

Klasy wewnętrzne i zagnieżdżone

Czego się dowiesz

- Czym są klasy wewnętrzne,
- jak działa słowo `this` w stosunku do klas lokalnych,
- czym są klasy zagnieżdżone.

Klasy wewnętrzne

Do tej pory wszystkie nowe klasy, czy interfejsy w naszych aplikacjach umieszczaliśmy w osobnych plikach `.java`, które miały nazwę zgodną z nazwą danej klasy, czy interfejsu. W Javie możliwe jest jednak definiowanie różnego rodzaju **klas wewnętrznych** (eng. *inner class*), które będą szczególnie przydatne w przypadku tworzenia struktur danych, o których dowiesz się więcej już niebawem.

Klasa wewnętrzna w odróżnieniu od "zwykłej" może być prywatna i niewidoczna dla kogoś korzystającego z klasy zewnętrznej. Dodatkowo może być oznaczana słowami kluczowymi, które już również znasz - `final`, `abstract`, `public`, `protected`.

Klasy wewnętrzne wykorzystywane są najczęściej wtedy, gdy chcemy w niej mieć dostęp do klasy otaczającej bez konieczności tworzenia i przekazywania jej instancji lub gdy chcemy ją wykorzystać tylko w celach pomocniczych i nie ma sensu jej wydzielać tak, aby inni również mogli z niej korzystać.

Dla przykładu - każdy samochód posiada silnik, więc w teorii powinniśmy stworzyć dwie klasy - `Car` i `Engine`. Po chwili zastanowienia można jednak dojść do wniosku, że przecież silnik jest integralnym elementem samochodu, więc można z niego zrobić wewnętrzną klasę, która będzie miała dzięki temu bezpośredni dostęp do pola oznaczającego ilość paliwa w baku.

```
class Car {
    private Engine engine;
    private int fuel;

    public Car() {
        engine = new Engine("Ferrari");
        System.out.println("Utworzono samochód z silnikiem " + engine.engineType);
    }

    public void go() throws InterruptedException {
        while (fuel > 0) {
            engine.consumeFuel();
            System.out.println("Pozostało " + fuel + " litrów paliwa");
            Thread.sleep(1000);
        }
        System.out.println("Brak paliwa");
    }

    public void refuel(int liters) {
        fuel = fuel + liters;
    }

    public class Engine {

        private String engineType;
        private static final int FUEL_CONSUMPTION = 20;

        public Engine(String type) {
            engineType = type;
        }

        public void consumeFuel() {
            fuel = fuel - FUEL_CONSUMPTION;
        }
    }
}
```

Zacznijmy tym razem od dołu. W klasie Car zdefiniowaliśmy **wewnętrzną publiczną klasę** Engine. Publiczna oznacza w tym momencie, że nie wykluczamy możliwości utworzenia jej obiektów poza klasą Car. W niej zdefiniowano pola prywatne engineType (typ silnika) i stałą FUEL_CONSUMPTION (spalanie na 100km), do której przypisujemy wartość 20. W klasie dostępny jest jeden konstruktor i metoda consumeFuel(), w której zmniejszamy stan paliwa, czyli zmienną prywatną klasy Car. Jedną z zalet stworzenia klasy Engine wewnątrz klasy Car jest to, że mamy dostęp do wszystkich pól klasy opakowującej, nawet jeśli nie udostępnia ona gettera.

Działa to także w drugą stronę, z klasy Car możemy odwoływać się do pól prywatnych klasy Engine. W konstruktorze

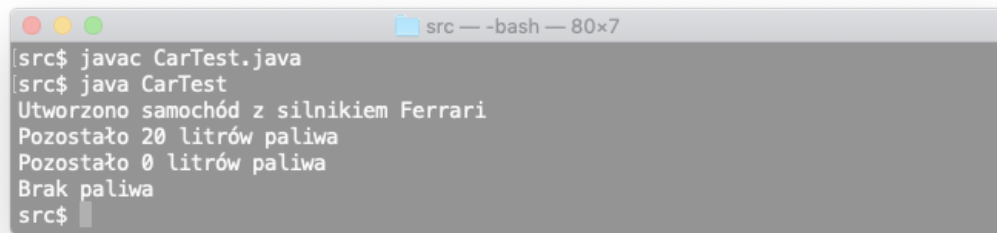
Car(), w którym utworzyliśmy nowy silnik, wyświetlamy komunikat o utworzeniu nowego samochodu i odwołujemy się do prywatnego pola engineType z klasy Engine.

W klasie Car znajdują się też dwie metody - go() i refuel(). Pierwsza symuluje poruszanie się samochodu i spalanie paliwa, a druga umożliwia zatankowanie samochodu. W metodzie go() wykorzystujemy pętlę, która sprawia, że silnik spala paliwo tak długo, dopóki jest ono jeszcze w baku. Dzięki metodzie statycznej **Thread.sleep(1000)** - wstrzymujemy wątek (czyli w naszej sytuacji cały program) na 1 sekundę. Jako argument tej metody podajemy czas, na jaki chcemy wstrzymać działania aplikacji w milisekundach, stąd liczba 1000. Jeżeli pętla skończy działanie, to wyświetlamy komunikat o braku paliwa. Ponieważ metoda sleep() może wygenerować wyjątek kontrolowany InterruptedException, a my nie możemy z tym faktem raczej nic sensownego w tym miejscu zrobić, deklarujemy w sygnaturze metody ten fakt dopisując throws InterruptedException.

