Podstawy Java SE

Hello

Tomasz Lisowski

Software developer, JIT Solutions

IT trainer

Agenda

- powtórka
- operacje na typach tekstowych
- dziedziczenie
- wyjątki
- interfejsy
- klasy abstrakcyjne

abstrakcja

Thinking in java - Bruce Eckel

- *wszystko jest obiektem*
  - obiekt przechowuje dane, wykonuje operacje
- *program, to zbiór obiektów, które mówią sobie nawzajem co robić*
- *każdy obiekt posiada własną pamięć, na którą składają się inne obiekty*
- *każdy obiekt posiada swój typ*
  - jest instancją pewnej klasy (szablonu)
- *wszystkie obiekty danego typu zachowują się tak samo*

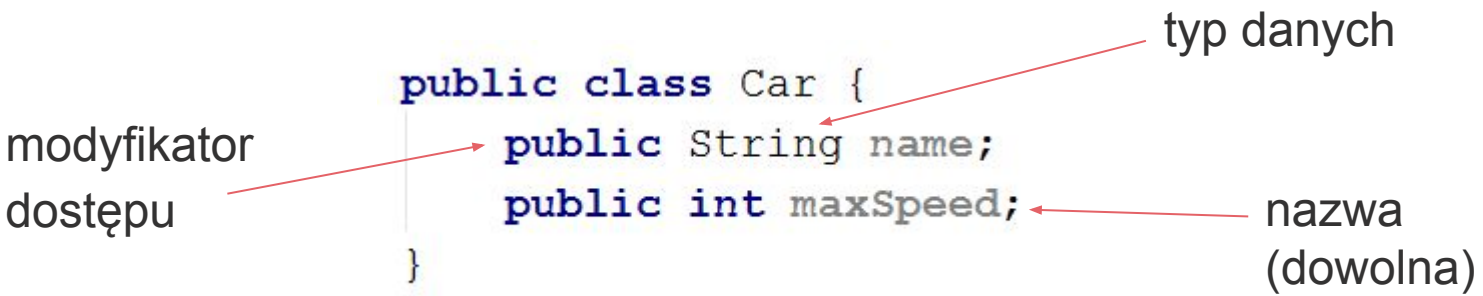
klasa

- podstawowy element składowy aplikacji
- typ danych
- szablon - konkretna definicja pewnego ‘bytu’
- zawiera pola i metody

```
public class Main {
    public static void main(String[] args) {
        System.out.println("infoShareAcademy - Java SE");
    }
}
```

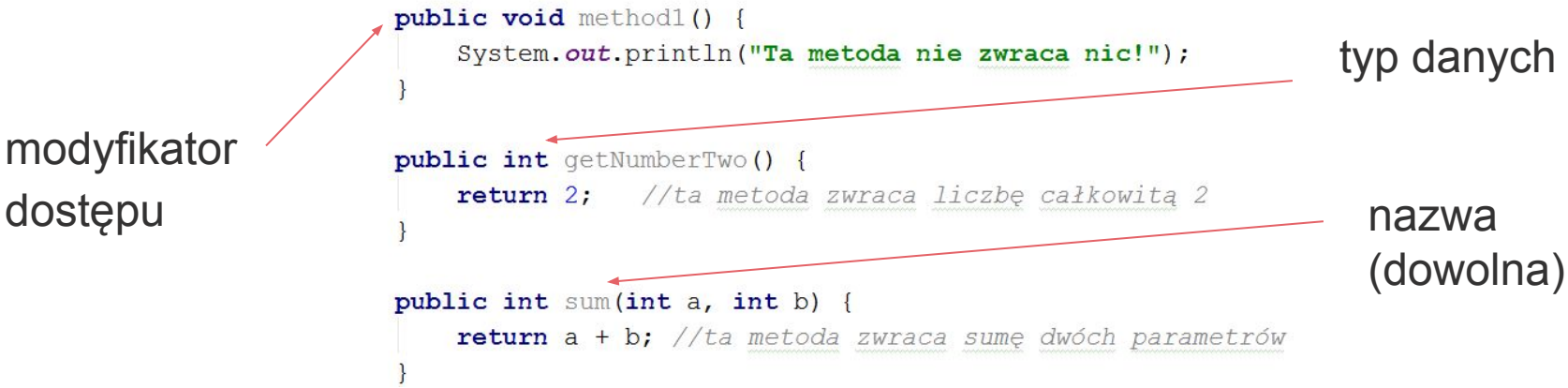
klasa - pole

- dana cecha naszej klasy
- może reprezentować dowolny typ (klasę)
- może być ich wiele lub wcale



