Notatki - Lombok - Intro

Spis treści

Czym jest lombok? . . . 1
- Jak skonfigurować Lombok w projekcie? . . . 2
  - Gradle . . . 2
  - Maven . . . 2
  - Annotation Processor . . . 3
  - Intellij . . . 4

Definicje . . . 5
- POJO . . . 5
- Java Bean . . . 5
- Boilerplate . . . 6

Czym jest lombok?

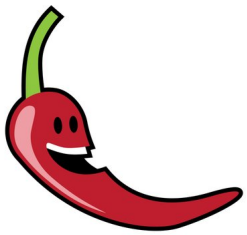
`Lombok` jest jedną z bibliotek, która ma za zadanie ułatwić nam pracę. Każda biblioteka rozwiązuje jakiś problem. Tak samo jest w tym przypadku.

Przypomnijmy, że biblioteka (library) to zestaw funkcji/metod, które można wywołać, które są zorganizowane w postaci klas. Inaczej sprawę ujmując są to gotowe klasy oraz metody, które możemy wykorzystywać w swoim kodzie/programie, które zostały napisane przez kogoś innego.

Java w swojej naturze jest **gadatliwa** (verbose). Stwierdzenia tego używa się w praktyce w stosunku do sytuacji, gdzie trzeba napisać dużo kodu bądź instrukcji aby osiągnąć jakiś cel. Jednocześnie też taki kod lub takie komendy nie przynoszą dużo wartości. Wartość w tym rozumieniu to kod, który służy do realizowania funkcji biznesowych aplikacji. W Javie poznaliśmy już dużo kodu, który "nie realizuje biznesu". Przykładem takiego kodu są konstruktory (takie proste nie realizujące nic innego jak przypisanie), gettery, settery, equals(), hashCode() lub toString().

Aby rozwiązać ten problem powstało coś takiego jak Project Lombok. Działa to w ten sposób, że zamiast pisać tego rodzaju kod, dołączamy bibliotekę, która na podstawie adnotacji (przykład adnotacji to `@Override`) określa, że w danej klasie mamy mieć konstruktor, getter itp. Wtedy w trakcie procesu budowania programu, na podstawie tych adnotacji automatycznie jest generowany bytecode w plikach `.class`, który zawiera wspomniane konstruktory, gettery itp. Inaczej mówiąc, nie musimy tworzyć tych konstruktorów ręcznie, wystarczy, że dodamy odpowiednie adnotacje w pliku `.java`.

Przypomnijmy, że programy w praktyce pisze się po to, żeby ktoś mógł dzięki nim szybciej bądź taniej zarabiać pieniądze. Tak samo jak automatyzacja fabryk lub linii produkcyjnych. Im szybciej możemy napisać faktyczny kod realizujący "biznes" tym lepiej, szybciej osiągamy finalny cel tworzenia takiego oprogramowania.



Obraz 1. Lombok logo. Źródło: https://medium.com/

Jak skonfigurować Lombok w projekcie?

Gradle

Jeżeli korzystamy z Gradle, tutaj znajdziemy instrukcję konfiguracji. Poniżej umieszczam gotowy plik `build.gradle`.

```
plugins {
    id 'java'
}

group = 'pl.zajavka'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

repositories {
    mavenCentral()
}

dependencies {
    compileOnly 'org.projectlombok:lombok:1.18.22'
    annotationProcessor 'org.projectlombok:lombok:1.18.22'

    testCompileOnly 'org.projectlombok:lombok:1.18.22'
    testAnnotationProcessor 'org.projectlombok:lombok:1.18.22'
}
```

Jak już się domyślasz, dopisek `test`, znaczy, że zależność będzie używana w testach. My na razie nie rozmawiamy o testach, więc ten fragment pominiemy (może zostać w `build.gradle`, przyda się na potem).

Wyjaśnienie czym jest `annotationProcessor` nastąpi później.

