

Dziedziczenie

Czego się dowiesz

Czym jest dziedziczenie,
w jaki sposób dziedziczyć klasy w Javie,
jakie są najważniejsze fakty związane z dziedziczeniem
w Javie,
jak zablokować możliwość dziedziczenia.

Wstęp

We wcześniejszych lekcjach przewinęło się stwierdzenie, że w Javie możliwe jest tworzenie typów, które odzwierciedlają nasze otoczenie. Nauczyliśmy się już tworzyć własne typy obiektowe definiując odpowiednie klasy, a w tej lekcji dowiemy się jak tworzyć między nimi dodatkowe relacje. Obiekty, które otaczają nas w naszym życiu można często zakwalifikować do różnych kategorii, np.:

człowiek, małpa, pies i delfin są ssakami,
ssaki, ptaki, gady, płazy, ryby to kręgowce,
kręgowce i bezkręgowce nazwiemy zwierzętami.

(możliwe, że podział ten da się przedstawić w lepszy sposób, ale tyle zapamiętałem z biologii w gimnazjum, czy liceum).

To co ważne to na każdym z takich poziomów można wyróżnić pewne elementy wspólne dla wszystkich wymienionych typów zwierząt, ale każdy z nich posiada również pewne cechy charakterystyczne. Przykładowo każdy z wymienionych ssaków ma oczy, ale tylko delfin ma płetwy. Wszystkie kręgowce mogą się poruszać, ale tylko ryby mają skrzela, a ptaki pióra.

Java pozwala na wygodne budowanie takiej abstrakcji otaczającego nas świata, która będzie odzwierciedlała podział elementów, które będą nam potrzebne w

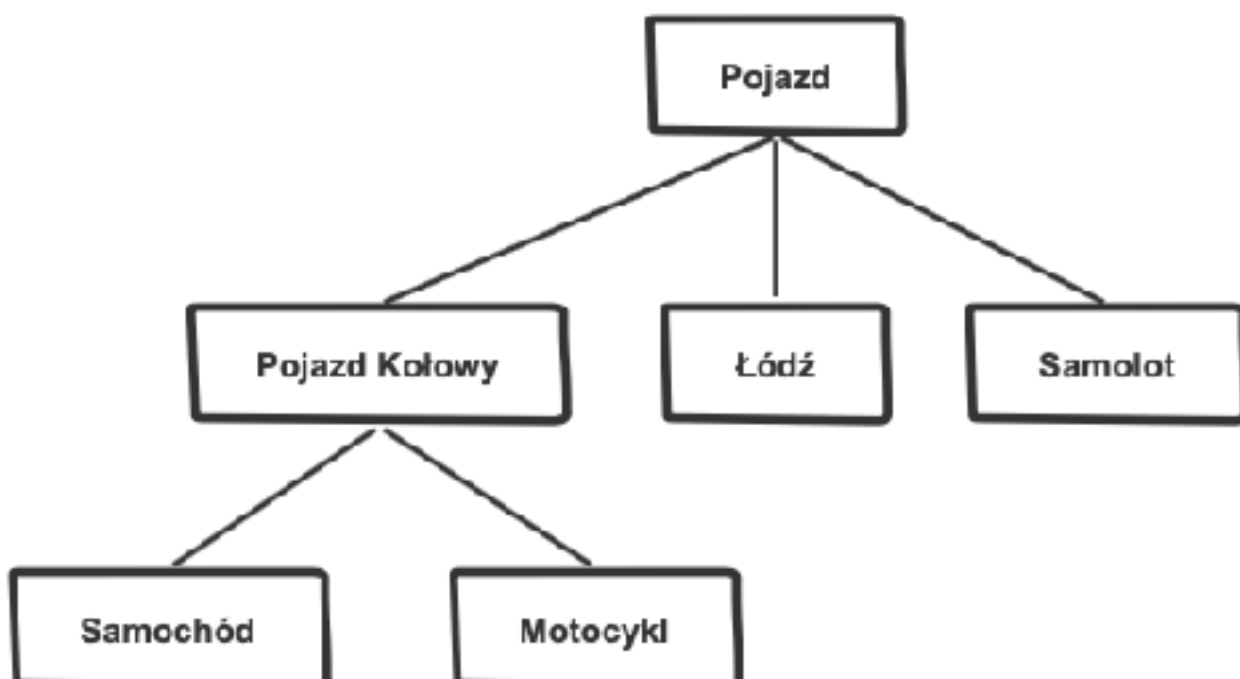
programie. Możemy wskazać, że "człowiek jest ssakiem", "karp jest rybą", a "ssak i ryba są kręgowcem". Część cech i funkcjonalności w programie może być w takiej sytuacji uniwersalna, ale w razie potrzeby każdy może mieć również swoje dodatkowe elementy charakterystyczne. Możliwość przejmowania pewnych cech i funkcjonalności z innych typów (klas) nazywać będziemy w programowaniu obiektowym **dziedziczeniem**.

