WYŻSZA SZKOŁA BANKOWA
Warszawa

Programowanie aplikacji w Java

Maciej Gowin

Zjazd 2 - dzień 1

Linki

Opis

https://maciejgowin.github.io/wsb-java/

Kod źródłowy przykładów oraz zadań

https://github.com/MaciejGowin/wsb-programowanie-aplikacji-java

Język Java: pętla for

Przy okazji tablic wspomnieliśmy o pętlach, które pozwalają na iterowanie po jej elementach. U swoich podstaw pętle służą do wykonywania danej operacji wielokrotnie.

Wyobraźmy sobie, że chcielibyśmy wypisać na ekran tekst `I like programming` dwadzieścia razy. Zamiast kopiować instrukcję:

```
System.out.println("I like programming");
```

Użylibyśmy czegoś na obraz:

```
EXECUTE 20 times:
    System.out.println("I like programming");
```

Język Java: pętla for

Z pomocą przychodzi pętla `for`, która pozwala na wykonywanie operacji wielokrotnie:

```
for (int i = 0; i < 20; i++) {
    System.out.println("I like programming");
}
```

W naszym przykładzie:

- `int i = 0` jest warunkiem początkowym, od którego rozpoczynamy odliczanie
- `i < 20` jest warunkiem testowym zwracającym `true` lub `false`
  - w przypadku `true` blok kodu jest wykonywany
  - w przypadku `false` wykonywanie pętli jest przerywane
- `i++` jest instrukcją zmieniającą wartość warunku początkowego

Język Java: pętla for

Możemy to uogólnić do:

```
FOR (initial expression; condition; update expression)
    Statement to be executed if condition met
```

Programowanie: przykład 12

Pętla `for` z warunkami oraz różnymi instrukcjami aktualizacyjnymi.

```
public class Main {

    public static void main(String[] args) {
        breakLine();

        int loop1Invocations = 0;
        for (int i = 0; i < 10; i++) {
            loop1Invocations++;
            printIteration("loop1", i);
        }
        printTotal("loop1", loop1Invocations);

        breakLine();

        int loop2Invocations = 0;
        for (int i = 0; i < 10; i = i + 3) {
            loop2Invocations++;
            printIteration("loop2", i);
        }
        printTotal("loop2", loop2Invocations);

        breakLine();
    }

    public static void printIteration(String loopName, int index) {
        System.out.printf("Loop: %s: current index: %d%n", loopName, index);
    }

    public static void printTotal(String loopName, int index) {
        System.out.printf("Loop: %s: invoked: %d times%n", loopName, index);
    }

    public static void breakLine() {
        System.out.println("----------");
    }
}
```

Programowanie: zadanie 10

Zdefiniuj tablicę 5 cen produktów o nazwie `prices` wraz z przypisanymi wartościami. Oblicz sumę cen przed obniżką (`totalPrice`) oraz po obniżce (`totalPriceDiscounted`) zakładając, że każda z cen podlega obniżce równej `10% * indeks ceny w tabeli` . Do obliczeń ceny po obniżce zdefiniuj funkcję `priceAfterDiscount(double price, int discountInPercent)` . Do obliczeń sumy użyj instrukcji `for` . Wynik wypisz na ekran.

Język Java: pętla for

Możliwe jest zdefiniowanie warunku, który będzie zawsze prawdą. Wtedy kod będzie wykonywał się w nieskończoność aż do wyczerpania się pamięci lub przerwania działania przez użytkownika. Podobnie dzieje się w przypadku pominięcia warunku.

```
        for (int i = 0; i >= 0; i++) {
            System.out.println("Run 1");
        }

        for (int i = 0;; i++) {
            System.out.println("Run 2");
        }

        for (;;) {
            System.out.println("Run 3");
        }
```

Język Java: pętla for...each

O pętli `for...each` mówiliśmy przy okazji tablic. Pozwalają one na podgląd wartości w kolekcjach i mają uproszczoną formę w porównaniu do klasycznej pętli `for`.

```
FOR EACH item IN collection
    Statement to be executed
```

