Programowanie obiektowe - cz.1

Spis treści

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

Klasa

class - "*klasa to jest schemat, na podstawie którego opisujemy świat*" (zanotowane zdanie od Karola). Opisuje cechy i zachowania, które będzie miał obiekt utworzony na podstawie takiego schematu.

Czy korzystaliśmy już z klas w Bootcampie? Tak, zarówno tych wbudowanych, dostarczanych przez Javę (najczęściej była to klasa `String`) jak i tworzyliśmy własne, choć niespecjalnie przywiązując uwagę, do czego nam to było ☺.

Struktura klasy - najprostsza

```
class Nazwa-Klasy {
    // ciało klasy
```

```
}
```

Klasa może składać się z:

- **pól/atrybutów** (zmienne bezpośrednio w klasie) - określają stan obiektu.
- **metod** - metoda opisuje zachowanie obiektu danej klasy (lub samej klasy).
- **konstruktor** - specjalna konstrukcja języka (podobna do metody), która jest wywoływana w momencie tworzenia obiektu (konstruktor, bo konstruuje). W klasie domyślnie znajduje się konstruktor, nawet jeśli go nie zdefiniujemy. Konstruktorów w klasie może być kilka, różnią się one wtedy od siebie listą parametrów.

O czym warto pamiętać:

- Metoda `main()` nie jest metodą wymaganą w klasie! Jest ona natomiast wymagana, jeżeli chcemy uruchomić nasz program. Program może mieć zdefiniowanych więcej niż jedną metodę `main()` i każda wtedy będzie uruchamiała ten program "ze swojej strony". Każda klasa może mieć tylko jedną metodę `main()`, ale najczęściej stosuje się tylko jedną metodę `main()`, w klasie, z której uruchamiamy program. Pozostałe klasy służą wtedy do realizacji logiki programu.
- Z klas należy korzystać jako z jednostki logicznej, która grupuje zachowania i dane jakiegoś aspektu programu. Konkretna Klasa powinna definiować tylko jeden element programu. Jeśli mamy definicje `uczeń` to klasa `Student` nie koniecznie powinna grupować bezpośrednio oceny tego `ucznia` czy przedmiotów których uczy się `student` - to można zrobić w innych klasach, a ich obiekty przypisać po prostu do `ucznia`.
- Klasa sama w sobie nie jest obiektem, ale umożliwia tworzenie obiektów (stworzony obiekt nazywamy **instancją** lub **egzemplarzem**)
- Klasa definiuje nowy typ danych o nazwie takiej samej jak nazwa klasy (klasa typ `Car`, definiuje nowy typ danych, którym jest `Car`).

Pole w klasie

Definiujemy je bezpośrednio w klasie. Zgodnie z konwencją powinniśmy to zrobić na początku klasy. Jest to tylko konwencja, zrobić to możemy również i na końcu, natomiast przyjęło się, że robione to jest na początku.

```
public class Person {
    String name;
    int age;
}
```

Jeśli nie określimy wartości pól w chwili tworzenia obiektu, to zostaną im przyporządkowane wartości domyślne. W przypadku typów numerycznych będzie to `0`, w przypadku obiektów `null` (`String` więc też miałby `null`), a w przypadku typu `boolean` będzie to `false`. Mówimy tutaj oczywiście o typach prymitywnych, tzn. `int`, `boolean`. Dla ich odpowiedników 'klasowych', czyli `Integer`, `Boolean` wartość domyślna będzie wynosiła `null`.

Składnia pola

```
modyfikator dostępu (opcjonalny) typ-zmiennej nazwa-zmiennej
```

Warto zapamiętać, że pola są dostępne w całej klasie. Dobrą praktyką jest tworzenie specjalnych metod w klasie, które pozwalają pobrać wartość pola lub je zmienić (o czym przekonamy się już niedługo).

