Day 17: The adapter design pattern

This pattern opens up the section on `Structural design patterns`! Structural design patterns are most concerned with stitching together components in a clear, extensible way.

Say we want to stitch **existing** incompatible components together. The adapter pattern shines here.

Take an already-existing `Logging` class like so:

```
class Logger {
  def log(message: String, severity: String): Unit = {
    System.out.println(s"${severity.toUpperCase}: $message")
  }
}
```

Now say we also have some functionality we want to inject and *adapt* to this existing class, such as:

```
trait Log {
  def info(message: String)
  def debug(message: String)
  def warning(message: String)
  def error(message: String)
}
```

We can create an `adapter class` out of these two like so:

```
class AppLogger extends Logger with Log {
  override def info(message: String): Unit = log(message, "info")

  override def warning(message: String): Unit = log(message, "warning")

  override def error(message: String): Unit = log(message, "error")

  override def debug(message: String): Unit = log(message, "debug")
}
```

Two things to note:

1. This seems a more typical Java-like way to add functionality via interfaces.
2. Note that this example wouldn't work if the `Logger` class was final. One way to handle this is to slightly tweak your `is-a` / `has-a` relationship as such:

```
class FinalAppLogger extends Log {
  private val logger = new FinalLogger

  override def info(message: String): Unit = logger.log(message, "info")

  override def warning(message: String): Unit = logger.log(message, "warning")

  override def error(message: String): Unit = logger.log(message, "error")

  override def debug(message: String): Unit = logger.log(message, "debug")
}
```

A small but subtle change. Scala also offers native adapter patterns via `implicit classes`:

```
  implicit class FinalAppLoggerImplicit(logger: FinalLogger) extends Log {

    override def info(message: String): Unit = logger.log(message, "info")

    override def warning(message: String): Unit = logger.log(message, "warning")

    override def error(message: String): Unit = logger.log(message, "error")

    override def debug(message: String): Unit = logger.log(message, "debug")
  }
```

Now we can pretend we're just using a `FinalLogger` object! Implicit classes are nice, and ultimately just syntactic sugar for implicit coversions (via `defs`).