Basics continued

This lesson covers:

apply methods

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

```
scala> class Foo {}
defined class Foo

scala> object FooMaker {
     |   def apply() = new Foo
     | }
defined module FooMaker

scala> val newFoo = FooMaker()
newFoo: Foo = Foo@5b83f762
```

or

```
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. Specifically, 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.

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

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

There is Function0 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 syntactic 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 every time 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 ().

```
scala> 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.

See Also Effective Scala has opinions about when to use pattern matching and pattern matching formatting. A Tour of Scala describes Pattern Matching

Matching on type

You can use `match` to handle values of different types differently.

```
def bigger(o: Any): Any = {
  o match {
    case i: Int if i < 0 => i - 1
    case i: Int => i + 1
    case d: Double if d < 0.0 => d - 0.1
    case d: Double => d + 0.1
    case text: String => text + "s"
  }
}
```

Matching on class members

Remember our calculator from earlier.

Let's classify them according to type.

Here's the painful way first.

```
def calcType(calc: Calculator) = calc match {
  case _ if calc.brand == "hp" && calc.model == "20B" => "financial"
  case _ if calc.brand == "hp" && calc.model == "48G" => "scientific"
  case _ if 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 unavailable. should have kept your trusty
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 unavailable. should have kept your trusty 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.