Dla przypomnienia:

```
String[] names = {"John", "Peter", "Andrew"};
for (String name: names) {
    System.out.println("Hello " + name);
}
```

Programowanie: przykład 13

Porównanie pętli `for` oraz `for...each`.

```
public class Main {

    public static void main(String[] args) {
        String[] names = {"John", "Peter", "Andrew"};

        for (String name: names) {
            System.out.println("Hello " + name);
        }

        for (int i = 0; i < names.length; i++) {
            System.out.println("Hello " + names[i]);
        }
    }
}
```

Język Java: pętla while

Pętla `while` w założeniu jest podobna do pętli `for`. Pozwala na wielokrotne wykonanie bloku kodu. W tym przypadku jednak kod jest wykonywany, dopóki dany warunek jest spełniony:

```
int i = 0;
while (i < 20) {
    System.out.println("I like programming");
    i++;
}
```

W naszym przykładzie:

- `i < 20` jest warunkiem testowym zwracającym `true` lub `false`
  - w przypadku `true` blok kodu jest wykonywany
  - w przypadku `false` wykonywanie pętli jest przerywane

Język Java: pętla while

Możemy to uogólnić do:

```
WHILE (condition)
    Statement to be executed if condition met
```

Język Java: pętla while

Możemy zdefiniować pętlę nieskończoną, podobnie jak w przypadku pętli `for`. Oba poniższe przykłady mają taki sam efekt:

```
for (;;) {
    System.out.println("I like programming");
}

while (true) {
    System.out.println("I like programming");
}
```

Język Java: pętla do...while

Pętla `do...while` działa podobnie do pętli `while` z jedną małą różnicą. W tym przypadku blok kodu jest wykonany po raz pierwszy przed sprawdzeniem warunku:

```
int i = 0;
do {
    System.out.println("I like programming");
    i++;
} while (i < 1);
```

Język Java: pętla do...while

Możemy to uogólnić do:

```
DO
    Statement to be executed if condition met
WHILE (condition)
```

Programowanie: przykład 14

Pobieranie numerów na standardowym wejściu do momentu wprowadzenia wartości oczekiwanej.

```
import java.util.Scanner;

public class Main {

    private static final int MAGIC_NUMBER = 7;

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        int inputInt = 0;
        do {
            System.out.print("Please specify an integer: ");
            inputInt = input.nextInt();
        } while (inputInt != MAGIC_NUMBER);

        System.out.println("Finally!");
    }
}
```

Język Java: break w pętlach

Język Java uzbraja nas w kolejne wyrażenia, które pozwalają na lepsze sterowanie pętlami. Wyrażenie `break` pozwala na natychmiastowe wykonywanie pętli, nawet jeżeli kolejne warunki będą spełnione. Dla przykładu:

```
        for (int i = 0; i < 10; i++) {
            if (i % 9 == 7) {
                break;
            }
            System.out.println("In " + i);
        }
```

Język Java: break w pętlach

Wyrażenie to jest często używane z nieskończoną pętlą `while` pozwalając na jej przerwanie, gdy dany warunek jest spełniony:

```
    while (true) {
        // Some statements

        if (state == DONE) {
            break;
        }

        // Some statements
    }
```

Przy zagnieżdżonych pętlach wyrażenie to przerywa wykonywanie najbardziej wewnętrznej pętli.

Język Java: break w pętlach

Java pozwala na definiowanie `labeled break`, które służą do przerywania wykonywania pętli i przekazania kontroli do wskazanego miejsca. Choć jest to możliwe, odradza się używania tej konstrukcji.

```
topBreak:
while (i < 1000) {
    while (j < 1000) {
        if (i * j == 4004) {
            break topBreak;
        }
    }
}
```

Język Java: continue w pętlach

Podczas przepływu rozkazów w pętlach może wystąpić konieczność pominięcia wykonania bloku dla pewnych iteracji oraz przejście do kolejnego warunku. Używamy do tego wyrażenia `continue` jak w przykładzie:

```
        for (int i = 0; i < 10; i++) {
            if (i % 9 == 7) {
                continue;
            }
            System.out.println("In " + i);
        }
```

Przy zagnieżdżonych pętlach wyrażenie to przerywa wykonywanie iteracji najbardziej wewnętrznej pętli.

