0.7.4 and Scala 2.7.7
>
```

Note that it's good form to start out with a SNAPSHOT version of your project.

Project Layout

- project – project definition files
  - project/build/.scala – the main project definition file
  - project/build.properties – project, sbt and scala version definitions
- src/main – your app code goes here, in a subdirectory indicating the code's language (e.g. src/main/scala, src/main/java)
- src/main/resources – static files you want added to your jar (e.g. logging config)
- src/test – like src/main, but for tests
- lib_managed – the jar files your project depends on. Populated by sbt update
- target – the destination for generated stuff (e.g. generated thrift

code, class files, jars)

Adding Some Code

We'll be creating a simple JSON parser for simple tweets. Add the following code to src/main/project/com/twitter/sample/SimpleParser.scala

```
package com.twitter.sample

case class SimpleParsed(id: Long, text: String)

class SimpleParser {

  val tweetRegex = "\"id\":(.*),\"text\":\"(.*)\"".r

  def parse(str: String) = {
    tweetRegex.findFirstMatchIn(str) match {
      case Some(m) => {
        val id = str.substring(m.start(1), m.end(1)).toInt
        val text = str.substring(m.start(2), m.end(2))
        Some(SimpleParsed(id, text))
      }
      case _ => None
    }
  }
}
```

This is ugly and buggy, but should compile.

Testing in the Console

SBT can be used both as a command line script and as a build console. We'll be primarily using it as a build console, but most commands can be run standalone by passing the command as an argument to SBT, e.g.

```
sbt test
```

Note that if a command takes arguments, you need to quote the entire argument path, e.g.

```
sbt 'test-only com.twitter.sample.SampleSpec'
```

It's weird that way.

Anyway. To start working with our code, launch sbt

```
[local ~/projects/sbt-sample]$ sbt
[info] Building project sample 1.0-SNAPSHOT against Scala 2.8.1
[info]    using sbt.DefaultProject with sbt 0.7.4 and Scala 2.7.7
>
```

SBT allows you to start a Scala REPL with all your project dependencies loaded. It compiles your project source before launching the console, providing us a quick way to bench test our parser.

```
> console
[info]
[info] == compile ==
[info]   Source analysis: 0 new/modified, 0 indirectly invalidated, 0 removed.
[info] Compiling main sources...
[info] Nothing to compile.
[info]   Post-analysis: 3 classes.
[info] == compile ==
[info]
[info] == copy-test-resources ==
[info] == copy-test-resources ==
[info]
[info] == test-compile ==
```

```
[info]   Source analysis: 0 new/modified, 0 indirectly invalidated, 0 removed.
[info] Compiling test sources...
[info] Nothing to compile.
[info]   Post-analysis: 0 classes.
[info] == test-compile ==
[info]
[info] == copy-resources ==
[info] == copy-resources ==
[info]
[info] == console ==
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.8.1.final (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_22).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Our code has compiled, and we're provide the typical Scala prompt. We'll create a new parser, an exemplar tweet, and ensure it "works"

```
scala> import com.twitter.sample._
import com.twitter.sample._

scala> val tweet = """{"id":1,"text":"foo"}"""
tweet: java.lang.String = {"id":1,"text":"foo"}

scala> val parser = new SimpleParser
parser: com.twitter.sample.SimpleParser = com.twitter.sample.SimpleParser@71060c3e

scala> parser.parse(tweet)
res0: Option[com.twitter.sample.SimpleParsed] = Some(SimpleParsed(1,"foo"}))

scala>
```

Adding Dependencies

Our simple parser works for this very small set of inputs, but we want to add tests and break it. The first step is adding the specs test library and a real JSON parser to our project. To do this we have to go beyond the default SBT project layout and create a project.

SBT considers Scala files in the project/build directory to be project definitions. Add the following to project/build/SampleProject.scala

```
import sbt._

class SampleProject(info: ProjectInfo) extends DefaultProject(info) {
  val jackson = "org.codehaus.jackson" % "jackson-core-asl" % "1.6.1"
  val specs = "org.scala-tools.testing" % "specs_2.8.0" % "1.6.5" % "test"
}
```

A project definition is an SBT class. In our case we extend SBT's DefaultProject.

You declare dependencies by specifing a val that is a dependency. SBT uses reflection to scan all the dependency vals in your project and build up a dependency tree at build time. The syntax here may be new, but this is equivalent to the maven dependency

```
<dependency>
  <groupId>org.codehaus.jackson</groupId>
  <artifactId>jackson-core-asl</artifactId>
  <version>1.6.1</version>
</dependency>
<dependency>
  <groupId>org.scala-tools.testing</groupId>
  <artifactId>specs_2.8.0</artifactId>
  <version>1.6.5</version>
  <scope>test</scope>
</dependency>
```

Now we can pull down dependencies for our project. From the command line (not the sbt console), run sbt update

```
[local ~/projects/sbt-sample]$ sbt update
[info] Building project sample 1.0-SNAPSHOT against Scala 2.8.1
[info]    using SampleProject with sbt 0.7.4 and Scala 2.7.7
[info]
[info] == update ==
```

```
[info] :: retrieving :: com.twitter#sample_2.8.1 [sync]
[info]  confs: [compile, runtime, test, provided, system, optional, sources, javadoc]
[info]  1 artifacts copied, 0 already retrieved (2785kB/71ms)
[info] == update ==
[success] Successful.
[info]
[info] Total time: 1 s, completed Nov 24, 2010 8:47:26 AM
[info]
[info] Total session time: 2 s, completed Nov 24, 2010 8:47:26 AM
[success] Build completed successfully.
```

You'll see that sbt retrieved the specs library. You'll now also have a lib_managed directory, and lib_managed/scala_2.8.1/test will have specs_2.8.0-1.6.5.jar

Adding Tests

Now that we have a test library added, put the following code in src/test/scala/com/twitter/sample/SimpleParserSpec.scala

```
package com.twitter.sample

import org.specs._

object SimpleParserSpec extends Specification {
  "SimpleParser" should {
    val parser = new SimpleParser()
    "work with basic tweet" in {
      val tweet = """{"id":1,"text":"foo"}"""
      parser.parse(tweet) match {
        case Some(parsed) => {
          parsed.text must be_==("foo")
          parsed.id must be_==(1)
        }
        case _ => fail("didn't parse tweet")
      }
    }
  }
}
```

```
```

In the sbt console, run test

```
> test
[info]
[info] == compile ==
[info]   Source analysis: 0 new/modified, 0 indirectly invalidated, 0 removed.
[info] Compiling main sources...
[info] Nothing to compile.
[info]   Post-analysis: 3 classes.
[info] == compile ==
[info]
[info] == test-compile ==
[info]   Source analysis: 0 new/modified, 0 indirectly invalidated, 0 removed.
[info] Compiling test sources...
[info] Nothing to compile.
[info]   Post-analysis: 10 classes.
[info] == test-compile ==
[info]
[info] == copy-test-resources ==
[info] == copy-test-resources ==
[info]
[info] == copy-resources ==
[info] == copy-resources ==
[info]
[info] == test-start ==
[info] == test-start ==
[info]
[info] == com.twitter.sample.SimpleParserSpec ==
[info] SimpleParserSpec
[info] SimpleParser should
[info]   + work with basic tweet
[info] == com.twitter.sample.SimpleParserSpec ==
[info]
[info] == test-complete ==
[info] == test-complete ==
[info]
```

```
[info] == test-finish ==
[info] Passed: : Total 1, Failed 0, Errors 0, Passed 1, Skipped 0
[info]
[info] All tests PASSED.
[info] == test-finish ==
[info]
[info] == test-cleanup ==
[info] == test-cleanup ==
[info]
[info] == test ==
[info] == test ==
[success] Successful.
[info]
[info] Total time: 0 s, completed Nov 24, 2010 8:54:45 AM
>
```

Our test works! Now we can add more. One of the nice things SBT provides is a way to run triggered actions. Prefacing an action with a tilde starts up a loop that runs the action any time source files change. Lets run ~test and see what happens.

```
[info] == test ==
[success] Successful.
[info]
[info] Total time: 0 s, completed Nov 24, 2010 8:55:50 AM
1. Waiting for source changes... (press enter to interrupt)
```

Now let's add the following test cases

```
    "reject a non-JSON tweet" in {
      val tweet = """"id":1,"text":"foo""""
      parser.parse(tweet) match {
        case Some(parsed) => fail("didn't reject a non-JSON tweet")
        case e => e must be_==(None)
      }
    }

    "ignore nested content" in {
```

```
    val tweet = """{"id":1,"text":"foo","nested":{"id":2}}"""
    parser.parse(tweet) match {
      case Some(parsed) => {
        parsed.text must be_==("foo")
        parsed.id must be_==(1)
      }
      case _ => fail("didn't parse tweet")
    }
  }

  "fail on partial content" in {
    val tweet = """{"id":1}"""
    parser.parse(tweet) match {
      case Some(parsed) => fail("didn't reject a partial tweet")
      case e => e must be_==(None)
    }
  }
```

After we save our file, SBT detects our changes, runs tests, and informs us our parser is lame

```
[info] == com.twitter.sample.SimpleParserSpec ==
[info] SimpleParserSpec
[info] SimpleParser should
[info]   + work with basic tweet
[info]   x reject a non-JSON tweet
[info]     didn't reject a non-JSON tweet (Specification.scala:43)
[info]   x ignore nested content
[info]     'foo","nested":{"id' is not equal to 'foo' (SimpleParserSpec.scala:31)
[info]   + fail on partial content
```

