Programowanie Obiektowe

Laboratorium 3 – podstawy języka Java

Tomasz Bieliński

2017-03-21

Przygotowanie do zajęć w domu

Programy napisane w języku Java są uruchamiane za pomocą maszyny wirtualnej Java (JVM). Maszyna wirtualna Javy jest zaimplementowana na wielu popularnych platformach. Są to m.in. Windows oraz Linux. Dzięki temu aplikacje dla maszyny wirtualnej są przenośne i mogą być uruchamiane pod różnymi systemami operacyjnymi oraz na różnych architekturach procesorów[1].

Do uruchamiania programów napisanych w języku Java konieczne jest środowisku uruchomieniowe (JRE). Natomiast do ich wytwarzania potrzebne jest środowisko deweloperskie zawierające kompilator (JDK). Do laboratoriów najlepiej zaopatrzyć się w JDK 1.8. Dodatkowo pomocny będzie edytor kodu źródłowego. Polecanym edytorem do ćwiczeń laboratoryjnych jest NetBeans[2].

Zadanie 1. Prosty program

Java jest językiem w pełni obiektowym i przez to wymusza podejście obiektowe (choć jeśli się ktoś uprze, to może naśladować programowanie proceduralne). Listing 1 przedstawia prosty program w języku Java. Wypisuje on na wyjście pytanie o imię, a następnie odczytuje z wejścia tekst wpisany przez użytkownika. Przykład uruchomienia tego programu przedstawia Listing 2.

Listing 1 Prosty program w języku Java

```
package pl.edu.pg.eti.ksg.po.lab1;
import java.util.Scanner;
public class Javalab1 {

    public static void main(String[] args)
    {
        Scanner inputScanner = new Scanner(System.in);
        System.out.println("Jak masz na imię?");
        String imie = inputScanner.nextLine();
        System.out.println("Witaj "+imie+"!");
    }
}
```

Listing 2 Przykładowy przebieg komunikacji z użytkownikiem programu zawartego w Listingu 1

```
Jak masz na imię?
Dave
Witaj Dave!
```

[1] Co to jest architektura procesora dowiedzą się Państwo w przyszłym semestrze ;)
[2] Program NetBeans można ściągnąć z strony internetowej https://netbeans.org/downloads/ lub wraz z JDK http://www.oracle.com/technetwork/articles/javase/jdk-netbeans-jsp-142931.html

Każda klasa publiczna w języku Java musi się znajdować w oddzielnym pliku o takiej samej nazwie. Przykładowo klasa *Javalab1* znajduje się w pliku *Javalab1.java*. Sam kod programu składa się z następujących elementów:

```
package pl.edu.pg.eti.ksg.po.lab1;
```

Jest to deklaracja informująca o tym w jakim pakiecie znajduje się definiowana właśnie klasa. Pakiety są odpowiednikami przestrzeni nazw i pomagają uniknąć kolizji nazw klas. W języku Java przyjęto zwyczaj nazywania pakietów poprzez odwrócony adres domenowy instytucji która wytworzyła oprogramowanie. W powyższym przykładzie jest to odwrócony adres domenowy wydziału ETI. Dalsza część nazwy pakietu zależy od zasad przyjętych w danej organizacji. Może zawierać nazwę projektu oraz jego części składowych.

```
import java.util.Scanner;
```

Jest do deklaracja mówiąca, że jeśli będziemy w dalszym kodzie używali nazwy *Scanner*, to będziemy mieli na myśli klasę *Scanner* z pakietu *java.util*. Ten element języka jest odpowiednikiem dwóch elementów z języka C++, mianowicie dyrektywy *#include* oraz *using namespace*. W języku Java nie ma plików nagłówkowych. Cały kod klasy znajduje się w pliku Java.

```
public class Javalab1 {
```

Jest to początek definicji klasy *Javalab1*. Wygląda podobnie jak w języku C++. Słowo kluczowe ***public*** oznacza, że klasa ta będzie dostępna spoza pakietu pl.edu.pg.eti.ksg.po.lab1.