Metoda

Z czego składa się metoda:

- nazwy,
- typu zwracanego lub słowa `void`, jeśli metoda nic nie zwraca,
- modyfikatora dostępu (opcjonalny),
- dodatkowych cech metody (opcjonalne),
- parametrów metody (opcjonalne).

```
modyfikator-dostępu typ-zwracany nazwa-metody (
    typ parametr1,
    typ parametr2 (itd)
) {
    // ciało metody
    return zwracana-wartość;
    // return jest konieczny tylko jeśli metoda ma coś zwracać
}
```

Mała uwaga co do **instrukcji return**. Return poza tym, że poprzedza zwracaną z metody wartość, to także kończy działanie metody. Dlatego może być używany także w metodach, które nic nie zwracają (`void`), a nawet użyty wielokrotnie np. w celu sterowaniem zakończenia metody (przykład poniżej). Powrót z metody do miejsca, w którym została wywołana następuje albo po zakończeniu ciała metody, albo gdy program napotka instrukcje `return`.

Przykładowo:

```
void example () {
    if(warunek) return;
    // ciało metody
    if(warunek) {
        return;
    }
    // ciało metody
}
```

W chwili deklarowania metody możemy także zdefiniować jej parametry, które stanowią zmienne lokalne tej metody (zmienne lokalne, czyli zmienne dostępne tylko w obrębie tej metody). Za pomocą parametrów możemy przekazywać dane do metody w chwili jej wywołania.

Definiowanie metody z parametrami

```
void example(int par1, String par2) {
    // ciało
}
```

Wywołanie metody z argumentami

```
example(arg1, arg2);
```

O czym warto pamiętać:

- Metoda powinna wykonywać jedno konkretne zadanie, choć oczywiście technicznie nic nie stoi na przeszkodzie, by była bardzo rozbudowana i robiła "mnóstwo rzeczy".
- Metoda może zwracać typ prosty albo obiekt (wtedy wskazujemy jaki typ zwracanego obiektu). Może też nic nie zwracać (wtedy musimy wskazać w jej deklaracji `void`).
- Parametry, które definiujemy w chwili deklaracji klasy to w praktyce lista deklaracji zmiennych lokalnych (zmiennych dostępnych w danej metodzie). Za ich sprawą (parametrów) przekazujemy do metody wartości i możemy ich używać w metodzie. Ich wartość jest nadawana w chwili wywołania metody (wprowadzone wartości określamy atrybutami).
- W Javie (na moment pisania tego materiału) nie ma czegoś takiego jak wartość domyślna parametrów w metodzie. Można takie zachowanie zastąpić przy wykorzystaniu mechanizmu **overloadingu**.

Konstruktor

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.
```

I jeszcze jeden przykład, zanim przejdziemy do wyjaśnienia słowa kluczowego `this`.

```
class Animal {
    int age = 4;
}
// Gdzieś w programie:
Animal fafik = new Animal();
System.out.println(fafik.age); // 4
```

Tutaj (powyżej) warto zauważyć, że każdy nowo tworzony obiekt miałby swoją kopię wartości `age`. W każdym przypadku byłoby to `4`, ale można by taką wartość oczywiście później nadpisać.

This

Przejdźmy więc to zrozumienia **this**, bo chyba to w tym momencie może to budzić pewne wątpliwości. Słówko `this` oznacza "*ten obiekt*", przy czym `this` nabiera znaczenia (referencji do obiektu) dopiero przy tworzeniu obiektu.

```
class Animal {
    int age;

    Animal(int age) {
        this.age = age;
        // na tym etapie this nie ma jeszcze zawartości,
        // to tylko deklaracja klasy
    }
}
```

Przyjrzyjmy się poszczególnym elementom kodu powyżej:

- `Animal(int age)` - określamy tu parametr konstruktora (może być więcej niż jeden lub może nie być ich wcale). W naszym przypadku definiujemy, że chcemy przy tworzeniu obiektu typu `Animal` określić wartość `int` i przypisać ją do lokalnej zmiennej `age`. Póki co tylko tyle.
- `this.age = age;` - tutaj wprowadzony (przy przyszłym wywołaniu konstruktora) argument (który jest zmienna lokalną o nazwie `age` w naszym przykładzie), chcemy przypisać do pola age zdefiniowanego w klasie.

W jaki sposób odwołać się do pola o tej samej nazwie co zmienna lokalna? Użyć `this`! This, co jak już wiesz, oznacza "*ten obiekt*", ten, czyli ten, na którym wykonywane jest działania, czyli ten, który powstanie dzięki instrukcji `new`. Oczywiście w chwili definiowania klasy i konstruktora, słowa `this` jeszcze nic nie zawiera (nie zawiera referencji do żadnego obiektu), ale w chwili wywołania będzie już zawierało referencje do nowo tworzonego obiektu. Zobaczmy:

```
class Animal {
    int age;
    Animal(int age) {
        this.age = age;
    }
}

// gdzieś w kodzie
Animal fafik = new Animal(12);
/* Tworzymy zmienną(referencję) typu Animal o nazwie fafik.
Dalej tworzony jest obiekt przypisywany do tej zmiennej (referencja do niego).
Następnie wywoływany jest konstruktor, w naszym przypadku Animal(int age).
W miejsce parametru podstawiony jest argument 12,
czyli tworzona jest zmienna lokalna (dostępna w konstruktorze) age = 12.
Następnie przechodzimy do ciała konstruktora i tutaj w miejsce this podstawiony zostaje nowy obiekt
(ten do którego prowadzi zmienna fafik)
a więc:
fafik.age = age czyli do pola age w obiekcie fafik zostaje przypisane 12.
Tylko tyle :) */
```

Pamiętaj, gdy umieścimy kropkę po referencji (np. `fafik.` czy `this.`) to odwołujemy się wtedy do pola obiektu, albo jego metody.

Warto zwrócić uwagę, że zmienna lokalna (parametr o nazwie `age`) nie nadpisuje pola `age` a jedynie sprawia, że pole age nie jest widziane w metodzie, bo przysłania je właśnie zmienna lokalna tworzona przez parametr o tej samej nazwie co pole. Innym rozwiązaniem tej sytuacji, zamiast użycia `this`, może być użycie innej nazwy zmiennej lokalnej, tak by nie "*gryzła się*" z polem. W takiej sytuacji, nawet jeśli

nie użyjemy `this.age` a po prostu `age`, konstruktor i tak odwoła się do pola `age`, ponieważ nie zostanie ono przesłonięte przez zmienną lokalną o tej samej nazwie. Tak w przykładzie poniżej:

```
class Animal {
    int age;
    Animal(int ageAnimal) {
        age = ageAnimal;
        // this.age też zadziała, ale w takim przypadku
        // nie jest potrzebne.
    }
}
```

O czym warto pamiętać:

Choć konstruktor przypomina metodę to też się od niej różni. Nie może być wywołany w programie poza sytuacją, gdy jest tworzony nowy obiekt. Musi mieć też określoną nazwę, taką samą jak nazwa klasy. Poza tym konstruktor nie może nic zwracać.

Nazwa metody i lista parametrów to **sygnatura**. Mogą istnieć metody o tej samej nazwie, ale z inną liczbą parametrów. Wtedy mówimy, że mają różne sygnatury.

Overloading

Stosując **overloading** metod (nie mylić z **overridingiem**, który będzie omówiony później) możemy mieć w jednym pliku kilka metod, które mają tę samą nazwę, ale różną listę parametrów.

```
int add(int x, int y) {
    return x + y;
}

double add(double x, double y) {
    return x + y;
}

long add(long x, long y) {
    return x + y;
}
```

Dzięki temu możemy mieć kilka metod, które różnią się listą parametrów i przez to mogą realizować w sumie kilka implementacji, które będą różnić się pewnymi szczegółami.

Obiekt

Obiekt możemy określić za pomocą trzech jego elementów:

- zachowanie (metody),
- stan (pola),
- miejsce w pamięci - referencja do zmiennych typu obiektowego. Dwa obiekty stworzone za pomocą

wywołania tego samego konstruktora, nie są tymi samymi obiektami, są odrębne.

Tworzenie obiektu

Obiekt tworzymy wg. poniższego schematu:

```
Typ(klasa) nazwa-zmiennej = new Nazwa-klasy;
```

Definicja klasy

```
public class Person {}
```

Tworzenie obiektu będącego instancją/egzemplarzem klasy Person

```
Person bartek = new Person();
Person jacek = new Person();
```

W tym wypadku zmienna bartek przechowuje referencję do obiektu typu `Person`. Podobnie jak zmienna `jacek`. I choć w tym wypadku obiekty są identyczne, to nie są tymi samymi obiektami. To zupełnie osobne instancje/egzemplarze.

equals() i hashcode()

equals(Object otherObject)

Tak jak nazwa sugeruje, metoda służy do sprawdzania równości dwóch obiektów. Implementacja domyślna sprawdza, czy obiekty są równe poprzez porównanie ich referencji, czyli inaczej mówiąc, czy dwie referencje wskazują na to samo miejsce w pamięci. No ale my możemy napisać swoją implementację ☺.

hashcode()

Zwraca unikalną wartość `int` dla danego obiektu, która może zostać obliczona w trakcie działania programu. Najczęściej, ale wcale nie musi to być, wartość `int` jest wyliczana na podstawie adresu obiektu w pamięci.

Obie metody możemy nadpisać, tylko pojawia się pytanie "po co?".

Skoro `equals()` służy do sprawdzania równości obiektów, można ją nadpisać, żeby sprawdzać równość obiektów "po naszemu". Jeżeli chcemy, aby dwie osoby które mają identyczny `pesel` były uznane za równe w rozumieniu metody `equals()` należy metodę `equals()` nadpisać.

Możemy wtedy stworzyć 2 obiekty klasy `Person` i wystarczy, aby miały one równe wartości pola `pesel`, aby po wywołaniu kodu `person1.equals(person2)` rezultat zwrócił `true`.

Przykład metody equals()

```
public class Person {

    private String pesel;

    public Person(final String pesel) {
        this.pesel = pesel;
    }

    // Metoda equals() ma parę założeń
    @Override
    public boolean equals(final Object o) {
        // Jeżeli porównany ze sobą ten sam obiekt,
        // equals() powinno zwrócić true, stąd pierwszy if
        if (this == o) {
            return true;
        }
        // Jeżeli porównamy ze sobą 2 obiekty innego typu
        // powinno zwrócić false,
        // stąd w drugim kroku sprawdzamy czy typ obiektu "o"
        // jest tego samego typu co "this".
        // Można to zrobić przykładowo przez porównanie
        // rezultatów metod getClass()
        if (o == null || this.getClass() != o.getClass()) {
            return false;
        }
        // Następnie musimy rzutować obiekt "o" na klasę
        // Person, bo w tej chwili "o" jest typu Object.
        // Chcemy zrobić z tego Person, żeby móc porównać Person
        // z Person, a nie Person z Object.
        // Więcej o rzutowaniu będzie później, natomiast można
        // to zrobić w ten sposób jak poniżej
        final Person person = (Person) o;
        // Tutaj następuje faktyczne porównanie wartości pól
        return Objects.equals(pesel, person.pesel);
    }

