Day 28: The chain of responsibility design pattern

This pattern is basically just about decoupling event handler components in your code. It's event-based processing, but in a synchronous, singe-JVM sort of way. The right way to do this is probably Akka, but you can also mimic it by making a linked-list-esque set of objects that will allow you to pass data through a set of methods. tl;dr —

```
val w1 = new Worker1(None)
val w2 = new Worker2(Some(w1))
val w3 = new Worker3(Some(w2))
new Worker4(Some(w3))
```

That's their example. Each object contains an instance of a linked object that will be the next thing in the process to evaluate some data. In this case, just imagine that each worker implements a common interface with a `process` method.

Of course, the Scala way to do this is via stackable traits and `abstract override` syntax, OR — a new functional approach is demonstrated using `PartialFunction`, which should get its own day. This approach is a little odd in the sense that it's just functions chained via `andThen`, but the partial functions allow conditioned `case` statements to shine - if it's not a match, exit `andThen` hit the next chained case. Code reproduced here for context:

```
trait PartialFunctionDispenser {

  def dispense(dispenserAmount: Int): PartialFunction[Money, Money] = {
    case Money(amount) if amount >= dispenserAmount =>
      val notes = amount / dispenserAmount
      val left = amount % dispenserAmount
      System.out.println(s"Dispensing $notes note/s of $dispenserAmount.")
      Money(left)
    case m @ Money(amount) => m
  }
}
class PartialFunctionATM extends PartialFunctionDispenser {

  val dispenser =
dispense(50).andThen(dispense(20)).andThen(dispense(10)).andThen(dispense(5))
}

```