Język Java: continue w pętlach

Java pozwala na definiowanie `labeled continue`, które służą do przerywania wykonywania pętli i przekazania kontroli do wskazanego miejsca. Choć jest to możliwe, odradza się używania tej konstrukcji.

```
topContinue:
while (i < 1000) {
    while (j < 1000) {
        if (i * j == 4004) {
            continue topContinue;
        }
    }
}
```

Programowanie: zadanie 11

Napisz program, który sczytuje liczby wpisane przez użytkownika do momentu wpisania liczby 0. Zsumuj liczby parzyste i nieparzyste oraz wypisz wynik na ekran.

Język Java: instrukcja warunkowa switch

Instrukcja `switch` przypomina swoim zachowanie instrukcję `if...elseif...else`. Pozwala jednak na lepsze uporządkowanie przypadków oraz poprawia czytelność kodu.

Możemy to uogólnić do:

```
SWITCH (value to be compared)
    CASE (value 1):
        Execute for value 1 if condition met
        BREAK;
    CASE (value 2):
        Execute for value 2 if condition met
        BREAK;
    DEFAULT:
        Exexcute if nothing else met
WHILE (condition)
```

Wartość porównywana jest z kolejnymi przypadkami. Kod jest wykonywany, tylko jeżeli dany warunek jest spełniony. W przeciwnym wypadku przy brakującym dopasowaniu wykonany zostanie przypadek domyślny.

Programowanie: przykład 15

Porównanie instrukcji warunkowej `switch` z instrukcją `if...elseif...else`.

```
public class Main {

    public static void main(String[] args) {
        for (int i = 0; i < 4; i++) {
            System.out.printf("=== Testing: %d%n", i);
            System.out.printf("State of switch: %s%n", getStateOfSwitch(i));
            System.out.printf("State of if: %s%n", getStateOfIf(i));
        }
    }

    public static String getStateOfSwitch(int i) {
        String state;
        switch (i) {
            case 0:
                state = "stop";
                break;
            case 1:
                state = "low-speed";
                break;
            case 2:
                state = "top-speed";
                break;
            default:
                state = "unknown";
        }
        return state;
    }


    public static String getStateOfIf(int i) {
        String state;
        if (i == 0) {
            state = "stop";
        } else if (i == 1) {
            state = "low-speed";
        } else if (i == 2) {
            state = "top-speed";
        } else {
            state = "unknown";
        }
        return state;
    }
}
```

Język Java: instrukcja warunkowa switch

Istotne w wyrażeniu `switch` było użycie słowa `break`, które to przerywa działanie całego bloku. Przy jego pominięciu wszystkie wyrażenia po pasującym przypadku zostaną wykonane.

```
        switch (i) {
            case 0:
                state = "stop";
            case 1:
                state = "low-speed";
            case 2:
                state = "top-speed";
                break;
            default:
                state = "unknown";
        }
```

Programowanie: przykład 16

Porównanie instrukcji warunkowych `switch` z instrukcją `break`.

```
public class Main {

    public static void main(String[] args) {
        getStateOfSwitchBreak(0);
        getStateOfSwitchNoBreak(0);
    }

    public static void getStateOfSwitchBreak(int i) {
        switch (i) {
            case 0:
                System.out.println("getStateOfSwitchBreak: index 0");
            case 1:
                System.out.println("getStateOfSwitchBreak: index 1");
            default:
                System.out.println("getStateOfSwitchBreak: index 2");
        }
    }

    public static void getStateOfSwitchNoBreak(int i) {
        switch (i) {
            case 0:
                System.out.println("getStateOfSwitchNoBreak: index 0");
                break;
            case 1:
                System.out.println("getStateOfSwitchNoBreak: index 1");
                break;
            default:
                System.out.println("getStateOfSwitchNoBreak: index 2");
        }
    }
}
```

Język Java: typy prymitywne a obiekty

Język Java dostarcza obiektowych odpowiedników dla typów prostych. Każdy z typów prymitywnych pomoże być reprezentowany przez adekwatną klasę.