    @Override
    public int hashCode() {
        return Objects.hash(pesel);
    }
}
```

W powyższym przykładzie sprawdzaliśmy tylko `pesel`, ale jeżeli chcielibyśmy dołożyć do sprawdzenia równości także inne pola, można to spokojnie napisać w ten sposób.

Jednocześnie pojawiło się coś takiego w przykładzie jak `hashCode()` i znowu, po co się tego używa? Wyobraźmy sobie, że mamy 1000 osób i każda nazywa się inaczej. Chcemy znaleźć w tej grupie konkretnie "Marka Kowalskiego", w jaki sposób możemy to zrobić?

W życiu codziennym można zwyczajnie krzyknąć, ale pisząc algorytmy to w sumie nie do końca tak działa ☺. Szukając kogoś takiego w algorytmie, musielibyśmy podejść bezpośrednio do każdej z osób i zapytać konkretnie, czy nazywa się ona "Marek Kowalski". Wtedy jest szansa, że albo znajdziemy taką osobę za 3 razem, ale może być też tak, że znajdziemy taką osobę za 989 razem. Jak można to przyspieszyć? Podzielić osoby o "podobnych" cechach (albo w jakiś inny sposób) na grupy, mając jakiś sposób na określenie, że jak ktoś nazywa się "Marek Kowalski" to będzie w grupie 7 z 10. Tym sposobem

jest `hashCode()`, ale to zaraz.

Wyobraźmy sobie teraz, że wiemy, że mamy 10 grup i umiemy określić na podstawie imienia i nazwiska osoby, do której grupy ma ona przynależeć. Wtedy bierzemy takie imię i nazwisko jak "Marek Kowalski", wiemy, że osoba o takim imieniu i nazwisku będzie w grupie 7 (bo policzyliśmy to wykorzystując metodę `hashCode()`) i idziemy bezpośrednio do grupy 7 szukać, ograniczając w taki sposób ilość osób do odpytania o imię i nazwisko. Po to właśnie może służyć `hashCode()`, konkretna implementacja wykorzystania nastąpi jak zaczniemy omawiać **kolekcje**.

I dodam jeszcze jedną rzecz, mówi się o czymś takim jak dobry `hashCode()`. Dobry `hashCode()` jest wtedy, jak możemy takich grup zrobić dużo różnorodnych, słaby jest wtedy, jeżeli do jednej grupy wrzucilibyśmy wszystkich.

Kontrakt między equals() i hashCode()

Kontrakt polega na tym, że:

- jeżeli kiedykolwiek nadpiszemy metodę `equals()` to musimy też nadpisać `hashCode()` bo jeżeli obiekty są sobie równe, to ich `hashCode()` też musi być równe. Zatem jak modyfikujemy `equals()` to musimy też zmodyfikować `hashCode()`,
- jeżeli wywołujemy `hashCode()` na tym samym obiekcie więcej niż raz, za każdym razem powinniśmy dostać ten sam rezultat,
- przy wywołaniu `hashCode()` przy kilku kolejnych uruchomieniach programu, nie powinna nam ona zwracać tej samej wartości przy kolejnych uruchomieniach programu,
- jeżeli 2 obiekty są równe w rozumieniu metody `equals()`, to wywołanie `hashCode()` dla tych 2 obiektów powinno zwrócić tę samą wartość.
- Jeżeli natomiast 2 obiekty nie są równe w rozumieniu `equals()`, ich `hashCode()` też nie musi być różne. Natomiast jest to dobrą praktyką w celu poprawy wydajności aplikacji.

toString()

Po co jest metoda `toString()`? Każdy obiekt można wydrukować. Domyślnie, jak będziemy próbowali wydrukować stworzony przez nas obiekt, dostaniemy na ekranie jakieś krzaki.

Jeżeli chcemy, żeby obiekt drukował nam się z uwzględnieniem wartości jego pól, należy dla niego napisać metodę `toString()`.

Dziedziczenie

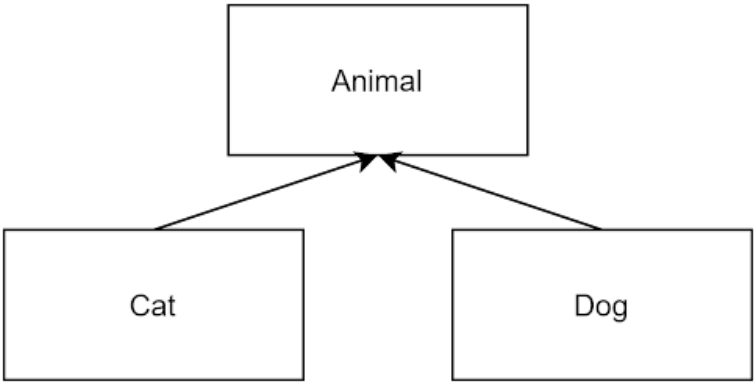
Dziedziczenie w Javie odnosi się do możliwości dziedziczenia pól oraz metod z jednej klasy przez drugą. W Javie nazywa się to również rozszerzaniem klasy, dlatego używane jest w tym celu słówko `extends`. Jedna klasa może rozszerzać drugą i takie zjawisko nazywa się dziedziczeniem. Dochodzi tutaj pewne nazewnictwo, gdyż klasa, z której dziedziczymy, nazywa się rodzicem (**parent**) albo superklasą (**superclass**). Klasa, która dziedziczy, czyli dziecko (**child**) może się też nazywać subklasą (**subclass**). Czyli subklasa rozszerza superklasę, albo inaczej subklasa dziedziczy z superklasy.

Samo dziedziczenie jest sposobem na reużywanie kodu. To znaczy, mamy w jednej klasie jakiś kod i

chcemy mieć podobną funkcjonalność w innym miejscu w naszym projekcie.