klasa - metoda

- funkcjonalność naszej klasy (tu logika - nie piszemy kodu poza metodami!)
- zwracają jakiś typ danych (lub nic - wtedy ‘zwracamy’ *void*)
- mogą przyjmować parametry



obiekt

- instancja klasy
- konkretny obiekt na podstawie definicji klasy

```
public class Car {
    public String name;
    public int maxSpeed;
}
```

```
Car myCar = new Car();
```

obiekt

wywołanie pól i metod

```
Car myCar = new Car();
//gdy pole name jest public
System.out.println(myCar.name);
//gdy pole name jest private
System.out.println(myCar.getName());
//wywołanie metody printName() z klasy Car, na obiekcie myCar
myCar.printName();
```

Czy obiekty są równe?

== *vs* **equals**

- instrukcje porównania
- == porównuje **referencję** (przestrzeń pamięci)
- *equals()* porównuje **wartość** dwóch pól

- *domyślna implementacja *equals()* z klasy *Object*

== *vs* **equals**

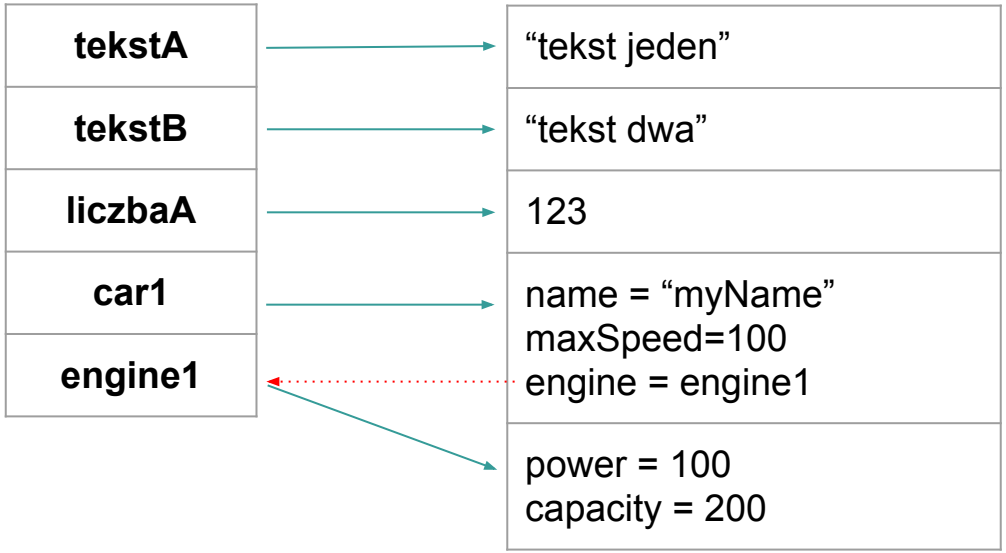
- **equals()** to metoda klasy Object

  jeśli obiekty są równe to muszą mieć ten sam hashCode

  jeśli obiekty mają ten sam hashCode to nie muszą być równe

- nadpisanie metody **hashCode()**
- kontrakt **hashCode()** ↔ **equals()**

pamięć


tekstA
tekstB
liczbaA
car1
engine1
“tekst jeden”
“tekst dwa”
123
name = “myName”
maxSpeed=100
engine = engine1
power = 100
capacity = 200

if else

- podstawowa operacja – instrukcja wyboru
- if = jeżeli
- *jeżeli warunek jest spełniony, to wykonaj instrukcje*

```
double wynik = liczbaA/liczbaB;

if (wynik > 0) {
    return "Liczba dodatnia";
}
```

if else

- warunek *if* można łączyć z *else*
- *else* wykonywane gdy pierwszy warunek nie jest spełniony
- można zagnieżdżać i/lub łączyć instrukcje *if* - *else*

```
if (wynik > 0) {
    return "Liczba dodatnia";
}
else if (wynik == 0) {
    return "Liczba 0";
}
else {
    return "Liczba ujemna";
}
```