Maven

Jeżeli korzystamy z Maven, możemy skorzystać z konfiguracji opisanej w tym źródle. Korzystając ze wspomnianego źródła, możemy napisać taką konfigurację:

```
<project>
    <modelVersion>4.0.0</modelVersion>

    <groupId>pl.zajavka</groupId>
    <artifactId>lombok-maven-examples</artifactId>
```

```
    <version>1.0.0</version>

    <properties>
        <java.version>17</java.version>
        <lombok.version>1.18.22</lombok.version>
        <maven-compiler-plugin.version>3.8.0</maven-compiler-plugin.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>${lombok.version}</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>${maven-compiler-plugin.version}</version>
                <configuration>
                    <release>${java.version}</release>
                    <annotationProcessorPaths>
                        <path>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                            <version>${lombok.version}</version>
                        </path>
                    </annotationProcessorPaths>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Tag `scope` i jego wartość wynikają z tego, że potrzebujemy tej zależności na etapie kompilacji, aby stworzyć pliki `.class` z wygenerowanym kodem. `Lombok` nie jest już natomiast potrzebny w trakcie działania programu.

Dokumentacja `Lombok` wspomina również, o dodaniu `annotationProcessorPaths` w przypadku wersji Javy wyższej niż 9.

Annotation Processor

Co oznaczają słówka `annotationProcessor` oraz `testAnnotationProcessor` w przypadku **Gradle** oraz analogicznie w przypadku **Maven**?

Annotation processing jest narzędziem wbudowanym w `javac` do skanowania i przetwarzania adnotacji w trakcie trwania kompilacji. Możemy zarejestrować własny procesor adnotacji. Procesor taki jest w stanie na podstawie zdefiniowanych adnotacji generować kod na wyjściu. Wspomniałem wcześniej, że `Lombok` daje nam możliwość generowania kodu na podstawie zdefiniowanych adnotacji, dlatego też w ogóle rozmawiamy o **Annotation processing**.

