Programowanie obiektowe - cz.2

Spis treści

Zapiski uczestnika Bootcampu Zajavka.pl w 12 tygodni by Bartek Borowczyk aka Samuraj Programowania. Dopiski na zielono od Karola Rogowskiego.

Modyfikatory dostępu

Modyfikatory dostępu skupiają się wokół słów kluczowych: `private`, `protected` i `public`. Są 4 modyfikatory dostępu, a w poprzednim zdaniu zostały wypisane 3. Jest tak, dlatego, że 4 modyfikator dostępu (czyli `package-private`, albo inaczej `default`), cechuje się tym, że nie piszemy go jako słowo kluczowe. Nie jest wtedy napisany żaden modyfikator dostępu.

Służą one do tego, żeby ograniczać widoczność pól, metod, konstruktorów lub klas z innych klas. I wyjaśniając każdy z nich:

- **private** - prywatny modyfikator dostępu oznacza, że pola, metody itp. są widoczne tylko z klasy, w której się znajdują,
- **default (package-private)** - defaultowy modyfikator dostępu określamy w ten sposób, że nie wpisujemy żadnego modyfikatora dostępu przy polu lub metodzie. Oznacza on natomiast, że pole lub metoda jest widoczna tylko dla klas znajdujących się w tej samej paczce co nasze pole lub metoda,
- **protected** - modyfikator dostępu protected oznacza, że pola, metody itp. mogą być dostępne tylko dla klas dziedziczących z klasy, w której napisaliśmy ten modyfikator dostępu, albo z klas, które są dostępne w tej samej paczce co klasa, w której napisaliśmy słówko protected,
- **public** - publiczny modyfikator dostępu oznacza, że pole, metoda itp. mogą być dostępne z każdego miejsca w kodzie.

Tabelka widoczności danego modyfikatora:

Modyfikatory dostępu				
Modifier	Class	Package	Subclass	World
private	Y	N	N	N
package-private	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

Obraz 1. Modyfikatory dostępu

Hermetyzacja, enkapsulacja

Hermetyzacja (enkapsulacja) jest jednym z fundamentów programowania obiektowego. Mechanizm ten polega na tym, że zamiast dawać dostęp do pól bezpośrednio, przez definiowanie ich jako publiczne, określamy te pola jako prywatne, a dostęp do nich odbywa się przy wykorzystaniu metod (np. getterów i setterów, o których powiemy sobie już niedługo). Dzięki temu nie "wycieka" nam stan tego obiektu na zewnątrz. Czyli operować na obiekcie możemy tylko poprzez jego własne zachowania (metody), a nie bezpośrednio na jego polach. Przykładowo:

```
public class Person {

    private String name;

    private String surname;

    public Person(final String name, final String surname) {
        this.name = name;
        this.surname = surname;
    }

    public String getName() {
        return name;
    }

    public String getSurname() {
        return surname;
    }

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

    public void setSurname(final String surname) {
        this.surname = surname;
    }
}
```

Konstruktor

Przypomnijmy sobie, jaki był cel istnienia konstruktorów.

Konstruktor służy do inicjalizacji danych obiektu w chwili jego tworzenia. Warto wiedzieć, że nawet jeśli nie tworzymy konstruktora w kodzie to i tak on będzie tworzony automatycznie w chwili tworzenia

obiektu. Konstruktor musi mieć taką samą nazwę jak klasa (w związku z tym jest też pisany wielką literą).

```
class Person() {
    // domyślny konstruktor - tak wygląda jeśli sami go nie zdefiniujemy
    Person() {}
}
```

Konstruktor jest wywoływany jedynie w chwili tworzenia obiektu. Obiekt tworzymy za pomocą instrukcji new i przypisanie nazwy konstruktora (konstruktor ma taką samą nazwę jak klasa).