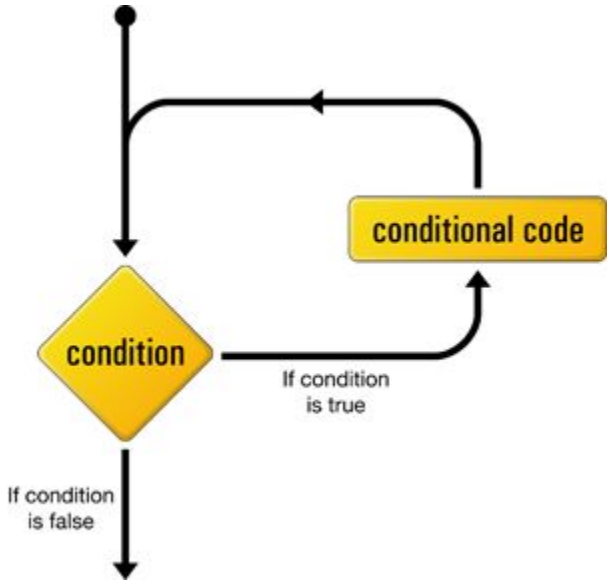
switch

- “wielowarunkowy if”
- switch pobiera parametr i sprawdza dowolną liczbę warunków

```
switch(liczba){
  case 1:
    jakieś_instrukcje_1;
    break;
  case 2:
    jakieś_instrukcje_2;
    break;
  ...
  default:
     instrukcje, gdy nie znaleziono żadnego pasującego przypadku
}
```

pętle

- podstawowa operacja – cykliczne wykonanie danych instrukcji
- niewiadoma ilość wykonań
- .. lub ściśle określona
- można przerwać lub pominąć dany obieg



while

- wykorzystywana, gdy nie znamy ilość obiegów pętli
- .. ale znamy warunek jej zakończenia
- pętla while może wykonać się nieskończenie wiele razy
- albo wcale, gdy warunek już na starcie nie jest spełniony

```
int liczba = -5;
while(liczba < 0) {
    liczba++; //liczba = liczba + 1;
}
```

do..while

- inna wersja pętli *while*
- pętla *do..while* zawsze wykona się co najmniej jeden raz

```
int liczba = 5;
do {
    liczba++; //liczba = liczba + 1;
} while(liczba < 0);
```

for

- zazwyczaj znamy liczbę iteracji w pętli
- 3 parametry:
  - wyrażenie początkowe → np. **int i = 0**
  - warunek → np. **i < 5**
  - modyfikator → np. **i++**

```
for (int i = 0; i < 5; i++) {
    System.out.println("i: " + i);
}
```

pakiety

- klasy pogrupowane w pakiety
- struktura hierarchiczna
- pakiety –> katalogi, klasy –> pliki
- implementacja klasy znajduje się w jakimś pakiecie
- informuje o tym instrukcja package

np. klasa znajduje się w pakiecie ***infoshareacademy***, który znajduje się w pakiecie ***com***

```
package com.infoshareacademy;
```

modyfikatory dostępu

- słowa kluczowe określające poziom dostępności pól/metod innym klasom
- **public** – dostęp do elementu dla wszystkich klas
- **protected** – dostęp tylko dla klas dziedziczących lub z tego samego pakietu
- **private** – brak widoczności elementów poza klasą
- default – dostęp pakietowy, nie istnieje takie słowo kluczowe
  package-private -> *publiczne w pakiecie, prywatne na zewnątrz*
- dobra praktyka – wszystkie pola prywatne

Tablica

Tablica

- struktura gromadząca uporządkowane dane
- tablice jedno lub wielowymiarowe
- wielkość jest stała (!)
- odwołanie do elementu po indeksie

typ[] nazwaTablicy = new typ[liczbaElemenetów]

Tablica

- tworzenie i odczyt danych z tablicy

```
int[] tab = new int[3];
int[] tab2 = {1,2,3};
```

```
int number = tab[1];
int number2 = tab2[1];
```