Konwersja pomiędzy typami prymitywnymi, a ich obiektowymi odpowiednikami zachodzi automatycznie. Mechanizm ten nazywamy `autoboxing`.

```
public static void test() {
    Integer i = 3;
    print(i);
}

public static void print(int i) {
    System.out.println(i);
}
```

Język Java: typy prymitywne a obiekty

Co ciekawe możemy przeciążać metody na podstawie różnicy typu nawet dla typów, dla których zachodzi `autoboxing`.

```
public static void print(int i) {
    System.out.println("Print for int: " + i);
}

public static void print(Integer i) {
    System.out.println("Print for Integer" + i);
}
```

Język Java: typy prymitywne a obiekty

Typ prymitywny	Odpowiednik obiektowy
boolean	Boolean
byte	Byte
short	Short
int	Integer

Język Java: typy prymitywne a obiekty

Typ prymitywny	Odpowiednik obiektowy
long	Long
double	Double
float	Float
char	Char

Język Java: klasa i konstruktor

Podczas tworzenia obiektów wywoływany jest konstruktor, który odpowiedzialny jest za jego inicjalizację. Możemy go zdefiniować w podobny sposób do tego, który znamy z definicji metody:

```
public class Person {

    public String firstName;

    public Person() {
    }
}
```

Konstruktor w odróżnieniu od klasycznej metody:

- ma nazwę tożsamą z nazwą klasy
- nie zwraca wartości

Język Java: klasa i konstruktor

Konstruktor, podobnie jak metoda, może przyjmować parametry. Usprawnia to inicjalizację obiektu znaną z poprzednich przykładów.

```
public class Person {

    public String firstName;
    public String lastName;

    public Person(String newFirstName, String newLastName) {
        firstName = newFirstName;
        lastName = newLastName;
    }
}
```

Język Java: klasa i konstruktor

W dotychczasowych przykładach nie definiowaliśmy konstruktora. W takim przypadku domyślny konstruktor zostanie automatycznie stworzony. To on jest odpowiedzialny za przypisywanie wartości początkowych.

```
public class Person {
}
```

Programowanie: przykład 17

Konstruktor wieloargumentowy oraz tworzenie obiektów.

```
class Person {
    String firstName;
    String lastName;

    public Person(String newFirstName, String newLastName) {
        firstName = newFirstName;
        lastName = newLastName;
    }

    public String getFullName() {
        return String.format("%s %s", dashOnNull(firstName), dashOnNull(lastName));
    }

    public static String dashOnNull(String value) {
        return value != null ? value: "-";
    }
}

public class Main {

    public static void main(String[] args) {
        Person[] persons = {
                new Person("Jan", "Kowalski"),
                new Person("Andrzej", null),
                new Person(null, "Nowak"),
                new Person(null, null)};

        for (Person person: persons) {
            System.out.printf("Person: %s%n", person.getFullName());
        }
    }
}
```

Język Java: klasa i konstruktor

Konstruktor domyślny

Konstruktor dodawany automatycznie, gdy żaden inny nie zostanie zdefiniowany.

Konstruktor bezargumentowy

Konstruktor zdefiniowany explicite nieposiadający żadnych parametrów.

Konstruktor wieloargumentowy

Konstruktor zdefiniowany explicite posiadający parametry.

