

Klasy abstrakcyjne

Czego się dowiesz

Dziedziczenie potrafi łączyć w sobie wiele zalet, ale także ma pewne wady. Jeżeli budujemy pewną hierarchię klas np. w celu oszczędzenia powtarzalności kodu, czy w celu umożliwienia wykorzystania polimorfizmu, to jednak mogą zdarzać się sytuacje, w których nasze klasy będą typami na tyle abstrakcyjnymi, że raczej nie chcielibyśmy, aby w ogóle można było tworzyć obiekty tego konkretnego typu.

Przykładowo mając klasę `Vehicle` (pojazd), po której dziedziczy `Car` (samochód) i `Plane` (samolot), to raczej możemy się spodziewać, że będziemy tworzyli obiekty w poniższy sposób:

```
Vehicle car = new Car();  
Vehicle plane = new Plane();  
car.speedUp();  
plane.speedUp();
```

przyspieszanie, czy zwalnianie samochodu i samolotu jest diametralnie różne, więc metoda `speedUp()` musi być nadpisana w każdej z klas, które dziedziczą po `Vehicle`.

Dodatkowo spójrzmy na taki fragment kodu:

```
Vehicle vehicle = new Vehicle();
```

Co reprezentuje w tym przypadku obiekt `vehicle`? Możemy powiedzieć, że pojazd, ale jaki? Łódź, samochód, czy może samolot? No właśnie nie wiadomo i w takiej sytuacji warto się zastanowić czy w ogóle w naszym programie chcemy, aby ktokolwiek zapisywał kiedykolwiek `new Vehicle()`.

Klasy abstrakcyjne

W Javie istnieje pojęcie **klas abstrakcyjnych**. Są to zwykłe klasy, oznaczone dodatkowo słowem kluczowym **abstract**.

Klasy takie mają jednak dwa ograniczenia / możliwości:

Nie można tworzyć instancji klas abstrakcyjnych.

Przykładowo, jeśli `Vehicle` z naszego przykładu będzie klasą abstrakcyjną to nie będzie można zapisać `new Vehicle()`.

Klasy abstrakcyjne mogą zawierać metody abstrakcyjne, czyli takie, które są oznaczone słowem `abstract` i definiują jedynie sygnaturę (to co metoda ma robić, czyli jaki jest typ zwracany, nazwa i parametry), ale nie mają ciała pomiędzy nawiasami klamrowymi.

W praktyce wygląda to tak:

Vehicle.java

```
abstract class Vehicle {
    private int speed;

    public int getSpeed() {
        return speed;
    }

    public void setSpeed(int speed) {
        this.speed = speed;
    }

    // metoda abstrakcyjna
    public abstract void speedUp();
}
```

W klasie `Vehicle` znajduje się pole `speed` typu `int`, które posiada swój getter i setter. Oprócz tego istnieje metoda `speedUp()`, którą oznaczyliśmy jako abstrakcyjną, ponieważ chcemy, aby każdy kto będzie dziedziczył po klasie `Vehicle` zaimplementował ją na swój sposób. Metoda jest abstrakcyjna, ponieważ nie jesteśmy w stanie powiedzieć "ogólnie każdy pojazd na świecie przyspiesza o XYZ km/h". Zupełnie inaczej przyspiesza rower, samochód, czy samolot.

Zauważ, że jeśli metoda jest abstrakcyjna, to nie posiada implementacji, co więcej nie posiada nawet nawiasów klamrowych, tylko od razu zakończona jest średnikiem. Dodatkowo ponieważ klasa `Vehicle` posiada co najmniej jedną metodę abstrakcyjną, to sama klasa również musiała zostać w taki sposób oznaczona i od tego momentu nikt w programie nie może zapisać `new Vehicle()`, co ogólnie jest dobre, bo jak wcześniej stwierdziliśmy, nie wiadomo jaki pojazd taki obiekt reprezentuje.