Tablica

- przypisanie wartości dla elementu tablicy

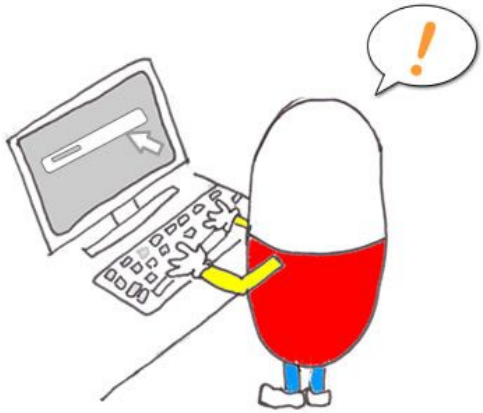
```
tab[1] = 2;

tab[i] = 2;
```

Tablice

ćwiczenie 19

- stwórz metodę przyjmującą parametr typu int
- wewnątrz metody stwórz tablicę 10-elementową typu int
- uzupełnij tablicę kolejnymi liczbami całkowitymi, zaczynając od tej podanej w parametrze
- wypisz wszystkie elementy tablicy

np. parametr = 4
tablica: 4,5,6…,13



Operacje na typach tekstowych

String
tworzenie

```java
//przypisanie wartości (by literal)
String s1 = "java";

//tablica pojedynczych znaków
char[] chars = {'i', 's', 'a'};

//zamiana tablicy znaków na String
String s2 = new String(chars);

//użycie słowa kluczowego new
String s3 = new String( original: "infoShare");
```

String
porównanie

```
String s1 = "INFOShare";
String s2 = new String( original: "infoShare");

System.out.println(s1.equals(s2));
System.out.println(s1.equalsIgnoreCase(s2));
System.out.println(s1 == s2);
System.out.println(s1.compareTo(s2));
System.out.println(s1.compareToIgnoreCase(s2));
System.out.println(s2.compareTo(s1));
```

StringBuilder

- specjalny builder do tworzenia Stringów
- kolejne Stringi dodajemy za pomocą metody *append*()
- udostępnia metody do manipulacji budowanego tekstu
- wynik możemy zapisać do zmiennej String (metoda *toString()*)

StringBuilder

```
String s1 = "info" + "Share" + "Academy";

StringBuilder stringBuilder = new StringBuilder();
stringBuilder.append("info");
stringBuilder.append("Share");
stringBuilder.append("Academy");

String s2 = stringBuilder.toString();
String s3 = stringBuilder.reverse().toString();
```

String
metody

```
String s = "  InfoShare  ";

s.toUpperCase(); //  INFOSHARE
s.toLowerCase(); //  infoshare
s.trim(); //InfoShare
s.startsWith("In"); //false
s.endsWith("are"); //false
s.charAt(5); //o
s.length(); //13
String.valueOf(10); //10
s.replace( target: "Share",  replacement: "Academy"); //  InfoAcademy
```

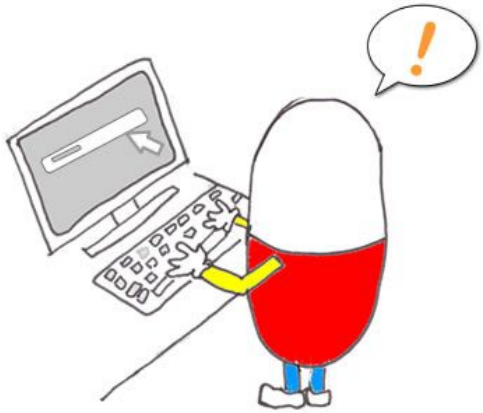
String
metody

```
String s = "string:separate:by:colons";
String[] sArray = s.split( regex: ":");

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

String

ćwiczenie 20

- pobierz z klawiatury dowolny tekst (jedna linia);
- w tekście mogą być kropki, ale nie muszą
- stwórz metodę, która zwróci ilość kropek w tekście
- wypisz na konsolę wyliczoną ilość kropek



OOP

OOP

- **polimorfizm** - “wielopostaciowość”
- “samochód jest pojazdem”
- dany typ, może rozszerzać inny obiekt i udostępniać metody obydwu typów

```
Part part1 = new Wheel();
Part part2 = new Door();
```

OOP

- **dziedziczenie**
- tworzy hierarchię klas
- współdzielenie funkcjonalności między klasami
- oprócz własnych atrybutów, obiekt posiada te pochodzące z klasy nadrzędnej/bazowej