Sprawdźmy działanie naszej klasy w prostej klasie testowej: *CarTest.java*

```
class CarTest {  
    public static void main(String[] args) throws InterruptedException {  
        Car car = new Car();  
        car.refuel(40);  
        car.go();  
    }  
}
```

Utworzyliśmy samochód, zatankowaliśmy go 40 litrami paliwa i uruchomiliśmy za pomocą metody go(). Na ekranie w odstępie 1 sekundy będzie wyświetlany komunikat o ubywającym paliwie:



```
src$ javac CarTest.java
src$ java CarTest
Utworzono samochód z silnikiem Ferrari
Pozostało 20 litrów paliwa
Pozostało 0 litrów paliwa
Brak paliwa
src$
```

Jeżeli z jakiegoś powodu chcielibyśmy stworzyć instancję klasy Engine poza klasą Car, to możemy to zrobić w następujący sposób:

```
Car.Engine engine = new Car().new Engine("Ferrari");
```

Całe szczęście nie jest to zbyt częsty widok i jest to raczej mało spotykana konstrukcja. Dodatkowo warto w tym miejscu dodać, że jeśli klasa wewnętrzna zostanie oznaczona jako prywatna, to dostęp do niej z zewnątrz będzie niemożliwy.

W przypadku klas wewnętrznych zmienia się także nieco działanie słowa `this`, które jak może pamiętasz służyło nam do odwoływania się do przesłoniętych przez argumenty metod, czy konstruktorów pól klasy. Jeżeli w klasie wewnętrznej użyjemy słowa `this` to oznacza ono odwołanie się do pól klasy wewnętrznej. Żeby odwołać się do pola klasy otaczającej należy zastosować zapis `KlasaZewnetrzna.this`. Przykładowo:

```
class Car {
    private int fuel;

    private class Engine {
        int fuel;

        void consumeFuel() {
            // pole klasy wewnętrznej - Engine
            this.fuel = 5;

            // pole klasy otaczającej - Car
            Car.this.fuel = 5;
        }
    }
}
```

Klasy zagnieżdżone

Klasy wewnętrzne w odróżnieniu od "zwykłych" mogą być również statyczne, nazywać będziemy je wtedy klasami zagnieżdżonymi (*eng. nested class*). Od standardowych klas wewnętrznych odróżnia je to, że mają one dostęp jedynie do statycznych pól klasy otaczającej. Dodatkowo jeśli klasa posiada element statyczny to sama musi zostać oznaczona jako statyczna.

Jest to logiczne, ponieważ, skoro klasa jest statyczna, to elementy instancyjne klasy otaczającej wcale nie muszą istnieć.

Inny jest także sposób inicjalizowania obiektu klasy zagnieżdżonej, gdyż nie wymaga on tworzenia instancji klasy opakowującej:

```
class Car {  
  
    public static class Engine {  
        int fuel;  
  
        void consumeFuel() {  
            this.fuel = 5;  
        }  
    }  
}
```

Obiekt powyższej klasy zagnieżdżonej utworzymy w następujący sposób:

```
Car.Engine eng = new Car.Engine();
```

Nie jest konieczne tworzenie obiektu klasy opakowującej, czyli wywołanie konstruktora klasy Car.

Podsumowanie

Klasy wewnętrzne i zagnieżdżone muszą posiadać swoją nazwę.

Klasy wewnętrzne mają dostęp zarówno do statycznych jak i niestatycznych elementów klasy opakowującej, jednak klasy zagnieżdżone mają dostęp tylko do elementów statycznych klasy opakowującej.

Klasy wewnętrzne i zagnieżdżone mogą być prywatne w odróżnieniu od "zwykłych" klas.

Jeśli klasa wewnętrzna posiada co najmniej jedną składową statyczną to sama również musi być statyczna (czyli musi być klasą zagnieżdżoną). Nie dotyczy to stałych (pól oznaczonych jako static final).

Klasa wewnętrzna może odwoływać się do wszystkich składowych klasy opakowującej, nawet jeśli te są oznaczone jako prywatne.

W praktyce klasy zagnieżdżone są raczej niespotykane.

Klasy wewnętrzne wykorzystywane są częściej, jednak w wielu sytuacjach są to tylko składowe pomocnicze podnoszące czytelność kodu lub wykorzystywane w strukturach danych, gdzie pomagają zbudować

odpowiednie zależności między danymi. Oznacza to tyle, że nawet jeśli klasy wewnętrzne gdzieś są wykorzystywane, to jako użytkownicy takich klas niekoniecznie będziemy o tym wiedzieli.