Jeżeli zdefiniujemy dowolną klasę dziedziczącą po `Vehicle`:

Car.java

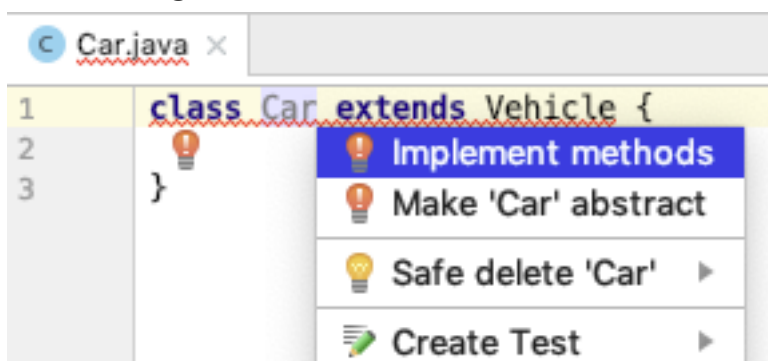
```
class Car extends Vehicle { }
```

To dziedziczymy metodę `speedUp()`. Ponieważ jest ona abstrakcyjna to mamy dwa wyjścia.

Oznaczamy klasę `Car` jako abstrakcyjną, ale przez to nie będziemy mogli tworzyć obiektów przez `new Car()`, więc nie wydaje się to dobrym pomysłem.

Implementujemy metodę `speedUp()`, czyli nadpisujemy metodę z klasy `Vehicle` i mówimy w jaki sposób konkretnie ma ona działać dla samochodów.

Jeśli korzystasz ze środowiska programistycznego, to klasa `Car` będzie w powyższej formie podkreślona na czerwono. Możesz na nią kliknąć i wybrać jedno z dwóch powyższych rozwiązań. W tym celu w eclipse wciśnij `Ctrl+1`, a w IntelliJ `Alt+Enter`.



Po zaimplementowaniu metody możemy w klasie `Car` zdefiniować, że np. w przypadku samochodu jednokrotne wywołanie metody `speedUp()` powoduje przyspieszenie o 5km/h.

Car.java

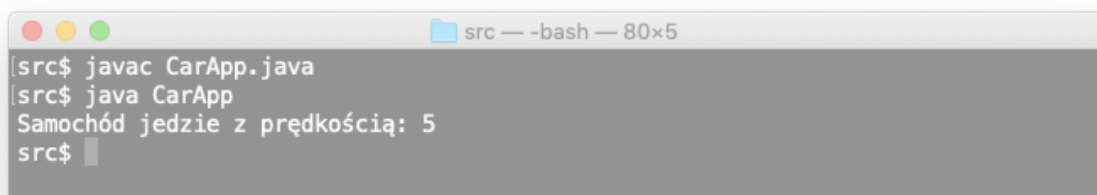
```
class Car extends Vehicle {
    @Override
    public void speedUp() {
        setSpeed(getSpeed() + 5);
    }
}
```

Stwórzmy jeszcze klasę testową, aby się przekonać, że wszystko działa.

CarApp.java

```
class CarApp {
    public static void main(String[] args) {
        // błąd nie można tworzyć obiektów typów abstrakcyjnych
        // Vehicle vehicle = new Vehicle();

        Vehicle vehicle = new Car(); //polimorfizm
        vehicle.speedUp();
        System.out.println("Samochód jedzie z prędkością: " +
vehicle.getSpeed());
    }
}
```



```
src — -bash — 80x5
[src$ javac CarApp.java
[src$ java CarApp
Samochód jedzie z prędkością: 5
src$
```

Interfejsy

W Javie istnieje także pojęcie interfejsów. Można o nich powiedzieć, że są klasami w pełni abstrakcyjnymi - mogą posiadać jedynie publiczne metody abstrakcyjne oraz stałe. W Javie 8 wprowadzono jednak pewne usprawnienie nazwane **metodami domyślnymi** (eng. *default methods*), które omówimy w dalszej części lekcji.