Przykład 1 (dziedziczenie pól)

Dobrym przykładem do przedstawienia idei dziedziczenia będą pojazdy - jak widać na Wikipedii, mają one dosyć klarowną hierarchię kategorii i powiedzmy, że w naszym programie będziemy potrzebowali informacje o:

- samochodach,
- motocyklach,
- łodziach,
- samolotach.

W ogólności możemy więc narysować prostą strukturę do sklasyfikowania powyższych typów pojazdów:



Normalnie, aby odwzorować każdy z tych typów, musielibyśmy stworzyć dla każdego z nich nową klasę i pomimo, że każdy z nich posiada cechy wspólne jak silnik (np. reprezentowany przez klasę *Engine*), czy ilość paliwa (reprezentowaną przez liczbę typu *double*), to muszą one być przepisywane w identycznej formie w każdej z nich:

Car.java (Samochód)

```
class Car {  
    public Engine engine;  
    public double fuel;  
}
```

Motorbike.java (Motocykl)

```
class Motorbike {  
    public Engine engine;  
    public double fuel;  
}
```

Boat.java (Łódź)

```
class Boat {  
    public Engine engine;  
    public double fuel;  
}
```

itd.

na szczęście w Javie pojawił się mechanizm dziedziczenia, który pozwala oddać powyższą strukturę i elementy współdzielone nie muszą być powtarzane w każdej klasie.

Wystarczy wskazać, że klasa posiada swoją **klasę nadrzędną / bazową**, a tym samym przejmuje jej wszystkie widoczne cechy. Można to zrobić wykorzystując słowo **extends** (co oznacza „rozszerzać”):

```
class B extends A{  
}
```

W powyższym kodzie:

Klasa B jest **klasą podrzędną lub pochodną** innej klasy A, czyli przejmuje wszystkie jej widoczne cechy i funkcjonalności.

Klasa A jest **klasą nadrzędną lub bazową** klasy B ale nie ma pojęcia o istnieniu klasy B.

Wracając do naszego przykładu kod kolejnych klas mógłby wyglądać następująco:

Vehicle.java (Pojazd)

```
class Vehicle {  
    public Engine engine;  
    public double fuel;  
}
```

WheeledVehicle.java (Pojazd kołowy)

```
class WheeledVehicle extends Vehicle {  
    // klasa dziedziczy pola engine i fuel z klasy Vehicle  
    public int wheels; // ilość kół  
}
```

Widzimy, że oprócz pól przejętych z klasy nadrzędnej możemy bez problemu dodać kolejne pole. W tym przypadku określamy ile kół ma pojazd.

Car.java (Samochód)

```
class Car extends WheeledVehicle{  
    //klasa dziedziczy pola engine, fuel oraz wheels  
    public String type;  
}
```

Samochód dziedziczy po pojeździe kołowym i dodatkowo ma informacje o typie samochodu (coupe, sedan itp.)

