Day 18: The decorator design pattern

Say you want to add functionality / behavior to a class, but you **don't** want to do it via inheritence (ie, kinda muddying up your class separation). The decorator pattern here serves well.

High-level, there are two ways to mix in functionality:

1. *Java-esque* — create a decorator class that takes some other class object in its constructor, and then uses `super` calls to enhance the original fubcntionality without directly extending the original class. This is a `has-a` model rather than `is-a`, and feels akin to functional composition, but with mix-in ordering determined by inheritence.
2. *Scala-esque* — this is simply stackable traits; use the `abstract override` syntax to mix in via `super` calls (preserves interfaces), and self-types to mix in with functional composition (maybe requires more careful method naming / architecture).

Java-esque

Given an interface and a basic implementation, we can use a decorator class as an `abstract class` to start implementing different branches of functionality:

```
// basic interface
trait InputReader {
  def readLines(): Stream[String]
}

// basic implementation
class AdvancedInputReader(reader: BufferedReader) extends InputReader {
  override def readLines(): Stream[String] = reader.lines().iterator().asScala.toStream
}

// decorator class - notice that it `has-a` reader
abstract class InputReaderDecorator(inputReader: InputReader) extends InputReader {
  override def readLines(): Stream[String] = inputReader.readLines()
}

// two examples from the book here
class Base64EncoderInputReader(inputReader: InputReader) extends
InputReaderDecorator(inputReader) {
  override def readLines(): Stream[String] = super.readLines().map {
    case line => Base64.getEncoder.encodeToString(line.getBytes(Charset.forName("UTF-8")))
  }
}

```

```
23 class CapitalizedInputReader(inputReader: InputReader) extends
   InputReaderDecorator(inputReader) {
24   override def readLines(): Stream[String] = super.readLines().map(_.toUpperCase)
25 }
26
27 // usage can be composed, although it's goofy
28 // CompressingInputReader not shown here
29 val reader = new CompressingInputReader(
30       new Base64EncoderInputReader(
31         new CapitalizedInputReader(
32           new AdvancedInputReader(stream)
33         )
34       )
35     )
```

Things to note:

1. This still outlines a `has-a` relationship — the original `InputReader` implementation has technically not been touched.
2. Effects are composed with `super` calls, which are resolved through the extensive wrapping seen on line 29.
3. Notice there's no `super` call to any abstract method, like we'll see below — the decorator abstract class assumes it'll be constructed with a concrete `InputReader` with a concrete `readLines`, thus negating any calls to abstract methods.
4. There are a million ways to achieve this, but this decomposes your functions. It does make types difficult, however, and is also clunky and overly distinct at the call site (what if you create a class that takes only a class object that is base64 and capitalization-friendly?

Scala-esque

Scala enables this pattern via `stackable traits` and `abstract override` syntax. Now instead of some abstract decorator class that takes and wraps the base class, we extend the base interface and attach the behaviors to some object at instantion via `with` keywords (subject, of course, to linearization!):

```
1 // one example behavior
2 trait CapitalizedInputReaderTrait extends InputReader {
3   abstract override def readLines(): Stream[String] = super.readLines().map(_.toUpperCase)
4 }
```

Notes are:

1. We can't do this without `abstract override`. Self-types will only help in calling another method, which requires planning.
2. This pattern can add complexity to your object's type, making it more cumbersome and verbose to pass around.
3. Like the previous pattern this makes your code much more one-off-y, which can lead to some code sprawl if it's not managed well.