# Języki i paradygmaty programowania: Laboratorium nr 10

## Podstawowe paradygmaty programowania obiektowego - wprowadzenie. Java - dziedziczenie, polimorfizm.

2017-2018

*mgr inż. Przemysław Walkowiak*
*dr inż. Michał Ciesielczyk*

# Instrukcja

W czasie pisania programu pamiętaj o:

1. dbaniu o czytelność kodu (odpowiednie formatowanie kodu, nazewnictwo zmiennych adekwatne do ich znaczenia, komentarze),
2. dbaniu o czytelność interfejsu z użytkownikiem (w sposób jawny pytaj użytkownika jakie dane ma podać oraz opisuj wyniki, które zwracasz),
3. przed fragmentem implementującym poszczególne zadania umieść komentarz: `/*Zadanie X */` oraz wypisz na ekranie analogiczny komunikat (X jest numerem zadania),
4. każde zadanie umieść w oddzielnej klasie z odpowiednimi metodami,
5. zaimplementuj menu wyboru zadania, a następnie wykorzystując pętle **do–while** oraz konstrukcję **switch** wykonaj odpowiedni fragment kodu,
6. w zadaniach wymagających udzielenia komentarza bądź odpowiedzi, należy umieścić go w kodzie programu (np. w postaci komentarza albo wydrukować na ekranie),
7. w zadaniach polegających na zaprojektowaniu klasy należy utworzyć jej instancję i wykorzystać zaimplementowaną funkcjonalność.

# Wprowadzenie

Java jest obiektowym językiem programowania cechującym się (podobnie jak C++) silnym typowaniem. Znaczna część składni i słów kluczowych w Javie zostały przejęte z języka C++. Przykładowa klasa reprezentująca punkt w dwuwymiarowym układzie współrzędnych mogłaby wyglądać następująco:

```
public class Point {

    private double x, y; // instance fields

    public Point(double x, double y) { // constructor
        this.x = x;
        this.y = y;
    }

    public Point() { // constructor
        this(0.0, 0.0);
    }

    public Point add(Point other) { // method
```

```
15          return new Point(x + other.x, y + other.y);
        }

        public double getX() { // method (getter)
            return x;
20      }

        public double getY() { // method (getter)
            return y;
        }
25  }
```

Zazwyczaj jedna klasa odpowiada jednemu plikowi o tej samej nazwie oraz rozszerzeniu **.java*. Przykładowo, klasa `Point` powinna zostać utworzona w pliku *Point.java*.

Konstruktor jest to metoda wewnątrz klasy wywoływana w momencie tworzenia instancji tej klasy. Konstruktor ma zawsze nazwę taką samą jak klasa i nie zwraca żadnej wartości. Wykorzystuje się ją do inicjalizacji wartości pól klasy. Konstruktor domyślny (linia 10) nie posiada żadnych argumentów i jest wywoływany w przypadkach jak poniżej:

```
Point p = new Point(); // konstruktor domyślny nadaje x i y wartości 0
```

Konstruktory tak jak każda funkcja czy metoda może mieć dowolną liczbę argumentów. Konstruktor z linii 5 posiada dwa argumenty, które są wykorzystane do inicjalizacji pól struktury. Przykład wykorzystania:

```
Point p = new Point(2, 5); // konstruktor nadaje x i y wartości 2 i 5
```

W przypadku programów napisanych w Javie, pamięć przydzielana dynamiczna jest automatycznie zarządzana przez JVM (ang. Java Virtual Machine). Za proces zwalniania odpowiedzialny jest nie programista, lecz wyłącznie programowy zarządca nazywany garbage collector. Klasy nie posiadają więc destruktorów, które można wywołać w kodzie programu.

Funkcje `getX()` oraz `getY()` (linie 18 i 22) pozwalają na pobranie współrzędnych punktu bez zmiany jego stanu. Zwróć uwagę, że bezpośredni dostęp do zmiennych `x` oraz `y` spoza klasy jest niedozwolony (pola prywatne).