Przykład tworzenia obiektu:

```
Person janek = new Person();
```

W przykładzie powyżej **instrukcja new** oznacza stworzenie nowego obiektu, znak równości oznacza przypisanie tego obiektu do zmiennej janek typu obiektowego `Person` (czyli przypisanie referencji do nowo powstałego obiektu). Konstruktor zaś jest wywoływany na tym nowym obiekcie i najczęściej ma za zadanie zainicjalizować wartości w polach obiektu. Za chwilę zobaczymy, jak to działa w praktyce.

```
class Animal {
    String name;
    Animal(String name) {
        this.name = name;
    }
}
// Gdzieś w programie:
// tworzymy obiekt typu Animal
Animal fafik = new Animal("faficzek");
System.out.println(fafik.name);
```

W naszym przykładzie stworzymy nowy obiekt, którego pole name, typu `String`, będzie zawierało tekst `faficzek`. Zerknijmy na przykład co by było, gdyby nie było tu konstruktora.

```
class Animal {
    String name;
    // Animal() {
        // domyślny konstruktor wywołany przy tworzeniu instancji
    // }
}
// Gdzieś w programie:
// nie możemy przekazać argumentu, bo domyślny konstruktor nie ma zdefiniowanych parametrów.
Animal fafik = new Animal();
System.out.println(fafik.name);
// ponieważ mamy tu typ String tak więc domyślną wartością będzie null,
// zostanie ona przypisana w chwili tworzenia obiektu.
```

Słówko this w konstruktorze

Powiedzieliśmy sobie o słówku `this` w kontekście odwołania do pól w klasie. Słowo kluczowe `this` może

również wystąpić w formie wywołania konstruktora:

```
public class Cat {
    private String name;

    public Cat() {
        this("Romek");
    }

    public Cat(String name) {
        this.name = name;
    }
}
```

W przykładzie powyżej, wywołanie `this("Romek");` służy do wywołania innego konstruktora z tej samej klasy. Java jest w stanie rozpoznać, który konstruktor chcemy wywołać na podstawie ilości i rodzaju parametrów konstruktora. Należy pamiętać, że w jednej klasie nie możemy mieć 2 metod o tej samej nazwie i sygnaturze, czyli o tej samej liście parametrów. To samo tyczy konstruktora, stąd też Java jest w stanie rozpoznać, który konstruktor ma być wywołany poprzez `this("Romek");` Należy też pamiętać, że przy takim zapisie wywołanie `this("Romek");` musi być pierwszą linijką w konstruktorze!

Słówko super w konstruktorze

No to super! No właśnie, jest też coś takiego jak `super`.

```
public class Cat extends Animal {

    public Cat() {
        super("Romek");
    }

}
```

Słówko `super` w konstruktorze służy do wywołania konstruktora w klasie `Animal`, z której dziedziczy klasa `Cat`. Skąd Java wie, który konstruktor wywołać? Tak samo, jak w przypadku wywołania `this("Romek");` I znowu, wywołanie `super("Romek");` musi być pierwszą linijką w konstruktorze.

Overriding

Zwracam uwagę, **overriding** często jest mylony z **overloadingiem**, a są to różne koncepcje.

Jeżeli klasa dziecka (subklasa) posiada zdefiniowaną taką samą metodę jak jest zdefiniowana w klasie rodzica (superclass), to zjawisko takie określamy jako method overriding. Innymi słowy, jeżeli w klasie dziedziczącej mamy konkretną implementację metody, która była deklarowana w klasie rodzica, nazywamy to overridingiem.

Używamy tego, gdy chcemy określić konkretną implementację metody w klasie Child, bo implementacja w klasie parent jest za mało szczegółowa (albo nie ma jej wcale, o czym dowiemy się później). Mechanizm ten jest również używany w przypadku polimorfizmu, o którym dowiemy się później.

Na co zwracać uwagę:

- metoda musi nazywać się tak samo jak w klasie parent,
- metoda musi mieć taką samą listę parametrów jaką ma metoda w klasie rodzica,
- musi mieć miejsce dziedziczenie (albo implementacja interfejsu, o której powiemy potem), żeby móc overridować metodę.

