Programowanie w Javie
ćwiczenia 08

Interfejsy

mgr Sara Jurczyk

Wyobraźmy sobie klasy Samochód oraz Rower. Obie klasy reprezentują różne pojazdy, jednak możemy wyodrębnić dla nich część wspólną – oba umożliwiają przemieszczanie się i oba posiadają funkcjonalności takie jak jedź czy zatrzymaj się. Co więcej, właściwość tą posiadają także inne pojazdy oraz na przykład zwierzęta, chociaż każde z nich realizuje poruszanie się na swój własny sposób. Metody związane z poruszaniem się możemy więc wyodrębnić w postaci interfejsu.

Interfejsy są swego rodzaju „kontraktem", który mówi co klasa go implementująca może robić, ale nie podaje jak. Za pomocą interfejsu wskazujemy jakie funkcjonalności będzie posiadać implementująca go klasa. Wszystkie metody w interfejsie są domyślnie publiczne i abstrakcyjne, a co za tym idzie nie podajemy ich definicji w interfejsie, natomiast każda klasa implementująca interfejs musi implementować te metody.

Dodatkowe uwagi: W interfejsie możemy także deklarować pola stałe (domyślnie każde pole będzie publiczne, statyczne i stałe - final). Nie można w nim deklarować metod statycznych. Warto zapamiętać, że dziedziczenie wielokrotne w Javie jest niedozwolone, jednak implementowanie wielu interfejsów jest poprawne.

Zadanie 1.

Utwórz

A. Interfejs **`Moveable`** zawierający tylko dwie metody abstrakcyjne:

```
void start();
void stop();
```

B. Klasę **`Rower`** implementującą interfejs `Moveable`

```
public class Rower implements Moveable{
        @Override
        public void start() {
           System.out.println("Rower rusza");
        }
        @Override
        public void stop() {
           System.out.println("Rower zatrzymał sie");
        }
     }
```

C. Klasę **`Samochod`** z polem **`marka`** implementującą interfejs `Moveable`

D. Interfejs **`Speakable`** {
z dwoma stałymi statycznymi

```
int QUIET = 0;      // <- publiczne stałe statyczne
int LOUD = 1;      // domyślnie public static final
```

i metodą abstrakcyjną

```
String getVoice(int voice);
```

E. Abstrakcyjną klasę **`Zwierze`** z polem

```
private String name = "bez imienia";
```

konstruktorami, metodą zwracającą wartość imienia (*getter*), nadpisaną metodą toString i abstrakcyjna metodą:

```
public abstract String getTyp();
//metoda abstrakcyjna- ogolne zwierze nie wiadomo jaki typ
```

F. Klasę **`Pies`** dziedziczącą po klasie `Zwierze` i implementującą oba interfejsy.

```
public String getTyp() {
    return "Pies";
}
```

Następnie napisz metodę **`wyscig(...)`**, która jako parametry ma dostać dowolną liczbę obiektów `Moveable` (skorzystaj z "varargs"), na których powinna uruchomić metodę **`start()`**.

Zadanie 2.

Napisz interfejs `Language` posiadający metody:

```
String sayGreeting();
String sayGoodbye();
String sayThanks();
```

(Będziemy realizować przywitanie, pożegnanie i podziękowanie w różnych językach).

Następnie utwórz klasy: `English`, `Spanish`, `Japanese`, `Polish` implementujące interfejs `Language`.

Napisz metodę `przywitajSie` która jako parametry ma dostać dowolną liczbę obiektów `Language` (skorzystaj z "varargs"), na których powinna uruchomić metodę `sayGreeting()` i wyświetlić przywitanie.

Zadanie 3.

Napisz program modelujcy istniejce środki transportu używane do przemieszczania się zarówno na lądzie, wodzie, jak i w powietrzu:

- Utwórz interfejs **`Plywa`** oraz **`Lata`** (metody odpowiednio **`plyn()`** i **`lec()`**).
- Zdefiniuj klasy implementujce każdy z interfejsów (np. **`Statek`**, **`Samolot`**), a następnie stwórz przykładowe obiekty wraz z wywołaniem zaimplementowanych metod.

