# Klasy abstrakcyjne i interfejsy

**Streszczenie**

Celem wykładu jest omówienie klas abstrakcyjnych i interfejsów w Javie.
Czas wykładu – 45 minut.

Rozwiązanie w miarę standardowego zadania matematycznego (i nie tylko matematycznego) składa się z trzech czynności: wprowadzenia danych, znalezienia rozwiązania i wypisania wyników. Oczywiście ważne jest, aby te czynności wykonane były zawsze. Ponadto zawsze w wymienionej kolejności. Nie ma sensu wypisywać wyniku przed znalezieniem rozwiązania, ani szukać tego rozwiązania zanim wprowadzi się dane wejściowe. Przykładowa klasa opisująca zadanie może więc wyglądać:

```
class Zadanie
{
    void wprowadzDane() { }
    void dyskusjaRozwiazania() { }
    void wypiszWyniki() { }
    void rozwiaz()
    {
        wprowadzDane();
        dyskusjaRozwiazania();
        wypiszWyniki();
    }
}
```

W celu ułatwienia stosowania tej klasy, wygodne jest, aby zawierała ona metodę `void rozwiaz()`. Użytkownik klasy, chcąc rozwiązać konkretne zadanie, zamiast wywoływania trzech metod

```
class ZADANIA
{
    public static void main(String[] args)
    {
        Zadanie z = new Zadanie();
        z.wprowadzDane();
        z.dyskusjaRozwiazania();
        z.wypiszWyniki();
    }
}
```

może zrealizować cel wywołując tylko jedną

```
class ZADANIA
{
    public static void main(String[] args)
    {
        Zadanie z = new Zadanie();
        z.rozwiaz();
    }
}
```

Zwalniamy użytkownika z konieczności pamiętania nazw kilku metod oraz pamiętania kolejności ich wywołania.

Zwróćmy uwagę, że na razie nie mówimy o żadnym konkretnym zagadnieniu, nie mówimy też o ilości danych wejściowych ani wyjściowych. Dlatego metody `void wprowadzDane()`, `void dyskusjaRozwiazania()` i `void wypiszWyniki()` mają puste implementacje. Ich nazwy muszą się jednak pojawić, aby mogły być użyte w funkcji `void rozwiaz()`.

### Metody i klasy abstrakcyjne

W sytuacjach, gdy pewne metody są definiowane w klasach tylko po to, aby wprowadzić nazwy, a ich definicje są puste, możemy zastosować tak zwane *metody abstrakcyjne*. Osoby po kursie programowania w C++ znają to pojęcie. W C++ metodę abstrakcyjną definiowało się opuszczając jej definicję, a stosując zapis `virtual void f() = 0;`, gdzie typ `void` oraz nazwa `f` mogą być dowolne.

W Javie metody abstrakcyjne deklaruje się stosując kwalifikator `abstract`, np.: `abstract void f();`. Ponownie, typ `void` oraz nazwa `f` mogą być dowolne. Ponadto, jeśli w klasie jest zadeklarowana przynajmniej jedna metoda abstrakcyjna, to cała klasa staje się *klasą abstrakcyjną*, i jako taka wymaga kwalifikatora `abstract` przed słowem `class`, np.

```
abstract class Zadanie
{
    abstract void wprowadzDane();
    abstract void dyskusjaRozwiazania();
    abstract void wypiszWyniki();
    void rozwiaz()
    {
        wprowadzDane();
        dyskusjaRozwiazania();
        wypiszWyniki();
    }
}
```

Klasy abstrakcyjne mają jedno bardzo ważne ograniczenie: **nie wolno generować obiektów klas abstrakcyjnych**. Na program

```
class ZADANIA
{
    public static void main(String[] args)
    { Zadanie z = new Zadanie(); }
}
```

kompilator reaguje następująco:

```
Rownania.java:46: Zadanie is abstract; cannot be instantiated
Zadanie r1 = new Zadanie();
```

Zatem w jaki sposób wykorzystać metody i klasy abstrakcyjne? Rozwiązanie jest w miarę oczywiste: należy zdefiniować klasę potomną klasy abstrakcyjnej i nadpisać w niej metody abstrakcyjne (ukonkretnić je). Przy czym nie ma obowiązku nadpisywania wszystkich metod abstrakcyjnych w bezpośrednim potomku, ale w którymś z potomków. Pierwsza klasa potomna klasy abstrakcyjnej, w której ukonkretni się wszystkie metody abstrakcyjne staje się klasą konkretną (nie wymagany jest kwalifikator `abstract`) i dopiero dla niej można generować obiekty.