Przykład overridowania metody:

```
class Animal {
    // W klasie rodzica definiujemy metodę
    void run() {
        System.out.println("Animal is running");
    }
}

class Dog extends Animal {
    // Definiujemy taką samą metodę jak w klasie rodzica i jeżeli
    // stworzymy obiekt Dog, to właśnie ta metoda zostanie wywołana
    void run() {
        System.out.println("Dog is running");
    }

    public static void main(String[] args) {
        Animal animal = new Dog();
        animal.run(); // wywoła się metoda z klasy Dog
    }
}
```

W praktyce poleca się stosowanie Adnotacji `@Override` (Adnotacje są mechanizmem, który pomaga przekazywać informację o kodzie. Takie meta informacje - czyli określenia odnośnie do naszego kodu. Nie chcę się tu skupiać nad samym mechanizmem Adnotacji). Adnotacja ta daje nam możliwość sprawdzenia, czy prawidłowo overridujemy metodę wyrzucając nam błąd kompilacji, jak coś jest nie tak.

Gettery i Settery

Bezpośrednio nie mamy dostępu do pól prywatnych spoza klasy, w której się znajdują. Możemy za to skorzystać z metod w danej klasie, by zwrócić czy zmienić te pola.

Gettery i settery to takie metody w klasie, które udostępniają Ci dostęp do pól prywatnych tej klasy. Oczywiście takie metody są publiczne (bo co do zasady mają być dostępne dla innych obiektów).

Najczęściej w nazwach znajduje się get (getter) i set (setter), ale to jest kwestia konwencji. Przykładowo często dla zmiennych typu boolean zamiast `getVariable()` pisze się `isVariable()`.

Przykład getterów:

```
public class Car {

    private String company;

    private String model;

    private int productionYear;

    public String getCompany() {
        return company;
    }

    public String getModel() {
        return model;
    }

    public int getProductionYear() {
        return productionYear;
    }
}
```

Oczywiście w setterach możemy także wstawiać więcej instrukcji, np. taką, która sprawdzi czy dana wartość nie jest `null`em, a jeśli jest, to zwrócone zostanie coś innego niż ta wartość. jak w przykładzie poniżej:

```
public String getModel() {
    return this.model == null ? "default" : this.model;
}
```

Przykład setterów:

```
public void setCompany(final String company) {
    this.company = company;
}

public void setModel(final String model) {
    this.model = model;
}
```

W innej klasie/obiekcie możemy wtedy ustawić wartości, wywołując na danym obiekcie settery.

```
car.setCompany("Mercedes");
car.setModel("GLA");
car.setProductionYear(2020);
```

Bardzo często w setterach stosujemy też walidację wprowadzanych wartości, czyli np. wykluczamy pewne wartości (czyli czy godzimy się by dana wartość była zapisana w danym polu).

```
public void setCompany(String company) {
    if ("brak".equals(company)) {
        System.out.println("No way mate!");
        return;
    }
    this.company = company;
}
```

Co do getterów i setterów. W praktyce stosuje się sposoby, żeby nie pisać ich bezpośrednio, tylko stosować generatory kodu, które robią to za nas, ale o tym dowiemy się w przyszłości.

Kolejność elementów w klasie

Istnieją różne konwencje umieszczania elementów w klasie. Karol poleca taką praktykę:

- **package i importy nad deklaracją klasy** - to akurat wymóg a nie konwencja.