```
Part part1 = new Wheel();
Part part2 = new Door();
```

OOP

- **abstrakcja**
- obiekt jako model “wykonawcy”
- wykonanie pracy, bez ujawniania implementacji

np. połączenie z bazą danych, niezależnie od silnika bazy
dbDriver.connectToDB()

OOP

- **hermetyzacja (enkapsulacja)**
  - ukrywanie implementacji przez obiekt
  - ukrywanie pewnych składowych (pól, metod) tak, aby były dostępne tylko metodom wewnętrznym klasy
  - “wszystkie pola są prywatne”

przeciążanie

- ang. ***overloading***
- mechanizm pozwalający na tworzenie metod o tej samej nazwie
- ..ale różniących się typem lub ilością parametrów
- konstruktory również mogą być przeciążane
- pułapka automatycznego rzutowania (która metoda ma się wykonać?)

przeciążanie

```
public class Calculator {

    public int add(int a, int b) {
        return a+b;
    }

    public int add(int a, int b, int c) {
        return add(a, b) + c;
    }

    public double add(double a, double b) {
        return a+b;
    }

    public double add(double a, double b, double c) {
        return add(a, b) + c;
    }
```

nadpisanie (przesłanianie)

- ang. ***overriding***
- inaczej nadpisanie
- mechanizm pozwalający modyfikować metodę klasy bazowej
- używany w celu stworzenia specyficznej implementacji
- przeładowane metody muszą mieć taką samą strukturę jak bazowe
- oraz posiadać adnotację @Override

nadpisanie (przesłanianie)

- metody *hashCode(), equals(), toString()* są metodami klasy *Object* i mogą być nadpisane w każdej innej klasie
- nie można przesłaniać metod statycznych

```
@Override
public String toString() {
    return "someString";
}
```

Dziedziczenie

dziedziczenie

- podstawowy mechanizm programowania obiektowego
- przekazanie cech innym klasom
- klasa potomna ma cechy klasy bazowej + swoje własne
- klasa potomna może rozszerzać tylko jedną klasę

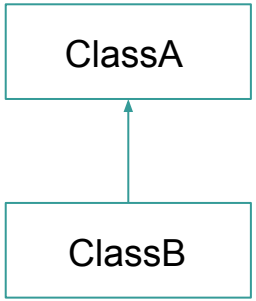
```
class Class extends OtherClass {

}
```

dziedziczenie

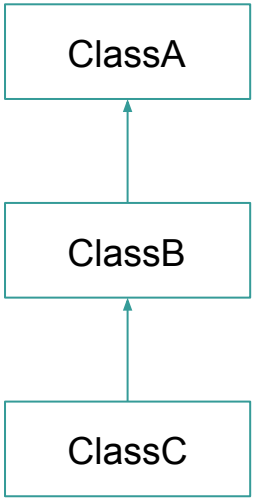
```
class Vehicle {
    String name;
    Date productionDate;
}

class Car extends Vehicle {
    int numberOfWheels;
    int enginePower;
}
```

```
Car myCar = new Car();
myCar.setName("name");
myCar.setEnginePower(9001);
```

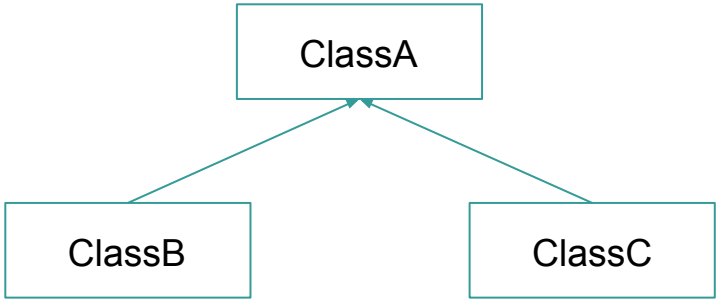
dziedziczenie



Single



Multilevel

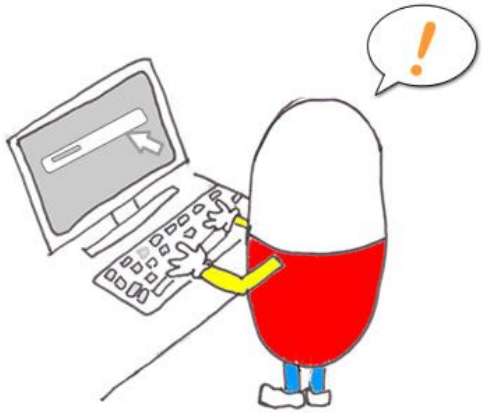


Hierarchical

Dziedziczenie

ćwiczenie 21

- stwórz klasę nadrzędną *Vehicle*, która posiada pole liczbowe *id* oraz metodę *printId()*
- niech klasa *Car* dziedziczy po klasie *Vehicle*
- ustaw wartość pola *id* dla obiekty typu *Car*
- stwórz obiekt typu *Car* i obiekt typu *Vehicle*
- ustaw wartość pola *id* dla obydwu obiektów
- wywołaj metodę *printId()* dla obydwu obiektów
- porównaj dostępne metody



Wyjątki

Wyjątki

- mechanizm pozwalający wyłapywać błędy
- umieszczenie zestawu instrukcji w bloku try..catch