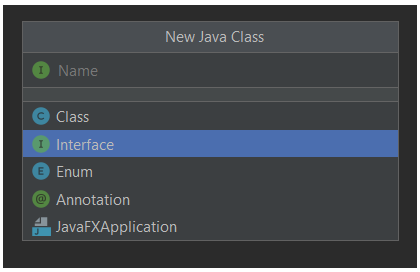
- Hydroplan to środek transportu umożliwiający przemieszczanie się zarówno w powietrzu, jak i na wodzie. Zdefiniuj klasę **`Hydroplan`**, która implementuje obydwa interfejsy. Sprawdź działanie przykładowego hydroplanu.

Rozwiązanie zadania 1 – tutorial:

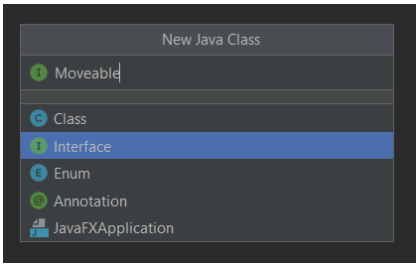
A) Zadeklarujmy interfejs `Moveable` posiadający dwie metody (abstrakcyjne):

```
void start();
void stop();
```

Aby dodać interfejs do projektu, klikamy prawym przyciskiem na pakiet (jeżeli nie zmieniliśmy domyślnej nazwy, będzie to com.company), a następnie wybieramy *New -> Java Class -> Interface*:



I podajemy nazwę naszego interfejsu: `Moveable`



Następnie dodajemy deklaracje metod:

```
void start();
void stop();
```

Domyślnie są to publiczne metody abstrakcyjne. Nie podajemy jak będzie realizowane poruszanie się pojazdów – to będzie uzależnione od tego z jakiego typu poruszającym się obiektem mamy do czynienia. Zaznaczamy jedynie, że każda implementująca ten interfejs klasa, będzie realizowała `start` i `stop`.

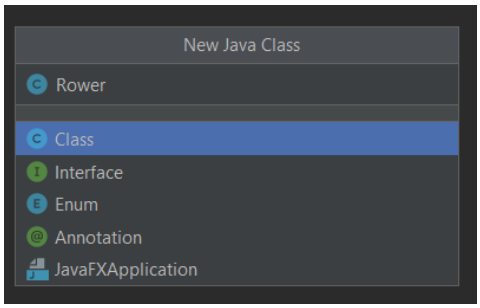
Nasz interfejs powinien wyglądać następujaco:

```
public interface Moveable {
void start();
void stop();
}
```

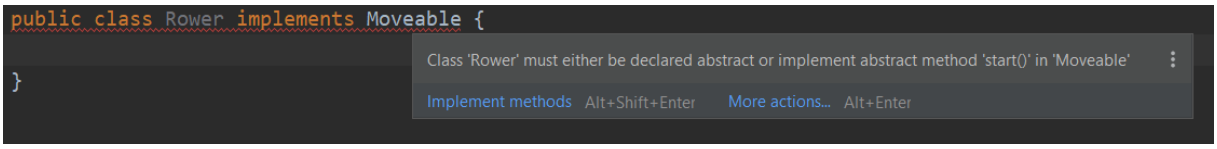
Dodajmy teraz klasy, które będą implementować ten interfejs.

B) Dodajmy do projektu klasę `Rower`, która realizuje poruszanie się – implementuje interfejs `Moveable`.

Sposób 1: Możemy ją dodać w standardowy sposób: wybieramy *New* -> *Java Class* -> *Class*:

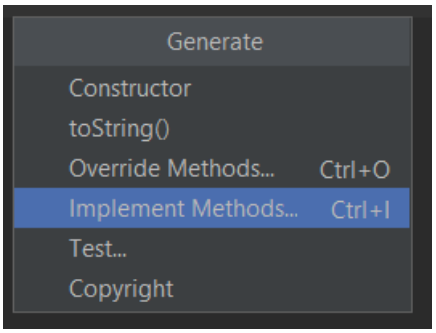


a następnie dopisujemy `implements Moveable.` Warto zwrócić uwagę na komunikat, który pojawia się po utworzeniu klasy:

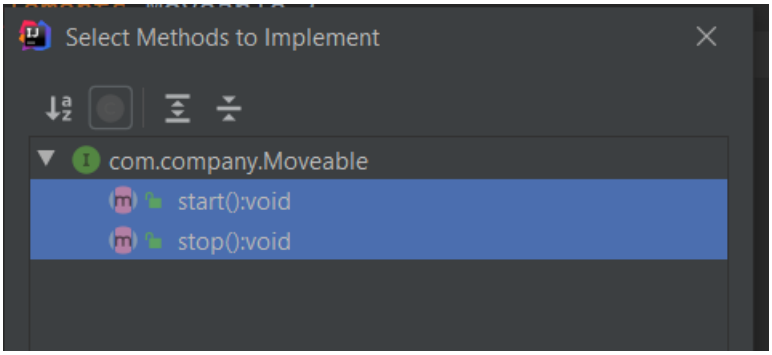


Jasno wskazuje nam on, iż jeżeli klasa Rower chce implementować interfejs `Moveable`, musi implementować metody abstrakcyjne `start, stop.`

Możemy teraz dodać te metody. Wybieramy ALT + Insert, a następnie *Implement Methods*...



i wybieramy obie metody



Do wygenerowanego kodu:

```
public class Rower implements Moveable {

    @Override
    public void start() {

    }

    @Override
    public void stop() {

    }
}
```

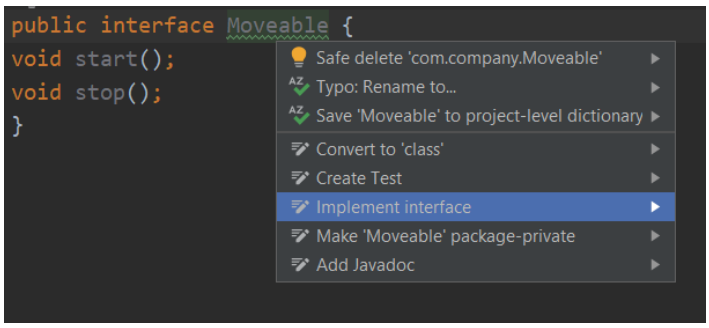
dodamy teraz odpowiednie definicje:

```
public class Rower implements Moveable {

    @Override
    public void start() {
        System.out.println("Rower rusza");
    }

    @Override
    public void stop() {
        System.out.println("Rower zatrzymał się");
    }
}
```

Sposób 2: Klasę `Rower` możemy też utworzyć w alternatywny sposób. Klikamy ALT + Enter po najechaniu na nazwę `Moveable` i wybieramy opcję `Implement interface`

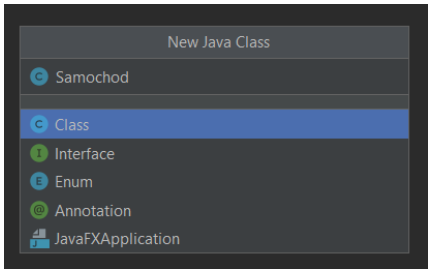


a następnie podajemy nazwę klasy, która implementuje dany interfejs: `Rower`.

Ten sposób automatycznie doda frazę `implements Moveable` oraz pozwala od razu wygenerować metody `start, stop`. Od razu możemy przejść do podania definicji tych metod.

C) Dodajmy teraz do projektu klasę `Samochod` - kolejną klasę, która realizuje poruszanie się, tzn. implementuje interfejs `Moveable` – jednak metody start i stop możemy zdefiniować inaczej, niż w klasie Rower. Klasa ta posiada także dodatkowe pole: `marka`.

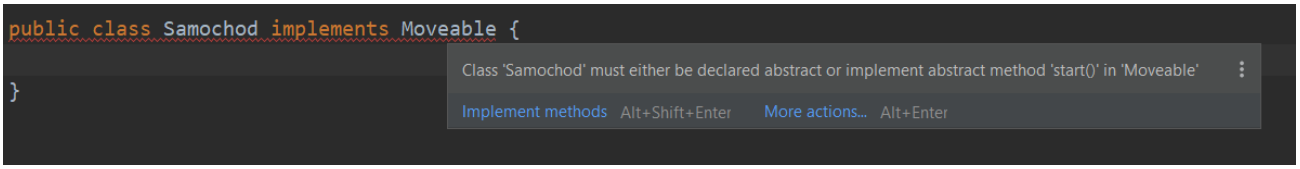
Sposób 1: Klasę możemy ponownie dodać w standardowy sposób: wybieramy *New -> Java Class -> Class*:



a następnie dopisujemy `implements Moveable`

```
public class Samochod implements Moveable
```

Po utworzeniu klasy w ten sposób, pojawia komunikat:



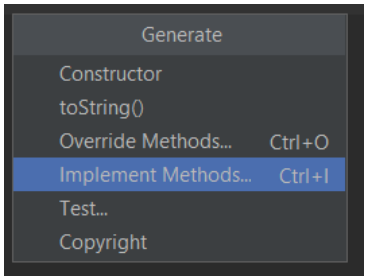
IntelliJ ponownie przypomina nam, że jeżeli klasa Samochód chce implementować interfejs `Moveable`, musi implementować metody abstrakcyjne: `start, stop`.