So let's rework our JSON parser to be real

```
package com.twitter.sample

import org.codehaus.jackson._
import org.codehaus.jackson.JsonToken._
```

```
case class SimpleParsed(id: Long, text: String)

class SimpleParser {

  val parserFactory = new JsonFactory()

  def parse(str: String) = {
    val parser = parserFactory.createJsonParser(str)
    if (parser.nextToken() == START_OBJECT) {
      var token = parser.nextToken()
      var textOpt:Option[String] = None
      var idOpt:Option[Long] = None
      while(token != null) {
        if (token == FIELD_NAME) {
          parser.getCurrentName() match {
            case "text" => {
              parser.nextToken()
              textOpt = Some(parser.getText())
            }
            case "id" => {
              parser.nextToken()
              idOpt = Some(parser.getLongValue())
            }
            case _ => // noop
          }
        }
        token = parser.nextToken()
      }
      if (textOpt.isDefined && idOpt.isDefined) {
        Some(SimpleParsed(idOpt.get, textOpt.get))
      } else {
        None
      }
    } else {
      None
    }
  }
}
```

This is a simple Jackson parser. When we save, SBT recompiles our code and reruns our tests. Getting better!

```
info] SimpleParser should
[info]   + work with basic tweet
[info]   + reject a non-JSON tweet
[info]   x ignore nested content
[info]     '2' is not equal to '1' (SimpleParserSpec.scala:32)
[info]   + fail on partial content
[info] == com.twitter.sample.SimpleParserSpec ==
```

Uhoh. We need to check for nested objects. Let's add some ugly
guards to our token reading loop.

```
  def parse(str: String) = {
    val parser = parserFactory.createJsonParser(str)
    var nested = 0
    if (parser.nextToken() == START_OBJECT) {
      var token = parser.nextToken()
      var textOpt:Option[String] = None
      var idOpt:Option[Long] = None
      while(token != null) {
        if (token == FIELD_NAME && nested == 0) {
          parser.getCurrentName() match {
            case "text" => {
              parser.nextToken()
              textOpt = Some(parser.getText())
            }
            case "id" => {
              parser.nextToken()
              idOpt = Some(parser.getLongValue())
            }
            case _ => // noop
          }
        } else if (token == START_OBJECT) {
```

```
          nested += 1
        } else if (token == END_OBJECT) {
          nested -= 1
        }
        token = parser.nextToken()
      }
      if (textOpt.isDefined && idOpt.isDefined) {
        Some(SimpleParsed(idOpt.get, textOpt.get))
      } else {
        None
      }
    } else {
      None
    }
  }
```

And… it works!

Packaging and Publishing

At this point we can run the package command to generate a jar file. However we may want to share our jar with other teams. To do this we'll build on StandardProject, which gives us a big head start.

The first step is include StandardProject as an SBT plugin. Plugins are a way to introduce dependencies to your build, rather than your project. These dependencies are defined in project/plugins/Plugins.scala. Add the following to the Plugins.scala file.

```
import sbt._

class Plugins(info: ProjectInfo) extends PluginDefinition(info) {
  val twitterMaven = "twitter.com" at "http://maven.twttr.com/"
  val defaultProject = "com.twitter" % "standard-project" % "0.7.14"
}
```

Note that we've specified a maven repository as well as a dependency. That's because the standard project library is hosted by us, which isn't one of the default repos sbt checks.

We'll also update our project definition to extend StandardProject, include an SVN publishing trait, and define the repository we wish to publish to. Alter SampleProject.scala to the following

```
import sbt._
import com.twitter.sbt._

class SampleProject(info: ProjectInfo) extends StandardProject(info) with SubversionPublisher {
  val jackson = "org.codehaus.jackson" % "jackson-core-asl" % "1.6.1"
  val specs = "org.scala-tools.testing" % "specs_2.8.0" % "1.6.5" % "test"

  override def subversionRepository = Some("http://svn.local.twitter.com/maven/")
}
```

Now if we run the publish action we'll see the following

```
[info] == deliver ==
IvySvn Build-Version: null
IvySvn Build-DateTime: null
[info] :: delivering :: com.twitter#sample;1.0-SNAPSHOT :: 1.0-SNAPSHOT :: release :: Wed Nov 24 10:26:45 PST 2010
[info]  delivering ivy file to /Users/mmcbride/projects/sbt-sample/target/ivy-1.0-SNAPSHOT.xml
[info] == deliver ==
[info]
[info] == make-pom ==
[info] Wrote /Users/mmcbride/projects/sbt-sample/target/sample-1.0-SNAPSHOT.pom
[info] == make-pom ==
[info]
[info] == publish ==
[info] :: publishing :: com.twitter#sample
[info] Scheduling publish to http://svn.local.twitter.com/maven/com/twitter/sample/1.0-SNAPSHOT/sample-1.0-SNAPSHOT.jar
[info]  published sample to com/twitter/sample/1.0-SNAPSHOT/sample-1.0-SNAPSHOT.jar
[info] Scheduling publish to http://svn.local.twitter.com/maven/com/twitter/sample/1.0-SNAPSHOT/sample-1.0-SNAPSHOT.pom
[info]  published sample to com/twitter/sample/1.0-SNAPSHOT/sample-1.0-SNAPSHOT.pom
[info] Scheduling publish to http://svn.local.twitter.com/maven/com/twitter/sample/1.0-SNAPSHOT/ivy-1.0-SNAPSHOT.xml
[info]  published ivy to com/twitter/sample/1.0-SNAPSHOT/ivy-1.0-SNAPSHOT.xml
[info] Binary diff deleting com/twitter/sample/1.0-SNAPSHOT
[info] Commit finished r977 by 'mmcbride' at Wed Nov 24 10:26:47 PST 2010
[info] Copying from com/twitter/sample/.upload to com/twitter/sample/1.0-SNAPSHOT
[info] Binary diff finished : r978 by 'mmcbride' at Wed Nov 24 10:26:47 PST 2010
[info] == publish ==
```

```
[success] Successful.
[info]
[info] Total time: 4 s, completed Nov 24, 2010 10:26:47 AM
```

And (after some time), we can go to binaries.local.twitter.com:http://binaries.local.twitter.com/maven/com/twitter/sample/1.0- SNAPSHOT/ to see our published jar.

Adding Tasks

Tasks are Scala functions. The simplest way to add a task is to include a val in your project definition using the task method, e.g.

```
lazy val print = task {log.info("a test action"); None}
```

If you want dependencies and a description you can add them like this

```
lazy val print = task {log.info("a test action"); None}.dependsOn(compile) describedAs("prints a line after compile")
```

If we reload our project and run the print action we'll see the following

```
> print
[info]
[info] == print ==
[info] a test action
[info] == print ==
[success] Successful.
[info]
[info] Total time: 0 s, completed Nov 24, 2010 11:05:12 AM
>
```

So it works. If you're defining a task in a single project this works just fine. However if you're defining this in a plugin it's fairly inflexible. I may want to

```
lazy val print = printAction
def printAction = printTask.dependsOn(compile) describedAs("prints a line after compile")
```

```
def printTask = task {log.info("a test action"); None}
```

This allows consumers to override the task itself, the dependencies and/or description of the task, or the action. Most built in SBT actions follow this pattern. As an example, we can modify the builtin package task to print the current timestamp by doing the following

```
lazy val printTimestamp = task { log.info("current time is " + System.currentTimeMillis); None}
override def packageAction = super.packageAction.dependsOn(printTimestamp)
```

There are many examples in StandardProject of tweaking SBT defaults and adding custom tasks.

Quick Reference

Common Commands

- actions – show actions available for this project
- update – downloads dependencies
- compile – compiles source
- test – runs tests
- package – creates a publishable jar file
- publish-local – installs the built jar in your local ivy cache
- publish – pushes your jar to a remote repo (if configured)

Moar Commands

- test-failed – run any specs that failed
- test-quick – run any specs that failed and/or had dependencies updated
- clean-cache – remove all sorts of sbt cached stuff. Like clean for sbt
- clean-lib – remove everything in lib_managed

Project Layout

TBD

Built at @twitter by @stevej and @marius with much help from @evanm, @sprsquish, @kevino, @zuercher, @timtrueman, @wickman and @mccv

This lesson covers testing with Specs, a BDD Framework for Scala.

extends Specfcation

Let's just jump in.

```
import org.specs._

object ArithmeticSpec extends Specification {
  "Arithmetic" should {
    "add two numbers" in {
      1 + 1 mustEqual 2
    }
    "add three numbers" in {
      1 + 1 + 1 mustEqual 3
    }
  }
}
```

Let's break this down

Arithmetic is the **System Under Specification**

add is a context.

add two numbers and **add three numbers** are examples.

`mustEqual` indicates an **expectation**

`1 mustEqual 1` is a common placeholder **expectation** before you start writing real tests. All examples should have at least one expectation.