Interfejsów rozpoczynamy słowem **interface**, a nie **class** jak przy klasach. W odróżnieniu od klas abstrakcyjnych nie musimy używać słowa **abstract** ani do metod, ani do samej nazwy interfejsu, ponieważ wszystkie te elementy są domyślnie abstrakcyjne. Co więcej wszystkie metody są domyślnie publiczne, a jeżeli wykorzystujemy pola, to są one domyślnie publicznymi stałymi, czyli **public static final**. Przykładem może być sytuacja, gdy chcemy stworzyć program potrafiący wyliczyć pole i obwód figur geometrycznych. Stworzymy interfejs **Shape** zawierający dwie metody **calculatePerimeter()** (oblicz obwód) oraz **calculateArea()** (oblicz pole). Dla wygody dodajmy często używaną wartość **PI** - korzysta z niej kilka kształtów. Interfejsy podobnie jak klasy umieszczamy w osobnych plikach o rozszerzeniu **.java** - w eclipse możesz je utworzyć korzystając z menu **New** podobnie jak przy klasach, czy pakietach, a w IntelliJ należy skorzystać z tego samego kreatora co dla klas, wybierając odpowiednią opcję przy wpisywaniu nazwy.

Shape.java

```
interface Shape {  
    public static final double PI = 3.14;  
  
    // metody są domyślnie publiczne i abstrakcyjne  
    double calculateArea();  
  
    public double calculatePerimeter();  
}
```

W kodzie pokazano, że można użyć słowa `public` lub też nie - w praktyce warto stosować jedną spójną konwencję. Obie metody w interfejsie są abstrakcyjne, pomimo że jawnie nie zapisaliśmy słowa `abstract`.

W przypadku interfejsów mówimy o ich implementacji przez klasy, a nie o dziedziczeniu. Fakt ten wskazujemy korzystając ze słowa kluczowego **`implements`**.

```
class Rectangle implements Shape {
    private double a;
    private double b;

    public Rectangle(double a, double b) {
        this.a = a;
        this.b = b;
    }

    public double getA() {
        return a;
    }

    public void setA(double a) {
        this.a = a;
    }

    public double getB() {
        return b;
    }

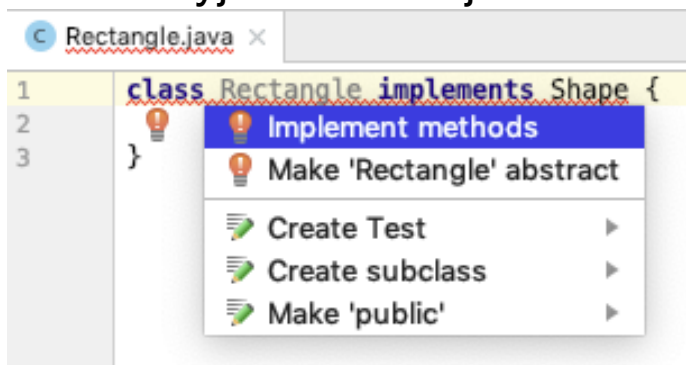
    public void setB(double b) {
        this.b = b;
    }

    @Override
    public double calculateArea() {
        return a*b;
    }

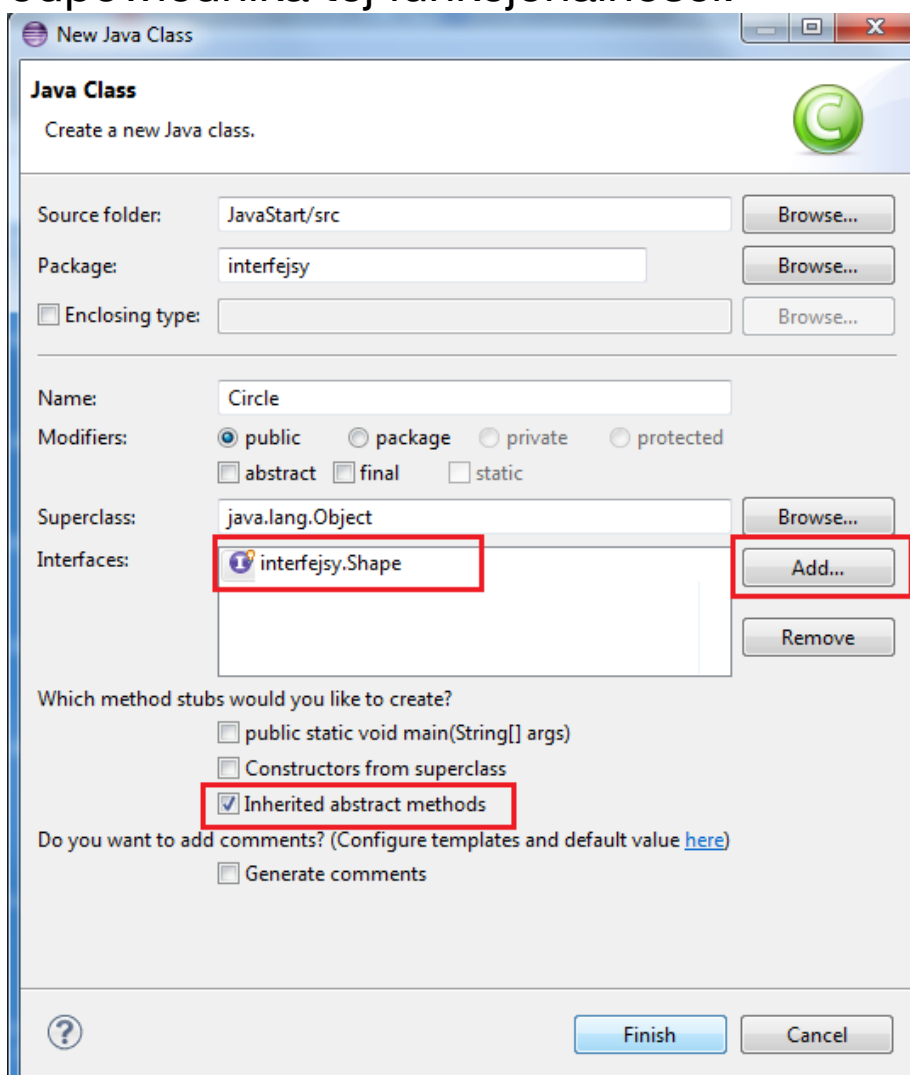
    @Override
    public double calculatePerimeter() {
        return 2*a + 2*b;
    }
}
```