Boat.java (Łódź)

```
class Boat extends Vehicle {  
    //pola odziedziczone z klasy Vehicle  
}
```

klasa *Motorbike* (motocykl) będzie wyglądała analogicznie do klasy *Car*, a klasa *Plane* (samolot) analogicznie do *Boat*. Ważne jest to, że w klasie podrzędnej mamy dostęp wyłącznie do składowych publicznych, a także w przypadku klas znajdujących się w tym samym pakiecie do składowych o zasięgu pakietowym i domyślnym. Dostęp do składowych prywatnych klasy nadrzędnej możemy uzyskać jedynie poprzez nieprywatne metody dostępowe (np.

settery i gettery). Jak to możliwe, że dostęp do pól prywatnych uzyskujemy przez metody rozpoczynające się od set, czy get? Otóż w przypadku Javy słowo dziedziczenie nie do końca pasuje do tego co właściwie się dzieje. Obiekt klasy podrzędnej posiada wszystkie pola własne oraz te z klasy nadrzędnej, nawet te prywatne, ale ponieważ private oznacza "prywatne w klasie", to w klasie dziedziczącej są one niedostępne. Jeśli dodamy dla nich publiczne gettery. settery, to problem ten omijamy.

Przykład 2 (dziedziczenie metod i konstruktorów)

W Javie **konstruktory nie są dziedziczone**, jednak jak się dowiesz w późniejszej lekcji, możliwe jest wywołanie konstruktora klasy nadrzędnej z konstruktora klasy pochodnej.

Nie ma jednak problemu z dziedziczeniem metod. Wracając do przykładu 1, jeśli istniałaby publiczna metoda startEngine() w klasie Vehicle, to również wszystkie klasy podrzędne mogłyby z niej korzystać.

Napiszmy inny przykład dziedziczenia, w którym powrócimy do podziału zwierząt. Powiedzmy, że istnieje klasa Animal, a po niej dziedziczą dwa inne typy - Cat (kot) i Bird (ptak). Każde zwierzę ma określony kolor i może wydawać dźwięk.

Animal.java

```
class Animal {
    private String color;

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public void makeSound() {
        System.out.println("Burp Burp");
    }
}
```

Klasa `Animal` posiada pole opisujące kolor, a także jedną metodę `makeSound()`, która określa, kiedy zwierzę wyda dźwięk (powiedzmy, że Burp Burp to ogólny dźwięk, który jest w stanie wydać każde zwierzę).

Moglibyśmy zdefiniować klasy `Cat` i `Bird` w następujący sposób:

Cat.java

```
class Cat extends Animal {  
  
}
```

Bird.java

```
class Bird extends Animal{  
  
}
```

Teraz możemy utworzyć obiekty `Cat` i `Bird` w innej klasie, przypisać do nich kolor, oraz zmusić do wydania dźwięku:

Zoo.java

```
class Zoo {  
    public static void main(String[] args) {  
        Cat cat = new Cat();  
        cat.setColor("Czarny");  
        Bird bird = new Bird();  
        bird.setColor("Niebieski");  
        System.out.println("Zwierzęta dają głos: ");  
        System.out.print("Kot: ");  
        cat.makeSound();  
  
        System.out.print("Ptak: ");  
        bird.makeSound();  
    }  
}
```



Wszystko byłoby super, tylko czy ktokolwiek z was widział kota lub ptaka, który robi "burp burp"? No właśnie ja też nie.

W Javie istnieje jednak możliwość **nadpisania** (eng. **override**) metody. Często spotkacie się z nazywaniem tej czynności także jako **przesłonięcie** metody, co nie do końca jest poprawne, ale na tyle powszechne, że już każdy się do tego przyzwyczaił i terminy te można traktować zamiennie. Nadpisanie oznacza, że w klasie pochodnej definiujemy metodę o identycznej sygnaturze co metoda w klasie bazowej, czyli metodę z takim samym typem zwracanym, nazwą i parametrami, ale innym zachowaniem. Poprawione klasy Cat i Bird wyglądałyby więc następująco:

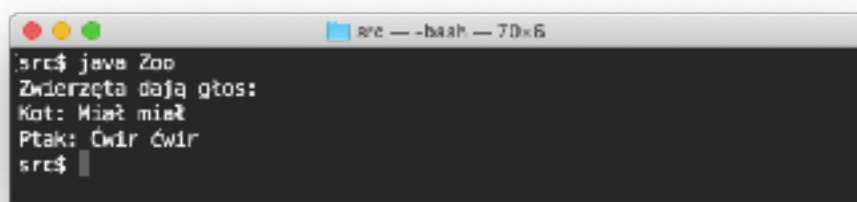
Cat.java

```
class Cat extends Animal {
    public void makeSound() {
        System.out.println("Miał miał");
    }
}
```

Bird.java

```
class Bird extends Animal {
    public void makeSound() {
        System.out.println("Ćwir ćwir");
    }
}
```

Teraz po wywołaniu tego samego kodu klasy Zoo co poprzednio widzimy już wynik, który jest bardziej sensowny:



```
src$ java Zoo
Zwierzęta dają głos:
Kot: Miał miał
Ptak: Ćwir ćwir
src$
```

W zależności od tego, czy wywołamy metodę `makeSound()` na obiekcie typu `Cat`, czy typu `Bird`, to wywoływana jest metoda z jednej z tych dwóch klas, a nie z klasy `Animal`.

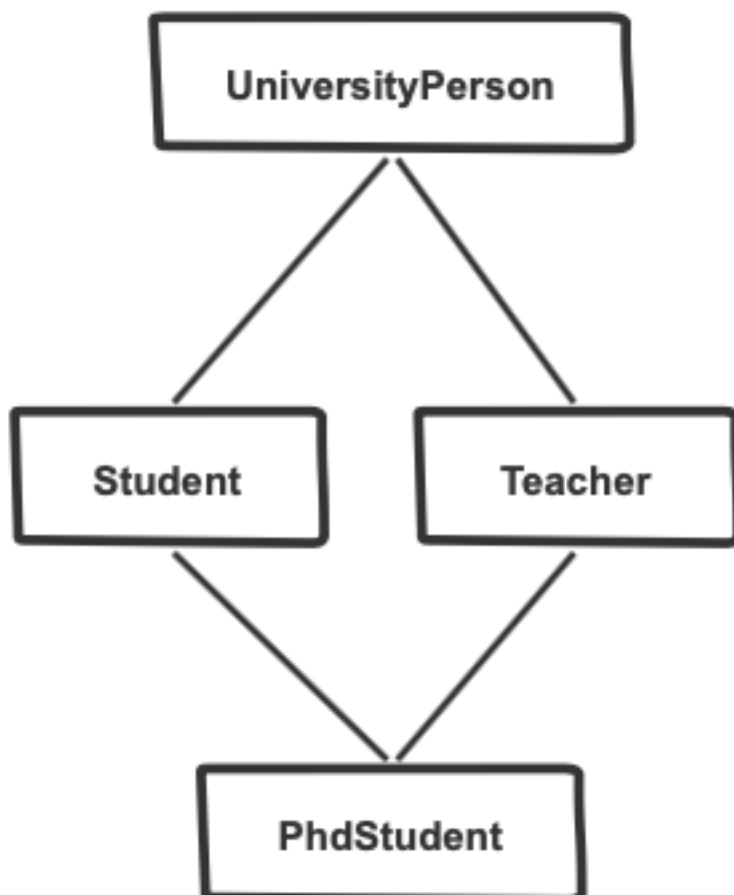
Przykład 3 (wielodziedziczenie)

A co zrobić w sytuacji, gdy chcielibyśmy klasę dziedziczącą z dwóch innych klas? Przykładowo możemy sobie to wyobrazić klasy:

UniversityPerson reprezentującą dowolną osobę na uczelni,

Student i *Teacher* - reprezentujące studenta oraz doktora lub profesora, który uczy studentów. Obie klasy dziedziczą cechy z *UniversityPerson*,