Duplication

Notice how two tests both have `add` in their name? We can get rid of that by **nesting** expectations.

```
import org.specs._

object ArithmeticSpec extends Specification {
  "Arithmetic" should {
    "add" in {
      "two numbers" in {
        1 + 1 mustEqual 2
      }
      "three numbers" in {
        1 + 1 + 1 mustEqual 3
      }
    }
  }
}
```

Execution Model

```
object ExecSpec extends Specification {
  "Mutations are isolated" should {
    var x = 0
    "x equals 1 if we set it." in {
      x = 1
```

```
      x mustEqual 1
    }
    "x is the default value if we don't change it" in {
      x mustEqual 0
    }
  }
}
```

Setup

doBefore & doAfter

```
"my system" should {
  doBefore { resetTheSystem() /** user-defined reset function */ }
  "mess up the system" in {...}
  "and again" in {...}
  doAfter { cleanThingsUp() }
}
```

NOTE `doBefore`/`doAfter` are only run on leaf examples.

doFirst & doLast

`doFirst`/`doLast` is for single-time setup. (need example, I don't use this)

```
"Foo" should {
  doFirst { openTheCurtains() }
  "test stateless methods" in {...}
  "test other stateless methods" in {...}
  doLast { closeTheCurtains() }
}
```

Matchers

You have data, you want to make sure it's right.

Let's tour the most commonly used matchers.

Matchers Guide

mustEqual

We've seen several examples of mustEqual already.

```
1 mustEqual 1

"a" mustEqual "a"
```

Reference equality, value equality.

elements in a Sequence

```
val numbers = List(1, 2, 3)

numbers must contain(1)
numbers must not contain(4)

numbers must containAll(List(1, 2, 3))
numbers must containInOrder(List(1, 2, 3))

List(1, List(2, 3, List(4)), 5) must haveTheSameElementsAs(List(5, List(List(4), 2, 3), 1))
```

Items in a Map

```
map must haveKey(k)
map must notHaveKey(k)

map must haveValue(v)
map must notHaveValue(v)
```

Numbers

```
a must beGreaterThan(b)
a must beGreaterThanOrEqualTo(b)

a must beLessThan(b)
a must beLessThanOrEqualTo(b)

a must beCloseTo(b, delta)
```

Options

```
a must beNone

a must beSome[Type]

a must beSomething

a must beSome(value)
```

throwA

```
a must throwA[WhateverException]
```

This is shorter than a try catch with a fail in the body.

You can also expect a specific message

```
a must throwA(WhateverException("message"))
```

You can also match on the exception:

```
a must throwA(new Exception) like {
  case Exception(m) => m.startsWith("bad")
}
```

Write your own Matchers

As a val

```
import org.specs.matcher.Matcher
```

```
“A matcher” should {

  “be created as a val” in {

    val beEven = new Matcher[Int] {

      def apply(n: => Int) = {

        (n % 2 == 0, “%d is even”.format(n), “%d is odd”.format(n))

      }

    }

    2 must beEven

  }
```

```
}
```

The contract is to return a tuple containing whether the expectation is true, and a message for when it is and isn't true.

As a case class

```
case class beEven(b: Int) extends Matcher[Int]() {
  def apply(n: => Int) =  (n % 2 == 0, "%d is even".format(n), "%d is odd".format(n))
}
```

Using a case class makes it more shareable.

Mocks

```
import org.specs.Specification
import org.specs.mock.Mockito

class Foo[T] {
  def get(i: Int): T
}

object MockExampleSpec extends Specification with Mockito {
  val m = mock[Foo[String]]

  m.get(0) returns "one"

  m.get(0)

  there was one(m).get(0)

  there was no(m).get(1)
}
```

Spies

Spies can also be used in order to do some “partial mocking” of real objects:

```
val list = new LinkedList[String]
val spiedList = spy(list)

// methods can be stubbed on a spy
spiedList.size returns 100

// other methods can also be used
spiedList.add("one")
spiedList.add("two")

// and verification can happen on a spy
there was one(spiedList).add("one")
```

However, working with spies can be tricky:

```
// if the list is empty, this will throws an IndexOutOfBoundsException
spiedList.get(0) returns "one"
```

`doReturn` must be used in that case:

```
doReturn("one").when(spiedList).get(0)
```

Run individual specs in sbt

```
> test-only com.twitter.yourservice.UserSpec
```

Will run just that spec.

```
> ~ test-only com.twitter.yourservice.UserSpec
```

Will run that test in a loop, with each file modification triggering a test run.

Built at @twitter by @stevej and @marius with much help from @evanm, @sprsquish, @kevino, @zuercher, @timtrueman, @wickman and @mccv

Runnable/Callable

Runnable has a single method that returns no value.

```
trait Runnable {
  def run(): Unit
}
```

Callable is similar to run except that it returns a value

```
trait Callable[V] {
  def call(): V
}
```

Threads

Scala concurrency is built on top of the Java concurrency model.

On Sun JVMs, with a IO-heavy workload, we can run tens of thousands of threads on a single machine.

A Thread takes a Runnable. You have to call `start` on a Thread in order for it to run the Runnable.

```
scala> val hello = new Thread(new Runnable {
  def run() {
    println("hello world")
  }
})
hello: java.lang.Thread = Thread[Thread-3,5,main]

scala> hello.start
hello world
```

When you see a class implementing Runnable, you know it's intended to run in a Thread somewhere by somebody.

Something single-threaded

Here's a code snippet that works but has problems.

```
import java.net.{Socket, ServerSocket}
import java.util.concurrent.{Executors, ExecutorService}
import java.util.Date

class NetworkService(port: Int, poolSize: Int) extends Runnable {
  val serverSocket = new ServerSocket(port)

  def run() {
    while (true) {
      // This will block until a connection comes in.
      val socket = serverSocket.accept()
      (new Handler(socket)).run()
    }
  }
}

class Handler(socket: Socket) extends Runnable {
  def message = (Thread.currentThread.getName() + "\n").getBytes
```

```
  def run() {
    socket.getOutputStream.write(message)
    socket.getOutputStream.close()
  }
}

(new NetworkService(2020, 2)).run
```

Each request will respond with the name of the current Thread, which is always `main`.

The main drawback with this code is that only one request at a time can be answered!

You could put each request in a Thread. Simply change

```
(new Handler(socket)).run()
```

to

```
(new Thread(new Handler(socket))).start()
```

but what if you want to reuse threads or have other policies about thread behavior?

Executors

With the release of Java 5, it was decided that a more abstract interface to Threads was required.

You can get an `ExecutorService` using static methods on the `Executors` object. Those methods provide you to configure an `ExecutorService` with a variety of policies such as thread pooling.

Here's our old blocking network server written to allow concurrent requests.

```
import java.net.{Socket, ServerSocket}
import java.util.concurrent.{Executors, ExecutorService}
import java.util.Date

class NetworkService(port: Int, poolSize: Int) extends Runnable {
```

```
  val serverSocket = new ServerSocket(port)
  val pool: ExecutorService = Executors.newFixedThreadPool(poolSize)

  def run() {
    try {
      while (true) {
        // This will block until a connection comes in.
        val socket = serverSocket.accept()
        pool.execute(new Handler(socket))
      }
    } finally {
      pool.shutdown()
    }
  }
}

class Handler(socket: Socket) extends Runnable {
  def message = (Thread.currentThread.getName() + "\n").getBytes

  def run() {
    socket.getOutputStream.write(message)
    socket.getOutputStream.close()
  }
}

(new NetworkService(2020, 2)).run
```

Here’s a transcript connecting to it showing how the internal threads are re-used.

```
$ nc localhost 2020
pool-1-thread-1

$ nc localhost 2020
pool-1-thread-2

$ nc localhost 2020
pool-1-thread-1
```

```
$ nc localhost 2020
pool-1-thread-2
```

Futures

A `Future` represents an asynchronous computation. You can wrap your computation in a Future and when you need the result, you simply call a blocking `get()` method on it. @Executor@s return a `Future`.

A `FutureTask` is a Runnable and is designed to be run by an `Executor`

```
val future = new FutureTask[String](new Callable[String]() {
  def call(): String = {
    searcher.search(target);
}})
executor.execute(future)
```

Now I need the results so let's block until its done.

```
val blockingResult = future.get()
```

Thread Safety

```
class Person(var name: String) {
  def set(changedName: String) {
    name = changedName
  }
}
```

This program is not safe in a multi-threaded environment. If two threads have references to the same instance of an Adder and call `add`, you can't predict what `i` will be at the end of both calls. It could be 2, it could be 1!

In the Java memory model, each processor is allowed to cache values in it's L1 or L2 cache so two threads running on different processors can each have their own view of data.

Let's talk about some of the tools that force threads to keep a consistent view of data.

Three tools

synchronization

Mutexes provide ownership semantics. When you enter a mutex, you own it. The most common way of using a mutex in the JVM is by synchronizing on something. In this case, we'll synchronize on our userMap.

In the JVM, you can synchronize on any instance that's not null.

```
class Person(var name: String) {
  def set(changedName: String) {
    this.synchronized {
      name = changedName
    }
  }
}
```

volatile

With Java 5's change to the memory model, volatile and synchronized are basically identical except with volatile, nulls are allowed.

`synchronized` allows for more fine-grained locking. `volatile` synchronizes on every access.

```
class Person(@volatile var name: String) {
  def set(changedName: String) {
    name = changedName
  }
}
```

AtomicReference

Also in Java 5, a whole raft of low-level concurrency primitives were added. One of them is an `AtomicReference` class

```
import java.util.concurrent.atomic.AtomicReference
```

```
class Person(val name: AtomicReference[String]) {
  def set(changedName: String) {
    name.set(changedName)
  }
}
```

Does this cost anything?