## Specyfikatory dostępu

Specyfikatory dostępu pozwalają na określenie poziomu dostępu do poszczególnych klas oraz ich składowych. Podobnie jak w C++:

- **private** oznacza, że elementy dostępne są tylko z wnętrza danej klasy.
- **protected** oznacza, że elementy dostępne są tylko z wnętrza danej klasy oraz klas dziedziczących.
- **public** powoduje, że składowe są publicznie dostępne.

Jeśli deklaracja klasy lub jej składowej nie jest poprzedzona żadnym specyfikatorem, to domyślnym (dla kompilatora) jest tzw. **package–private**. Oznacza on, że elementy dostępne są tylko z wnętrza danego pakietu Javy.

## Pakiety

Pakiety w Javie pozwalają na organizację class w odpowiednie przestrzenie nazw. W pliku źródłowym Java, pakiet do którego należy dana klasa (lub zbiór klas) jest definiowany poprzez słowo kluczowe **package**. Zazwyczaj jest ono umieszczane na początku pliku źródłowego, np.:

```
package pl.poznan.put.jipp;
```

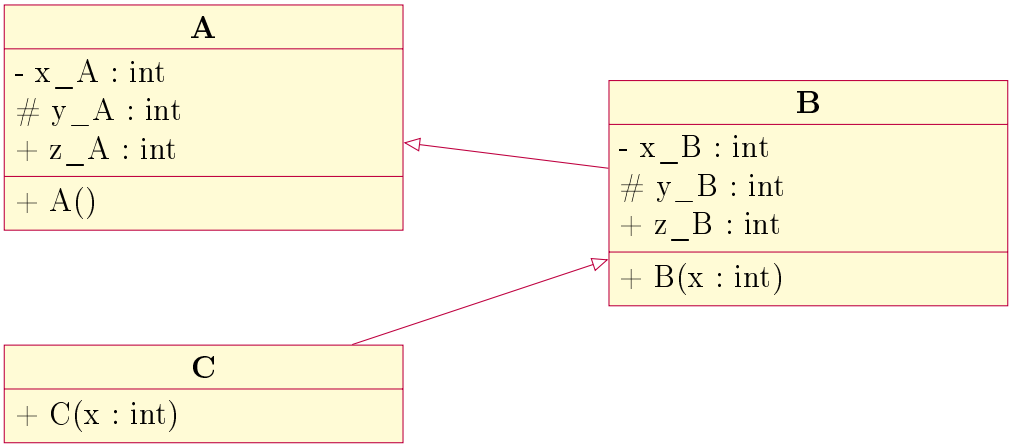
Aby w kodzie źródłowym wykorzystać klasę znajdującą się w innym pakiecie należy użyć słowa kluczowego **import**. Przykładowo:

```
import pl.poznan.put.jipp.Point;
```

importuje klasę `Point` z pakietu `pl.poznan.put.jipp`.

## Dziedziczenie

Na rysunku 1 został umieszczony diagram w notacji UML przedstawiający trzy klasy `A`, `B` oraz `C`. Każda klasa jest reprezentowana przez jeden prostokąt, podzielony na trzy części: nazwę klasy, listę atrybutów (pól) oraz listę metod. Symbole $-, \#, +$ przed nazwą pola lub metody oznaczają tryb widoczności danego składnika, odpowiednio: **private**, **protected** i **public**. Strzałki pomiędzy poszczególnymi klasami symbolizują relację **dziedziczenia**.




Rysunek 1: Diagram zależności pomiędzy klasami A, B i C.

Więcej informacji na temat dziedziczenia w Javie można znaleźć tutaj: `https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html`.

