Day 7: Aspect-Oriented Programming

The term `aspect-oriented programming` (AOP) refers to good practices for writing cross-cutting code. Examples of such code might include:

- Logging
- Timing / benchmarking
- Serializing

We'll focus on an example of timing different implementations of the same functionality here, per the chapter.

Imagine you're reading and parsing JSON, and you want to compare two implementations. You might try implementing timing without AOP as such (note this requires json4s):

```
case class Person(firstName: String, lastName: String, age: Int)

trait DataReader {
  def readData(): List[Person]
  def readDataInefficiently(): List[Person]
}

class DataReaderImpl extends DataReader {
  private def readUntimed(): List[Person] = List(Person("a","b",27))

  override def readData(): List[Person] = {
    val startMillis = System.currentTimeMillis()
    val result = readUntimed()
    val time = System.currentTimeMillis() - startMillis
    println(s"readData took ${time} milliseconds.")
    result
  }

  override def readDataInefficiently(): List[Person] = {
    val startMillis = System.currentTimeMillis()
    (1 to 10000).foreach {
      case num =>
        readUntimed()
    }
    val result = readUntimed()
    val time = System.currentTimeMillis() - startMillis
    System.err.println(s"readDataInefficiently took ${time} milliseconds.")
    result
  }
}
```

```
31
32 val dataReader = new DataReaderImpl
```

Of course the main problem here is that our timing code is intermixed with our reading code, and totally clutters and duplicates. With AOP, let's try separating out our concerns better - leave the core reading alone, and mix in the timing functionality on top of the reads!

```
1  // first revert DataReaderImpl to no-logging state
2  class DataReaderImpl extends DataReader {
3    private def readUntimed(): List[Person] = List(Person("a","b",27))
4
5    override def readData(): List[Person] = {
6      readUntimed()
7    }
8
9    override def readDataInefficiently(): List[Person] = {
10     (1 to 10000).foreach {
11       case num =>
12         readUntimed()
13     }
14     readUntimed()
15   }
16 }
17
18 trait LoggingDataReader extends DataReader {
19
20   abstract override def readData(): List[Person] = {
21     val startMillis = System.currentTimeMillis()
22     val result = super.readData()
23     val time = System.currentTimeMillis() - startMillis
24     System.err.println(s"readData took ${time} milliseconds.")
25     result
26   }
27
28   abstract override def readDataInefficiently(): List[Person] = {
29     val startMillis = System.currentTimeMillis()
30     val result = super.readDataInefficiently()
31     val time = System.currentTimeMillis() - startMillis
32     System.err.println(s"readDataInefficiently took ${time} milliseconds.")
33     result
34   }
35 }
36
37 val dataReader = new DataReaderImpl with LoggingDataReader
```

What's happened here is much nicer — our timing code has been injected without interfering with the parent trait's implementations. This is enabled by the `super.readData()` calls, which is allowing us to essentially wrap the original function without changing how it's invoked (of course you could wrap in a separate function, but this

is cleaner!).

Notice here the `abstract override` override syntax - this will not work without it! Think about it this way - you're overriding a method **before** it gets initialized, so by definition you're overriding something abstract. Why / how are you overriding before initialization? Well, remember linearization:

- `DataReaderImpl LoggingDataReader`
- `l(DataReaderImpl) l(LoggingDataReader)`
- `DataReaderImpl DataReader LoggingDataReader DataReader`
- `DataReaderImpl LoggingDataReader DataReader`

So `LoggingDataReader` references via `super` the `readData()` of `DataReader`, which at that point of initialization (right to left) is still abstract. Then when `DataReaderImpl` provides an implementation, `LoggingDataReader`'s `super` call references and wraps **that** implementation. This is why the `abstract override` special syntax is necessary — the first concrete implementation is provided by `DataReaderImpl`, **NOT** `LoggingDataReader`!