Język Java: klasa i konstruktor - przeciążanie

Podobnie jak w przypadku metod, konstruktor może zostać przeciążony na podstawie parametrów przekazanych.

```
public class Person {

    String firstName;
    String lastName;

    public Person(String newLastName) {
        firstName = null;
        lastName = newLastName;
    }

    public Person(String newFirstName, String newLastName) {
        firstName = newFirstName;
        lastName = newLastName;
    }
}
```

Język Java: słowo kluczowe this

W sytuacji, w której chcemy odnieść się do referencji obiektu z jego wnętrza używamy słowa `this`. W skrócie możemy go rozumieć jako `obecny obiekt`. Poniższe przykłady są równoznaczne:

```
public class Person {

    String firstName;
    String lastName;

    public Person(String newFirstName, String newLastName) {
        firstName = newFirstName;
        lastName = newLastName;
    }

    public String getFullName() {
        return firstName + " " + this.lastName;
    }
}
```

Język Java: słowo kluczowe this

Odwołanie do obecnego obiektu jest szczególnie przydatne podczas używania tej samej nazwy zmiennej dla pól klasy oraz parametrów metody, czy też konstruktora. Związane jest to z zasięgiem zmiennej i bez użycia słowa `this` byłoby to niemożliwe.

```
public class Person {

    String name;

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

    public void setName(String name) {
        this.name = name;
    }

    public String getFullName() {
        return "Full name: " + this.name;
    }
}
```

Język Java: słowo kluczowe this

Przy definicji kilku konstruktorów możemy również użyć odwołania do innego konstruktora. W tym przypadku również użyjemy słowa `this`. Jest to przydatne w przypadku gdy konstruktory posiadają bardziej skomplikowaną logikę.

```
public class Person {

    String fullName;

    public Person(String firstName, String lastName) {
        fullName = firstName + " " + lastName;
    }

    public Person(String lastName) {
        this(null, lastName);
    }
}
```

> Wywołanie konstruktora z poziomu innego konstruktora musi być pierwszą instrukcją.

Język Java: słowo kluczowe this

Słowo kluczowe `this` jest używane w dwóch przypadkach:

- referencja do obecnego obiektu i jego pól oraz metod
- wywołanie konstruktora z definicji innego konstruktora

Programowanie: przykład 18

Klasa z kilkoma konstruktorami oraz prześledź jej działanie.

```
class Person {

    String firstName;
    String lastName;

    public Person(String lastName) {
        this(null, lastName);
        System.out.println("Invoking: Person(lastName)");
    }

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
        System.out.println("Invoking: Person(firstName, lastName)");
    }
}

public class Main {

    public static void main(String[] args) {
        System.out.println("== Defined 1");
        Person person1 = new Person("Jan", "Kowalski");

        System.out.println("== Defined 2");
        Person person2 = new Person("Kowalski");
    }
}
```

Język Java: zasięg

Do tej pory w przykładach posługiwaliśmy się zasięgiem `public`. Oznacza to, że odwoływać się do pól i metod może każda inna klasa, z dowolnego miejsca oraz dowolnego innego pakietu.

Java wprowadza koncepcję zasięgu, który pozwala na sterowanie tym, kto i w jaki sposób może odwoływać się do klas, jak i ich poszczególnych składowych. Innymi słowy, definiujemy ich `widoczność`.

Język Java: zasięg

Wyróżniamy następujące zasięgi:

Zasięg	Opis
public	widoczność na każdym z poziomów
protected	widoczność na poziomie pakietu oraz wszystkich klas dziedziczących
private	widoczność na poziomie klasy definiującej
domyślny	widoczność na poziomie pakietu

Język Java: zasięg

```
class Person {
    public String firstName;
    private String lastName;
    protected int yearOfBirth;
    int height;
}
```

Język Java: zasięg

Dobrą praktyką w programowaniu jest ukrywanie szczegółów implementacji oraz udostępnianie jedynie elementów, która powinny być dostępne przez użytkowników danej klasy oraz jej pól i metody.

Bardzo często pola klasy są prywatne, a dostęp do nich jest definiowany poprzez publiczne metody tzw. `settery` oraz `gettery`.

Język Java: hermetyzacja

Do `hermetyzacji` (ang. `encapsulation`) dochodzi podczas grupowania cech oraz zachowań w jednej klasie. Efektem hermetyzacji jest ukrywanie danych oraz szczegółów implementacji przed zewnętrznymi klasami.