PhdStudent (doktorant) - łączy cechy klasy *Student* (bo wciąż się uczy) oraz *Teacher* (bo prowadzi pewne zajęcia)



Mogłoby się wydawać, że wystarczy w tej sytuacji zapisać:

```
class PhdStudent extends Student, Teacher {  
  
}
```

Jednak zrobić tak nie możemy. **W Javie nie istnieje wielodziedziczenie klas.** Jest to spowodowane kilkoma problematycznymi kwestiami, jednak najbardziej znany jest tzw. problem diamentu. Prowadzi on do konfliktu wyboru odpowiedniej metody z klas nadrzędnych, a jego nazwa wywodzi się od wyglądu powyższego diagramu, który nieco taki diament przypomina. Jeśli klasy *Student* i *Teacher* posiadałyby metodę o identycznej sygnaturze, ale wykonywałyby różne operacje, np:

Student.java

```
class Student extends UniversityPerson {  
    public void printInfo() {  
        //wyświetla informacje o studencie  
    }  
}
```

Teacher.java

```
class Teacher extends UniversityPerson {  
    public void printInfo() {  
        //wyświetla informacje o nauczycielu  
    }  
}
```

to w tej sytuacji nie byłoby wiadomo, którą metodę `printInfo()` powinna odziedziczyć klasa *PhdStudent*.

Dziedziczenie a specyfikator domyślny i `protected`

W temacie o specyfikatorach dostępu wspomnieliśmy o subtelnej różnicy zachodzącej pomiędzy specyfikatorem domyślnym (czyli jego braku) oraz *protected*. Różnica polega na tym, kiedy odpowiednie składowe klasy są dziedziczone, a kiedy nie. Otóż w przypadku, gdy klasa *Parent* i klasa *Child* znajdowałyby się w dwóch różnych

pakietach, przy czym *Child* dziedziczy po *Parent*, to odziedziczone zostaną pola, czy metody ze specyfikatorem *protected*, natomiast do składowych bez żadnego oznaczenia nie będziemy mieli dostępu.



W powyższej sytuacji zachodzi sytuacja jaką opisano komentarzami.

Parent.java

```
package first;

public class Parent {
    protected String name;
    int value;
}
```

Child.java

```
package second;
import first.Parent;

public class Child extends Parent {

    public Child() {
        name = "Hello"; // wszystko ok, name ma specyfikator
protected
        value = 5; // błąd pole value nie jest dziedziczone w
klasie Child
    }

}
```

W praktyce raczej nigdy nie spotkasz się z powyższym przypadkiem.

Klasa Object

Nawet jeśli tego nie zapiszesz, to w Javie każda klasa dziedziczy po klasie Object. Poświęćmy jej więcej czasu niebawem, na razie musisz wiedzieć, że zapisując dowolną klasę, np.:

```
public class User {  
  
    //...  
}
```

w rzeczywistości należy ją sobie wyobrazić w ten sposób:

```
public class User extends Object{  
  
    //...  
}
```

Dziedziczenie a klasy finalne

W rzadkich sytuacjach może się zdarzyć, że będziesz chciał stworzyć klasę, której nikt inny nie powinien rozszerzać.

Jest to raczej niezbyt intuicyjne zachowanie, biorąc pod uwagę, że przecież jedną z głównych zalet programowania obiektowego jest właśnie dziedziczenie.

Jeżeli jednak koniecznie będziesz chciał to zrobić, oznacz klasę słowem kluczowym **final**. Dzięki temu nikt nie będzie mógł rozszerzyć Twojej klasy i przesłonić domyślnej implementacji metod. W bibliotece standardowej Javy można znaleźć przykłady takich klas - jedną z nich, z którą masz już styczność niemal od samego początku swojej przygody z programowaniem jest klasa String.

Przy pomocy słowa final możesz oznaczyć także metody. W takiej sytuacji w klasach pochodnych nie będzie można ich nadpisać. Stosowanie takiej konwencji jest jednak bardzo rzadko spotykane.