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

Jest to początek deklaracji metody *main* w klasie *Javalab1*. Metoda ta jest publiczna, czyli jest dostępna z innych klas. Jest metodą statyczną, czyli nie jest związana z żadną instancją klasy *Javalab1*. Jako argument przyjmuje tablice referencji do obiektów typu *String* (*java.lang.String*). Nic nie zwraca. Jak widać sposób definicji metody również jest podobny do tego używanego w języku C++. Metoda statyczna *main* przyjmująca tablicę referencji do obiektu typu String jest punktem wejścia do programu w języku Java, analogicznie do funkcji main w językach C oraz C++.

```
Scanner inputScanner = new Scanner(System.in);
```

W tej linii jest tworzona instancja klasy *java.util.Scanner*, jako parametr jest podawane pole statyczne *in* klasy *java.lang.System*. Referencja[3] do tego obiektu została zapisana w zmiennej o nazwie *inputScanner*. Obiekt klasy *Scanner* pomoże nam odczytać odpowiedź użytkownika.

```
System.out.println("Jak masz na imię?");
```

Wypisanie na wyjście pytania do użytkownika. W polu statycznym out klasy *java.lang.System* znajduje się obiekt klasy *java.io.PrintStream*. Obsługuje on standardowe wyjście. Do obsługi

[3] Referencja w języku Java najbardziej przypomina wskaźnik z jeżyków C oraz C++. Może ona wskazywać na obiekt lub mieć wartość ***null***. W odróżnieniu od wskaźnika nie można na niej wykonywać operacji ++ oraz --, a także wykonywać odejmowania referencji lub dodawania do nie liczby. Do obiektów wskazywanych przez referencję odwołujemy się przez kropkę „.”.

standardowego wyjścia błędów służy *System.err*. Metoda *println* drukuje zadany obiekt lub wartość na wyjście, a następnie przechodzi do nowej linii. Metoda ta jest wielokrotnie przeciążona.

```
String imie = inputScanner.nextLine();
```

Uruchomienie metody *nextLine* obiektu, na który wskazuje referencja *inputScanner*. Metoda ta zwraca obiekt typu *java.lang.String*. Wynik jest przypisywany do referencji do obiektu *String* o nazwie *imie*.

```
System.out.println("Witaj "+imie+"!");
```

Wypisywanie na wyjście tekstu, którym program wita się z użytkownikiem. Operator + słuzy m.in. do łączenia obiektów typu String w większe napisy.

```
    }
}
```

Są to zamykające nawiasy klamrowe. Zamykają one definicję metody oraz klasy. Proszę zwrócić uwagę na to, że nie ma na końcu średnika.

W powyższym programie można zauważyć że występowały tam odwołania do klas z pakietu ***java.lang*** mimo że nie pojawiła się stosowana linia ***import***. Dzieje się tak ponieważ wszystkie klasy z tego pakietu zawsze są niejawnie importowane, dlatego deklaracja import nie jest potrzebna.

Zadanie 2. Definicja własnej klasy

Zdefiniujmy w pakiecie *pl.edu.pg.eti.ksg.po.lab1* nową klasę *A*. Definicję klasy zawiera Listing 3.

Listing 3 Definicja klasy A

```
package pl.edu.pg.eti.ksg.po.lab1;
public class A {
    private int liczba;

    public A(int liczba) {
        this.liczba = liczba;
    }

    public void setLiczba(int liczba) {
        this.liczba = liczba;
    }

    public int getLiczba() {
        return liczba;
    }

    @Override
    public String toString() {
        return "Instancja klasy A zawierająca liczbę " + liczba;
    }
}
```

Klasa *A* ma jedno pole prywatne typu *int* o nazwie liczba:

```
private int liczba;
```

Ponadto posiada konstruktor przyjmujący jeden parametr typu *int*:

```
public A(int liczba) {
        this.liczba = liczba;
}
```

Metody set oraz get pozwalające na modyfikację wartości pola *liczba*:

```
public void setLiczba(int liczba) {
    this.liczba = liczba;
}

public int getLiczba() {
    return liczba;
}
```

Ostatnią pozycją w klasie A jest metoda *toString*:

```
@Override
public String toString() {
    return "Instancja klasy A zawierająca liczbę " + liczba;
}
```

Metoda ta formuje napis, który ma reprezentować obiekt klasy *A*. Linia `@Override` Informuje, ze metoda *toString* została nadpisana. Powstaje jednak pytanie jak można nadpisać metodę skoro w definicji klasy *A* nie została wspomniana żadna klasa bazowa. Okazuje się, że klasą bazową dla klasy A jest klasa *java.lang.Object*. W języku Java wszystkie klasy bezpośrednio lub pośrednio dziedziczą po tej klasie, nawet jeśli nie została ona wymieniona w definicji klasy. Sama metoda *toString* służy do zamiany obiektu na napis. Klasa *A* nadpisuje tą metodę własną implementacją.