@AtomicReference is the most costly of these two choices since you have to go through method dispatch to access values.

`volatile` and `synchronized` are built on top of Java's built-in monitors. Monitors cost very little if there's no contention. Since `synchronized` allows you more fine-grained control over when you synchronize, there will be less contention so `synchronized` tends to be the cheapest option.

When you enter synchronized points, access volatile references, or deference AtomicReferences, Java forces the processor to flush their cache lines and provide a consistent view of data.

PLEASE CORRECT ME IF I'M WRONG HERE. This is a complicated subject, I'm sure there will be a lengthy classroom discussion at this point.

Other neat tools from Java 5

As I mentioned with `AtomicReference`, Java 5 brought many great tools along with it.

CountDownLatch

A `CountDownLatch` is a simple mechanism for multiple threads to communicate with each other.

```
val doneSignal = new CountDownLatch(2)
doAsyncWork(1)
doAsyncWork(2)

doneSignal.await()
println("both workers finished!")
```

Among other things, it's great for unit tests. Let's say you're doing some async work and want to ensure that functions are completing. Simply have your functions `countDown` the latch and `await` in the test.

AtomicInteger/Long

Since incrementing Ints and Longs is such a common task, `AtomicInteger` and `AtomicLong` were added.

AtomicBoolean

I probably don’t have to explain what this would be for.

ReadWriteLocks

`ReadWriteLock` lets you take reader and writer locks. reader locks only block when a writer lock is taken.

Let’s build an unsafe search engine

Here’s a simple inverted index that isn’t thread-safe. Our inverted index maps parts of a name to a given User.

This is written in a naive way assuming only single-threaded access.

Note the alternative default constructor `this()` that uses a `mutable.HashMap`

```
import scala.collection.mutable

case class User(name: String, id: Int)

class InvertedIndex(val userMap: mutable.Map[String, User]) {

  def this() = this(new mutable.HashMap[String, User])

  def tokenizeName(name: String): Seq[String] = {
    name.split(" ").map(_.toLowerCase)
  }

  def add(term: String, user: User) {
    userMap += term -> user
  }

  def add(user: User) {
    tokenizeName(user.name).foreach { term =>
      add(term, user)
    }
  }
}
```

I've left out how to get users out of our index for now. We'll get to that later.

Let's make it safe

In our inverted index example above, userMap is not guaranteed to be safe. Multiple clients could try to add items at the same time and have the same kinds of visibility errors we saw in our first `Person` example.

Since userMap isn't thread-safe, how we do we keep only a single thread at a time mutating it?

You might consider locking on userMap while adding.

```
def add(user: User) {
  userMap.synchronized {
    tokenizeName(user.name).foreach { term =>
      add(term, user)
    }
  }
}
```

Unfortunately, this is too coarse. Always try to do as much expensive work outside of the mutex as possible. Remember what I said about locking being cheap if there is no contention. If you do less work inside of a block, there will be less contention.

```
def add(user: User) {
  // tokenizeName was measured to be the most expensive operation.
  val tokens = tokenizeName(user.name)

  tokens.foreach { term =>
    userMap.synchronized {
      add(term, user)
    }
  }
}
```

SynchronizedMap

We can mixin synchronization with a mutable HashMap using the SynchronizedMap trait.

We can extend our existing InvertedIndex to give users an easy way to build the synchronized index.

```
import scala.collection.mutable.SynchronizedMap

class SynchronizedInvertedIndex(userMap: mutable.Map[String, User]) extends InvertedIndex(userMap) {
  def this() = this(new mutable.HashMap[String, User] with SynchronizedMap[String, User])
}
```

If you look at the implementation, you realize that it's simply synchronizing on every method so while it's safe, it might not have the performance you're hoping for.

Java ConcurrentHashMap

Java comes with a nice thread-safe ConcurrentHashMap. Thankfully, we can use JavaConversions to give us nice Scala semantics.

In fact, we can seamlessly layer our new, thread-safe InvertedIndex as an extension of the old unsafe one.

```
import java.util.concurrent.ConcurrentHashMap
import scala.collection.JavaConversions._

class ConcurrentInvertedIndex(userMap: collection.mutable.ConcurrentMap[String, User])
    extends InvertedIndex(userMap) {

  def this() = this(new ConcurrentHashMap[String, User])
}
```

Let's load our InvertedIndex

The naive way

```
trait UserMaker {
  def makeUser(line: String) = line.split(",") match {
    case Array(name, userid) => User(name, userid.trim().toInt)
  }
}
```

```
class FileRecordProducer(path: String) extends UserMaker {
  def run() {
    Source.fromFile(path, "utf-8").getLines.foreach { line =>
      index.add(makeUser(line))
    }
  }
}
```

For every line in our file, we call `makeUser` and then `add` it to our InvertedIndex. If we use a concurrent InvertedIndex, we can call add in parallel and since makeUser has no side-effects, it's already thread-safe.

We can't read a file in parallel but we *can* build the User and add it to the index in parallel.

A solution: Producer/Consumer

A common pattern for async computation is to separate producers from consumers and have them only communicate via a `Queue`. Let's walk through how that would work for our search engine indexer.

```
import java.util.concurrent.{BlockingQueue, LinkedBlockingQueue}

// Concrete producer
class Producer[T](path: String, queue: BlockingQueue[T]) implements Runnable {
  public void run() {
    Source.fromFile(path, "utf-8").getLines.foreach { line =>
      queue.put(line)
    }
  }
}

// Abstract consumer
abstract class Consumer[T](queue: BlockingQueue[T]) implements Runnable {
  public void run() {
    while (true) {
      val item = queue.take()
      consume(item)
    }
  }
```

```
  def consume(x: T)
}

val queue = new LinkedBlockingQueue[String]()

// One thread for the consumer
val producer = new Producer[String]("users.txt", q)
new Thread(producer).start()

trait UserMaker {
  def makeUser(line: String) = line.split(",") match {
    case Array(name, userid) => User(name, userid.trim().toInt)
  }
}

class IndexerConsumer(index: InvertedIndex, queue: BlockingQueue[String]) extends Consumer[String](queue) with UserMaker {
  def consume(t: String) = index.add(makeUser(t))
}

// Let's pretend we have 8 cores on this machine.
val cores = 8
val pool = Executors.newFixedThreadPool(cores)

// Submit one consumer per core.
for (i <- i to cores) {
  pool.submit(new IndexerConsumer[String](index, q))
}
```

This lesson covers Java interoperability.

- Javap
- Classes
- Exceptions
- Traits
- Objects
- Closures and Functions
- Variance

Javap

javap is a tool that ships with the JDK. Not the JRE. There's a difference. Javap decompiles class definitions and shows you what's inside. Usage is pretty simple

```
[local ~/projects/interop/target/scala_2.8.1/classes/com/twitter/interop]$ javap MyTrait
Compiled from "Scalaisms.scala"
public interface com.twitter.interop.MyTrait extends scala.ScalaObject{
    public abstract java.lang.String traitName();
    public abstract java.lang.String upperTraitName();
}
```

If you're hardcore you can look at byte code

```
[local ~/projects/interop/target/scala_2.8.1/classes/com/twitter/interop]$ javap -c MyTrait\$class
Compiled from "Scalaisms.scala"
public abstract class com.twitter.interop.MyTrait$class extends java.lang.Object{
public static java.lang.String upperTraitName(com.twitter.interop.MyTrait);
  Code:
   0: aload_0
   1: invokeinterface #12,  1; //InterfaceMethod com/twitter/interop/MyTrait.traitName:()Ljava/lang/String;
   6: invokevirtual #17; //Method java/lang/String.toUpperCase:()Ljava/lang/String;
   9: areturn
```

```
public static void $init$(com.twitter.interop.MyTrait);
  Code:
   0: return

}
```

If you start wondering why stuff doesn’t work in Java land, reach for javap!

Classes

The four major items to consider when using a Scala *class* from Java are

- Class parameters
- Class vals
- Class vars
- Exceptions

We’ll construct a simple scala class to show the full range of entities

```
package com.twitter.interop

import java.io.IOException
import scala.throws
import scala.reflect.{BeanProperty, BooleanBeanProperty}

class SimpleClass(name: String, val acc: String, @BeanProperty var mutable: String) {
  val foo = "foo"
  var bar = "bar"
  @BeanProperty
  val fooBean = "foobean"
  @BeanProperty
  var barBean = "barbean"
  @BooleanBeanProperty
  var awesome = true

  def dangerFoo() = {
    throw new IOException("SURPRISE!")
  }
```

```
  @throws(classOf[IOException])
  def dangerBar() = {
    throw new IOException("NO SURPRISE!")
  }
}
```

Class parameters

- by default, class parameters are effectively constructor args in Java land. This means you can't access them outside the class.
- declaring a class parameter as a val/var is the same as this code

```
class SimpleClass(acc_: String) {
  val acc = acc_
}
```

which makes it accessible from Java code just like other vals

Vals

- vals get a method defined for access from Java. You can access the value of the val "foo" via the method "foo()"

Vars

- vars get a method _$eq defined. You can call it like so

```
foo$_eq("newfoo");
```

BeanProperty

You can annotate vals and vars with the @BeanProperty annotation. This generates getters/setters that look like POJO getter/setter definitions. If you want the isFoo variant, use the BooleanBeanProperty annotation. The ugly foo$_eq becomes

```
setFoo("newfoo");
```

```
getFoo();
```

Exceptions

Scala doesn’t have checked exceptions. Java does. This is a philosophical debate we won’t get into, but it **does** matter when you want to catch an exception in Java. The definitions of dangerFoo and dangerBar demonstrate this. In Java I can’t do this

```
        // exception erasure!
        try {
            s.dangerFoo();
        } catch (IOException e) {
            // UGLY
        }
```

Java complains that the body of s.dangerFoo never throws IOException. We can hack around this by catching Throwable, but that’s lame.

