Day 15: The builder design pattern

The **builder pattern** allows the user more flexibility in how an object is constructed - useful when building individual constructors would be laborious or rigid. The typical pattern in Java OOP usually involves two classes, one that can build the other. So, for example, you might have a `PersonBuilder` class that builds `Person` objects, where `PersonBuilder` would have methods like `setFirstName` and `setAge` and some kind of `build()` for actually producing a `Person` object.

In Scala, we can represent the builder pattern in three ways:

1. Classic mutability - a builder and a target. Not recommended due to aforementioned mutability. A variation on the builder pattern could be okay for sets / collections of things mutability updated, though, right?
2. Case classes with defaults. Within that - one variation validates its parameters and one doesn't.
3. Use generalized type constraints.

Java-Like Approach -- Classic Mutability

Basic code example:

```scala
class Person(builder: PersonBuilder) {
  val firstName = builder.firstName
  val lastName = builder.lastName
  val age = builder.age
}

class PersonBuilder {
  var firstName = ""
  var lastName = ""
  var age = 0
  def setFirstName(firstName: String): PersonBuilder = {
    this.firstName = firstName
    this
  }

  def setLastName(lastName: String): PersonBuilder = {
    this.lastName = lastName
    this
  }

  def setAge(age: Int): PersonBuilder = {
    this.age = age
    this
  }
  def build(): Person = new Person(this)
}

val person: Person = new PersonBuilder()
  .setFirstName("Ivan")
  .setLastName("Nikolov")
  .setAge(26)
  .build()
```

A couple things to note:

- All fields have defaults.
- Each setter feels pretty boilerplate-y.
- The actual usage is fluent and looks nice, but requires adding too much code for each new field.
- No constructors needed!

Case Class Approach

Instead of all the above boilerplate, use Scala's features!

```scala
case class Person(
  firstName: String = "",
  lastName: String = "",
  age: Int = 0
)
```

This accomplishes the same as the first option, but is much cleaner. The first option *could* allow some variation after an object is constructed by reconstructing it with a set member field - which a case class can't do. But we should prefer immutability as much as possible! When someone looks at an object, they shouldn't have to guess what sort of state it carries, based on what has been mutably transformed after the fact.

Generalized Type Constraints

Let's use a type-safe builder pattern to validate our dependencies in objet construction! But before we do, here's a simple bit of validation using `require`:

```scala
case class Person(
  firstName: String = "",
  lastName: String = "",
  age: Int = 0
) {
  require(firstName != "", "First name is required.")
  require(lastName != "", "Last name is required.")
}
```

This will fail objects at **runtime** that don't meet the requirements. This is nice and simple, but maybe we want compile-time requirement safety. How can we do that? Well, buckle your seatbelts, because we gotta do some evidence and type constraints.

Let's start with an ADT that will help us determine whether or not a `Person` object has a last name and / or first name. We'll also change our `PersonBuilder` to a parameterized class with covariance to this ADT, with a private constructor, a companion object, and a couple other defaults.

```scala
sealed trait BuildStep
sealed trait HasFirstName extends BuildStep
sealed trait HasLastName extends BuildStep

class Person(
  val firstName: String,
  val lastName: String,
  val age: Int
)

class PersonBuilder[PassedStep <: BuildStep] private (
  var firstName: String,
  var lastName: String,
  var age: Int
) {
  protected def this() = this("","",0)

  protected def this(pb: PersonBuilder[_]) = this(
    pb.firstName,
    pb.lastName,
    pb.age
  )

  def setFirstName(firstName: String): PersonBuilder[HasFirstName] = {
    this.firstName = firstName
    new PersonBuilder[HasFirstName](this)
  }

  def setLastName(lastName: String)(implicit ev: PassedStep =:= HasFirstName): PersonBuilder[HasLastName] = {
    this.lastName = lastName
    new PersonBuilder[HasLastName](this)
  }

  def setAge(age: Int): PersonBuilder[PassedStep] = {
    this.age = age
36     this
37   }
38 
39   def build()(implicit ev: PassedStep =:= HasLastName): Person = new Person(
40     firstName,
41     lastName,
42     age
43   )
44 }
45 
46 object PersonBuilder {
47   def apply() = new PersonBuilder[BuildStep]()
48 }
49 
```

Okay, this is all kinda complicated and hasn't gotten us anything yet. Here's the main thing to remember: **we use type constraints to enforce builder safety at compile time, but at a cost!** Let's break it down by line numbers.

L1:3 — We use an ADT to determine the kinds of relationships that will be enforced. This ADT could be extended.

L11:15 — we now have a private class constructor, and use aux constructors to build a `PersonBuilder` object that's covariant on our previously-defined ADT.

L24:27 — `setFirstName` does one key thing: it returns a `PersonBuilder[_]` parameterized to our ADT via `HasFirstName`. This is useful when...

L29:32 — ...when `setLastName` asks for parameter evidence! This is a type-safe, compile-time way to prove to Scala that some type requirement (here, that we've passed through `setFirstName` **strictly before** we call `setLastName`) has been satisfied. We meet the fun `=:=` operator, too — for more info on generalized type constraints like this, see this article.

So, pros and cons (see if you can tell how I feel about this):

Pros:

1. This is compile-time type safety.
2. It introduces some neat Scala concepts?

Cons:

1. It's way overly complicated. Have fun on your code review.
2. It relies on the order in which you chain your methods for the type evidence relationships to work here. It shouldn't matter whether your library user calls `setFirstName` or `setLastName` first, but here, it does.
3. It handles a use case that's really strange and shouldn't really exist — just, like, don't give your library users the option to construct the case class without a first and last name, right?
4. What the heck is the difference between `<:` and `<:<`? (Answer: see that article above again.)