W języku Java wszystkie metody niestatyczne oraz nieprywatne są wirtualne – można je nadpisywać. Aby zablokować nadpisywanie metody trzeba dodać do niej słowo kluczowe ***final***.

Zastosowanie klasy *A* przestawia Listing 4. Przedstawione zostały w nim ponadto sposoby porównywania obiektów. Wyszczególnić można dwa sposoby charakterystyczne dla języka Java:

- Poprzez porównywanie referencji: *a==a1*
- Poprzez metodę *equals*: *a.equals(a1)*

Porównywanie referencji sprawdza jedynie czy referencje wskazują na **ten sam** obiekt. Jeśli jest to **ten sam** obiekt wynikiem będzie *true*. Jeśli nie wynikiem jest *false*. Zadaniem metody *equals* jest sprawdzenie czy obiekty są **takie same**, nawet jeśli są dwoma oddzielnymi obiektami. Aby ta metoda działała poprawnie należy ją nadpisać, gdyż domyślna implementacja z klasy *java.lang.Object*

porównuje referencje. Ponadto została zaprezentowana metoda *hashCode*. Jej zadaniem jest obliczenie skrótu (hash) na podstawie zawartości obiektu. Metoda ta jest wykorzystywana m.in. w kontenerach takich jak *java.util.HashSet* oraz *java.util.HashMap*. Dla dwóch obiektów, dla których metoda *equals* zwraca *true* metoda *hashCode* powinna zwrócić taką samą wartość. Domyślna implementacja metody *hashCode* bazuje na referencji obiektu.

Ponadto można zauważyć, że metoda *println* klasy *java.io.PrintStream* jako argument może przyjmować obiekty. Takie obiekty zostaną zamienione na napis za pomocą metody *toString*, a następnie wypisane na strumień.

Listing 4 Użycie klasy A. Wyjście przedstawia Listing 5

```
package pl.edu.pg.eti.ksg.po.lab1;
public class Javalab1 {

    public static void main(String[] args)
    {
        A a = new A(5);
        System.out.println("Pobieranie liczby:");
        System.out.println(a.getLiczba());
        System.out.println("Drukowanie całego obiektu:");
        System.out.println(a);
        a.setLiczba(6);
        System.out.println("Drukowanie całego obiektu po ustawieniu
liczby:");
        System.out.println(a);

        A a1 = new A(6);
        System.out.println("Porównywanie referencji: a == a");
        System.out.println(a == a);
        System.out.println("Porównywanie referencji: a == a1");
        System.out.println(a == a1);
        System.out.println("Metoda equals: a.equals(a)");
        System.out.println(a.equals(a));
        System.out.println("Metoda equals: a.equals(a1)");
        System.out.println(a.equals(a1));

        System.out.println("Metoda hashCode: a.hashCode()");
        System.out.println(a.hashCode());
        System.out.println("Metoda hashCode: a1.hashCode()");
        System.out.println(a1.hashCode());
    }
}
```

Listing 5 Wyjście programu testującego właściwości obiektów klasy A

```
Pobieranie liczby:
5
Drukowanie całego obiektu:
Instancja klasy A zawierająca liczbę 5
Drukowanie całego obiektu po ustawieniu liczby:
Instancja klasy A zawierająca liczbę 6
```

```
Porównywanie referencji: a == a
true
Porównywanie referencji: a == a1
false
Metoda equals: a.equals(a)
true
Metoda equals: a.equals(a1)
false
Metoda hashCode: a.hashCode()
460141958
Metoda hashCode: a1.hashCode()
1163157884
```

Aby zaprezentować działa metod *equals* oraz *hashCode* stworzono klasę *B*. Jej kod zawiera Listing 6.

