Objects Everywhere

Principles of Functional Programming

Pure Object Orientation

A pure object-oriented language is one in which every value is an object.

If the language is based on classes, this means that the type of each value is a class.

Is Scala a pure object-oriented language?

At first glance, there seem to be some exceptions: primitive types, functions.

But, let's look closer:

Standard Classes

Conceptually, types such as `Int` or `Boolean` do not receive special treatment in Scala. They are like the other classes, defined in the package `scala`.

For reasons of efficiency, the Scala compiler represents the values of type `scala.Int` by 32-bit integers, and the values of type `scala.Boolean` by Java's Booleans, etc.

Pure Booleans

The `Boolean` type maps to the JVM's primitive type `boolean`.

But one *could* define it as a class from first principles:

```
package idealized.scala
abstract class Boolean extends AnyVal:
  def ifThenElse[T](t: => T, e: => T): T

  def && (x: => Boolean): Boolean = ifThenElse(x, false)
  def || (x: => Boolean): Boolean = ifThenElse(true, x)
  def unary_!: Boolean            = ifThenElse(false, true)

  def == (x: Boolean): Boolean    = ifThenElse(x, x.unary_!)
  def != (x: Boolean): Boolean    = ifThenElse(x.unary_!, x)
  ...
end Boolean
```

Boolean Constants

Here are constants `true` and `false` that go with `Boolean` in `idealized.scala`:

```scala
package idealized.scala

object true extends Boolean:
  def ifThenElse[T](t: => T, e: => T) = t

object false extends Boolean:
  def ifThenElse[T](t: => T, e: => T) = e
```

Exercise

Provide an implementation of an implication operator `==>` for class `idealized.scala.Boolean`.

Exercise

Provide an implementation of an implication operator ==> for class `idealized.scala.Boolean`.

```scala
extension (x: Boolean):
  def ==> (y: Boolean) = x.ifThenElse(y, true)
```

That is, if `x` is true, `y` has to be true also, whereas if `x` is false, `y` can be arbitrary.

The class Int

Here is a partial specification of the class `scala.Int`.

```
class Int:
  def + (that: Double): Double
  def + (that: Float): Float
  def + (that: Long): Long
  def + (that: Int): Int         // same for -, *, /, %

  def << (cnt: Int): Int         // same for >>, >>>  */

  def & (that: Long): Long
  def & (that: Int): Int         // same for |, ^ */
```

The class Int (2)

```
    def == (that: Double): Boolean
    def == (that: Float): Boolean
    def == (that: Long): Boolean   // same for !=, <, >, <=, >=
    ...
  end Int
```

Can it be represented as a class from first principles (i.e. not using primitive `int`s?

Exercise

Provide an implementation of the abstract class Nat that represents non-negative integers.

```
abstract class Nat:
  def isZero: Boolean
  def predecessor: Nat
  def successor: Nat
  def + (that: Nat): Nat
  def - (that: Nat): Nat
end Nat
```

Exercise (2)

Do not use standard numerical classes in this implementation.

Rather, implement a sub-object and a sub-class:

```
object Zero extends Nat:
  ...
class Succ(n: Nat) extends Nat:
  ...
```

One for the number zero, the other for strictly positive numbers.

(this one is a bit more involved than previous quizzes).