Jeżeli chcemy taki procesor zarejestrować, to możemy to zrobić na 2 sposoby. Ręcznie, albo przez narzędzie.

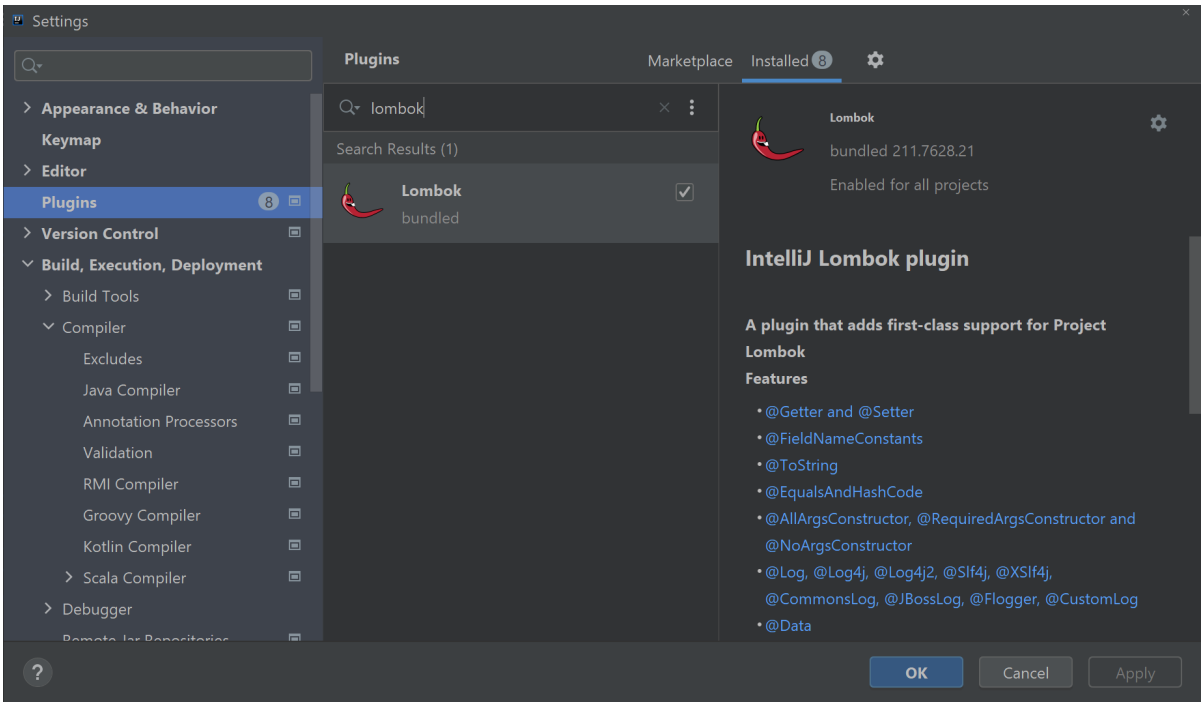
Jeżeli chcemy zrobić to ręcznie, należy dodać opcję `--processor-path path` lub `-processorpath path` do wywołania komendy `javac`. Cytując dokumentację:

> `--processor-path path` or `-processorpath path`
>
> Specifies where to find annotation processors. If this option is not used, then the class path is searched for processors.

Jeżeli natomiast chodzi o rejestrację takiego procesora, dokumentacja dedykowana pod **Maven** wspominała o ręcznej konfiguracji takiego procesora. Wspomina o tym również dokumentacja dedykowana pod **Gradle**. Na wszelki wypadek, przed konfiguracją `Lombok` upewnij się co mówi dokumentacja.

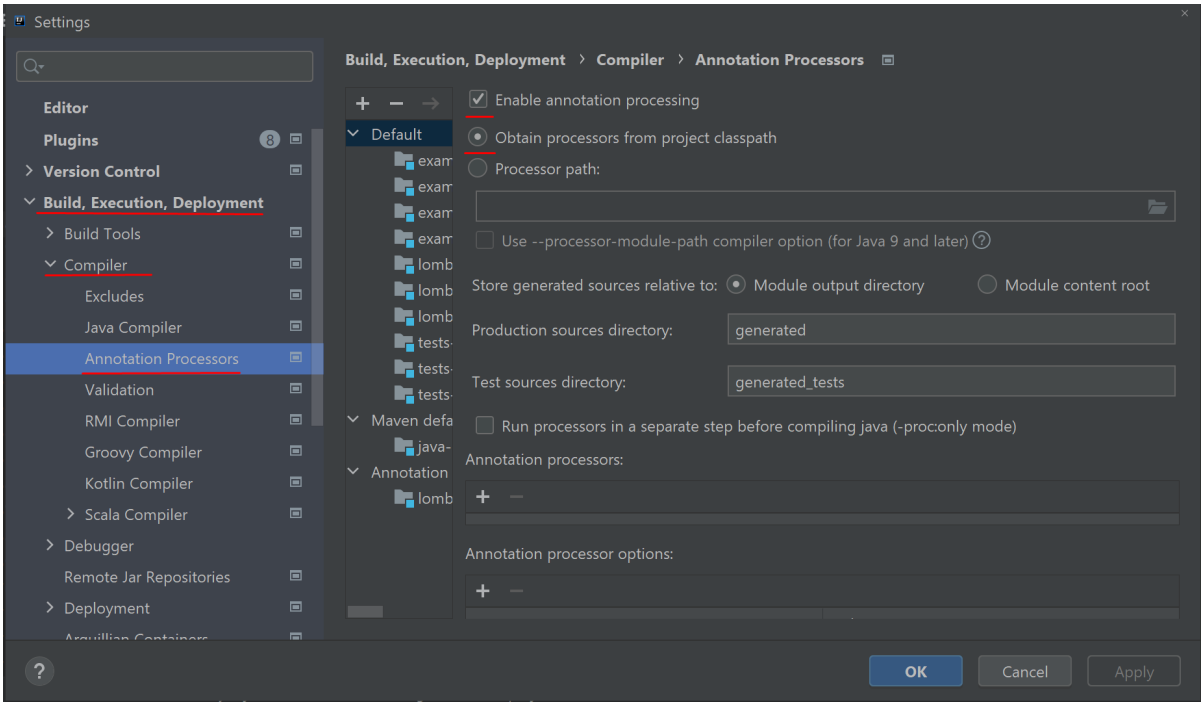
Intellij

Do tego należy dodać plugin `Lombok` do Intellij (*CTRL* + *ALT* + *S* > *Plugins*). Jeżeli nie będziemy mieli dodanego pluginu `Lombok`, Intellij będzie pokazywał nam błędy kompilacji, bo np. nie będzie widział danego konstruktora, gettera itp.



Obraz 2. Lombok Intellij Plugin

Jeżeli nadal coś nie będzie działało, można zobaczyć ustawienia, które pokazuję poniżej:



Obraz 3. Lombok Intellij Settings

Definicje

POJO

POJO czyli **P**lain **O**ld **J**ava **O**bject. Oznacza klasę, która w żaden sposób nie jest zależna od zewnętrznych bibliotek. Nie oznacza to stosowania żadnej konwencji, tzn, posiadania getterów, setterów itp. Ważne jest to, że klasa taka może być używana beż zależności do jakichkolwiek bibliotek czy frameworków.