Programowanie: zadanie 12

Klasa `Person` z ukrytymi szczegółami implementacji. Zdefiniuj metodę zwracającą:

- pełną nazwę użytkownika na podstawie imienia i nazwiska,
- jego wiek na podstawie roku urodzenia.

Pozwól na zmianę jedynie roku urodzenia użytkownika. Pozostałe pola pozostaw ukryte oraz pozwól na ich zdefiniowanie tylko podczas konstrukcji.

> Do pobrania obecnego roku użyj `LocalDate.now().getYear()` z pakietu java.time.

Język Java: dziedziczenie

Klasy nie są osobnymi bytami. Często posiadają one zbiór podobnych cech oraz zachowań charakterystycznych dla danej grupy.

Dla przykładu:

- Pies jest zwierzęciem domowym, które posiada imię oraz potrafi jeść i spać.
- Kot jest zwierzęciem domowym, które posiada imię oraz potrafi jeść i spać.

Wspólne cechy moglibyśmy zdefiniować dla obu zwierząt w klasie nadrzędnej. Mechanizm ten nazywamy `dziedziczeniem`.

Język Java: dziedziczenie

W przykładzie `Cat` oraz `Dog` dziedziczą po klasie `Animal` automatycznie przejmując zachowania klasy nadrzędnej. Do definicji dziedziczenia używamy słowa kluczowego `extends`.

```
class Animal {
    public void eat() {
    }
}

class Cat extends Animal {
}

class Dog extends Animal {
}
```

> Dana klasa może dziedziczyć tylko po jednej klasie.

Język Java: dziedziczenie

Dziedziczenie realizuje koncepcję relacji `jest` (ang. `is a`). Możemy powiedzieć, że:

- Pies jest Zwierzęciem.
- Kot jest Zwierzęciem.
- Trójkąt jest Figurą Geometryczną.
- Kwadrat jest Prostokątem.

Będziemy mówić, że:

- klasa jest nadklasą (superclass), jeżeli istnieje klasa, która po niej dziedziczy
- klasa jest podklasą (subclass), jeżeli jest klasą, która dziedziczy po innej klasie

> Dziedziczenie nie ogranicza się do jednego poziomu. W strukturze dziedziczenia jedna klasa może być nadklasą, jak i podklasą.

Programowanie: przykład 19

Dziedziczenie wspólnego zachowania.

```
class Animal {
    public void eat() {
        System.out.println("Eating");
    }
}

class Dog extends Animal {
}

class Cat extends Animal {
}

public class Main {

    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();

        Cat cat = new Cat();
        cat.eat();
    }
}
```

Język Java: dziedziczenie i przesłanianie metod

W sytuacji, gdy nadklasa i podklasa posiada zdefiniowaną metodę o tej samej sygnaturze mamy do czynienia z `przesłanianiem metod` (ang. `method overriding`). W takim przypadku zostanie wywołana metoda specyficzna dla danej klasy (jeżeli taka istnieje).

Programowanie: przykład 20

Nadpisywanie metod.

```
class Animal {
    public void voice() {
        System.out.println("Voice");
    }
}

class Dog extends Animal {
    public void voice() {
        System.out.println("Barking");
    }
}

class Cat extends Animal {
}

public class Main {

    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.voice();

        Cat cat = new Cat();
        cat.voice();
    }
}
```

Język Java: dziedziczenie a własności

Możemy też zdefiniować wspólne cechy na poziomie klasy nadrzędnej. W naszym przykładzie wspólną cechą może zostać `name`. Aby klasy podrzędne miały bezpośredni dostęp do pola `name` musimy je zdefiniować z zasięgiem `protected` (dla klas z innego pakietu) lub domyślnym, jeżeli podklasy znajdują się w tym samym pakiecie.

```
class Animal {
    protected String name;
}
```