Implementacja klas `A`, `B` oraz `C` z rysunku 1 w języku C++ wygląda następująco:

```
class A {

    private int x_A;
    protected int y_A;
    public int z_A;

    public A() {
        System.out.println("ctor A");
    }

    void printValuesA() {
        System.out.println(x_A);    // ok
        System.out.println(y_A);    // ok
        System.out.println(z_A);    // ok
    }
};

class B extends A { // dziedziczenie po klasie A

    private int x_B;
    protected int y_B;
    public int z_B;

    public B(int x) {
        super();
        System.out.println("ctor B");
    }

    void printValuesB() {
        // System.out.println(x_A); // błąd: x_A jest polem prywatnym
                                    // klasy bazowej A
        System.out.println(y_A);    // ok
        System.out.println(z_A);    // ok
        System.out.println(x_B);    // ok
        System.out.println(y_B);    // ok
        System.out.println(z_B);    // ok
    }
}

class C extends B { // dziedziczenie po klasie B

    public C(int x) {
        super(x); // wywołanie konstruktora klasy bazowej
        System.out.println("ctor C");
    }

    void printValuesC() {
        // System.out.println(x_A); // błąd: x_A jest polem prywatnym
```

```
                                // klasy bazowej A
50          System.out.println(y_A);    // ok
            System.out.println(z_A);    // ok
            // System.out.println(x_B); // błąd: x_B jest polem prywatnym
                                // klasy bazowej B
            System.out.println(y_B);    // ok
55          System.out.println(z_B);    // ok
    }
}
```

Korzystając z paradygmatu dziedziczenia w języku Java należy pamiętać o wywołaniu konstruktora klasy nadrzędnej (linie 25 oraz 43). W tym celu wykorzystywane jest słowo kluczowe **super**. W przypadku pominięcia jawnego wywołania, zostanie wywołany konstruktor domyślny o ile taki istnieje - jeżeli w tym konstruktor domyślny nie istnieje to w tym przypadku kompilator zgłosi błąd. Innymi słowy, linia 25 mogłaby zostać pominięta.

Przykłady wykorzystanie zdefiniowanych klas:

```
    A a1 = new A();
    B b1 = new B(5);
    C c1 = new C(2);

5   b1.printValuesA();// wypisuje na ekranie wartości składowych
                      // x_A, y_A, z_A. Ponieważ metoda znajduje się
                      // w klasie A ma również dostęp do pól prywatnych
                      // klasy A czyli m.in. do x_A

10  b1.printValuesB(); // wypisuje na ekranie wartości składowych
                      // y_A, z_A oraz x_B, y_B, z_B

    c1.printValuesC(); // wypisuje na ekranie wartości składowych
                       // y_A, z_A oraz y_B, z_B
15
    c1.printValuesB(); // wypisuje na ekranie wartości składowych
                       // y_A, z_A oraz x_B, y_B, z_B. Ponieważ metoda
                       // znajduje się w klasie B ma również dostęp do
                       // pól prywatnych klasy B czyli m.in. do x_B
20
    A a2 = b1;
    a2.printValuesA(); // ok
    a2.printValuesB(); // błąd kompilacji, klasa A nie ma tej metody.
    ((B) a2).printValuesB(); //ok
```

## Metody wirtualne

W Javie, wszystkie niestatyczne metody są domyślnie funkcjami wirtualnymi. Wyłącznie metody oznaczone jako prywatne lub finalne (słowo kluczowe **final**) – tzn. takie, które nie są dziedziczone – nie są wirtualne.

Rozważmy analogiczny do poprzedniego przykład:

```
class A {
    protected double x, y;

    public A(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double compute() {
        return x + y;
    }
}

class B extends A {

    public B(double x, double y) {
        super(x,y);
    }

    @Override
    public double compute() {
        return x * y;
    }
}
```

Podczas zarzutowania instancji klasy B na klasę A i wywołaniu metody `compute` zostanie wywołana wersja przynależna do klasy B.

```
    A a1 = new A(5, 4);
    System.out.println(a1.compute()); // wywołanie metody A.compute();
    // wynik: 9

    B b1 = new B(5, 4);
    System.out.println(b1.compute()); // wywołanie metody B.compute();
    // wynik: 20

    A a2_b1 = b1;
    System.out.println(a2_b1.compute()); // wywołanie metody B.compute();
    // wynik: 20
```

## Obsługa wejścia-wyjścia

Platforma Java wspiera trzy standardowe strumienie:

- `System.in` – Standard Input,
- `System.out` – Standard Output,

- `System.err` – Standard Error.