Framework można na tym etapie rozumieć jak dużą bibliotekę, która w uproszczeniu może służyć jako szkielet aplikacji.

Java Bean

Czytając o przeróżnych aspektach, napotkasz w końcu stwierdzenie `Java Bean`. Określenie to narzuca pewnego rodzaju standard klasy Java, w której:

- Wszystkie pola są prywatne, a dostęp do nich jest realizowany przez gettery i settery
- Nazewnictwo getterów i setterów trzyma się konwencji `getX()` i `setX()`
- Mamy dostępny publiczny bezargumentowy konstruktor
- Klasa ta implementuje interfejs `Serializable`

Nazwa **Bean** została nadana, aby uwypuklić ten standard, którego celem jest tworzenie reużywalnych komponentów (klas). Reużywalnych w tym rozumieniu, że działanie bibliotek możemy wtedy oprzeć o obiekty z naszego projektu pod warunkiem, że spełniają one standardy. Można się też pokusić o stwierdzenie, że Intellij wspiera ten standard, dając nam możliwość generowania getterów, setterów i konstruktorów.

Boilerplate

Cytując Wikipedię:

> In computer programming, boilerplate code or just boilerplate are sections of code that are repeated in multiple places with little to no variation. When using languages that are considered verbose, the programmer must write a lot of code to accomplish only minor functionality. Such code is called boilerplate.

Czyli jest to taki kod, o którym wspominałem wcześniej. Proste konstruktory, gettery, settery, toString itp.

Notatki - Lombok - Adnotacje cz.1

Spis treści

Adnotacje - cz.1 . . . 1
Gettery i Settery . . . 1
@NonNull . . . 3
Konstruktory . . . 4
@NoArgsConstructor . . . 4
@RequiredArgsConstructor . . . 5
@AllArgsConstructor . . . 6
Delombok . . . 6
staticName . . . 6

Adnotacje - cz.1

Gettery i Settery

Biorąc na przykład klasy, w której mamy zdefiniowaną bardzo dużą ilość pól. Przykłady będziemy pokazywać na przykładzie piesków ☺.

Klasa Dog bez użycia Lombok