```
try {
    // wykonywany kod, który może powodować wyjątek
} catch (Exception e){
    // zachowanie w przypadku wystąpienia wyjątku
} finally {
    // zachowanie po wykonaniu try lub catch
}
```

Dziedziczenie

ćwiczenie 22

- pobierz z konsoli (Scanner) wartość liczby całkowitej
- łap wyjątki, gdy ktoś poda niepoprawny parametr (np. tekst)
- pobieraj wartość do momentu, aż będzie liczbą całkowitą

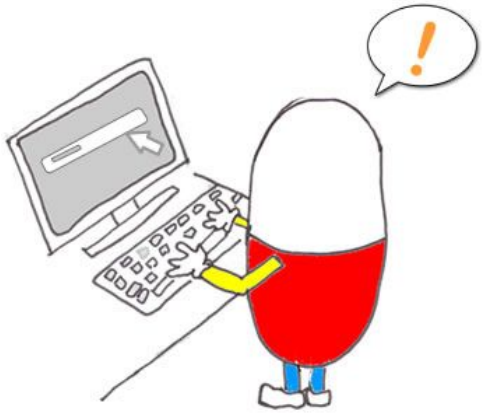
np.

podaj liczbę:

1,3

abcd

11



Wyjątki

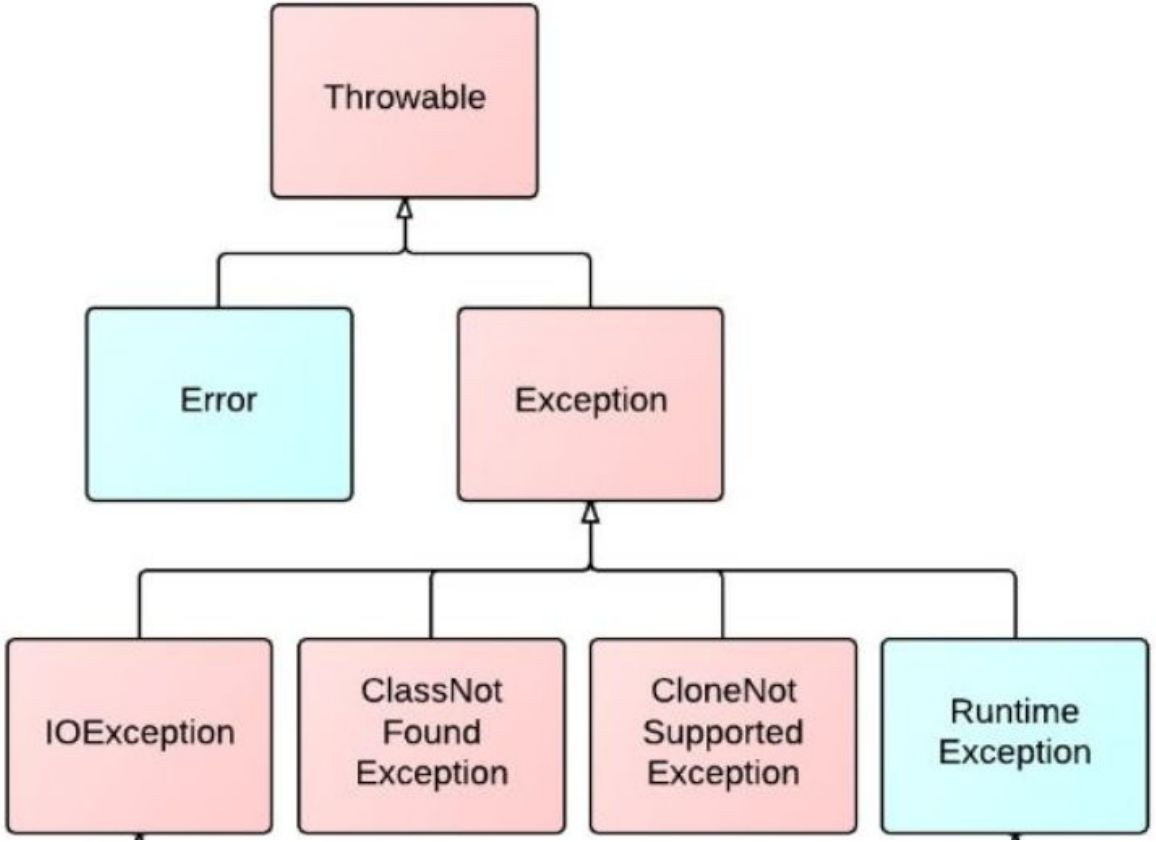
- metody mogą *rzucać* wyjątki
- słowo kluczowe *throws*