Przykładowo, metody `print` oraz `println` (napis dodatkowo zakończony nową linią) pozwalają na wypisanie informacji do konsoli:

```
String s = "This is an example string.";
System.out.println(s);
int x = 10;
System.out.println("x is equal to " + x);
```

Do wczytywania informacji z klawiatury można wykorzystać klasę `Scanner`. Przykładowo:

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
boolean b = sc.nextBoolean();
String s = sc.next();
String line = sc.nextLine();
```

gdzie:

- `nextInt()` – zwraca wartość typu **int** (linia 2),
- `nextBoolean()` – zwraca wartość typu **boolean** (linia 3),
- `next()` – zwraca ciąg znaków do następnego znaku białego (linia 4),
- `nextLine()` – zwraca ciąg znaków zakończony nową linią (linia 5).

Alternatywnie, można również skorzystać z bardziej zaawansowanej klasy Console.

## Przykład: "Hello World!"

Poniżej przedstawiono przykładową implementację aplikacji wyświetlającej na ekranie napis "*Hello World!*".

```
public class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

W linii 1 zdefiniowana jest klasa o nazwie `HelloWorldApp`. Podobnie jak w C++, w języku programowania Java, każda aplikacja musi posiadać główną metodę (linia 2). Zmienna `String[] args` oznacza listę argumentów przekazywanych do aplikacji (każdy napis na tej liście odpowiada jednemu argumentowi z konsoli). W tym przykładzie jest ona ignorowana. Ostatecznie, w linii 3, wykorzystywana jest klasa `System` do wydrukowania wiadomości "*Hello World!*" na standardowym wyjściu.

**Dodatkowe informacje:**

- Tutorial Java: `https://docs.oracle.com/javase/tutorial/index.html`
- Specyfikacja Java Platform API: `https://docs.oracle.com/javase/8/docs/api/overview-summary.html`

# Zadania

## Zadanie 1

Zdefiniuj klasę o nazwie `Osoba` zawierającą informacje takie jak imię i nazwisko (typu `String`, niezmienne, ustawiane wyłącznie w konstruktorze).

**Wskazówka** W Netbeans, po dodaniu do klasy odpowiednich pól, konstruktor możesz wygenerować automatycznie. W tym celu, w kodzie źródłowym wciśnij Ctrl+Space i wybierz odpowiedni konstruktor, lub kliknij prawym przyciskiem myszy na kod źródłowy i wybierz Insert Code... > Constructor....

## Zadanie 2

W funkcji głównej utwórz 3 przykładowe instancji klasy `Osoba` w następujący sposób:

```
Osoba jan = new Osoba("Jan", "Kowalski");
Osoba jan2 = new Osoba("Jan", "Kowalski");
Osoba maria = new Osoba("Maria", "Nowak");
```

Następnie przyrównaj wszystkie instancje do zmiennej `jan` z wykorzystaniem operatora `==` oraz metody `equals`:

```
System.out.printf("jan == jan : %b%n", jan == jan);
System.out.printf("jan == jan2 : %b%n", jan == jan2);
System.out.printf("jan == maria : %b%n", jan == maria);
System.out.printf("jan.equals(jan) : %b%n", jan.equals(jan));
System.out.printf("jan.equals(jan2) : %b%n", jan.equals(jan2));
System.out.printf("jan.equals(maria) : %b%n", jan.equals(maria));
```

Czym różnią się oba sposoby?

Zwróć uwagę, że obiekty, które zawierają te same dane są rozpoznawane jako różne. Aby to zmienić przeciąż metodę `equals` w klasie `Osoba`. Przetestuj swoją implementację ponownie uruchamiając powyższy kod.

