Day 27: The command design pattern

The command design pattern is about using an explicit class called an **invoker** to execute a series of related commands with additional state / logging available on those commands. In short, you construct a few things:

- **Receiver** — a class that has the actual implementations of methods. This will be used by all `Commands`.
- **Command** — a common interface for executing the functionality of a receiver.
- **Invoker** — the thing that actual executes commands.

This will be more clear with code examples. Let's see we have the following **receiver**. It's just a normal class with implementations of various functionality:

```scala
case class Robot() {
  def cleanUp(): Unit = System.out.println("Cleaning up.")

  def pourJuice(): Unit = System.out.println("Pouring juice.")
}
```

Now we need **command** classes to put the `Robot` functionality behind a common interface. This allows our eventual invoker to call all commands the same way — in this case, by just calling the `execute` method on any commands passed in.

```scala
trait RobotCommand {
  def execute(): Unit
}

case class PourJuiceCommand(robot: Robot) extends RobotCommand {
  override def execute(): Unit = robot.pourJuice()
}

case class CleanUpCommand(robot: Robot) extends RobotCommand {
  override def execute(): Unit = robot.cleanUp()
}
```

Last — we need an **invoker** to actual execute any `RobotCommand` instances passed in, and to also help us with logging and history and literally whatever else we need.

```scala
import scala.collection.mutable.ListBuffer

class RobotController {
  val history = ListBuffer[RobotCommand]()

  def issueCommand(command: RobotCommand): Unit = {
    history.append(command)
    command.execute()
  }

  def showHistory(): Unit = {
    history.foreach(println)
  }
}
```

We use a mutable `ListBuffer` to maintain state on what's been executed. You can also imagine that instead of executing right away on `issueCommand`, we could have some methods like `executeTopCommand`, `executeAllCommands`, and so on. Basically, this is truly all this pattern is — store some state and interact with it later. In this case, our state consists of actual actions, so it can be pretty powerful! I mean, this is basically just imitating a CPU in some ways.

Here's how we can use it:

```
object RobotExample {
  def main(args: Array[String]): Unit = {
    val robot = Robot()
    val robotController = new RobotController

    robotController.issueCommand(MakeSandwichCommand(robot))
    robotController.issueCommand(PourJuiceCommand(robot))
    System.out.println("I'm eating and having some juice.")
    robotController.issueCommand(CleanUpCommand(robot))

    System.out.println("Here is what I asked my robot to do:")
    robotController.showHistory()
  }
}
```

Once again - BUT WAIT! This is all OOP, how does it look functionally? The same, except we send pure functions to the invoker as such:

```
class RobotByNameController {
  val history = ListBuffer[() => Unit]()

  def issueCommand(command: => Unit): Unit = {
    history.append(command _)
    command
  }

  def showHistory(): Unit = {
    history.foreach(println)
  }
}
```

This makes use of Scala's `by-name` parameter syntax, which is similar to a `() => Unit` call, but has some interesting different semantics and usage patterns. We'll save that for the next day!