```
        try {
            method();
        } catch (Exception e) {
            e.printStackTrace();
        }
}

static void method() throws Exception {
    System.out.println("metoda");
}
```

Wyjątki

- tzw. **Checked Exception** - metoda **musi** deklarować możliwość rzucenia wyjątku (np. IOException)
- **Runtime Exception** - metoda **nie musi** deklarować możliwość rzucenia wyjątku (np. NullPointerException)
- unikaj tworzenia własnych wyjątków
- złapany wyjątek powinno się zawsze obsłużyć

Wyjątki
hierarchia



Interfejsy

Interfejsy

- szablon klasy
- definiuje metody, które klasa musi implementować (wszystkie metody interfejsu)
- klasa może implementować wiele interfejsów

```
public interface Vehicle {
    void run(int velocity);
    void stop();
}
```

Interfejsy

- wszystkie metody są domyślnie publiczne
- wszystkie pola są domyślnie *public static final*
- interfejsy mogą rozszerzać tylko inne interfejsy
- interfejsy nie mogą implementować innych interfejsów
- deklaracja za pomocą słowa *interface*

Interfejsy

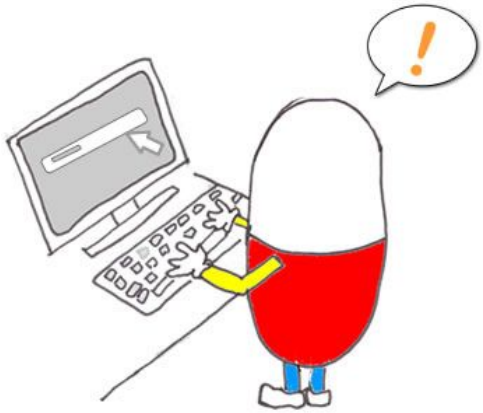
```
public class Car implements Vehicle {
    @Override
    public void run(int velocity) {}

    @Override
    public void stop() {}
```

```
public class Car implements Vehicle, Property {
```

Interfejsy

ćwiczenie 23

- zmień klasę *Vehicle* w interfejs
- usuń pole *id* i metodę *printId()*
- dodaj metodę *printName()* w *Vehicle*
- klasa *Car* powinna tym razem **implementować** *Vehicle*
- przetestuj program



Interfejsy

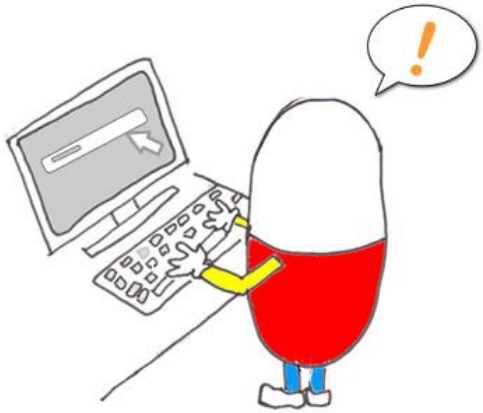
- metody domyślne (mogą posiadać implementację)
- słowo kluczowe **default**