Listing 6 Kod klasy B

```
package pl.edu.pg.eti.ksg.po.lab1;
public class B {
    private double liczba;

    public B(double liczba) {
        this.liczba = liczba;
    }

    public double getLiczba() {
        return liczba;
    }

    public void setLiczba(double liczba) {
        this.liczba = liczba;
    }

    @Override
    public String toString() {
        return "Instancja klasy B zawierająca liczbę " + liczba;
    }

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof B)
        {
            B other = (B)obj;
            return liczba == other.liczba;
        }
        return false;
    }

    @Override
    public int hashCode() {
        return 59 * Double.hashCode(liczba) + 7;
    }
}
```

Klasa B jest podobna do klasy A. Zawiera jednak nadpisane dodatkowo metody *equals* oraz *hashCode*. Metoda ta jako parametr przyjmuje referencje do obiektu klasy Object. Pierwsza linia w metodzie sprawdza czy obiekt ten jest klasy B:

```
if(obj instanceof B)
```

Jest to specjalna operacja będąca elementem języka Java.

```
B other = (B)obj;
return liczba == other.liczba;
```

Następnie referencja *obj* jest rzutowana na referencję na klasę B. Wynikiem działania metody *equals* jest porównanie liczb zawartych w obiektach klasy B.

Natomiast w metodzie *hashCode* obliczany jest skrót obiektu klasy B w następujący sposób:

```
return 59 * Double.hashCode(liczba) + 7;
```

Zastosowana została do tego metoda statyczna klasy *java.lang.Double* czyli referencyjnej wersji typu prostego *double*[4]. Dodatkowo wynik został przemnożony przez liczbę 57 oraz zwiększony o 3. Pozwala to nieco zmodyfikować wynik metody *hashCode*, tak aby nie był on identyczny jak w przypadku liczby typu *double*. To takich modyfikacji najlepiej stosować liczby pierwsze, zwłaszcza przy mnożeniu.

Jak widać w powyższym przykładzie metody equals oraz hashCode wykorzystują te same pole (pola) do uzyskania swoich wyników.

Test właściwości klasy *B* przestawia Listing 7. Przykładowy wynik tego testu uwidacznia Listing 8.

Listing 7 Test właściwości klasy B

```
package pl.edu.pg.eti.ksg.po.lab1;
public class Javalab1 {

    public static void main(String[] args)
    {
        B b = new B(2.5);
        System.out.println("Pobieranie liczby:");
        System.out.println(b.getLiczba());
        System.out.println("Drukowanie całego obiektu:");
        System.out.println(b);
        b.setLiczba(3.14);
        System.out.println("Drukowanie całego obiektu po ustawieniu
liczby:");
        System.out.println(b);
```

[4] Każdy typ prosty posiada klasę będącą wersją referencyjną tego typu. Czyli taką do której można odwołać się poprzez referencję zamiast przez wartość. I tak klasą opakowującą typ prosty int jest java.lang.Integer, zaś klasą opakowująca typ char jest java.lang.Character.

```
        B b1 = new B(3.14);

        System.out.println("Porównywanie referencji: b == b");
        System.out.println(b == b);
        System.out.println("Porównywanie referencji: b == b1");
        System.out.println(b == b1);
        System.out.println("Metoda equals: b.equals(b)");
        System.out.println(b.equals(b));
        System.out.println("Metoda equals: b.equals(b1)");
        System.out.println(b.equals(b1));

        System.out.println("Metoda hashCode: b.hashCode()");
        System.out.println(b.hashCode());
        System.out.println("Metoda hashCode: b1.hashCode()");
        System.out.println(b1.hashCode());
    }
}
```

Listing 8 Wyjście programu testującego właściwości obiektów klasy B

```
Pobieranie liczby:
2.5
Drukowanie całego obiektu:
Instancja klasy B zawierająca liczbę 2.5
Drukowanie całego obiektu po ustawieniu liczby:
Instancja klasy B zawierająca liczbę 3.14
Porównywanie referencji: b == b
true
Porównywanie referencji: b == b1
false
Metoda equals: b.equals(b)
true
Metoda equals: b.equals(b1)
true
Metoda hashCode: b.hashCode()
523886468
Metoda hashCode: b1.hashCode()
523886468
```

Jak pokazuje Listing 8 porównanie *b.equals(b1)* daje wynik *true*, gdyż obiekty te mają taką samą zawartość. Widać to także w wynikach metody *hashCode*. Natomiast porównanie referencji *b==b1* dało rezultat *false*, gdyż są to różne obiekty.

Na koniec pozostaje do omówienia kwestia zarządzania pamięcią. Maszyna wirtualna Javy została wyposażona w odśmiecacz (ang. Garbage Collector). Odśmiecacz monitoruje pamięć maszyny wirtualnej w tle. Jeśli zabraknie pamięci lub nastąpi inne ważne zdarzenie inicjujące uruchomiony zostanie mechanizm odśmiecania. Przejrzy on wszystkie referencje używane w całym programie i oznaczy używane obiekty. Pozostałe obiekty zostaną usunięte z pamięci. Mechanizm ten zwalnia programistę z odpowiedzialności za dealokację pamięci. Dzięki temu kod staje się prostszy i czytelniejszy.