Instead, as a good Scala citizen it’s a decent idea to use the throws annotation like we did on dangerBar. This allows us to continue using checked exceptions in Java land.

Further Reading

A full list of Scala annotations for supporting Java interop can be found here http://www.scala-lang.org/node/106.

Traits

How do you get an interface + implementation? Let’s take a simple trait definition and look

```
trait MyTrait {
  def traitName:String
  def upperTraitName = traitName.toUpperCase
}
```

This trait has one abstract method (traitName) and one implemented method (upperTraitName). What does Scala generate for us? An interface named MyTrait, and a companion implementation named MyTrait$class.

The implementation of MyTrait is what you’d expect

```
[local ~/projects/interop/target/scala_2.8.1/classes/com/twitter/interop]$ javap MyTrait
Compiled from "Scalaisms.scala"
public interface com.twitter.interop.MyTrait extends scala.ScalaObject{
    public abstract java.lang.String traitName();
    public abstract java.lang.String upperTraitName();
}
```

The implementation of MyTrait$class is more interesting

```
[local ~/projects/interop/target/scala_2.8.1/classes/com/twitter/interop]$ javap MyTrait\$class
Compiled from "Scalaisms.scala"
public abstract class com.twitter.interop.MyTrait$class extends java.lang.Object{
    public static java.lang.String upperTraitName(com.twitter.interop.MyTrait);
    public static void $init$(com.twitter.interop.MyTrait);
}
```

MyTrait$class has only static methods that take an instance of MyTrait. This gives us a clue as to how to extend a Trait in Java.

Our first try is the following

```
package com.twitter.interop;

public class JTraitImpl implements MyTrait {
    private String name = null;

    public JTraitImpl(String name) {
        this.name = name;
    }

    public String traitName() {
        return name;
    }
}
```

And we get the following error

```
[info] Compiling main sources...
[error] /Users/mmcbride/projects/interop/src/main/java/com/twitter/interop/JTraitImpl.java:3: com.twitter.interop.JTraitImpl is
not abstract and does not override abstract method upperTraitName() in com.twitter.interop.MyTrait
[error] public class JTraitImpl implements MyTrait {
[error]        ^
```

We *could* just implement this ourselves. But there's a sneakier way.

```
package com.twitter.interop;

    public String upperTraitName() {
        return MyTrait$class.upperTraitName(this);
    }
```

We can just delegate this call to the generated Scala implementation. We can also override it if we want.

Objects

Objects are the way Scala implements static methods/singletons. Using them from Java is a bit odd. There isn't a stylistically perfect way to use them, but in Scala 2.8 it's not terrible

A Scala object is compiled to a class that has a trailing "$". Let's set up a class and a companion object

```
class TraitImpl(name: String) extends MyTrait {
  def traitName = name
}

object TraitImpl {
  def apply = new TraitImpl("foo")
  def apply(name: String) = new TraitImpl(name)
}
```

We can naïvely access this in Java like so

```
MyTrait foo = TraitImpl$.MODULE$.apply("foo");
```

Now you may be asking yourself, WTF? This is a valid response. Let's look at what's actually inside TraitImpl$

```
local ~/projects/interop/target/scala_2.8.1/classes/com/twitter/interop]$ javap TraitImpl\$
Compiled from "Scalaisms.scala"
public final class com.twitter.interop.TraitImpl$ extends java.lang.Object implements scala.ScalaObject{
    public static final com.twitter.interop.TraitImpl$ MODULE$;
    public static {};
    public com.twitter.interop.TraitImpl apply();
    public com.twitter.interop.TraitImpl apply(java.lang.String);
}
```

There actually aren't any static methods. Instead it has a static member named MODULE$. The method implementations delegate to this member. This makes access ugly, but workable if you know to use MODULE$.

Forwarding Methods

In Scala 2.8 dealing with Objects got quite a bit easier. If you have a class with a companion object, the 2.8 compiler generates forwarding methods on the companion class. So if you built with 2.8, you can access methods in the TraitImpl Object like so

```
MyTrait foo = TraitImpl.apply("foo");
```

Closures Functions

One of Scala's most important features is the treatment of functions as first class citizens. Let's define a class that defines some methods that take functions as arguments.

```
class ClosureClass {
  def printResult[T](f: => T) = {
    println(f)
  }
```

```
  def printResult[T](f: String => T) = {
    println(f("HI THERE"))
  }
}
```

In Scala I can call this like so

```
val cc = new ClosureClass
cc.printResult { "HI MOM" }
```

In Java it's not so easy, but it's not terrible either. Let's see what ClosureClass actually compiled to:

```
[local ~/projects/interop/target/scala_2.8.1/classes/com/twitter/interop]$ javap ClosureClass
Compiled from "Scalaisms.scala"
public class com.twitter.interop.ClosureClass extends java.lang.Object implements scala.ScalaObject{
    public void printResult(scala.Function0);
    public void printResult(scala.Function1);
    public com.twitter.interop.ClosureClass();
}
```

This isn't so scary. "f: => T" translates to "Function0", and "f: String => T" translates to "Function1". Scala actually defines Function0 through Function22, supporting this stuff up to 22 arguments. Which really should be enough.

Now we just need to figure out how to get those things going in Java. Turns out Scala provides an AbstractFunction0 and an AbstractFunction1 we can pass in like so

```
    @Test public void closureTest() {
        ClosureClass c = new ClosureClass();
        c.printResult(new AbstractFunction0() {
                public String apply() {
                    return "foo";
                }
            });
        c.printResult(new AbstractFunction1<String, String>() {
                public String apply(String arg) {
```

```
                    return arg + "foo";
                }
            });
    }
```

Note that we can use generics to parameterize arguments.

Finagle is Twitter's RPC system. This blog post explains its motivations and core design tenets, the finagle README contains more detailed documentation. Finagle aims to make it easy to build robust clients and servers.

Futures

Finagle uses `com.twitter.util.Future`[1] to express delayed operations. Futures are highly expressive and composable, allowing for the succinct expression of concurrent and sequential operations with great clarity. Futures are a handle for a value not yet available, with methods to register callbacks to be invoked when the value becomes available. They invert the "traditional" model of asynchronous computing which typically expose APIs similar to this:

```
Callback<R> cb = new Callback<R>() {
  void onComplete(R result) { … }
  void onFailure(Throwable error) { … }
}

dispatch(req, cb);
```

Here, the `Callback.onComplete` is invoked when the result of the `dispatch` operation is available, and `Callback.onFailure` if the operation fails. With futures, we instead invert this control flow:

```
val future = dispatch(req)
future onSuccess { value => … }
future onFailure { error => … }
```

Futures themselves have combinators similar to those we've encountered before in the various collections APIs. Combinators work by exploiting a uniform API, wrapping some underlying `Future` with new behavior without modifying that underlying `Future`.

Sequential composition

The most important `Future` combinator is `flatMap`[2]:

> `def Future[A].flatMap[B](f: A => Future[B]): Future[B]`

`flatMap` sequences two features. The method signature tells the story: given the succesful value of the future `f` must provide the next `Future`. The result of this operation is another `Future` that is complete only when both of these futures have completed. If either `Future` fails, the given `Future` will also fail. This implicit interleaving of errors allow us to handle errors only in those places where they are semantically significant. `flatMap` is the standard name for the combinator with these semantics. Scala also has syntactic shorthand to invoke it: the `for` comprehension.

As an example, let's assume we have methods `authenticate: Request -> User`, and `rateLimit: User -> Boolean`, then the following code:

```
val f = authenticate(request) flatMap { u =>
  rateLimit(u) map { r => (u, r)
}
```

With the help of for-comprehensions, we can write the above as:

```
val f = for {
  u <- authenticate(request)
  r <- rateLimit(u)
} yield (u, r)
```

produces a future `f: Future[(User, Boolean)]` that provides both the user object and and a boolean indicating whether that user has been rate limited. Note how sequential composition is required here: `rateLimit` takes as an argument the output of `authenticate`

Concurrent composition

There are also a number of concurrent combinators. Generally these convert a sequence of `Future` into a `Future` of sequence, in slightly different ways:

```
object Future {
  …
  def collect[A](fs: Seq[Future[A]]): Future[Seq[A]]
  def join(fs: Seq[Future[_]]): Future[Unit]
  def select(fs: Seq[Future[A]]) : Future[(Try[A], Seq[Future[A]])]
}
```

`collect` is the most straightforward one: given a set of `Future`s of the same type, we are given a `Future` of a sequence of values of that type. This future is complete when all of the underlying futures have completed, or when any of them have failed.