Możemy zadeklarować teraz pole `marka` i dodać konstruktor:

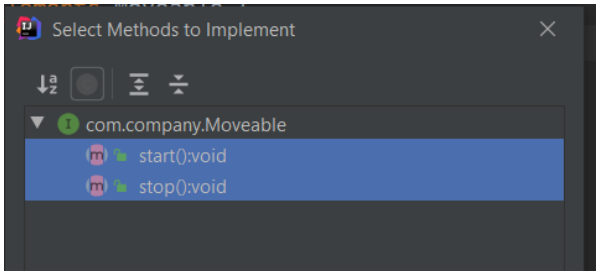
```
public class Samochod implements Moveable {
    private String marka;

    public Samochod(String marka) {
        this.marka = marka;
    }
```

a następnie wygenerować wymagane metody. Wybieramy ALT + Insert, a następnie *Implement Methods…*



i wybieramy obie metody



Uzupełniamy implementacje:

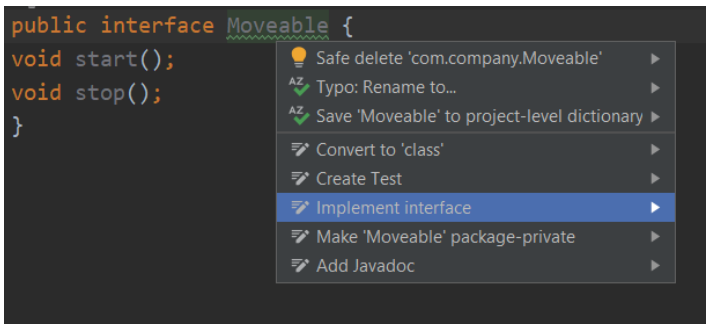
```
public class Samochod implements Moveable {
    private String marka;

    public Samochod(String marka) {
        this.marka = marka;
    }

    @Override
    public void start() {
        System.out.println("Samochód marki " + marka + "rusza");
    }

    @Override
    public void stop() {
        System.out.println("Samochód marki " + marka + "zatrzymał się");
    }
}
```

Sposób 2: Klasę `Samochod` możemy też utworzyć w alternatywny sposób. Klikamy ALT + Enter po najechaniu na nazwę `Moveable` i wybieramy opcję `Implement interface`



a następnie podajemy nazwę klasy, która implementuje dany interfejs: tym razem jest to `Samochod`.

Ten sposób automatycznie doda frazę `implements Moveable` oraz pozwala od razu wygenerować metody `start, stop`. Od razu możemy przejść do podania definicji tych metod oraz zadeklarowania pola `marka`.

D) Dodajmy teraz do naszego projektu kolejny interfejs – `Speakable`:

```
public interface Speakable {

    int QUIET = 0;      // publiczne stałe statyczne
    int LOUD = 1;      // domyślnie public static final
    String getVoice(int voice); // publiczna abstrakcyjna metoda

}
```

Będziemy rozróżniać głośnie i ciche „danie głosu”. Taki interfejs mogą implementować m.in. zwierzęta, na przykład pies.

E) Własności wspólne dla wszystkich zwierząt możemy wyodrębnić do klasy `Zwierze`. Klasy dziedziczące reprezentować konkretne gatunki (na przykład `Pies`, czy `Kot`). Klasa `Zwierze` jest klasą bardzo ogólną, dlatego zapiszemy ją jako abstrakcyjną - nie chcemy tworzyć nienazwanego zwierzęcia, nie przypisanego do żadnego gatunku.