```
package pl.zajavka;

public class Dog {

    private final String name;

    private final Integer age;

    private final Owner owner;

    public Dog(final String name, final Integer age, final Owner owner) {
        this.name = name;
        this.age = age;
        this.owner = owner;
    }

    public String getName() {
        return name;
    }

    public Integer getAge() {
        return age;
    }

    public Owner getOwner() {
        return owner;
```

```
    }

}
```

Zwróć uwagę, że kod, który jest odpowiedzialny za konstruktor i gettery (nie ma setterów, bo pola są `final`) zajmuje więcej miejsca niż pola, które faktycznie realizują naszą "potrzebę biznesową". Utrudnia to jednocześnie czytelność tego kodu.

Wykorzystajmy zatem Lombok:

Klasa Dog z użyciem adnotacji @Getter

```
package pl.zajavka;

import lombok.Getter;

@Getter
public class Dog {

    private final String name;

    private final Integer age;

    private final Owner owner;

    public Dog(final String name, final Integer age, final Owner owner) {
        this.name = name;
        this.age = age;
        this.owner = owner;
    }

}
```

Widać, że kod skrócił się już w znaczny sposób. W ten sam sposób możemy określić settery (`@Setter`). W tym przypadku, nawet jeżeli pola są określone jako `final`, możemy dodać adnotację `@Setter` oprócz adnotacji `@Getter`. Natomiast gdy będziemy chcieli użyć taki setter, czyli zwyczajnie go wywołać, to zobaczymy, że nie mamy takiej metody dostępnej. Wynika to z tego, że pola są `final`, czyli nie możemy ponownie przypisać do nich wartości po utworzeniu obiektu.

Adnotacje `@Getter` i `@Setter` mogą zostać również napisane nad konkretnym polem. W przypadku poniżej określamy, że nie chcemy mieć gettera dla pola `name` i faktycznie go nie będzie. Osiągamy to poprzez dodanie dopisku `AccessLevel.NONE`. Ta informacja może być rozumiana jako modyfikator dostępu gettera, gdzie w tym przypadku możemy również napisać, że dane pole ma nie mieć gettera w ogóle. Możemy ustawić również inny **AccessLevel**, mówiący, że getter może być `PUBLIC`, `PROTECTED`, `PACKAGE`, lub `PRIVATE`. To samo ma zastosowanie dla adnotacji `@Setter`.

```
public class Dog {

    @Getter(AccessLevel.NONE)
    private final String name;

    //... pozostałe elementy klasy
}
```

Należy tutaj również dodać, że jeżeli będziemy dodawać lub usuwać pola z klasy to nie musimy nic więcej robić. gettery i settery są generowane na etapie kompilacji. Więc na etapie pisania możemy założyć, że nic nie musimy robić bo zostaną one wygenerowane automatycznie na podstawie określonej definicji/konfiguracji w momencie kompilacji. Nie musimy sami dodawać lub usuwać getterów lub setterów jak zmienimy coś w klasie. A normalnie byśmy musieli ☺.

Lombok dostarcza również dokumentację, w której możemy zobaczyć przykłady: getter i setter. W dokumentacji stwierdzenie `Vanilla Java` oznacza Javę bez użycia Lomboka.

@NonNull

Adnotacja `@NonNull` może być używana, jeżeli chcemy dodać sprawdzenie, czy wartość nie jest `null` i jeżeli jest to wyrzucić wyjątek **NullPointerException**. Adnotacja ta może być używana na polu, parametrze metody lub konstruktora. Zostanie wtedy automatycznie wygenerowane sprawdzenie, czy pole nie jest `null`.

Przykład z wykorzystaniem Lombok

```
package pl.zajavka;

import lombok.NonNull;

public class Person {

    private String name;

    public Person(@NonNull String name) {
        this.name = name;
    }
}
```

Przykład bez wykorzystania Lombok

```
package pl.zajavka;

import lombok.NonNull;

public class Person {

    private String name;

    public Person(@NonNull String name) {
        if (name == null) {
            throw new NullPointerException("name is marked @NonNull but is null");
        }
        this.name = name;
    }
}
```

Link do dokumentacji `Lombok` @NonNull.

Konstruktory

Kolejną wartością dodaną jest możliwość stosowania adnotacji generujących konstruktory. Każdy z tych wygenerowanych konstruktorów robi to co robi konstruktor - przypisuje wartości pól w obiekcie.

@NoArgsConstructor

Adnotacja `@NoArgsConstructor` dodaje konstruktor bezparametrowy, przykład poniżej:

```
@Getter
@Setter
@NoArgsConstructor
public class Dog {

    private String name;

    private Integer age;

    private Owner owner;

