Updated: 2021-03-31

6.7
Advanced Concepts

© 2015 Robin Hillyard

Northeastern University

Type Classes

- We looked at these briefly before. Type classes are the way in which Scala programs (Haskell has these too) allow the imposition of behavior on another class
  - In pure O-O, we always have to *extend* traits to add behavior.
  - But that's often impossible, inconvenient, or just plain wrong.

Type classes: review

- Let's say you have in mind a trait but there's nothing appropriate for it to extend:

```
trait Parseable[T] {
  def parse(s: String): Try[T]
}
object Parseable {
  trait ParseableInt extends Parseable[Int] {
    def parse(s: String): Try[Int] = Try(s.toInt)
  }
  implicit object ParseableInt extends ParseableInt
}
object TestParseable {
  def parse[T : Parseable](s: String): Try[T] = implicitly[Parseable[T]].parse(s)
}
```

This form is called a "context bound". But we can also write it as follows:

```
def parse[T](s: String)(implicit ev: Parseable[T]): Try[T] = ev.parse(s)
```

- What we are doing here is adding the behavior of *Parseable* to type *T* without requiring *T* to extend anything.

- Note that you cannot add a context bound to a trait. Why not?

A case in point

- Renderable:
  - In my *LaScala* library, I created a trait *Renderable* which affords a better way to render objects as Strings (better than *toString*).
    - In particular, any type that extends *Renderable* supports the *render* method:

```
def render(indent: Int = 0)(implicit tab: Int => Prefix): String
```

    - Additionally, there are implicit *Renderables* for containers such as:

```
implicit def renderableTraversable(xs: Traversable[_]): Renderable =
RenderableTraversable(xs, max = Some(MAX_ELEMENTS))
```

    - where:

```
case class RenderableTraversable(xs: Traversable[_], bookends: String = "(,)", linear:
Boolean = false, max: Option[Int] = None) extends Renderable {
  def render(indent: Int)(implicit tab: (Int) => Prefix): String = {
    val (p, q, r) =
      if (linear || xs.size <= 1)
        ("" + bookends.head, "" + bookends.tail.head, "" + bookends.last)
      else
        (bookends.head + nl(indent + 1), bookends.tail.head + nl(indent + 1), nl(indent) +
bookends.last)
    Renderable.elemsToString(xs, indent, p, q, r, max)
  }
}
```

This works great but…

- …there is a bit of a problem:
  - it only works for types which extend *Renderable or* which have been provided for using an implicit;
  - in particular, if you want to write a library class based on a parametric type which is *Renderable*, you have to specify that as a type constraint;
  - instead, you can simply apply a “context bound” to the parametric type requiring “evidence” of a *Renderable*—this is so much more convenient.

Type class *Renderable*

- New Renderable:
  - I am in the process of changing *LaScala* to use a type class *Renderable* instead of a polymorphic trait.

```
trait Renderable[A] {
  /**
  * Method to render this object in a human-legible manner.
  *
  * @param indent the number of "tabs" before output should start (when writing on a new line).
  * @return a String that, if it has embedded newlines, will follow each newline with (possibly empty) white space,
  *        which is then followed by some human-legible rendering of *this*.
  */
 def render(a: A)(indent: Int = 0): String
  /**
  * Method to translate the tab number (i.e. indent) to a String of white space.
  * Typically (and by default) this will be uniform. But you're free to set up a series of tabs
  * like on an old typewriter where the spacing is non-uniform.
  *
  * @return a String of white spade.
  */
 def tab(x: Int): Prefix = Prefix(" " * x)
}
```

  - The most obvious differences are that *Renderable* now takes a parametric type *A* and the *render* method itself takes a value of type *A*.

Specification

- Here's part of *RenderableSpec*:

```
class RenderableSpec extends FlatSpec with Matchers with Inside {
    behavior of "Renderable"
  import RenderableInstances._
    it should "render String the hard way" in {
      import Renderable._
      render("Hello")() shouldBe "Hello"
  }
    it should "render String" in {
      import RenderableSyntax._
      "Hello".render shouldBe "Hello"
  }
    it should "render Int" in {
      import RenderableSyntax._
      1.render shouldBe "1"
  }
}
```

- It is slightly annoying to have to specify those imports but there may be a better way.

Specification (part 2)

- Here's another part of *RenderableSpec*:

```
class RenderableSpec extends FlatSpec with Matchers with Inside {
  behavior of "MockContainer"
  it should "render correctly" in {
    val target = MockContainer(Seq(1,2,3))
    target.toString shouldBe "(\n 1,\n 2,\n 3\n)"
 }
}

case class MockContainer[A: Renderable](as: Seq[A]) {
  import RenderableSyntax._
  override def toString: String = as.render
}
```

Using Type classes

- Best Practices:
  - Define the minimum number of methods (often just one!) in your type class trait;
  - Use a type class to add behavior to a parametric type in some other class—don't use one to add behavior to the other class—be strict about that.
  - If a parametric type needs more than one behavior imposed on it, add additional context bounds:

```
case class MockNumericContainer[A: Renderable: Numeric](as: Seq[A]) {
  import RenderableSyntax._
  override def toString: String = as.render
  def total: A = as.sum
}
```

Type-class Libraries

- Cats
  - https://typelevel.org/cats
  - Cats uses type-classes extensively. There are type-classes for just about everything.