Napiszmy więc abstrakcyjną klasę **`Zwierze`** z polem
`private String name = "bez imienia";`
konstruktorami, metodą zwracającą wartość imienia (*getter*), nadpisaną metodą toString i abstrakcyjna metodą:
`public abstract String getTyp();`

```
public abstract class Zwierze {
    private String name = "bez imienia";

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

    public Zwierze() {
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "Zwierze{" +
                "name='" + name + '\'' +
                '}';
    }

    public abstract String getTyp(); // metoda abstrakcyjna
    // na tą chwilę, to zwierze ogólnie, nie wiadomo jaki typ

}
```

F) Możemy teraz utworzyć nową klasę: klasę **`Pies`** rozszerzającą klasę `Zwierze` i implementującą oba interfejsy (pies może zarówno poruszać się, jak i dawać głos).

```
public class Pies extends Zwierze implements Moveable,Speakable
```

W Javie nie występuje dziedziczenie wielokrotne, jednak możliwe jest implementowanie więcej niż jednego interfejsu (oddzielamy ich nazwy przecinkiem). W klasie implementujemy konstruktory, a następnie musimy podać definicje wszystkich dostępnych metod abstrakcyjnych. Klasa `Zwierze` posiada abstrakcyjną metodę `getTyp`. teraz możemy wskazać typ zwierzęcia jako „Pies”:

```
@Override
public String getTyp() {
    return "Pies";
}
```

Ponieważ implementujemy interfejs `Moveable`, konieczne jest także zdefiniowane metod `start` i `stop`:

```
@Override
public void start() {
    System.out.println("Pies zaczął biec");
}

@Override
public void stop() {
    System.out.println("Pies zatrzymał się");
}
```

Natomiast implementacja interfejsu `Speakable` wymaga zdefiniowania metody `getVoice`:

```
@Override
public String getVoice(int voice) {
    if(LOUD == voice)
        return "HAU HAU";
    else if(QUIET == voice)
        return "hau hau";
    else
        return null;
}
```

Przy definiowanie powyższych metod możemy skorzystać ze skrótu ALT + Enter, który pozwoli wybrać metody możliwe do nadpisania.

Finalny wygląd klasy `Pies`:

```
public class Pies extends Zwierze implements Moveable,Speakable {

    public Pies() {
    }
    public Pies(String name) {
        super(name);
    }
    @Override
    public String getTyp() {
        return "Pies";
    }
    @Override
    public void start() {
        System.out.println("Pies zaczął biec");
    }
    @Override
    public void stop() {
        System.out.println("Pies zatrzymał się");
    }
    @Override
    public String getVoice(int voice) {
        if(LOUD == voice)
            return "HAU HAU";
        else if(QUIET == voice)
            return "hau hau";
        else
            return null;
    }
}
```

G) Teraz możemy przetestować działanie napisanych klas. Oczywiście nie możemy utworzyć obiektu klasy `Zwierze` – jest to klasa abstrakcyjna.

```
public static void main(String[] args) {

    Samochod ford = new Samochod( marka: "Mustang");
    ford.start();
    ford.stop();

    Pies goofy = new Pies( name: "Goofy");
    System.out.println(goofy.getVoice(1));
    goofy.start();
    goofy.stop();
    System.out.println(goofy.getTyp());

}
```

H) Pozostało nam jeszcze napisać metodę **`wyscig`**`(...)`, która jako parametry ma dostać dowolną liczbę obiektów `Moveable` (skorzystać z "varargs"), na których powinna uruchomić metodę **`start`**`()`.

Wyścig będzie publiczną statyczną metodą typu void. Dzięki skorzystaniu z "varargs" (zapis …) metoda zadziała dla dowolnej liczby obiektów `Moveable`. Napiszemy zatem pętle for-each, która dla każdego przekazanego obiektu `m` wywoła na nim metodę `start`.

Jeżeli chcemy dodatkowo wywołać jakąś metodę tylko dla określonych klas implementujących `Moveable` (na przykład wyświetlić dodatkowo imię psa) możemy skorzystać z operatora `instanceof`. `Instanceof` pozwala sprawdzić czy obiekt jest instancją określonego typu (klasy, podklasy, interfejsu).

```
public static void wyscig(Moveable... moveables) {
    for (Moveable m : moveables) {
        m.start();
        if(m instanceof Pies)
            System.out.println("Imię psa: " + ((Pies) m).getName());
    }

}
```

Przykładowe wywołanie metody `wyscig`:

```
// test wyscig:
System.out.println("Wyścig:");
wyscig(new Rower(), new Pies( name: "Reksio"), new Samochod( marka: "BMW X5"));
```