}
```

Zwróć uwagę, że pola nie są już `final`, dlaczego? Bo jeżeli zrobimy je teraz `final` to dostaniemy błąd kompilacji wynikający z tego, że pola `final` nie mają przypisanej wartości. Dzieje się tak bo konstruktor domyślny nie przypisuje wartości domyślnych polom. Jeżeli chcemy wymusić wartość domyślną na tych polach, należy zastosować `@NoArgsConstructor(force = true)`. Wtedy pola zostaną zainicjowane odpowiednio wartościami domyślnymi `0` / `false` / `null`.

Jeżeli natomiast zastosujemy taką kombinację:

Wykorzystanie adnotacji @NoArgsConstructor(force = true)

```
@Getter
@Setter
@NoArgsConstructor(force = true)
public class Dog {

    private final String name;

    private int age;

    private final Owner owner;

}
```

Spowoduje to wygenerowanie takiego kodu:

Analogiczny przykład kodu bez wykorzystania Lombok

```
public class Dog {

    private final String name;
```

```
    private int age;

    private final Owner owner;

    public Dog() {
        this.name = null;
        this.owner = null;
    }

    public String getName() {
        return this.name;
    }

    public Integer getAge() {
        return this.age;
    }

    public Owner getOwner() {
        return this.owner;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
}
```

Nie ma potrzeby przypisywania wartości `age` w konstruktorze, gdyż domyślnie jest to 0.

Link do dokumentacji `Lombok` @NoArgsConstructor.

@RequiredArgsConstructor

Adnotacja `@RequiredArgsConstructor` generuje konstruktor z polami klasy, które są albo `final`, albo oznaczone jako `@NonNull`. Do pól `@NonNull` dodawane jest również sprawdzenie czy pole nie jest `null`. Czyli konstruktor wyrzuci **NullPointerException** jeżeli przekażemy wartość jakiegoś pola jako `null`. Kolejność pól w konstruktorze zgadza się z kolejnością pól w klasie.

Wykorzystanie adnotacji @RequiredArgsConstructor

```
@Getter
@Setter
@RequiredArgsConstructor
public class Dog {

    @NonNull
    private String name;

    private Integer age;

    private final Owner owner;

}
```

Kod bez wykorzystania Lombok

```
public class Dog {
```

```
    @NonNull
    private String name;

    private Integer age;

    private final Owner owner;

    public Dog(@NonNull String name, Owner owner) {
        this.name = name;
        this.owner = owner;
    }

    public @NonNull String getName() {
        return this.name;
    }

    public Integer getAge() {
        return this.age;
    }

    public Owner getOwner() {
        return this.owner;
    }

    public void setName(@NonNull String name) {
        this.name = name;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
}
```

Zwróć uwagę, że adnotacja `@RequiredArgsConstructor` stworzyła konstruktor tylko dla parametrów oznaczonych jako `@NonNull` lub `final`. Pole `age` nie zostało wykorzystane.

Link do dokumentacji Lombok @RequiredArgsConstructor.

@AllArgsConstructor

Adnotacja `@AllArgsConstructor` generuje konstruktor z każdym polem w naszej klasie. Dla pól `@NonNull` dodawane jest sprawdzenie czy pole nie jest `null`. @AllArgsConstructor

Delombok

Jeżeli na którymś etapie zaczniesz zastanawiać się, co tak właściwie zostanie na końcu wygenerowane albo skąd są brane przykłady tego jak będzie wyglądał kod bez wykorzystania `Lombok`. Można to osiągnąć stosując `Lombok` plugin, który w zakładce `Refactor` (w paskach na górze ekranu), daje mi opcje *Delombok* > *All lombok annotations*.

staticName

Pamiętasz klasy `LocalDateTime` itp? A konkretnie sposób tworzenia tych obiektów? Wykorzystywana była metoda `of()`. `Lombok` daje nam możliwość dodania takiej metody przy wykorzystaniu adnotacji. Robi się

to w ten sposób:

Wykorzystanie Lombok i metody of()

```
@Getter
@Setter
@AllArgsConstructor(staticName = "of")
public class Dog {

    private String name;

    private Integer age;

    private final Owner owner;