Język Java: dziedziczenie i słowo kluczowe super

Podobnie jak w przypadku słowa kluczowego `this` słowo kluczowe `super` definiuje referencję obiektu. Tutaj do obiektu nadrzędnego. Dzięki niemu mamy bezpośredni dostęp do pół oraz metod klasy nadrzędnej z poziomu klasy podrzędnej.

```
class Dog extends Animal {
    public void getName() {
        return "Dog: " + super.name;
    }
}
```

Język Java: dziedziczenie i słowo kluczowe super

Słowo kluczowe `super` ma jeszcze jedno zastosowanie. Możemy go użyć w przypadku wywołania konstruktora klasy nadrzędnej.

```
class Animal {
    private String name;

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

class Dog extends Animal {

    public Dog(String name) {
        super(name);
    }
}
```

> Podczas konstrukcji klasy podrzędnej klasa nadrzędna musi zostać poprawnie zainicjalizowana, stąd też konieczne jest wywołanie jej konstruktora.

Programowanie: zadanie 13

Stwórz klasę `animal.Animal` definiującą nazwę zwierzęcia inicjalizowaną przy pomocy konstruktora oraz metody:

- `String getName()` służącą do pobrania nazwy zwierzęcia
- `void voice()` służącą do imitacji odgłosu zwierzęcia
- `void eat()` służącą do imitacji procesu jedzenia

Zdefiniuj klasy dziedziczące po `animal.Animal`:

- `animal.dog.Dog`: nadpisującą `void voice()`
- `animal.cat.Cat`: nadpisującą `void voice()`

Przetestuj zdefiniowane klasy.

Język Java: polimorfizm

W programowaniu obiektowym przez polimorfizm rozumiemy dynamiczny wybór metody wywołania. W ogólnym rozumieniu polimorfizm to traktowanie różnych podtypów danego typu w taki sam sposób.

Dla przykładu język automatycznie dobierze wywołanie odpowiedniej metody na podstawie typu.

```
Dog dog = new Dog();
dog.voice();

Animal animal = new Dog();
animal.voice();
```

Język Java: hierarchia dziedziczenia

W przykładach definiowaliśmy hierarchię dziedziczenia, gdzie klasy mogła dziedziczyć po innej.

Jeżeli natomiast klasa nie dziedziczy explicite po żadnej klasie to de facto jest automatycznie podklasą `Object`.

Wynika z tego, że klasa `Object` jest korzeniem w hierarchii dziedziczenia i każda z klas automatycznie przejmuje wszystkie jej własności oraz zachowania.

Język Java: kompozycja

Wspomnieliśmy już, że dziedziczenie realizuje koncepcję relacji `jest` (ang. `is a`). W przykładach używaliśmy też innego mechanizmu służącego do ponownego użycia zdefiniowanych już klas i obiektów, czyli kompozycji.

Kompozycja realizuje koncepcję `zawiera` (ang. `has a`). Możemy powiedzieć, że:

- Samochód zawiera silnik i koła.
- Kot zawiera głowę, nogi, tułów i ogon.

Język Java: dziedziczenie a kompozycja

Dziedziczenie stosowane jest w sytuacjach, gdy pomiędzy klasami zachodzi relacja "uogólnienie - specjalizacja".

Kompozycja stosowana jest w sytuacjach, gdy pomiędzy klasami zachodzi relacja "całość - składowa".

Zwykle przy tworzeniu struktury klas używamy dwóch mechanizmów równocześnie. Dla przykładu:

```
Dom jest budynkiem (dziedziczenie) oraz zawiera: okna, drzwi, ściany i dach (kompozycja).
```

```
class House extends Building {
    private Window[] windows;
    private Door[] doors;
    private Wall[] walls;
    private Roof roof;
}
```

Język Java: instanceof

Operatora `instanceof` używany do sprawdzenia, czy dany obiekt jest danego typu. Jest to szczególnie przydatne podczas pracy z dziedziczeniem oraz polimorfizmem.

```
Dog dog = new Dog();
boolea isDog = dog instanceof Dog;
boolean isAnimal = dog instanceof Animan;
```