Zadania na laboratorium

Zadanie 1 (1 punkt)

Uzupełnij kod klasy Punkt (Listing 9). Pamiętaj o umieszczeniu pliku z kodem klasy w odpowiednim pakiecie. Zaprezentuj działanie klasy Punkt w metodzie main.

Listing 9 Klasa Punkt

```
package pl.edu.pg.eti.ksg.po.lab1.transformacje;
public class Punkt {
    /*
     * Słowo kluczowe final oznacza, że po pierwszym przypisaniu
     * zawartość pól x oraz y nie może zostać zmieniona
     * w tym miejscu jest to odpowiednik słowa kluczowego const
     * z języka C++.
     */
    private final double x,y;
    public Punkt(double x, double y) {
        /*
         * TODO
         */
    }
    public double getX() {
        /*
         * TODO
         */
    }
    public double getY() {
        /*
         * TODO
         */
    }
    @Override
    public boolean equals(Object obj) {
        /*
         * TODO
         */
    }
    @Override
    public int hashCode() {
        /*
         * TODO
         */
    }
    @Override
    public String toString() {
        /*
         * TODO
         */
    }
```

```
    /**
     * Początek układu współrzędnych.
     * W tym miejscu słowo kluczowe final oznacza, że statyczne
     * pole klasy Punkt nie może zostać zmienione.
     * Dotyczy to jednak samej referencji, a nie obiektu, na który
     * wskazuje.
     */
    public static final Punkt O = new Punkt(0, 0);

    /**
     * Punkt (1,0) w układzie współrzędnych
     */
    public static final Punkt E_X = new Punkt(1, 0);

    /**
     * Punkt (0,1) w układzie współrzędnych
     */
    public static final Punkt E_Y = new Punkt(0, 1);
}
```

Zadanie 2 (2 punkty)

Dodaj do projektu klasę BrakTransformacjiOdwrotnejException, która implementuje wyjątek (Listing 10).

Listing 10 Klasa BrakTransformacjiOdwrotnejException - wyjątek

```
package pl.edu.pg.eti.ksg.po.lab1.transformacje;
/*
 * Klasa ta jawnie dziedziczy po klasie java.lang.Exception
 */
public class BrakTransformacjiOdwrotnejException extends Exception
{
    public BrakTransformacjiOdwrotnejException()
    {
        //Niejawne wywołanie konstruktora klasy nadrzędnej
        //super();
    }

    /*
     * Konstruktor przyjmujący jako parametr tekst opisujący błąd
     */
    public BrakTransformacjiOdwrotnejException(String message) {
        super(message);
    }

    /*
     * Konstruktor przyjmujący jako parametr referencje do innego
     * wyjątku, który spowodował błąd
     */
    public BrakTransformacjiOdwrotnejException(Throwable cause) {
        super(cause);
    }
```

```
    /*
     * Konstruktor przyjmujący jako parametr tekst opisujący błąd
     * oraz wyjątek, który spowodował błąd.
     */
    public BrakTransformacjiOdwrotnejException(String message,
                                               Throwable cause) {
        super(message, cause);
    }
}
```

Wyjątki służą do zgłaszania błędów. Wszystkie wyjątki dziedziczą po klasie java.lang.Throwable[5]. Te najczęściej wykorzystywane wyjątki dziedziczą bezpośrednio lub pośrednio po java.lang.Exception.

Dodaj do projektu interfejs Transformacja (Listing 11).

Listing 11 Interfejs Transfsormacja

```
package pl.edu.pg.eti.ksg.po.lab1.transformacje;
public interface Transformacja{
    Punkt transformuj(Punkt p);
    Transformacja getTransformacjaOdwrotna()
                  throws BrakTransformacjiOdwrotnejException;
}
```

Interfejs służy do określenia jakie metody musi implementować klasa aby zapewnić jakąś funkcjonalność. W interfejsie Transformacja zdefiniowane są dwie metody:

```
Punkt transformuj(Punkt p);
```

Metoda ta przyjmuje jeden parametr będący referencją na obiekt klasy Punkt oraz zwraca referencję na obiekt klasy punkt. Jak można się domyślić dokonuje przekształcenia jednego punktu na inny.

```
Transformacja getTransformacjaOdwrotna()
                  throws BrakTransformacjiOdwrotnejException;
```

Ta metoda zwraca referencję do obiektu implementującego interfejs Transformacja oraz w razie problemów zgłasza wyjątek BrakTransformacjiOdwrotnejException. Metoda ta zwraca obiekt reprezentujący transformację odwrotną.