```
default void defaultImplementation() {
    System.out.println("some implementation");
}
```

Interfejsy

ćwiczenie 24

- rozbuduj interfejs *Vehicle* o metodę domyślną
- wykorzystaj tę metodę na obiektach typu *Car*
- czy można nadpisać metodę domyślną?



Klasy abstrakcyjne

Klasy abstrakcyjne

- inny sposób tworzenia abstrakcji
- mogą posiadać metody abstrakcyjne (bez implementacji)
- mogą posiadać zwykłe metody
- klasy rozszerzające muszą implementować wszystkie metody abstrakcyjne
- nie można tworzyć instancji takiej klasy (!)

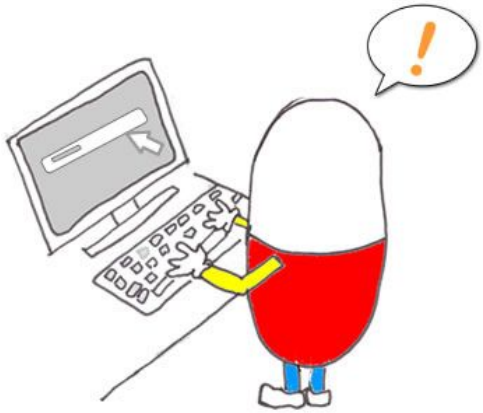
Klasy abstrakcyjne

```
public abstract class AbstractCar {
    public static final String CODE = "ISA";
    private String name;

    public abstract void run(int velocity);

    public void showName() {
        System.out.println(name);
    }
}
```

Klasy abstrakcyjne

ćwiczenie 25

- stwórz klasę *AbstractCar*
- stwórz w niej jedną metodę abstrakcyjną i jedną zwykłą
- klasa *Car* powinna tym razem **rozszerzać** *AbstractCar*
- przetestuj program



Klasy abstrakcyjne *vs* interfejsy

- **można dziedziczyć tylko z jednej klasy**
- **ale implementować wiele interfejsów**
- zmienne mogą istnieć tylko w klasach


JAVA JAVA JAVA JAVA
MOAR MOAR MOAR MOAR

Parametry metod

- możliwość stosowania zmiennej liczby parametrów

gdy nie znamy ilości, np. wczytywanie z klawiatury x imion

```
    method( ...names: "Andrzej", "Stefan");
    method( ...names: "Andrzej", "Mariusz", "Bogdan", "Domino");
}

private static void method(String... names) {
    for (String name : names) {
        System.out.println(name);
    }
```

Klasy wewnętrzne

- możliwość tworzenia klasy wewnątrz klasy
- gdy klasa wewnętrzna nie ma sensu istnienia bez klasy zewnętrznej i są one ściśle powiązane
- ukrywanie implementacji
- ma dostęp do prywatnych pól klasy otaczającej

Klasy wewnętrzne

```
public class OuterClass {
    class InnerClass {}

    public InnerClass initializeInnerClass() {
        return new InnerClass();
    }
```

Klasy wewnętrzne

```
OuterClass outerClass = new OuterClass();

OuterClass.InnerClass innerClass
        = outerClass.initializeInnerClass();

OuterClass.InnerClass innerClass2
        = outerClass.new InnerClass();
```

Final

- oznacza niezmienność elementu
- zmiennym finalnym można tylko raz przypisać wartość
- klasa oznaczona jako ***final*** nie może być dziedziczona
- metoda oznaczona jako ***final*** nie może być implementowana w klasie pochodnej

Static

- zmienne i metody statyczne istnieją zawsze
- nawet gdy nie została utworzona instancja klasy
- konstruktory i interfejsy **nie mogą** być statyczne
- w metodach statycznych nie można odwoływać się do zmiennych nie statycznych
- stałe definiujemy poprzez static final

```
private static final String CONSTANT_STRING = "some constant";
```

Files / Paths

- klasy do operacji na plikach
- **Path** - reprezentuje ścieżkę do pliku lub katalogu w systemie
- **Paths** - służy do tworzenia instancji obiektu **Path**
- **Files** - służy do operacji na plikach

  np. *copy(), createFile(), readAllLines()*

Files / Paths

- **Files** i **Paths** zamiast starych klas jak np. **Reader**, **Writer**
- tworząc obiekt **Path**, buduj ścieżkę ‘krok po kroku’;
  zamiast: **Paths.get(“/home/user/dir”)**;
  użyj: **Paths.get(“home”, “user”, “dir”)**;
  pozwoli to na uniknięcie problemu z separatorem na różnych systemach

Podstawy JSE

materiały

- https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html
- http://www.samouczekprogramisty.pl/porownywanie-obiektow-metody-equals-i-hashcode-w-jezyku-java/
- https://www.samouczekprogramisty.pl/interfejsy-w-jezyku-java/
- https://www.samouczekprogramisty.pl/wyjatki-w-jezyku-java/

Pytania?

Thanks!

Q&A

tomasz.lisowski@protonmail.ch