Opcja pierwsza, zrobić kopiuj - wklej ☺. Opcja druga, odziedziczyć pewne zachowania i cechy obiektu. Jest do tego też opcja trzecia zwana **kompozycją**, ale ona zostanie wyjaśniona później.

Pod spodem został umieszczony diagram pokazujący jak może takie zachowanie wyglądać dla klas `Animal`, `Cat` i `Dog`.



Obraz 1. Diagram hierarchii klas

Klasa `Cat` dziedziczy cechy i zachowania i klasy `Animal`, to samo dzieje się w przypadku klasy `Dog`. Klasa `Animal` jest superklasą, natomiast `Cat` i `Dog` są subklasami. Jeżeli wszystkie zwierzęta jedzą i dodamy metodą umożliwiającą jedzenie w klasie `Animal`, a następnie dziedziczymy to zachowanie w klasie `Cat` oraz `Dog`, będzie to oznaczało, że klasy `Cat` oraz `Dog` mogą jeść ☺.

Jak w praktyce podjąć decyzję, czy dziedziczyć?

Określając w jaki sposób, potrzebujesz pracować z klasami i obiektami w swoim programie. Czy Kot oraz Pies powinny być traktowane jednakowo w kontekście pewnego zachowania?

Inaczej mówiąc, czy potrzebujesz, żeby "przetwarzać" obie klasy w jednakowy sposób np. w kontekście ich zachowań? Jeżeli tak, to dziedziczenie może być dobrym sposobem na wyciągnięcie takiego zachowania i umieszczenie go we wspólnym miejscu. Jeżeli natomiast byłoby to robione na siłę, czyli nigdy te 2 klasy nie będą dzieliły zachowań, ale mają podobne metody i chcemy tylko uniknąć duplikowania kodu, dlatego piszemy dziedziczenie? Wtedy dziedziczenie jest bez sensu, będzie prowadziło w przyszłości do dziwnych relacji między klasami i trudność w przyszłej reorganizacji kodu.

Hierarchia klas

Hierarchia klas - Jeżeli mówimy o superklasach i subklasach, mówimy też o czymś takim jak hierarchia klas. Na górze tej hierarchii mamy superklasę (klasę, z której dziedziczą klasy). Niżej mamy subklasy (klasy dziedziczące). Subklasy mogą dziedziczyć z innych subklas, które mogą dziedziczyć z superklasy. Subklasa może też być superklasą dla klas, które dziedziczą z niej samej. Takie dziedziczenie może mieć kilka poziomów, czyli np. `Animal` → `Mammal` → `Cat` → `Persian`.

Co się dzieje z polami i metodami, gdy dokonujemy dziedziczenia?

Gdy klasa dziedziczy z jakiejś superklasy, dziedziczy też jej pola oraz metody.

Metody, czyli zachowania mogą zostać zredefiniowane (overridowane). Czyli możemy napisać jakąś

metodę ponownie, zmieniając jej zachowanie w stosunku do zachowania z metody w klasie bazowej (superklasie). Pól nie możemy **overridować**, w przypadku pól przykrywamy pola z klasy bazowej, natomiast w praktyce odradzam robienie takich rzeczy, bo powoduje to później duże problemy z czytaniem takiego kodu.

Konstruktory nie są dziedziczone w subklasach, ale subklasa wywołując swój konstruktor, domyślnie najpierw wywołuje konstruktor z klasy bazowej. Dzieje się to przy wykorzystaniu słówka `super();` wyjaśnionego we fragmencie notatek o konstruktorach.

Modyfikatory dostępu

Do samego dziedziczenia ma też odniesienie kwestia modyfikatorów dostępu. Jeżeli rozszerzamy superklasę, wszystkie metody z modyfikatorem `public` oraz `protected` są automatycznie dziedziczone. Oznacza to, że klasa dziedzicząca zaczyna mieć do nich dostęp, tak jakby klasa dziedzicząca sama je definiowała.

Jeżeli chodzi o modyfikator `package-private`, dostęp do takich metod mamy tylko, gdy subklasa znajduje się w tej samej paczce co superklasa. Prywatne metody nie są dziedziczone, bo tylko superklasa je widzi. (No bo są prywatne 😄). Za moment wspomnimy o modyfikatorach ponownie.

Wielokrotne dziedziczenie

Java pozwala subklasie dziedziczyć tylko z jednej superklasy. Są języki, które pozwalają rozszerzać kilka klas jednocześnie, natomiast Java pozwala tylko na jedną klasę. Twórcy podjęli taką decyzję ze względu na problemy, jakie się pojawiają, gdy dziedziczymy z kilku klas jednocześnie. Jeżeli przykładowo mamy 2 superklasy i pojawią się w nich dwie metody o identycznej nazwie, to którą z nich odziedziczyć? Żeby wyjść naprzeciw i nie mieć takich problemów, w Javie można rozszerzyć tylko jedną klasę. Można natomiast implementować kilka interfejsów, ale jest to temat na później.

Do zadeklarowania dziedziczenia służy słówko **"extends"**:

