Day 26: The strategy design pattern

Say we have multiple different algorithms for a problem, each of which is most suitable for a particular kind of data (or related access pattern). The **strategy design pattern** allows us to *chooe at runtime what algorithm to run, based entirely off the input data.*

Are you thinking about `case` and `if`? If so, don't stop - that's all this pattern is. tl;dr, here's what it gets down to:

```scala
object Parser {
  def apply(filename: String): Parser[Person] =
    filename match {
      case f if f.endsWith(".json") => new JsonParser
      case f if f.endsWith(".csv") => new CSVParser
      case f => throw new RuntimeException(s"Unknown format: $f")
    }
}
```

You don’t need to know what `Parser` here is. All that’s important - either way parse our `filename` as JSON or as CSV, depending on how `filename` ends (re: file type). So any code that invokes this function will determine how to parse its file at runtime.

But wait! We have two implementation options here. One of them is object-oriented, and looks like this:

```scala
object ParserExample {
  def main(args: Array[String]): Unit = {
    val csvPeople = Parser("people.csv")
    val jsonPeople = Parser("people.json")

    val applicationCsv = new PersonApplication(csvPeople)
    val applicationJson = new PersonApplication(jsonPeople)

    System.out.println("Using the csv: ")
    applicationCsv.write("people.csv")

    System.out.println("Using the json: ")
    applicationJson.write("people.json")
  }
}
```

The `csvPeople` and `jsonPeople` variables are instances of parsers determined dynamically by their file extensions. This is normal OOP - pass an instance of a class to construct another. But can we be functional? Yep:

```scala
object StrategyFactory {
  implicit val formats = DefaultFormats

  def apply(filename: String): (String) => List[Person] =
    filename match {
      case f if f.endsWith(".json") => parseJson
      case f if f.endsWith(".csv") => parseCsv
      case f => throw new RuntimeException(s"Unknown format: $f")
    }

  def parseJson(file: String): List[Person] =
JsonMethods.parse(StreamInput(this.getClass.getResourceAsStream(file))).extract[List[Person]
]

  def parseCsv(file: String): List[Person] = CSVReader.open(new
InputStreamReader(this.getClass.getResourceAsStream(file))).all().map {
      case List(name, age, address) => Person(name, age.toInt, address)
    }
}
```

What have we gained here?

- ANYBODY can create a parse method. It must simply take a `String` and returns a `List[Person]`, and then add it in as a way to potentially parse files. In other words - we're now much less rigid!
- We're passing functions as first-class objects around. Which, in Scala, is clearly appropriate - all "methods" / "functions" are simply instances of the corresponding `FunctionN` traits, where a method that takes one parameter is an instance of `Function1`. So clearly we're first-class — we're literally objects.

What have we lost?

- ANYBODY can create a parse method. Allowing total free will is a good way to get code explosion / sprawl. Maybe you should be forcing your coders to think about what's really different between your various parse methods.