}
```

Dzięki fragmentowi `staticName = "of"` możemy teraz napisać kod w ten sposób:

```
public class Example {
    public static void main(String[] args) {
        Dog dog = new Dog(); ①
        Dog dog2 = Dog.of("name", 12, null); ②
    }
}
```

① Ten konstruktor przestaje być dostępny, staje się on prywatny.

② Od teraz możemy ten obiekt tworzyć w ten sposób.

Notatki - Lombok - Adnotacje cz.2

Spis treści

Adnotacje - cz.2 . . . 1
@ToString . . . 1
@EqualsAndHashCode . . . 2
@With . . . 4

Adnotacje - cz.2

@ToString

Od tego momentu nie musimy już generować tej metody w Intellij. Wystarczy dodać coś takiego:

```
@Getter
@Setter
@AllArgsConstructor(staticName = "of")
@ToString
public class Dog {
    // kod klasy
}
```

Jeżeli teraz użyjemy pluginu, żeby zrobić **Delombok**, zobaczymy taką metodę:

```
public class Dog {
    // kod klasy

    public String toString() {
        return "Dog(name=" + this.getName() +
            ", age=" + this.getAge() +
            ", owner=" + this.getOwner() +
            ")";
    }
}
```

Możemy również jawnie podać, które pola mają zostać wyłączone z metody `toString()`.

```
@Getter
@Setter
@AllArgsConstructor(staticName = "of")
@ToString
public class Dog {

    @ToString.Exclude
    private String name;
```

```
    private Integer age;

}
```

Jeżeli teraz użyjemy pluginu, żeby uruchomić **Delombok**, zobaczymy taką metodę:

```
public class Dog {
    // kod klasy

    public String toString() {
        return "Dog(age=" + this.getAge() + ")";
    }
}
```

Możemy również zastosować adnotację `@ToString(onlyExplicitlyIncluded = true)`, a wraz z nią adnotację `@ToString.Include`. Przykład:

```
@Getter
@Setter
@AllArgsConstructor(staticName = "of")
@ToString(onlyExplicitlyIncluded = true)
public class Dog {

    @ToString.Include
    private String name;

    private Integer age;

}
```

Przy wykorzystaniu **Delombok**, dostaniemy taki kod:

```
public class Dog {
    // kod klasy

    public String toString() {
        return "Dog(name=" + this.getName() + ")";
    }
}
```

Sama adnotacja `@ToString` pozwala stosować więcej parametrów niż tutaj poruszone. W tym celu zapraszam do poczytania dokumentacji @ToString ☺.

@EqualsAndHashCode

Od teraz nie musimy już również generować metod `equals()` oraz `hashCode()` z poziomu Intellij. `Lombok` pomoże nam również w tym. W tym celu należy dodać adnotację `@EqualsAndHashCode`.

```
@Getter
@Setter
@AllArgsConstructor(staticName = "of")
```

```
@ToString
@EqualsAndHashCode
public class Dog {

    @ToString.Exclude
    private String name;

    private Integer age;

}
```

Przy wykorzystaniu **Delombok**, zobaczymy teraz coś takiego:

```
public class Dog {
    // kod klasy

    public boolean equals(final Object o) {
        if (o == this) return true;
        if (!(o instanceof Dog)) return false;
        final Dog other = (Dog) o;
        if (!other.canEqual((Object) this)) return false;
        final Object this$name = this.getName();
        final Object other$name = other.getName();
        if (this$name == null ? other$name != null : !this$name.equals(other$name)) return false;
        final Object this$age = this.getAge();
        final Object other$age = other.getAge();
        if (this$age == null ? other$age != null : !this$age.equals(other$age)) return false;
        return true;
    }

    protected boolean canEqual(final Object other) {
        return other instanceof Dog;
    }