Logging functional style

- Logging styles:
  - Libraries such as log4j are ideally suited to a statement-oriented language such as Java…
  - …but Scala is functional, not statement-oriented.
  - Why not use a functional style of logging (as in *LaScala*)?

```
trait Spy
object Spy {
 def apply(x: Unit): Spy = new Spy() {}
  lazy private val configuration = ConfigFactory.load()
 var spying: Boolean = configuration.getBoolean("spying")
  def spy[X](message: => String, x: => X, b: Boolean = true)(implicit spyFunc: String => Spy, isEnabledFunc: Spy => Boolean): X = {
    val xy = Try(x) // evaluate x inside Try
 if (b && spying && isEnabledFunc(mySpy)) doSpy(message, xy, b, spyFunc) // if spying is turned on, log an appropriate message
 xy.get // return the X value or throw the appropriate exception
 }
 def log(w: => String, b: Boolean = true)(implicit spyFunc: String => Spy, isEnabledFunc: Spy => Boolean) {spy(w, (), b); ()}
 def noSpy[X](x: => X): X = {
    val safe = spying
 spying = false
 val r = x
    spying = safe
    r
  }
 private val prefix = "spy: "
val brackets: String = "{}"
implicit val defaultLogger: Logger = getLogger(getClass)
implicit def spyFunc(s: String)(implicit logger: Logger): Spy = if (logger != null) Spy(logger.debug(prefix + s)) else Spy()
implicit def isEnabledFunc(x: Spy)(implicit logger: Logger): Boolean = logger != null && logger.isDebugEnabled
def getLogger(clazz: Class[_]): Logger = LoggerFactory.getLogger(clazz)
def getPrintlnSpyFunc(ps: PrintStream = System.out): String => Spy = { s => Spy(ps.println(prefix + s)) }
 private def doSpy[X](message: String, xy: => Try[X], b: Boolean, spyFunc: (String) => Spy) = {
    …
  }
  private def formatMessage[X](x: X, b: Boolean): String = x match {
    case () => "()"
 …
 // NOTE: If the value to be spied on is Future(_) then we invoke spy on the underlying value when it is completed
 case f: Future[_] =>
      import scala.concurrent.ExecutionContext.Implicits.global
  f.onComplete(spy("Future", _, b))
      "to be provided in the future"
 // NOTE: If the value to be spied on is a common-or-garden object, then we simply form the appropriate string using the toString method
 case _ => if (x != null) x.toString else "<<null>>"
 }
  private val mySpy = apply(())
}
```

Using *Spy*

- Here's the Spec file:

```
class SpySpec extends FlatSpec with Matchers {

behavior of "Spy.spy"
  it should "work with implicit (logger) (with default logger) spy func" in {
    import Spy._
    Spy.spying = true
  (for (i <- 1 to 2) yield Spy.spy("i", i)) shouldBe List(1, 2)
    // you should see log messages written to console (assuming your logging level, i.e. logback-test.xml, is set to DEBUG)
 }
}
```

- Note what it would like look without spying:

```
class SpySpec extends FlatSpec with Matchers {

behavior of "for comprehension"
  it should "work" in {
 (for (i <- 1 to 2) yield i) shouldBe List(1, 2)
 }
}
```

Another functional logger

```
object Flog {
    implicit class Flogger(message: => String)(implicit logFunc: LogFunction = Flog.loggingFunction) {
  def !![X: Loggable](x: => X): X = Flog.logLoggable(logFunc, message)(x)
  def !|[X](x: => X): X = Flog.logX(logFunc, message)(x)
  def |![X](x: => X): X = x
  }
 var enabled =  true
 implicit var loggingFunction: LogFunction = getLogger[Flogger]
   def getLogger[T: ClassTag]: LogFunction = LogFunction(LoggerFactory.getLogger(implicitly[ClassTag[T]].runtimeClass).debug)
 def logLoggable[X: Loggable](logFunc: LogFunction, prefix: => String)(x: => X): X = {
     lazy val xx: X = x
     if (enabled) logFunc(s"log: $prefix: ${implicitly[Loggable[X]].toLog(xx)}")
     xx
  }
 def logX[X](logFunc: LogFunction, prefix: => String)(x: => X): X = {
     lazy val xx: X = x
     if (enabled) logFunc(s"log: $prefix: $xx")
     xx
  }
}
```