Interfejs można rozumieć jak swoistą klasę abstrakcyjna z samymi czysto wirtualnymi metodami. Jedyna różnica między taką klasą a interfejsem w języku Java jest to, że klasa może implementować wiele interfejsów natomiast może dziedziczyć tylko po jednej klasie[6].

[5] Drugą gałęzią klas dziedziczącą po java.lang.Throwable są klasy pochodne java.lang.Error, które reprezentują poważne błędy maszyny wirtualnej i inne podobne zdarzenia, które nie powinny być obsługiwane przez kod programu.

[6] Dziedziczenie po wielu klasach na raz jest niedozwolone w języku Java

Dodaj do projektu klasę Translacja implementującą interfejs Transformacja (Listing 12).

Listing 12 Klasa Translacja (implementująca przesunięcie)

```
package pl.edu.pg.eti.ksg.po.lab1.transformacje;
/*
 * Klasa Translacja implementuje interfejs transformacja
 */
public class Translacja implements Transformacja{
    private final double dX,dY;

    public Translacja(double dX, double dY) {
        this.dX = dX;
        this.dY = dY;
    }

    @Override
    public Transformacja getTransformacjaOdwrotna() {
        return new Translacja(-dX, -dY);
    }

    @Override
    public Punkt transformuj(Punkt p) {
        return new Punkt(p.getX() + dX, p.getY() + dY);
    }

    public double getdX() {
        return dX;
    }

    public double getdY() {
        return dY;
    }

    @Override
    public String toString() {
        return "Translacja o wektor ("+dX+","+dY+")";
    }
}
```

Dodaj to projektu klasę Skalowanie implementującą interfejs Transformacja (Listing 13).

Listing 13 Klasa Skalowanie

```
package pl.edu.pg.eti.ksg.po.lab1.transformacje;
public class Skalowanie implements Transformacja{

    private final double skalaX,skalaY;

    public Skalowanie(double skalaX, double skalaY) {
        this.skalaX = skalaX;
        this.skalaY = skalaY;
    }
```

```
    /*
     * W przypadku klasy Skalowanie metoda getTransformacjaOdwrotna
     * zgłasza błąd w przypadku braku transformacji odwrotnej
     * (z powodu dzielenia przez 0). Błąd ten jest zgłaszany za
     * pomocą specjalnej instrukcji throw która to powoduje
     * propagacje informacji o błędzie do metod wywołujących
     * do momentu aż maszyna wirtualna natrafi na fragment kodu
     * obsługujący wyrzucony wyjątek
     */
    @Override
    public Transformacja getTransformacjaOdwrotna() throws
BrakTransformacjiOdwrotnejException {
        if(skalaX == 0 || skalaY == 0)
            throw new BrakTransformacjiOdwrotnejException("Brak
transformacji odwrotnej. Przynajmniej jeden z czynników skalowania jest
równy 0.");

        return new Skalowanie(1/skalaX,1/skalaY);
    }

    @Override
    public Punkt transformuj(Punkt p) {
        return new Punkt(skalaX * p.getX(), skalaY * p.getY());
    }

    public double getSkalaX() {
        return skalaX;
    }

    public double getSkalaY() {
        return skalaY;
    }

    @Override
    public String toString() {
        return "Skalowanie "+skalaX+"x oraz "+skalaY+"y";
    }

}
```

Przeprowadź test klas Translacja oraz Skalowanie (Listing 14).