Klasa `Rectangle` (prostokąt) implementuje interfejs `Shape`. Zdefiniowaliśmy w niej pola `a` i `b`, które reprezentują długości boków. Jeżeli zdefiniowalibyśmy klasę `Rectangle` i nie zaimplementowali metod `calculateArea()` oraz `calculatePerimeter()`, to środowisko będzie podkreślało dokładnie taki sam błąd jak przy klasach abstrakcyjnych, informując nas o tym, że albo musimy oznaczyć klasę jako

abstract, albo powinniśmy dodać wszystkie metody abstrakcyjne z interfejsu.



Jeżeli korzystasz z eclipse i tworzysz klasę, która ma implementować interfejs, możesz w oknie definiowania nowego typu wskazać interfejsy, które chcesz implementować. Dzięki temu szkielet niezbędnych metod zostanie od razu wygenerowany. W IntelliJ niestety nie ma odpowiednika tej funkcjonalności.



W ten sposób powstanie szkielet klasy do uzupełnienia. Po dodaniu do klasy Circle informacji o promieniu koła otrzymujemy:

Circle.java

```
class Circle implements Shape {  
    private double r;  
  
    public Circle(double r) {  
        this.r = r;  
    }  
  
    public double getR() {  
        return r;  
    }  
  
    public void setR(double r) {  
        this.r = r;  
    }  
  
    @Override  
    public double calculateArea() {  
        return Shape.PI * r * r;  
    }  
  
    @Override  
    public double calculatePerimeter() {  
        return 2 * Shape.PI * r;  
    }  
}
```

Teraz możemy skorzystać z naszych klas w innych miejscach tak jak robiliśmy to już w poprzednich lekcjach.

```
class ShapeCalculator {  
    public static void main(String[] args) {  
        Shape circle = new Circle(5);  
        Shape rect = new Rectangle(5, 10);  
  
        System.out.println("Pole koła: " + circle.calculateArea());  
        System.out.println("Obwód prostokąta: " +  
rect.calculatePerimeter());  
    }  
}
```