Programowanie: przykład 21

Przykład wykorzystania operatora `instanceof`.

```
class Animal {
}

class Dog extends Animal {
}

class Cat extends Animal {
}

public class Main {

    public static void main(String[] args) {
        Dog dog = new Dog();
        Animal animal = new Dog();

        System.out.printf("dog instanceof Dog: %s%n", dog instanceof Dog);
        System.out.printf("dog instanceof Animal: %s%n", dog instanceof Animal);
        System.out.printf("animal instanceof Dog: %s%n", animal instanceof Dog);
        System.out.printf("animal instanceof Animal: %s%n", animal instanceof Animal);
        System.out.printf("animal instanceof Cat: %s%n", animal instanceof Cat);
    }
}
```

Język Java: klasy abstrakcyjne

W przypadku dotychczasowych definicji klas nadrzędnych zawsze dostarczaliśmy domyślną implementację każdej z metod. Jeżeli jest to niemożliwe z pomocą przychodzą `metody abstrakcyjne`.

Metodę abstrakcyjną definiujemy z pominięciem jej ciała. Każda z podklas jest odpowiedzialna za dostarczenie jej implementacji.

```
abstract Animal {
    abstract void voice();
}
```

Co istotne, jeżeli klasa posiada choć jedną metodę abstrakcyjną sama też musi zostać zdefiniowana jako `abstrakcyjna`.

Język Java: klasy abstrakcyjne

Głównym celem klas abstrakcyjnych jest definiowanie wspólnego zachowań oraz ukrywanie niepotrzebnych szczegółów. Pomijamy to, co złożone, udostępniając tylko wysokopoziomowy interfejs.

Język Java: klasy abstrakcyjne

Z klasami abstrakcyjnymi związane jest kilka założeń

- metoda abstrakcyjna nie posiada ciała
- nie możemy stworzyć obiektów klas abstrakcyjnej
- jeżeli klasa posiada choć jedną metodę abstrakcyjną sama staje się abstrakcyjna
- każda podklasa dziedzicząca po klasie abstrakcyjnej musi implementować wszystkie metody abstrakcyjne
- jeżeli podklasa nie implementuje metod abstrakcyjnych sama musi zostać zdefiniowana jako abstrakcyjna

Język Java: interfejsy

Interfejs to nic innego jak klasa abstrakcyjna nieposiadająca żadnych właściwości niestatycznych z wszystkimi metodami abstrakcyjnymi. Definiujemy go przy pomocy słowa kluczowego `interface`. W tym przypadku pomijamy słowo `abstract`.

```
interface Animal {
    public static final int MAX_AGE;
    public void voice();
}
```

> Metody interfejsu automatycznie uzyskują zasięg `public`. Każda z własności staje się automatycznie `public static final`.

Język Java: interfejsy

Klasy mogą `imlementować` dany `interfejs`. W przeciwieństwie do klas abstrakcyjnych gdzie są one `rozszerzane`.

```
class Dog implements Animal {
    public void voice() {
        System.out.println("Barking");
    }
}
```

> Co ważne jedna klasa może implementować więcej niż jeden interfejs. W przeciwieństwie do dziedziczenia gdzie dziedziczyć możemy tylko po jednej klasie.

Język Java: interfejsy

Co więcej, interfejsy podlegają rozszerzeniom przy użyciu słowa `extends`. Interfejs może rozszerzać więcej niż jeden interfejs.

```
interface Animal {
    void eat();
}

interface Mammal extends Animal {
    void breath();
}

class Dog implements Mammal {
    public void eat() {
        System.out.println("Eating");
    }

    public void breath() {
        System.out.println("Breathing");
    }
}
```

Programowanie: przykład 22

Użycie klas abstrakcyjnych oraz interfejsów. Kod źródłowy dostępny na stronie.

```
interface Animal { ...
interface Mammal extends Animal { ...
interface Nameable { ...
abstract class Pet implements Mammal, Nameable { ...
class Dog extends Pet { ...
class Cat extends Pet { ...
```