`join` takes a sequence of `Future`s whose types may be mixed, yielding a `Future[Unit]` that is completely when all of the underlying futures are (or fails if any of them do). This is useful for indicating the completion of a set of heterogeneous operations.

`select` returns a `Future` that is complete when the first of the given `Future`s complete, together with the remaining uncompleted futures.

In combination, this allows for powerful and concise expression of operations typical of network services. This hypothetical code performs rate limiting (in order to maintain a local rate limit cache) concurrently with dispatching a request on behalf of the user to the backend:

```
def serve(request: Request): Future[Response] = {
  val userLimit: Future[(User, Boolean)] =
    for {
      user    <- auth(request)
      limited <- isLimit(user)
    } yield (user, limited)

  val done =
    dispatch(request) join userLimit

  done flatMap { case (rep, (usr, lim)) =>
    if (lim) {
      updateLocalRateLimitCache(usr)
      Future.exception(new Exception("rate limited"))
    } else {
      Future.value(rep)
    }
  }
}
```

This hypothetical example combines both sequential and concurrent composition. Also note how there is no explicit error handling other than converting a rate limiting reply to an exception. If any future fails here, it is automatically propagated to the returned `Future`.

Service

A `Service` is a function `Req => Future[Rep]` for some request and reply types. `Service` is used by both clients and servers: servers implement `Service` and clients use builders to create one used for querying.

> `abstract class Service[-Req, +Rep] extends (Req => Future[Rep])`

A simple HTTP client might do:

```
service: Service[HttpRequest, HttpResponse]
```

```
val f = service(HttpRequest("/", HTTP_1_1))
f onSuccess { res =>
  println("got response", res)
} onFailure { exc =>
  println("failed :-(", exc)
}
```

Servers implement `Service`:

```
class MyServer
  extends Service[HttpRequest, HttpResponse]
{
  def apply(request: HttpRequest) = {
    request.path match {
      case "/" =>
        Future.value(HttpResponse("root"))
      case _ =>
        Future.value(HttpResponse("default"))
    }
  }
}
```

Combining them is easy. A simple proxy might look like this:

```
class MyServer(client: Service[..])
  extends Service[HttpRequest, HttpResponse]
{
  def apply(request: HttpRequest) = {
    client(rewriteReq(request)) map { res =>
      rewriteRes(res)
    }
  }
}
```

where `rewriteReq` and `rewriteRes` can provide protocol translation, for example.

Filters

Filters are service transformers. They are useful both for providing functionality that's *service generic* as well as factoring a given service into distinct phases.

```
abstract class Filter[-ReqIn, +RepOut, +ReqOut, -RepIn]
  extends ((ReqIn, Service[ReqOut, RepIn]) => Future[RepOut])
```

Its type is better viewed diagramatically:

```
    ((ReqIn, Service[ReqOut, RepIn])
         => Future[RepOut])


           (*   Service   *)
[ReqIn -> (ReqOut -> RepIn) -> RepOut]
```

Here's how you might write a filter that provides a service timeout mechanism.

```
class TimeoutFilter[Req, Rep](
    timeout: Duration, timer: util.Timer)
  extends Filter[Req, Rep, Req, Rep]
{
  def apply(
    request: Req, service: Service[Req, Rep]
  ): Future[Rep] = {
    service(request).timeout(timer, timeout) {
      Throw(new TimedoutRequestException)
    }
  }
}
```

This example shows how you might provide authentication (via an authentication service) in order to convert a `Service[AuthHttpReq, HttpRep]` into

`Service[HttpReq, HttpRep]`.

```
class RequireAuthentication(authService: AuthService)
  extends Filter[HttpReq, HttpRep, AuthHttpReq, HttpRep]
{
  def apply(
    req: HttpReq,
    service: Service[AuthHttpReq, HttpRep]
  ) = {
    authService.auth(req) flatMap {
      case AuthResult(AuthResultCode.OK, Some(passport), _) =>
        service(AuthHttpReq(req, passport))
      case ar: AuthResult =>
        Future.exception(
          new RequestUnauthenticated(ar.resultCode))
    }
  }
}
```

Filters compose together with `andThen`. Providing a `Service` as an argument to `andThen` creates a (filtered) `Service` (types provided for illustration).

```
val authFilter: Filter[HttpReq, HttpRep, AuthHttpReq, HttpRep]
val timeoutfilter[Req, Rep]: Filter[Req, Rep, Req, Rep]
val serviceRequiringAuth: Service[AuthHttpReq, HttpRep]

val authenticateAndTimedOut: Filter[HttpReq, HttpRep, AuthHttpReq, HttpRep] =
  authFilter andThen timeoutFilter

val authenticatedTimedOutService: Service[HttpReq, HttpRep] =
  authenticateAndTimedOut andThen serviceRequiringAuth
```

Builders

Finally, builders put it all together. A `ClientBuilder` produces a `Service` instance given a set of parameters, and a `ServerBuilder` takes a `Service` instance and dispatches incoming requests on it. In order to determine the type of `Service`, we must provide a `Codec`. Codecs provide the underlying protocol implementation (eg. HTTP, thrift, memcached). Both builders have many parameters, and require a few.

Here's an example `ClientBuilder` invocation (types provided for illustration):

```
val client: Service[HttpRequest, HttpResponse] = ClientBuilder()
  .codec(Http)
  .hosts("host1.twitter.com:10000,host2.twitter.com:10001,host3.twitter.com:10003")
  .hostConnectionLimit(1)
  .tcpConnectTimeout(1.second)
  .retries(2)
  .reportTo(new OstrichStatsReceiver)
  .build()
```

This builds a client that load balances over the 3 given hosts, establishing at most 1 connection per host, and giving up only after 2 failures. Stats are reported to ostrich. The following builder options are required (and their presence statically enforced): `hosts` or `cluster`, `codec` and `hostConnectionLimit`.

```
val myService: Service[HttpRequest, HttpResponse] = // provided by the user
ServerBuilder()
  .codec(Http)
  .hostConnectionMaxLifeTime(5.minutes)
  .readTimeout(2.minutes)
  .name("myHttpServer")
  .bindTo(new InetSocketAddress(serverPort))
  .build(myService)
```

This will serve, on port `serverPort` an HTTP server which dispatches requests to `myService`. Each connection is allowed to stay alive for up to 5 minutes, and we require a request to be sent within 2 minutes. The required `ServerBuilder` options are: `name`, `bindTo` and `codec`.

1 distinct from `java.util.concurrent.Future`

2 this is equivalent to a monadic bind

We're going to build a very simple distributed search engine using Finagle. First, create a skeleton project using scala-bootstrapper

`scala-bootstrapper` will create a very simple finagle based scala service that exports an in-memory key-value store. We'll extend this to support searching of the values, and then extend it to support searching multiple in-memory stores over several processes.

```
$ mkdir searchbird; cd searchbird
$ scala-bootstrapper -p searchbird
$ find . -type f
./Capfile
./config/development.scala
./config/production.scala
./config/staging.scala
./config/test.scala
./Gemfile
./project/build/SearchbirdProject.scala
./project/build.properties
./project/plugins/Plugins.scala
./run
./src/main/scala/com/twitter/searchbird/config/SearchbirdServiceConfig.scala
./src/main/scala/com/twitter/searchbird/Main.scala
./src/main/scala/com/twitter/searchbird/SearchbirdServiceImpl.scala
./src/main/thrift/searchbird.thrift
./src/scripts/console
./src/scripts/searchbird.sh
./src/test/scala/com/twitter/searchbird/AbstractSpec.scala
./src/test/scala/com/twitter/searchbird/SearchbirdServiceSpec.scala
```

Exploring the default bootstrapper project

Let's first explore the default project `scala-bootstrapper` creates for us. This is meant as a template. You'll end up substituting most of it, but it serves as a convenient scaffold. It defines a simple (but complete) key-value store. Configuration, a thrift interface, stats export and logging are all included.

Since searchbird is a thrift service (like most of our services), its external interface is defined in the thrift IDL.

src/main/thrift/searchbird.thrift

```
service SearchbirdService {
  string get(1: string key) throws(1: SearchbirdException ex)
  void put(1: string key, 2: string value)
}
```

This is pretty straightforward: our service `SearchbirdService` exports 2 RPC methods, `get` and `put`. They comprise a simple interface to a key-value store.

Now let's run the default service, and explore it through this interface.