Pamiętaj jednak, że nadal masz dostęp wyłącznie do metod dostępnych w typie referencji, więc jeśli chcesz odwoływać się do metod `getA()`, czy `getR()` z klas `Rectangle` lub `Circle`, to niezbędne będzie wcześniejsze rzutowanie na konkretny typ.

```
((Circle)circle).getR();  
((Rectangle)rect).getA();
```


Jeśli pojawia się u ciebie jednak potrzeba rzutowania, to rozważ zrezygnowanie z deklarowania zmiennej jako typu ogólniejszego (Shape) i użyj po prostu od razu typów Circle lub Rectangle.

Stosowanie interfejsów jako elementu wyjściowego do późniejszych definicji klas jest dobrą praktyką, ponieważ: pozwala na wykorzystywanie zalet polimorfizmu (np. możemy stworzyć tablicę różnych kształtów, a nie samych prostokątów), umożliwia dobrą kontrolę nad "czystością kodu". Jeżeli typy definiujemy od razu jako klasy, często dochodzi do sytuacji, kiedy tworzymy w nich dodatkowe metody, które niekoniecznie się tam powinny znaleźć. Za pomocą interfejsu sami narzucamy sobie "łańcuch" tego, które metody muszą być nadpisane, dodawanie kolejnych metod do klasy będzie na nas wymuszało albo rzutowanie, albo dodanie metody do interfejsu - i wtedy zauważymy, że nie ma to sensu, bo inne klasy go implementujące wcale takiej metody nie potrzebują.

Interfejsy traktuj jako definiowanie kontraktów. W naszym przypadku kontrakt brzmi "każdy kształt musi mieć możliwość obliczenia pola i obwodu". Jeśli tworzysz klasę, która implementuje interfejs Shape, to klasa taka musi spełniać ten kontrakt poprzez implementowanie wszystkich metod abstrakcyjnych interfejsu. Interfejsy są użyteczne jeżeli korzystamy z polimorfizmu, same w sobie nie niosą większej wartości. Na razie raczej rzadko będziesz zauważać sens ich stosowania, ale później przy rozwijaniu dużych aplikacji pozwalają one pisać rozwiązania bardziej podatne na rozbudowę i łatwiejsze w utrzymaniu. Na prostych przykładach jest to niestety ciężko pokazać.

Metody domyślne (Java 8)

Wcześniej napisałem, że interfejsy mogą posiadać tylko metody abstrakcyjne, ale coś - trochę skłamałem. Od Javy w wersji 8 wprowadzono tzw. metody domyślne (*eng. default methods*), które oznaczamy słowem kluczowym **default**. Są to metody, które **nie są abstrakcyjne**, a więc nie muszą być nadpisywane w klasach implementujących dany interfejs, a co najważniejsze mogą wykonywać pewne obliczenia.

Przykładowo mając interfejs `Vehicle` nie wiemy o ile przyspiesza samochód, czy samolot, ale możemy zaimplementować domyślną metodę `speedUp()`, która zwiększy prędkość o jedną jednostkę.

```
public interface Vehicle {  
    default public int speedUp(int speed) {  
        return speed++;  
    }  
}
```

To co ważne, to fakt, że dodając metodę domyślną do interfejsu, nie "psujemy" reszty kodu, który z tego interfejsu korzysta.

Możliwość taką wprowadzono ponieważ w innym przypadku nie dało się dodać do istniejących już w Javie interfejsów nowych metod bez niszczenia kompatybilności wstecznej. Przykładowo jeśli w naszym wcześniejszym przykładzie z kształtami dopisalibyśmy sobie dodatkową metodę w interfejsie `Shape` i metoda ta nie byłaby domyślna, to automatycznie nasz projekt przestanie się kompilować dopóki nie zaimplementujemy tych metod w klasach `Rectangle` i `Circle`.

Metody statyczne (Java 8)

Java 8 wprowadziła jeszcze jedną nowość. Od teraz interfejsy mogą posiadać także metody statyczne.