W samej klasie:

- pola statyczne (będzie o tym w przyszłości)
- pola o zasięgu public
- pola o zasięgu protected
- pola o zasięgu domyślnym (bez modyfikatora dostępu)
- pola o zasięgu private

Tak jak poniżej:

```
public class Cabriolet {

    public String company;

    protected String model = initModel();

    String model2;

    private String color = initColor();
}
```

I dalej:

- bloki inicjalizacyjne
- konstruktory

```
{
    printDuringInitBlock(4);
}

{
    printDuringInitBlock(2);
}

{
    this.company = "Audi";
    printDuringInitBlock(1);
}

{
    printDuringInitBlock(3);
}

public Cabriolet() {
    System.out.println(Cabriolet.class + " constructor called");
}

public Cabriolet(String as) {
    System.out.println(Cabriolet.class + " constructor called");
}

public Cabriolet(int i) {
    System.out.println(Cabriolet.class + " constructor called");
}

public Cabriolet(double a) {
    System.out.println(Cabriolet.class + " constructor called");
}
```

Dalej powinny pojawić się metody:

- Jeśli jakieś metody są przeciążone (ta sama nazwa, ale inne parametry), powinny być obok siebie (zgrupowane). Nawet jeżeli mają różne modyfikatory dostępu.

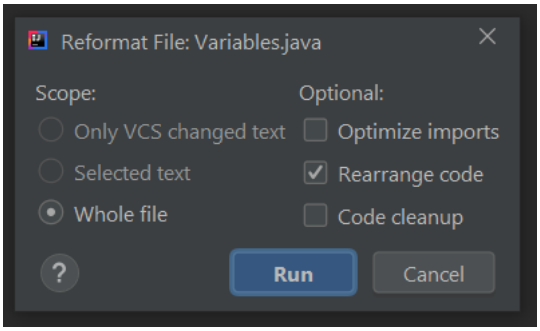
```
private void printDuringInitBlock(int i) {
    System.out.println("I'm printing during init block in " + Cabriolet.class + " block number " + i);
}

public void printDuringInitBlock(String i) {
    System.out.println("I'm printing during init block in " + Cabriolet.class + " block number " + i);
}
```

- Metody nadpisywane np. `toString()`:

```
@Override
public String toString() {
    return "Cabriolet{" +
        "company='" + company + '\'' +
        ", model='" + model + '\'' +
        '}';
}
```

- gettery i settery (tu już bez screena)

Co do samej konwencji formatowania, z pomocą przychodzi nam Intellij. Ma on określone ustawienia domyślne, w jakiej kolejności formatować elementy w klasie, możemy też oczywiście te ustawienia zmienić. W praktyce najczęściej stosowane są te domyślne. Jeżeli chcemy użyć automatycznego formatowania kodu, żeby Intellij "posprzątał" plik za nas, należy w danej klasie użyć skrótu `ctrl + alt + shift + L` i wybrać opcję "Rearrange code".



Klasy abstrakcyjne

To taka klasa, z której nie da się utworzyć obiektu (instancji klasy). Struktura:

```
public abstract class nameClass {}
```

Można je wykorzystać wtedy, gdy chcemy mieć wspólne pola i metody, ale by jednocześnie nie dało się stworzyć takiego obiektu. Np. chcemy tworzyć obiekt konkretnej marki samochodu, ale nie samochód jako obiekt ogólny, przy czym potrzebujemy takiego uniwersalnego zestawu metod i pól, z których mogą skorzystać inne klasy. Wtedy tworzymy klasę `Car` i dziedziczymy z niej do konkretnych klas marek samochodowych.

Metody, które stworzymy w klasie abstrakcyjnej, mogą być albo "normalne", albo abstrakcyjne. Metoda abstrakcyjna to taka, która nie ma zdefiniowanego ciała, przykładowo:

```
public abstract class Animal {

    private String color;

    public Animal(final String color) {
        this.color = color;
    }

    abstract String gimmeVoice();
}
```

Założeniem takiej metody jest to, żeby zdefiniować jej ciało (czyli co ma ona w praktyce robić) w klasie, która będzie dziedziczyła z naszej klasy abstrakcyjnej. Natomiast zapisanie, metody abstrakcyjnej w klasie abstrakcyjnej ma "wymusić" zakodowanie jak ma wyglądać ciało takiej metody w klasie implementującej. Zwróćmy też uwagę na podobieństwo do interfejsów, o których jest napisane poniżej.