First build the project, and run the service (which is also the default "main" method that sbt will run).

```
$ sbt
…
> compile
> run -f config/development.scala
```

We've also written a simple client library that you can run from the `sbt console`

```
$ sbt console
+ exec java -Djava.net.preferIPv4Stack=true -Dhttp.connection.timeout=2 -Dhttp.connection-manager.timeout=2 -
Dhttp.socket.timeout=6 -Dcom.sun.management.jmxremote.port=0 -Dcom.sun.management.jmxremote.ssl=false -
Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote '-Xmx512m -XX:MaxPermSize=256m' -
XX:+AggressiveOpts -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+CMSParallelRemarkEnabled -XX:+CMSClassUnloadingEnabled -
XX:MaxPermSize=256m -XX:SurvivorRatio=128 -XX:MaxTenuringThreshold=0 -Xss200M -Xms512M -Xmx2G -ea -server -jar
/Users/stevej/bin/sbt-launch-0.7.4.jar console
[info] Standard project rules 0.12.7 loaded (2011-05-24).
[info] Building project searchbird 1.0.0-SNAPSHOT against Scala 2.8.1
[info]    using SearchbirdProject with sbt 0.7.4 and Scala 2.7.7
[info]
[info] == copy-resources ==
[info] == copy-resources ==
[info]
[info] == write-build-properties ==
[info] == write-build-properties ==
```

```
[info]
[info] == console ==
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.8.1.final (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_26).
Type in expressions to have them evaluated.
Type :help for more information.

scala> import com.twitter.searchbird.Client
import com.twitter.searchbird.Client

scala> val client = new Client()
client: com.twitter.searchbird.Client = com.twitter.searchbird.Client@24759265

scala> client.put("marius", "Marius Eriksen")
res0: ...

scala> client.put("stevej", "Steve Jenson")
res1: ...
```

The server also exports runtime statistics. These are convenient both for inspecting individual servers as well as aggregating into global service statistics (a machine-readable json interface is also provided).

```
$ curl localhost:9900/stats.txt
counters:
  Searchbird/connects: 2
  Searchbird/requests: 5
  Searchbird/success: 5
  jvm_gc_ConcurrentMarkSweep_cycles: 2
  jvm_gc_ConcurrentMarkSweep_msec: 102
  jvm_gc_ParNew_cycles: 9
  jvm_gc_ParNew_msec: 210
  jvm_gc_cycles: 11
  jvm_gc_msec: 312
gauges:
  Searchbird/connections: 0
```

```
  Searchbird/pending: 0
  jvm_fd_count: 147
  jvm_fd_limit: 10240
  jvm_heap_committed: 588251136
  jvm_heap_max: 3220570112
  jvm_heap_used: 39530208
  jvm_nonheap_committed: 81481728
  jvm_nonheap_max: 1124073472
  jvm_nonheap_used: 69312424
  jvm_num_cpus: 4
  jvm_post_gc_CMS_Old_Gen_used: 5970824
  jvm_post_gc_CMS_Perm_Gen_used: 46407832
  jvm_post_gc_Par_Eden_Space_used: 0
  jvm_post_gc_Par_Survivor_Space_used: 0
  jvm_post_gc_used: 52378656
  jvm_start_time: 1314124442749
  jvm_thread_count: 14
  jvm_thread_daemon_count: 8
  jvm_thread_peak_count: 14
  jvm_uptime: 404221
labels:
metrics:
  Searchbird/connection_duration: (average=25115, count=2, maximum=52068, minimum=142, p25=142, p50=142, p75=52068, p90=52068, p95=52068, p99=52068, p999=52068, p9999=52068, sum=50230)
  Searchbird/connection_received_bytes: (average=84, count=2, maximum=142, minimum=29, p25=29, p50=29, p75=142, p90=142, p95=142, p99=142, p999=142, p9999=142, sum=169)
  Searchbird/connection_requests: (average=2, count=2, maximum=4, minimum=1, p25=1, p50=1, p75=4, p90=4, p95=4, p99=4, p999=4, p9999=4, sum=5)
  Searchbird/connection_sent_bytes: (average=61, count=2, maximum=95, minimum=23, p25=23, p50=23, p75=95, p90=95, p95=95, p99=95, p999=95, p9999=95, sum=123)
  Searchbird/request_latency_ms: (average=20, count=5, maximum=95, minimum=1, p25=1, p50=2, p75=8, p90=95, p95=95, p99=95, p999=95, p9999=95, sum=103)
```

In addition to our own service statistics, we are also given some generic JVM stats that are often useful.

.../config/SearchbirdServiceConfig.scala

Configurations are simply any scala trait that has a method `apply: RuntimeEnvironment => T` for some `T` we want to create. In this sense, they are "factories". At runtime, a configuration file is evaluated as a script (by using the scala compiler as a library), and is expected to produce such a configuration object. `RuntimeEnvironment`s are objects queried for various runtime parameters (command line flags, JVM version, build timestamps, etc.).

The `SearchbirdServiceConfig` class specifies such a class. It specifies configuration parameters together with their defaults.

```
class SearchbirdServiceConfig extends ServerConfig[SearchbirdServiceServer] {
  var thriftPort: Int = 9999

  def apply(runtime: RuntimeEnvironment) = new SearchbirdServiceImpl(this)
}
```

In our case, we want to create a `SearchbirdServiceServer`. This is the server type generated by the thrift code generator [1].

.../Main.scala

The `main` function is very simple: it reads the configuration, creates a `SearchbirdServiceServer` and starts it. `RuntimeEnvironment.loadRuntimeConfig` performs the configuration evaluation and calls its `apply` method with itself as an argument [2].

```
object Main {
  def main(args: Array[String]) {
    val env = RuntimeEnvironment(this, args)
    val service = env.loadRuntimeConfig[SearchbirdServiceServer]
    service.start()
  }
}
```

.../SearchbirdServiceImpl.scala

This is the meat of the service: we extend the `SearchbirdServiceServer` with our custom implementation. Recall that `SearchbirdServiceServer` has been created for us by the thrift code generator. It generates a scala method per thrift method. In our example so far, the generated interface is:

```
trait SearchbirdService {
  def put(key: String, value: String): Future[Void]
  def get(key: String): Future[String]
}
```

`Future[Value]`s are returned instead of the values directly so that their computation may be deferred.

The default implementation provided by `scala-bootstrapper` is quite simple:

```
class SearchbirdServiceImpl(config: SearchbirdServiceConfig) extends SearchbirdServiceServer {
  val serverName = "Searchbird"
  val thriftPort = config.thriftPort

  val database = new mutable.HashMap[String, String]()

  def get(key: String) = {
    database.get(key) match {
      case None =>
        log.debug("get %s: miss", key)
        Future.exception(new SearchbirdException("No such key"))
      case Some(value) =>
        log.debug("get %s: hit", key)
        Future(value)
    }
  }

  def put(key: String, value: String) = {
    log.debug("put %s", key)
    database(key) = value
    Future.void
  }
}
```

It implements a simple thrift interface to a scala `HashMap`.

A simple search engine

Now we'll extend our example so far to create a simple search engine. We'll then extend it further to become a *distributed* seach engine consisting of multiple shards so that we can fit a corpus larger than what can fit in memory of a single machine. To keep things simple, we'll extend our current thrift service minimally in order to support a search operation.

src/main/thrift/searchbird.thrift

```
service SearchbirdService {
  string get(1: string key) throws(1: SearchbirdException ex)
  void put(1: string key, 2: string value)
  list<string> search(1: string query)
}
```

We’ve added a `search` method that searches the current hashtable, returning the list of keys whose values match the query. The implementation is also straightforward:

.../SearchbirdServiceImpl.scala

We first add another `HashMap` to hold the reverse index, giving us maps in both the forward (key to document) and reverse (token to set of documents) directions.

```
  val forward = new mutable.HashMap[String, String]
    with mutable.SynchronizedMap[String, String]
  val reverse = new mutable.HashMap[String, Set[String]]
    with mutable.SynchronizedMap[String, Set[String]]
```

The `get` method remains the same (it only performs forward lookups), but we need to populate the reverse index on `put`.

```
  def put(key: String, value: String) = {
    log.debug("put %s", key)

    forward(key) = value

    // admit only one updater.
    synchronized {
      (Set() ++ value.split(" ")) foreach { token =>
        val current = reverse.get(token) getOrElse Set()
        reverse(token) = current + key
      }
    }

    Future.void
```

```
}
```

This is simple: we tokenize by splitting the value, and then store a reference to the key for each token. This will allow us to perform lookups by the tokens themselves.

Note that since we do a retrieve-modify-update, we need to synchronize the update even though the underlying `HashMap` is synchronized. Also, this implementation has a bug: when overwriting keys, we're not collecting references to the old value in the reverse index. Fixing this is an excercise for the reader.

Now to the meat of the search engine: the new `search` method. It should tokenize its query, look up all of the matching documents and then intersect these lists. This will yield the list of documents that contain all of the tokens in the query. This is also pretty straightforward to express in Scala:

```
def search(query: String) = Future.value {
  val tokens = query.split(" ")
  val hits = tokens map { token => reverse.getOrElse(token, Set()) }
  val intersected = hits reduceLeftOption { _ & _ } getOrElse Set()
  intersected.toList
}
```

A few things are worth calling out in this short piece of code. When constructing the hit list, `getOrElse` will supply the value in the 2nd parameter if the key (`token`) is not found. We perform the actual intersection using a left-reduce. The particular flavor, `reduceLeftOption` will not attempt to perform the reduce if `hits` is empty, returning instead a `None`. This allows us to supply a default value instead of experiencing an exception. In fact, this is equivalent to:

```
def search(query: String) = Future.value {
  val tokens = query.split(" ")
  val hits = tokens map { token => reverse.getOrElse(token, Set()) }
  if (hits.isEmpty)
    Nil
  else
    hits reduceLeft { _ & _ } toList
}
```

Which to use is mostly a matter of taste, though functional style often eschews conditionals for sensible defaults.