Zachowują się one dokładnie tak samo jak w przypadku tradycyjnych metod statycznych w klasach i najczęściej będą przydatne w przypadku definiowania prostych metod użytkowych. Metody statyczne w interfejsach są domyślnie publiczne. Przykład:

```
public interface StringUtil {  
    static String revert(String original) {  
        return new  
StringBuilder(original).reverse().toString();  
    }  
}
```

Powyższa metoda odwraca tekst przekazany jako argument.

Interfejsy a dziedziczenie i wielodziedziczenie

W odróżnieniu od klas interfejsy mogą dziedziczyć po wielu innych interfejsach:

```
interface D extends A, B, C {}
```

(gdzie A, B, C to również interfejsy),

a dodatkowo klasy mogą implementować wiele interfejsów:

```
class MotorBike implements Bike, Motorcycle, Vehicle {}
```

W przypadku wielodziedziczenia klas pisałem o jednej ważnej kwestii - problemie diamentu. W przypadku interfejsów w starszej wersji Javy nie było z tym problemu, ponieważ interfejsy posiadały tylko i wyłącznie metody abstrakcyjne, a więc trzeba je było przesłonić, żeby robiły cokolwiek użytecznego.

Problem pojawia się od Javy 8, która za pomocą interfejsów i metod domyślnych daje teoretyczną możliwość wielodziedziczenia. Załóżmy, że mamy interfejsy Car i Boat (łódź) z metodą domyślną o takiej samej sygnaturze:

Car.java

```
interface Car {  
    default public void printName() {  
        System.out.print("Car");  
    }  
}
```

Boat.java

```
interface Boat {  
    default public void printName() {  
        System.out.print("Boat");  
    }  
}
```

Teraz tworząc klasę implementującą dwa powyższe interfejsy doprowadzamy do konfliktu, ponieważ kompilator nie wie, którą z metod printName() wybrać do dziedziczenia:

```
public class Amphibia implements Car, Boat {  
  
}
```

✖ Duplicate default methods named printName with the parameters () and () are inherited from the types Boat and Car

Press 'F2' for focus

Problem można jednak rozwiązać dzięki nadpisaniu problematycznej metody:

```
class Amphibia implements Car, Boat {  
    @Override  
    public void printName() {  
        System.out.println("Amphibia");  
    }  
}
```

Ostatnim istotnym elementem, który może być przydatny w kontekście metod domyślnych jest odwołanie się do konkretnej implementacji. Czyli w jaki sposób wywołać metodę printName() z klasy Boat lub Car w klasie Amphibia. Otóż możemy to zrobić przez zapis:
NazwaKlasy.super.nazwaMetody()

W naszym przykładzie:

```
class Amphibia implements Car, Boat {
    @Override
    public void printName() {
        System.out.print("Amfibia to trochę ");
        Car.super.printName();
        System.out.print(" a trochę ");
        Boat.super.printName();
    }
}
```

W wyniku zobaczymy tekst *"Amfibia to trochę Car a trochę Boat"*.

Metody prywatne (Java 9)

W Javie 8 i jej wcześniejszych wersjach wszystkie metody w interfejsach były domyślnie publiczne. Od Javy 9 możliwe jest jednak definiowanie także metod prywatnych co sprawia, że stają się one jeszcze bardziej podobne do klas abstrakcyjnych. Metody prywatne mają sens z powodu wprowadzenia w Javie 8 metod domyślnych. Wyobraź sobie, że jakaś metoda domyślna ma np. 200 linii kodu. Z punktu widzenia programisty Javy oznacza to, że raczej jest to paskudny kod i przydałoby się go podzielić na mniejsze kawałki (metody). Bez metod prywatnych nie dało się jednak tego osiągnąć i stąd ta nowość.

Przykład:

```
interface Example {
    default void complicatedMethod() {
        read(); //np. wczytanie czegoś z pliku
        calculate(); //wykonanie obliczeń
        save(); //zapisanie czegoś w pliku
    }

    private void read() {
        //wczytujemy
    };

    private void calculate() {
        //obliczamy
    };

    private void save() {
        //zapisujemy
    };
}
```