Scala Functional Abstraction

Function values

Partially Applied Functions

A `def` is not a function value.

```scala
def dubbel(x: String): String = s"two ${x}s"

// Won't compile because dubbel is not a function value
val wontCompile = dubbel
```

To turn the dubbel method in to a Function value, partially apply it

```scala
val dubbelFun = dubbel _
```

Don't forget the space between the name of the function and the underscore.

The partial function application above is equivalent to:

```scala
val dubbelFun = (x: String) => dubbel(x)
```

Parial Function Short Forms

You can leave off the underscore if target type is a function. These three are equivalent

```
List("Honey", "Boo", "Boo").foreach(x => print(x))
List("Honey", "Boo", "Boo").foreach(print _)
List("Honey", "Boo", "Boo").foreach(print)
```

- The third example above works because `foreach` takes a function value, so print is lifted to a function (another term for partial function application)

Note that this form is not technically a partially applied function, it's just a short-form of a function literal using placeholder syntax:

```
List("Honey", "Boo", "Boo").foreach(print(_))
```

Closures

```
def makeDecorator(
    leftBrace: String,
    rightBrace: String): String => String =
  (middle: String) => leftBrace + middle + rightBrace

val squareBracketer = makeDecorator("[", "]")
```

In the function literal

```
(middle: String) => leftBrace + middle + rightBrace
```

- `middle` is bound variable because it's in the parameter list
- `leftBrace` and `rightBrace` are free variables

A function literal with only bound variables is called a closed term.

A function literal with free variables is called an open term becuase values for the free variables must be captures from an enclosing environment, thereby *closing* the term.

Schönfinkeling, a.k.a., Currying

Scala syntax for curried functions: multiple param lists

```scala
def curry(chicken: String)(howard: String): String =
  s"Love that $chicken from $howard!"
```

Above is equivalent to:

```scala
def explicitCurry(chicken: String): String => String =
 (howard: String) => s"Love that $chicken from $howard!"
```

You can partially apply second parameter list to get another function

```scala
val eleganceFrom = curry("elegence")_
eleganceFrom("provability")
```

Control Abstraction

A common idiom in programming with resources is the loan pattern:

1. open a resource,
2. operate on the resource, and
3. close the resource.

You can capture this pattern in a function:

```scala
def withPrintWriter(file: File, op: PrintWriter => Unit) = {
  val writer = new PrintWriter(file)
  try {
    op(writer)
  } finally {
    writer.close()
  }
}

withPrintWriter(
  new File("date.txt"),
  writer => writer.println(new java.util.Date)
)
```

Control Abstraction with Multiple Parameter Lists

In the previous example we had to use standard function call syntax. If we use multiple parameter lists:

```scala
def withPrintWriter(file: File)(op: PrintWriter => Unit) = {
  val writer = new PrintWriter(file)
  try {
    op(writer)
  } finally {
    writer.close()
  }
}
```

we can use a nicer syntax with curly braces for the second argument list:

```scala
withPrintWriter(new File("date.txt")) { writer =>
  writer.println(new java.util.Date)
}
```

Thunks

In the previous example we needed a parameter in the second parameter list. But sometimes you don't:

```scala
val assertionsEnabled = true

def myAssert(predicate: () => Boolean) =
   if (assertionsEnabled && !predicate())
    throw new AssertionError
```

Using this function is awkward:

```scala
myAssert(() => 5 > 3)
```

The function passed to `myAssert` is called a "thunk", which is a piece of code that is not evaluated until it is needed – in particular when `predicate` is called in the body of `myAssert`. Scala provides another way to achieve the same effect ...

By-Name Parameters

Specify a *by-name* parameter by putting a => between the colon and the type:

```scala
def byNameAssert(predicate: => Boolean) =
  if (assertionsEnabled && !predicate)
    throw new AssertionError
```

When byNameAssert is called predicate is not evaluated until it is used. Contrast this with a *by-value* parameter:

```scala
def boolAssert(predicate: Boolean) =
  if (assertionsEnabled && !predicate)
    throw new AssertionError
```

Abstraction with Higher-order Functions

▶