We can now experiment with our store in using the console.

```
$ src/scripts/console
Hint: the client is in the variable `$client`
No servers specified, using 127.0.0.1:9999
>
```

Paste the lecture descriptions in.

```
client.put("basics", " values functions classes methods inheritance try catch finally expression oriented")
client.put("basics", " case classes objects packages apply update functions are objects (uniform access principle) pattern")
client.put("collections", " lists maps functional combinators (map foreach filter zip")
client.put("pattern", " more functions! partialfunctions more pattern")
client.put("type", " basic types and type polymorphism type inference variance bounds")
client.put("advanced", " advanced types view bounds higher kinded types recursive types structural")
client.put("simple", " all about sbt the standard scala build")
client.put("more", " tour of the scala collections")
client.put("testing", " write tests with specs a bdd testing framework for")
client.put("concurrency", " runnable callable threads futures twitter")
client.put("java", " java interop using scala from")
client.put("searchbird", " building a distributed search engine using")
```

We can now perform some searches.

```
> client.search("functions")
res0: Seq("basics")
> client.search("java")
res1: Seq("java")
> client.search("java scala")
res2: Seq("java")
> client.search("functional")
res3: Seq("collections")
> client.search("sbt")
res4: Seq("simple")
> client.search("types")
res5: Seq("type", "advanced")
```

Distributing our service

Our simple in-memory search engine won't be able to search corpuses larger than the size of memory on a single machine. We'll not venture to remedy this by distributing nodes with a simple sharding scheme.

Abstracting

To aid our work, we'll first introduce another abstraction: an `Index` in order to decouple the index implementation from the SearchBirdService. This is a straightforward refactor.

.../Index.scala

```
package com.twitter.searchbird

import scala.collection.mutable
import com.twitter.util._
import com.twitter.logging.Logger

trait Index {
  def get(key: String): Future[String]
  def put(key: String, value: String): Future[Unit]
  def search(key: String): Future[List[String]]
}

class ResidentIndex extends Index {
  val log = Logger.get(getClass)

  val forward = new mutable.HashMap[String, String]
    with mutable.SynchronizedMap[String, String]
  val reverse = new mutable.HashMap[String, Set[String]]
    with mutable.SynchronizedMap[String, Set[String]]

  def get(key: String) = {
    forward.get(key) match {
      case None =>
        log.debug("get %s: miss", key)
        Future.exception(new SearchbirdException("No such key"))
      case Some(value) =>
        log.debug("get %s: hit", key)
```

```
        Future(value)
      }
    }

    def put(key: String, value: String) = {
      log.debug("put %s", key)

      forward(key) = value

      // admit only one updater.
      synchronized {
        (Set() ++ value.split(" ")) foreach { token =>
          val current = reverse.get(token) getOrElse Set()
          reverse(token) = current + key
        }
      }

      Future.Unit
    }

    def search(query: String) = Future.value {
      val tokens = query.split(" ")
      val hits = tokens map { token => reverse.getOrElse(token, Set()) }
      val intersected = hits reduceLeftOption { _ & _ } getOrElse Set()
      intersected.toList
    }
  }
```

We now convert our thrift service to a simple dispatch mechanism: it provides a thrift interface to any `Index` instance. The power of this will soon be apparent.

.../SearchbirdServiceImpl.scala

```
class SearchbirdServiceImpl(config: SearchbirdServiceConfig, index: Index) extends SearchbirdServiceServer {
  val serverName = "Searchbird"
  val thriftPort = config.thriftPort
```

```
  def get(key: String) = index.get(key)
  def put(key: String, value: String) =
    index.put(key, value) flatMap { _ => Future.void }
  def search(query: String) = index.search(query)
}
```

Finally we adjust our configuration class to match the new convention.

…/config/SearchbirdServiceConfig.scala

```
class SearchbirdServiceConfig extends ServerConfig[SearchbirdServiceServer] {
  var thriftPort: Int = 9999
  def apply(runtime: RuntimeEnvironment) = new SearchbirdServiceImpl(this, new ResidentIndex)
}
```

We'll set up our simple distributed system so that there is one distinguished node that coordinates queries to its child nodes. In order to achieve this, we'll need two new `Index` types. One represents a remote index, the other is a composite index over several other `Index` instances. This way we can construct the distributed index by instantiating a composite index of the remote indices.

First we define a `CompositeIndex`.

```
class CompositeIndex(indices: Seq[Index]) extends Index {
  require(!indices.isEmpty)

  def get(key: String) = {
    val queries = indices.map { idx =>
      idx.get(key) map { r => Some(r) } handle { case e => None }
    }

    Future.collect(queries) flatMap { results =>
      results.find { _.isDefined } map { _.get } match {
        case Some(v) => Future.value(v)
        case None => Future.exception(new SearchbirdException("No such key"))
      }
    }
  }
```

```
  def put(key: String, value: String) =
    Future.exception(new SearchbirdException("put() not supported by CompositeIndex"))

  def search(query: String) = {
    val queries = indices.map { _.search(query) rescue { case _=> Future.value(Nil) } }
    Future.collect(queries) map { results => (Set() ++ results.flatten) toList }
  }
}
```

The composite index works over a set of underlying `Index` instances. Note that it doesn't care how these are actually implemented. This type of composition allows for great flexibility in constructing various querying schemes. We don't define a sharding scheme, and so the composite index doesn't support `put` operations. These are instead issued directly to the child nodes. `get` is implemented by querying all of our child nodes and picking the first succesful result. If there are none, we throw an exception. Note that since the absence of a value is communicated by throwing an exception, we `handle` this on the `Future`, converting any exception into a `None` value. In a real system, we'd probably have proper error codes for missing values rather than using exceptions. Exceptions are convenient and expedient for prototyping, but compose poorly. In order to distinguish between a real exception and a missing value, I have to examine the exception itself. Rather it is better style to embed this distinction directly in the type of the returned value.

`search` works in a similar way. Instead of picking the first result, we combine them, ensuring their uniqueness by using a `Set` construction.

`RemoteIndex` provides an `Index` interface over a number of hosts.

```
class RemoteIndex(hosts: String) extends Index {
  val transport = ClientBuilder()
    .name("remoteIndex")
    .hosts(hosts)
    .codec(ThriftClientFramedCodec())
    .hostConnectionLimit(1)
    .timeout(500.milliseconds)
    .build()
  val client = new SearchbirdServiceClientAdapter(
      new thrift.SearchbirdService.ServiceToClient(
        transport, new TBinaryProtocol.Factory))

  def get(key: String) = client.get(key)
  def put(key: String, value: String) = client.put(key, value) map { _ => () }
  def search(query: String) = client.search(query) map { _.toList }
}
```

This constructs a finagle thrift client with some sensible defaults, and just proxies the calls, adjusting the types slightly.

Putting it all together

We now have all the pieces we need. We'll need to adjust the configuration in order to be able to invoke a given node as either a distinguished node or a data shard node. In order to do so, we'll enumerate the shards in our system by creating a new config item for it. We'll then use command line arguments (recall that the `Config` has access to these) to start the server up in either mode.

```
class SearchbirdServiceConfig extends ServerConfig[SearchbirdServiceServer] {
  var thriftPort: Int = 9999
  var shards: Seq[String] = Seq()

  def apply(runtime: RuntimeEnvironment) = {
    val index = runtime.arguments.get("shard") match {
      case Some(arg) =>
        val which = arg.toInt
        if (which >= shards.size || which < 0)
          throw new Exception("invalid shard number %d".format(which))

        // override with the shard port
        val Array(_, port) = shards(which).split(":")
        thriftPort = port.toInt

        new ResidentIndex

      case None =>
        require(!shards.isEmpty)
        val remotes = shards map { new RemoteIndex(_) }
        new CompositeIndex(remotes)
    }

    new SearchbirdServiceImpl(this, index)
  }
}
```

And finally we'll adjust the configuration itself:

```
new SearchbirdServiceConfig {
  shards = Seq(
    "localhost:9000",
    "localhost:9001",
    "localhost:9002"
  )
  …
```

Now if we invoke our server without any arguments, it starts a distinguished node that speaks to all of the given shards. If we specify a shard argument, it starts a server on the port belonging to that shard index.

Let’s try it! We’ll launch 2 shards and 1 distinguished node.

```
$ sbt 'run -f config/development.scala -D shard=0'&
$ sbt 'run -f config/development.scala -D shard=1'&
$ sbt 'run -f config/development.scala'&
```

Then interact with it through the console. First let’s populate some data in the two shard nodes.

```
$ src/scripts/console localhost:9000
> $client.put("fromShardA", "a value from SHARD_A")
> $client.put("hello", "world")
^D
$ src/scripts/console localhost:9001
> $client.put("fromShardB", "a value from SHARD_B")
> $client.put("hello", "world again")
```

And now let’s query our database from the distinguished node.

```
$ src/scripts/console
No servers specified, using 127.0.0.1:9999
> $client.get("hello")
"world"
> $client.get("fromShardC")
```

```
Searchbird::SearchbirdService::Client::ApplicationException: Searchbird::SearchbirdService::Client::ApplicationException
…
> $client.get("fromShardA")
"a value from SHARD_A"
> $client.search("hello")
[]
> $client.search("world")
["hello"]
> $client.search("value")
["fromShardA", "fromShardB"]
```

1 In `target/gen-scala/com/twitter/searchbird/SearchbirdService.scala`

2 See Ostrich's README:https://github.com/twitter/ostrich/blob/master/README.md for more information.