```
public class Animal {

    protected String name;

    public Animal(String name) {
        this.name = name;
    }
}

public class Cat extends Animal {

    public String surname;

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

    void printName() {
        System.out.println("name:surname = " + name + ":" + surname);
    }
}
```

W przykładzie klasa `Cat` dziedziczy (rozszerza klasę) z klasy `Animal`. Z racji wprowadzenia dziedziczenia, pole `name` z klasy `Animal` jest również dostępne w klasie `Cat`. Skoro pole name jest odziedziczone z klasy `Animal` (i nie jest ono prywatne), klasa `Cat` może odwołać się do pola `name`, jakby to było jej własne pole. Dlatego metoda `printName()` może odwołać się bez problemu do pola `name`.

Jednocześnie w powyższym przykładzie pole `surname` nie jest dostępne z klasy `Animal`, gdyż jest ono zdefiniowane w klasie `Cat`. Bo to klasa `Cat` dziedziczy z klasy `Animal`, a nie odwrotnie!

Dziedziczenie z klasy Object

Skoro poruszyliśmy już temat dziedziczenia, warto wspomnieć, że każda klasa w Javie automatycznie dziedziczy z klasy `Object`. Stąd właśnie możemy pisać takie konstrukcje jak `equals()`, `hashCode()` czy `toString()`. Klasa `Object` ma zdefiniowane te metody z ich domyślnymi implementacjami. Jeżeli chcemy mieć swoje implementacje tych metod, musimy je zdefiniować i nadpisać swoimi implementacjami, tak jak jest wspomniane we fragmentach odpowiednio o `equals()`, `hashCode()` i `toString()`.

Autoboxing

Autoboxing polega na automatycznej konwersji, która jest dokonywana przez kompilator Javy pomiędzy typami prymitywnymi (`byte`, `char`, `int`, `long`), a ich odpowiednikami w postaci klas, inaczej zwanymi **wrapper classes**. Przykładowo możemy używać ze sobą wymiennie `int` i `Integer`, `char` i `Character`.

Pamiętajmy, że `String` jest klasą, nie ma on typu prostego jak `int` dla `Integera`. Najprostszy przykład:

```
Integer a = 123;
```

W przykładzie powyżej przypisaliśmy wartość prymitywną `123` do zmiennej typu `Integer` (czyli klasy).