    public int hashCode() {
        final int PRIME = 59;
        int result = 1;
        final Object $name = this.getName();
        result = result * PRIME + ($name == null ? 43 : $name.hashCode());
        final Object $age = this.getAge();
        result = result * PRIME + ($age == null ? 43 : $age.hashCode());
        return result;
    }
}
```

Mamy również do dyspozycji `@EqualsAndHashCode.Include` oraz `@EqualsAndHashCode.Exclude`. Jeżeli stosujemy `@EqualsAndHashCode.Include` należy wykorzystać adnotację `@EqualsAndHashCode(onlyExplicitlyIncluded = true)` analogicznie jak w `@ToString`.

Tak samo jak w przypadku adnotacji `@ToString`, mamy możliwość określenia, czy mamy wołać metodę `super.equals()` oraz `super.hashCode()` jeżeli dziedziczymy z jakiegoś obiektu. Z racji rzadkości używania tej kombinacji nie poruszam jej tutaj. Jeżeli będzie to potrzebne, zapraszam do przeczytania dokumentacji @EqualsAndHashCode ☺.

@With

Lombok pozwala nam w bardzo szybki sposób dodać metody `withX()`, gdzie `X` jest nazwą pola, do którego generujemy daną metodę. Metoda ta pozwala nam zmienić wartość pola (jak setter), ale jednocześnie możemy takie metody chainować. Należy jednak pamiętać, że w przypadku Lomboka, każda metoda `with()`, zwraca kopię poprzedniego obiektu ze zmienionym tym jednym polem, wrócimy jeszcze do tego.

Jeżeli realizowałeś/realizowałaś z nami bootcamp, pamiętasz pewnie projekt kalkulatora kredytu hipotecznego. Pamiętasz, że pisaliśmy tam ręcznie metody **with**, przykładowo `withName()`? `@With` jest podobne do tamtych metod.

```
@Getter
@Setter
@AllArgsConstructor(staticName = "of")
@With
public class Dog {

    @ToString.Include
    private String name;

    private Integer age;

}
```

Dzięki temu dostajemy teraz podobne zachowanie. Piszę podobne, gdyż w projekcie o kredycie hipotecznym, metoda `with()` była zdefiniowana w ten sposób:

```
public class Dog {
    public InputData withType(MortgageType type) {
        this.rateType = type;
        return this;
    }
}
```

Widać tutaj, że `with()` był wzbogaconym setterem. Po wywołaniu `with()`, operowaliśmy dalej na tym samym obiekcie, gdyż zwracaliśmy `this`. Inaczej mówiąc, operowaliśmy na obiekcie mutowalnym, czyli zmieniającym stan. Przy stosowaniu adnotacji `@With` tworzymy kopię poprzedniego obiektu z nową wartością. Nie mutujemy obiektu, tylko tworzymy nowy.

Żeby pokazać różnicę, zdefiniujmy metodę `with()` w klasie `Dog` w ten sposób:

```
@Getter
@Setter
@ToString
@AllArgsConstructor(staticName = "of")
public class Dog {

    private String name;

    public Dog withName(String name) {
        this.name = name;
        return this;
```

```
    }

}
```

I użyjemy tej metody:

```
public class DogRunner {

    public static void main(String[] args) {
        Dog dog = Dog.of("Burek");
        Dog dog2 = dog.withName("Romek");
        System.out.println(dog);
        System.out.println(dog2);
    }
}
```

To na ekranie wydrukuje się dwukrotnie:

```
Dog(name=Romek)
Dog(name=Romek)
```

Jeżeli natomiast zastosujemy adnotację `@With`, to przy tym samym wywołaniu, na ekranie wydrukuje się:

```
Dog(name=Burek)
Dog(name=Romek)
```

Czyli w poprzednim przypadku operowaliśmy cały czas na tym samym obiekcie w pamięci. W przypadku adnotacji `@With`, najpierw tworzymy nowy obiekt będący kopią obiektu oryginalnego i dopiero tę kopię modyfikujemy.

Dokumentacja adnotacji @With

Notatki - Lombok - Adnotacje cz.3

Spis treści

Adnotacje - cz.3 . . . 1
@Data . . . 1
@Value . . . 1
@Builder . . . 1
@SneakyThrows . . . 3
Minusy stosowania Lombok . . . 3

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 ☺.