Metoda `abstract` może przyjąć modyfikator `public`, `protected` czy domyślny (bez modyfikatora), ale nie

może być `private`. Deklarowanie takiej metody jako `private` nie ma żadnego sensu, bo przecież z założenia musimy ją nadpisać w klasie implementującą naszą klasę abstrakcyjną, ale nie możemy tego zrobić, bo metoda jest `private`, więc jej nie widać.

Jeżeli jakakolwiek metoda w klasie jest abstrakcyjna, to sama klasa też musi być abstrakcyjna.

Przykład dziedziczenia z klasy abstrakcyjnej `Animal` przez klasę `Cat`:

```
public class Cat extends Animal {

    private String name;

    public Cat(String name, String color) {
        super(color);
        this.name = name;
    }

    @Override
    public String gimmeVoice() {
        return "Cat meow!";
    }
}
```

I w tym miejscu, gdy mamy już do czynienia z klasą "normalną" musimy dokonać implementacji metody, przy czym nie może ona mieć modyfikatora zawężającego względem deklaracji w klasie abstrakcyjnej - jeśli tam był `public` to tutaj też musi być `public` (w przykładzie `gimmeVoice()` mamy modyfikator `default` co oznacza, że na pewno nie mieliśmy public w deklaracji metody w klasie abstrakcyjnej).

Interfejsy

Konstrukcja w Javie, która służy, by określić kontrakt zachowań. Ta konstrukcja mówi, że dany obiekt ma wykonać dane zachowania. Interfejs mówi jakie zachowanie ma być zapewnione, a implementujący obiekt to dostarcza. Przykład z pilotem i telewizorem Karola pokazuje, o co chodzi. Pilot (interfejs) ma przycisk przełącz kanał (interfejs mówi, że ma być takie zachowania), a obiekt takie zachowanie implementuje.

Interfejs to struktura, która posiada zestaw "wypisanych" metod, które nie mają implementacji. Klasa, która implementuje interfejs musi już mieć zdefiniowane te metody.

Chyba, że jest abstrakcyjna, to wtedy nie musi. Implementować wszystkie metody abstrakcyjne musi dopiero klasa, która w hierarchii dziedziczenia jest pierwszą klasą nieabstrakcyjną.

Przykład prostego interfejsu i definiowania wymaganej metody:

```
public interface Voiceable {

    String gimmeVoice();
}
```

Istnieje konwencja, by interfejsy nazywać przez określenie co on może zrobić (czyli z końcówką -able np. `Voiceable` jak w przykładzie). W praktyce natomiast często bardzo ciężko jest to tak zrobić niestety ☺.

Wszystkie metody zdefiniowane w interfejsie są `public abstract`. Nie ma zatem potrzeby pisać w interfejsach:

```
public abstract String gimmeVoice()
```

Dodam tutaj, że mówiąc wszystkie, mamy na myśli metody, które nie są `default` ani `static`. `Static` zostanie wyjaśniony już niedługo, natomiast koncepcja metody `default` jest wyjaśniona poniżej.

By zaimplementować interfejs w danej klasie, należy użyć słowa kluczowego `implements`. W jednej klasie możemy zaimplementować więcej niż jeden interfejs.

```
public class Cat implements CatStrokeable, Voiceable {}
```

W klasach implementujemy metody z implementowanego interfejsu. Override oznacza, że nadpisujemy (implementujemy) tu metodę z interfejsu.

```
public class Cat implements CatStrokeable, Voiceable {

    @Override
    public void doSomethingAsStrokeableCat() {
        System.out.println("I like being stroked!");
    }

    @Override
    public void beStroked() {
        System.out.println("I am stroked and it's really nice man!");
    }

    @Override
    public String gimmeVoice() {
        return "I'm giving You my voice";
    }
}
```

Interfejs może rozszerzać `extends` inny interfejs. Implementacje metod robimy dopiero w pierwszej klasie, która zaimplementuje dany interfejs.

```
public interface CatStrokeable extends Strokable {

    void doSomethingAsStrokeableCat();

}
```

Interfejs jest "kontraktem". Chcę danego zachowania, ale nie interesuje mnie jak to zaimplementujesz - to tego rodzaju kontrakt.

