Day 10: The factory method design pattern

This pattern allows us to define common interfaces via subtype polymorphism for related classes. Its main strength is to hide specific implementation details from library users, such that the users can interact with many related classes in the same way. This allows for clear APIs, and easy refactoring.

Let's take the simple example of database connections, per the book:

```scala
trait SimpleConnection {
  def getName(): String
  def executeQuery(query: String): Unit
}

class SimpleMysqlConnection extends SimpleConnection {
  override def getName(): String = "SimpleMysqlConnection"

  override def executeQuery(query: String): Unit = {
    System.out.println(s"Executing the query '$query' the MySQL way.")
  }
}

class SimplePgSqlConnection extends SimpleConnection {
  override def getName(): String = "SimplePgSqlConnection"

  override def executeQuery(query: String): Unit = {
    System.out.println(s"Executing the query '$query' the PgSQL way.")
  }
}
```

This is already summing up the factory pattern - some other class / object can take a `SimpleConnection`, and then implement / pass in either a MySQL or PostgreSql connection, while maintaning the same method calls and return types. Here's a specific factory example that uses this pattern:

```scala
abstract class DatabaseClient {
  def executeQuery(query: String): Unit = {
    val connection = connect()
    connection.executeQuery(query)
  }

  protected def connect(): SimpleConnection
}

```

```scala
class MysqlClient extends DatabaseClient {
  override protected def connect(): SimpleConnection = new SimpleMysqlConnection
}

class PgSqlClient extends DatabaseClient {
  override protected def connect(): SimpleConnection = new SimplePgSqlConnection
}
```

Now a `DatabaseClient` can be created that uses one or the other connection type! Also imagine if the connections had different constructors - this can all be hidden and generalized as far as the user goes!

This pattern can be tricky if there are lots of intertwined factory points (can make logical errors), but generally - if you're looking to abstract away details, this is your ticket!