Notatki - Lombok - Adnotacje cz.3

Spis treści

Adnotacje - cz.3

@Data

A co jeżeli uznamy, że zrobiło nam się za dużo adnotacji i chcemy zapisać to w sposób kompaktowy? Nie ma problemu. Wystarczy użyć `@Data`. Użycie adnotacji `@Data` oznacza jednoczesne użycie adnotacji `@ToString`, `@EqualsAndHashCode`, `@Getter`, `@Setter` oraz `@RequiredArgsConstructor`. Innymi słowy, stosujemy tę adnotację, gdy chcemy stworzyć **POJO** z całym boilerplate code, który normalnie byśmy stworzyli, bez wprowadzania specyficznych przypadków, zastosujemy adnotację `@Data`.

```
@Data
public class Dog {
    private String name;
}
```

Prawda, że prościej? Dokumentacja adnotacji @Data.

@Value

Adnotacja `@Value` jest stosowana jeżeli chcemy w prosty sposób stworzyć klasę **immutable**. `@Value` może być rozumiane jako odmiana **immutable** adnotacji `@Data`. Jeżeli stosujemy adnotację `@Value`, wszystkie pola w klasie stają się domyślnie `private` i `final`. Do tego nie są generowane settery (skoro pola są `final` to i tak bez sensu). Klasa zostaje oznaczona słówkiem `final`. Użycie tej adnotacji powoduje zachowanie analogiczne do posiadania adnotacji `@ToString`, `@EqualsAndHashCode`, `@Getter` oraz `@AllArgsConstructor` wraz z zachowaniem kwestii wspomnianych wcześniej.

Dokumentacja adnotacji @Value.

@Builder

Wyobraźmy sobie przypadek, w którym mamy klasę, która ma 10 pól, ale nie wszystkie są zawsze używane. W podejściu jakie znamy dotychczas, mamy 3 możliwości jak podejść do tworzenia obiektu

takiej klasy:

- stworzyć obiekt z konstruktora bezargumentowego i ustawić wartości tych pól przy pomocy setterów. Ale co jeżeli nie chcemy mieć setterów w naszej klasie? Wtedy ta możliwość odpada.
- stworzyć wszystkie możliwe kombinacje konstruktorów, z 1 argumentem, z 2 argumentami, z 3 argumentami itp. Do tego mamy możliwość różnych parametrów zestawionych ze sobą. Chyba nie muszę tłumaczyć, dalej, że raczej się tego tak nie robi 😉.
- stworzyć konstruktor ze wszystkimi dostępnymi parametrami i tam gdzie nie chcemy przekazywać wartości parametru, przekazać `null`. Powiedziałbym, że to rozwiązanie również odpada, bo jest nieczytelne. Do tego wywołując konstruktor z 10 parametrami, po jakimś trzecim idzie się zgubić co już jest podstawione, a co jeszcze zostało.

Wtedy pojawia się wzorzec projektowy `Builder`. Nie chcę opisywać tutaj czym są wzorce projektowe bo do tego jeszcze przejdziemy. Na teraz skupmy się natomiast nad sposobem rozwiązania opisanego wyżej problemu. Czyli chcemy mieć możliwość wyboru, które pola inicjujemy wartościami na etapie tworzenia obiektu. Jeżeli poszukasz materiałów dotyczących wzorca `Builder`, znajdziesz rozwiązania opisujące jak samodzielnie napisać kod dający taką możliwość. `Lombok` pozwala nam natomiast nie pisać takiego kodu ręcznie, tylko użyć adnotacji `@Builder`.

Nie zagłębiam się w tym momencie w to czym są wzorce projektowe ani na czym polega sam wzorzec projektowy `Builder`. Dokładnie zostanie to wyjaśnione gdy przejdziemy do omawiania wzorców projektowych. Na ten moment skupmy się na samym sposobie rozwiązania problemu opisanego wyżej.

```
@Data
@Builder
public class Employee {
    private String name;
    private String surname;
    private String email;
    private BigDecimal salary;
    private LocalDate dateOfBirth;
    private String address;
}
```

Mając klasę `Employee` zdefiniowaną jak wyżej, możemy teraz utworzyć obiekt tej klasy w ten sposób:

```
public class EmployeeRunner {

    public static void main(String[] args) {
        Employee employee1 = Employee.builder()
            .name("name")
            .surname("surname")
            .build();
        Employee employee2 = Employee.builder()
            .name("name")
            .surname("surname")
            .salary(BigDecimal.TEN)
            .build();
        Employee employee3 = Employee.builder()
            .name("name")
            .surname("surname")
```

```
            .dateOfBirth(LocalDate.of(2000, 1, 1))
            .build();
    }
}
```

Widzisz teraz, że możesz stworzyć instancje obiektu podając pola wybiórczo. Dokumentacja adnotacji @Builder.

@SneakyThrows

A może denerwują nas wyjątki checked? Czyli te wyjątki, które trzeba albo redeklarować, albo obsłużyć. Inaczej dostajemy błąd kompilacji. Lombok pozwala nam zrobić to wygodniej. Wyobraźmy sobie przykład, w którym staramy się złapać wyjątek checked i opakować go wyjątkiem runtime:

```
public class NonSneakyThrowsExample {

    public static void fileSize(Path path) {
        try {
            System.out.println(Files.size(path));
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Możemy zrobić to samo (ale krócej) stosując adnotację @SneakyThrows. Nie musimy wtedy deklarować bloku catch i dodawać obsługi catch (IOException e).

```
public class SneakyThrowsExample {

    @SneakyThrows
    public static void fileSize(Path path) {
        System.out.println(Files.size(path));
    }
}
```

Wspomniany wyżej przykład (ten bez Lombok) nie jest dokładną odwzorowaniem tego co robi Lombok. Gdy opakujemy wyjątek IOException wyjątkiem RuntimeException to otrzymany Stacktrace wygląda inaczej niż w przypadku zastosowania samego @SneakyThrows. Przykład ten ma pokazać sposób myślenia jaki stoi za adnotacją @SneakyThrows, a nie dokłady kod 1:1, który realizuje to samo zachowanie co adnotacja @SneakyThrows.

Dokumentacja adnotacji @SneakyThrows.

Minusy stosowania Lombok

Jak wszędzie, nie ma rozwiązań idealnych, każde podejście ma swoje plusy i minusy. Ewidentnym plusem jest to, że możesz napisać mniej kodu. Jednocześnie ten kod jest czystszy. Lombok już trochę czasu na rynku i można powiedzieć, że przetrwał, przeszedł próbę czasu ☺.

Problemem może być sytuacja, w której w połowie trwania projektu decydujemy się na wycofanie `Lomboka` z projektu. Wtedy możemy albo próbować stosować pluginy do **"delombokingu"** albo przepisać wszystko ręcznie. Kolejna kwestia jest taka, że przy bardziej skomplikowanych przypadkach, `Lombok` może nie obsłużyć ich w taki sposób jak oczekujemy - całe szczeście wtedy można napisać taki kod ręcznie. Dodatkowo, `Lombok` jak każda biblioteka może zawierać błędy. Błędy są najczęściej eliminowane w kolejnych wersjach bibliotek, ale czasem trzeba też na to trochę poczekać. Ostatni argument na minus jaki bym podał to kwestia, że dla osób, które nie znają tej biblioteki, początkowo taki kod może być trudny do zrozumienia.

Jak z każdą filozofią, każdy musi wyrobić swoje zdanie. Zapraszam Cię do wyrobienia swojego ☺.