Metoda defaultowa w interfejsie

Od Javy 8 mamy dostępną możliwość określenia domyślnego zachowania metod w interfejsach.

Jeśli nie byłoby implementacji w klasie, to zostałaby zastosowana metoda domyślna zadeklarowana w interfejsie.

```
public interface Voiceable {

    String gimmeVoice();

    default void sing(String songName) {
        singMeASongWithName(songName);
    }

    private void singMeASongWithName(String songName) {
        System.out.println("Default singing song method: " + songName);
    }

}
```

Różnica między klasą abstrakcyjną a interfejsem? Tych różnic nie ma wcale tak wiele. Szczególnie od czasu Java 8 gdzie wprowadzono metodę defaultową.

Główne różnice między klasą abstrakcyjną a interfejsem po zmianach w Javie 8 to:

- Klasa abstrakcyjna jest klasą, a interface jest interfacem ☺,
- Klasa może mieć stan, który można modyfikować metodami nieabstrakcyjnymi, ale interfejs nie może mieć takiego stanu, bo nie można w nim definiować pól,
- Interfejs nie może mieć konstruktora, klasa abstrakcyjna może,
- Wszystkie metody w interfejsach z założenia są abstrakcyjne (oprócz default i static), podczas gdy w klasie abstrakcyjnej wcale takie być nie muszą.

Polimorfizm

Zasada mająca zastosowanie do OOP. Podklasy mogą mieć własne zachowania, ale i dzielić zachowania z klasą rodzica.

```
public class Car {

    protected String color;

    public Car(final String color) {
        this.color = color;
    }

    public void describe() {
        System.out.print("Car color: " + color);
    }
}
```

Mamy klasę, z której dziedziczy np. klasa Cabriolet.

```
public class Cabriolet extends Car {

    private boolean roofOpened;

    public Cabriolet(final String color, final boolean roofOpened) {
        super(color);
        this.roofOpened = roofOpened;
    }
}
```

W której nadpisujemy metodę `describe()` (ale to jeszcze nie jest polimorfizm)

```
@Override
public void describe() {
    System.out.println("Cabriolet description start");
    super.describe();
    System.out.println("Cabriolet has opened roof: " + roofOpened);
    System.out.println("Cabriolet description end");
}
```

I teraz przechodzimy już do **polimorfizmu**. Na początku pamiętajmy, że każda klasa w Javie rozszerza też klasę Object. Zobaczmy jak możemy stworzyć obiekt i referencję.

```
public static void main(String[] args) {
    Cabriolet cabriolet1 = new Cabriolet("red", true);
    Car cabriolet2 = new Cabriolet("blue", false);
    Object cabriolet3 = new Cabriolet("white", true);
}
```

Każdy z tych obiektów w tym wypadku będzie mógł skorzystać z metod, które są dostępne w danym typie (klasie).

- `cabriolet1.describe()` - będzie ok, wywołana zostanie metoda describe z klasy `Cabriolet`.
- `cabriolet2.describe()` - będzie ok, ale nie wywoła metody z klasy `Car` (jest tam metoda `describe()`), a z klasy `Cabriolet`, bo został stworzony obiekt typu `Cabriolet`, choć jest przechowywany w referencji typu `Car`. To jest właśnie typowe zachowanie polimorficzne. Na etapie wywołania Java decyduje, jaki to jest obiekt i decyduje, którą metodę wywołać.
- `cabriolet3.describe()` - nie możemy tej metody wywołać, bo `Object` w klasie nie ma zdefiniowanej metody `.describe()` - na takie przypadki trzeba uważać, jak posługujemy się klasami bazowymi albo interfejsami w definicjach zmiennych (referencjach).