Rozwijając nasz przykład możemy zdefiniować klasę modelującą na przykład układ dwóch równań z dwoma niewiadomymi następująco:

```
class UkladRownanZDwomaZmiennymi extends Zadanie
{
    @Override void wprowadzDane() { }
    @Override void dyskusjaRozwiazania() { }
    @Override void wypiszWyniki() { }
}
```

i wykorzystać ją do rozwiązania konkretnego równania:

```
class ZADANIA
{
    public static void main(String[] args)
    {
        Zadanie r = new UkladRownanZDwomaZmiennymi();
        r.rozwiaz();
    }
}
```

Celem wykładu nie jest prezentacja rozwiązywania układów równań, zatem definicje metod pozostawiamy puste.

### Interfejsy

Rozszerzeniem języka Java w stosunku do C++ jest wprowadzenie nowej kategorii klas—interfejsów. ***Interfejs*, to klasa, której domyślnie wszystkie metody są abstrakcyjne i publiczne, a pola statyczne i finalne.** Deklarując metody w interfejsie nie wymaga się kwalifikatora `abstract`, a sam interfejs definiuje się stosując słowo `interface` zamiast `class`, np.:

```
interface Interfejs
{
    void f1();
    void f2();
    int czerwony = 3;
}
```

W jaki sposób wykorzystuje się interfejsy?

Chcąc użyć dany interfejs, po nazwie klasy stosuje się słowo `implements` i nazwę interfejsu, np.

```
class Klasa implements Interfejs
{
    public void f1() { }
    public void f2() { }
}
```

W klasie implementuje się metody zdefiniowane w użytym interfejsie, pamiętając o kwalifikatorze `public`. Przypomnijmy, że wszystkie metody interfejsów są publiczne. Jeśli dana klasa nie ukonkretni wszystkich metod użytego interfejsu, to taka klasa pozostaje klasą abstrakcyjną z wszelkimi tego konsekwencjami opisanymi wyżej.

Java jest językiem obiektowym jednobazowym. To znaczy może mieć maksymalnie jednego bezpośredniego przodka. Reguła ta jest złamana w przypadku interfejsów—klasa może implementować wiele interfejsów. Ich nazwy podajemy jako lista rozdzielana przecinkami po słowie `implements`. Klasa, aby nie być abstrakcyjną, musi ukonkretnić wszystkie metody wszystkich implementowanych interfejsów.

### Dziedziczenie interfejsów przez interfejsy

Interfejs może być dziedziczony przez inne interfejsy, np.:

```
interface I1 { }
interface I2 { }
interface J1 extends I1 { }

class K implements I2,J1 { }
```

Interfejsy `I1`, `I2` są niezależne między sobą. Interfejs `J1` dziedziczy interfejs `I1`. Klasa `K` implementuje interfejsy `I2` oraz `J1`.

### Generowanie obiektów typu interfejsu

Aby wykorzystać interfejs, nie jest konieczne tworzenie klasy implementującej dany interfejs. Java dopuszcza generowanie obiektów typu interfejsu, wykorzystując tak zwane klasy anonimowe, klasy bez nazw. Mając interfejs o nazwie `Interfejs` z przykładu wyżej, generujemy obiekt wskazywany przez referencję `i` następująco:

```
class X
{
    public static void main(String[] args)
    {
        Interfejs i = new Interfejs()
            { //początek klasy anonimowej
                public void f1() { }
                public void f2() { }
            };
        i.f1();
    }
}
```

### Kompletny przykład

```
interface IZadanie
{
    void wprowadzDane();
    void dyskusjaRozwiazania();
    void wypiszWyniki();
}

abstract class Zadanie implements IZadanie
{
    void rozwiaz()
    {
        wprowadzDane();
        dyskusjaRozwiazania();
        wypiszWyniki();
    }
}

class UkladRownanZDwomaZmiennymi extends Zadanie
{
    @Override public void wprowadzDane() { }
    @Override public void dyskusjaRozwiazania() { }
    @Override public void wypiszWyniki() { }
}

class ZADANIA
{
    public static void main(String[] args)
    {
        Zadanie r = new UkladRownanZDwomaZmiennymi();
        r.rozwiaz();
    }
}
```

**Podsumowanie**

- Metodę abstrakcyjną definiuje się stosując kwalifikator `abstract` przed typem wyniku funkcji.
- Jeśli klasa posiada przynajmniej jedną metodę abstrakcyjną, to klasa jest abstrakcyjna i wymagany jest kwalifikator `abstract` przed słowem `class`.
- Nie można generować obiektów klas abstrakcyjnych.
- Interfejsy definiuje się stosując słowo `interface`.
- Wszystkie metody interfejsów są domyślnie abstrakcyjne i publiczne.
- Wszystkie pola interfejsów są domyślnie statyczne i finalne i jako takie wymagają nadania wartości w momencie definicji.
- Klasa wykorzystuje interfejs stosując słowo `implements`.
- Klasa może implementować kilka interfejsów.
- Interfejsy mogą być dziedziczone przez interfejsy.
- Interfejsy nie mogą dziedziczyć klas.
- Można generować obiekty typu interfejs.