**Wskazówka** W Netbeans możesz automatycznie wygenerować klikając prawym przyciskiem myszy na kod źródłowy i wybierając Insert Code... > equals() and hashcode()... . Przeanalizuj wygenerowany kod – czy sam/a zaimplementowałbyś/łabyś to tak samo? Zwróć uwagę, że razem z metodą `equals()` generowana jest metoda `hashcode()` – jak myślisz dlaczego zalecane jest przygotowanie obu metod jednocześnie?

## Zadanie 3

Zdefiniuj nową klasę o nazwie `Pracownik` dziedziczącą po klasie `Osoba`. Dodaj do nowo zdefiniowanej klasy pola prywatne z informacjami o stanowisku i pensji (odpowiednio typu `String` oraz **int**). Następnie, dodaj odpowiednie metody typu `getter` oraz `setter`, pozwalające odpowiednio na pobieranie oraz ustawianie wartości pól.

**Wskazówka** W Netbeans, po dodaniu do klasy odpowiednich pól, metody typu `getter` oraz `setter` możesz wygenerować automatycznie. W tym celu, w kodzie źródłowym wciśnij Ctrl + Space i wybierz odpowiednią metodę, lub kliknij prawym przyciskiem myszy na kod źródłowy i wybierz Insert Code... » Getter and Setter....

## Zadanie 4

Zaimplementuj klasę `VerboseObject` posiadającą:

1. prywatne pole typu napisowego przechowujące nazwę obiektu,
2. konstruktor jednoargumentowy (poprzez argument należy zainicjalizować powyższe pole), wypisujący na ekranie informacje, że został wywołany (należy również wyświetlić nazwę obiektu),
3. przeciążenie metody `finalize`, będącej odpowiednikiem destruktora w C++, która wyświetla na ekranie informację o tym, że obiekt jest niszczony (również uwzględniającą nazwę obiektu),
4. metodę, która wypisuje na ekranie informację, że została wywołana (również uwzględniającą nazwę obiektu).

Przykładowo, wywołanie:

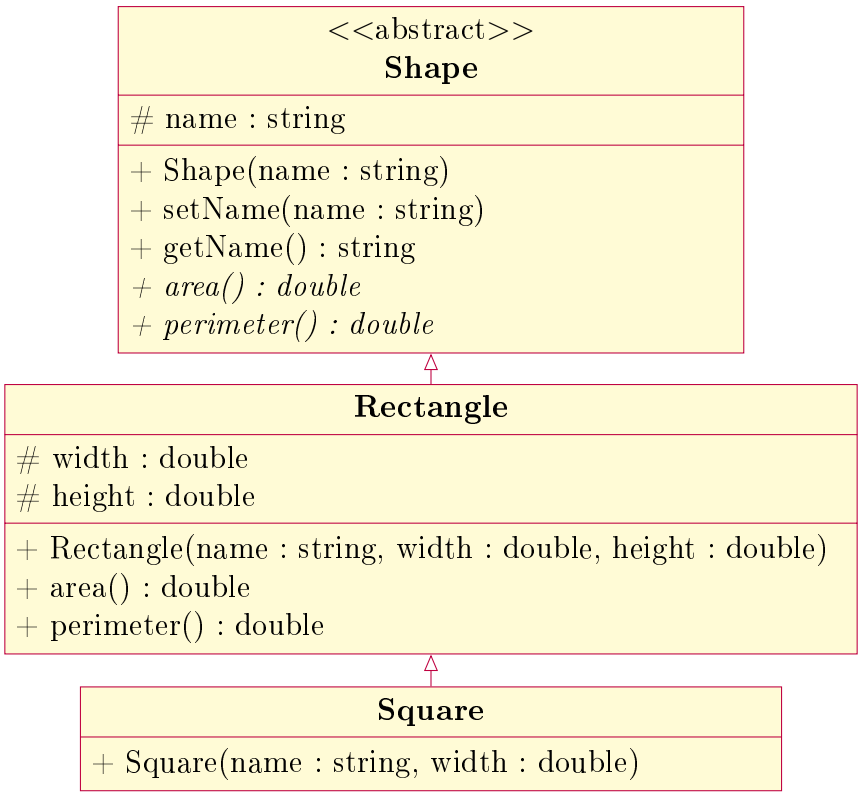
```
{
    VerboseObject o1 = new VerboseObject("Object 1");
    o1.saySomething();
}
```