Listing 14 Test klas Translacja oraz Skalowanie

```
package pl.edu.pg.eti.ksg.po.lab1;

import pl.edu.pg.eti.ksg.po.lab1.transformacje.Translacja;
import pl.edu.pg.eti.ksg.po.lab1.transformacje.Skalowanie;
import
pl.edu.pg.eti.ksg.po.lab1.transformacje.BrakTransformacjiOdwrotnejExcepti
on;
import pl.edu.pg.eti.ksg.po.lab1.transformacje.Punkt;
import pl.edu.pg.eti.ksg.po.lab1.transformacje.Transformacja;
```

```
public class Javalab1 {

    public static void main(String[] args)
    {
        /*
         * Konstrukcja językowa try {} catch (...){} służy do
         * obsługi wyjątków. Kod w bloku try jest monitorowany
         * pod kątem wystąpienia wyjątku bądź wyjątków
         * wspomnianych na początku bloku/bloków catch.
         * Jeżeli gdzieś w bloku try wystąpi wyjątek, to sterowanie
         * zostanie natychmiast przeniesione do bloku catch.
         * Tam powinien znajdować się kod obsługujący wyjątek.
         * Może to być np. wypisanie stosu wywołań na wyjście błędów
         * lub zapisanie wyjątku w logach, lub wyrzucenie (zgłoszenie)
         * innego wyjątku lepiej opisującego sytuacje (można załączyć
         * wyjątek który zainicjował to zdarzenie patrz. Konstruktor
         * klasy java.lang.Exception)
         */
        try {
            Punkt p1 = Punkt.E_X;
            System.out.println(p1);
            Transformacja tr = new Translacja(5, 6);
            System.out.println(tr);
            Punkt p2 = tr.transformuj(p1);
            System.out.println(p2);
            Transformacja trr = tr.getTransformacjaOdwrotna();
            System.out.println(trr);
            Punkt p3 = trr.transformuj(p2);
            System.out.println(p3);

        } catch (BrakTransformacjiOdwrotnejException ex)
        {
            ex.printStackTrace();
        }
        System.out.println();

        try
        {
            Punkt p1 = new Punkt(2, 2);
            System.out.println(p1);
            Transformacja tr2 = new Skalowanie(5, 4);
            System.out.println(tr2);
            Punkt p2 = tr2.transformuj(p1);
            System.out.println(p2);
            Transformacja trr2 = tr2.getTransformacjaOdwrotna();
            System.out.println(trr2);
            Punkt p3 = trr2.transformuj(p2);
            System.out.println(p3);
        }
        catch(BrakTransformacjiOdwrotnejException ex)
        {
            ex.printStackTrace();
        }
        System.out.println();
```

```
        try
        {
            Punkt p1 = new Punkt(2, 2);
            Transformacja tr2 = new Skalowanie(5, 0);
            System.out.println(tr2);
            System.out.println(p1);
            Punkt p2 = tr2.transformuj(p1);
            System.out.println(p2);
            Transformacja trr2 = tr2.getTransformacjaOdwrotna();
            System.out.println(trr2);
            Punkt p3 = trr2.transformuj(p2);
            System.out.println(p3);
        }
        catch(BrakTransformacjiOdwrotnejException ex)
        {
            ex.printStackTrace();
        }
        System.out.println();
    }
}
```

Przeanalizuj i uruchom kod. Które bloki try wykonają w całości, a które nie. Dlaczego?

Zadanie 3 (1 punkt)

Stwórz klasę Obrot, która będzie implementowała interfejs transformacja. Klasa ta ma wykonywać obrót na płaszczyźnie wokół punktu (0,0).

Zaprezentuj kod analogicznie tak jak ilustruje to Listing 14.

Obrót na płaszczyźnie wokół punktu (0,0):

$$\begin{cases} x' = x\cos\alpha - y\sin\alpha \\ y' = x\sin\alpha + y\cos\alpha \end{cases}$$

Potrzebne funkcje matematyczne znajduje się w klasie java.lang.Math.

Zadanie 4 (1 punkt)

Stwórz klasę ZlozenieTransformacji, która będzie implementowała interfejs transformacja. Klasa ta ma wykonywać kolejnych n transformacji na danym Punkcie. Użyj do tego tablicy.

Tablica jest to specjalny typ referencyjny języka Java. Tablice można podzielić na te dla typów prostych np.:

```
double[] tab = new double[5];
```

Utworzenie tablicy 5-cio elementowej zawierającej liczby typu double.

Oraz dla typów referencyjnych:

```
Transformacja[] tab = new Transformacja[6];
```

Utworzenie tablicy 6-ciu referencji do obiektów implementujących interfejs Transformacja.

Każda tablica ma wbudowane pole **lenght** w którym zawarta jest jej długość.

Podpowiedź: zastosuj tablice w konstruktorze klasy ZlozenieTransformacji.

Zaprezentuj kod analogicznie tak jak ilustruje to Listing 14. Zastosuj złożenie dwóch różnych transformacji, np. skalowania oraz translacji.