mogłoby wypisać na ekranie:

```
Object 1 constructed.
Object 1 says hello.
```

Przeprowadź eksperyment (i napisz w komentarzu wnioski) polegający na przetestowaniu tworzenia kilku obiektów. Prześledź w jakiej kolejności wywoływane są konstruktory, a w jakiej odpowiedniki „destruktorów". Zwróć uwagę, że w Javie nie ma żadnej możliwości ręcznego zwolnienia pamięci po obiekcie.

Zastanów się dlaczego w językach opartych na automatycznym odśmiecaniu pamięci (ang. *garbage collection*) nie powinno się polegać na metodach typu `finalize`.




Rysunek 2: Diagram zależności pomiędzy klasami Shape, Rectangle, Square.

## Zadanie 5

Zaimplementuj klasy `Shape`, `Rectangle` oraz `Square` ze składowymi jak na rysunku 2 i następującą funkcjonalnością:

a) każdy konstruktor wypisuje na ekranie komunikat, że jest wywoływany,
b) oznacz metody `area()` oraz `perimeter()` w klasie `Shape` jako abstrakcyjne (**abstract**),
c) w klasie `Rectangle` zaimplementuj metody `area()` oraz `perimeter()` liczące odpowiednio pole oraz obwód figury.

Utwórz instancje przygotowanych klas i wyświetl na ekranie ich pola. Następnie zarzutuj obiekty każdej z klas `Rectangle` i `Square` na wszystkie klasy wyżej w hierarchii oraz wywołaj metodę `area()`. Co zauważyłeś/aś?

Wykonaj powyższe polecenia (wnioski napisz w komentarzu). Prześledź w jakiej kolejności dla obiektów automatycznych wywoływane są konstruktory.

**Wskazówka 1** W konstruktorach klas dziedziczących wywołaj konstruktor klasy bazowej. Przykładowo, implementacja konstruktora `Square` mogłaby wyglądać następująco:

```
public Square(String name, double width) {
    super(name, width, width);
}
```

Zwróć uwagę, że wywołanie **super** musi być pierwszą instrukcją w konstruktorze.

## Zadanie 6

Zdefiniuj klasę `ShapeContainer`, która jest odpowiedzialna za zarządzanie kolekcją różnych figur (obiekty typu `Shape`). Zaimplementuj w niej metody, które wypisują na ekranie obwody oraz pola wszystkich figur w bazie. Klasa powinna również posiadać metodę `add(Shape)` pozwalającą na dodanie dowolnej figury.

Przetestuj swoją implementację przykładowymi danymi.

**Wskazówka** Do przechowywania kolekcji figur możesz skorzystać z klasy `ArrayList`.

```
import java.util.ArrayList;
import java.util.List;

/* ... */
List<Shape> shapes = new ArrayList<>(); // inicjalizacja listy
shapes.add(s); // dodawanie nowej figury s
for (Shape s : shapes) { // przeglądanie listy
    // dla każdej figury s w kolekcji figures wykonaj ...
}
```

## Zadanie 7*

Wykorzystując mechanizmy dziedziczenia oraz definicje klas z poprzednich zadań zaprojektuj klasy `Elipse` dla elipsy (dziedziczącego po `Shape`) oraz `Circle` dla koła (dziedziczącego po `Elipse`). Następnie, spróbuj je dodać do `ShapeContainer` oraz ponownie przetestować tę implementację.

**Wskazówka 1** Wzór na pole powierzchni ograniczonej przez elipsę: $\pi ab$, oraz przybliżony wzór na obwód elipsy: $\pi(\frac{3}{2}(a+b) - \sqrt{ab}))$, gdzie $a$ i $b$ to odpowiednio półoś wielka i półoś mała elipsy.