Podstawy Java SE

Hello!

Tomasz Lisowski

Software developer, JIT Solutions
IT trainer

Agenda

- równość obiektów
- kolekcje
- modyfikatory dostępu
- interfejsy
- wyjątki
- strumienie
- wątki

Java SE
Ćwiczenie

- stwórz obiekt menu z wartością *number* = 1
- napisz metodę 2 parametrową, która:
  - przyjmuje parametry boolean oraz int
  - w zależności od flagi dodaje lub odejmuje wartość z parametru od wartości number

Java SE
Scanner

- podstawowe pobranie danych od użytkownika
- obiekt korzysta ze strumienia wejściowego:
  Scanner scanner = new Scanner(System.in);
- popularne metody:
  - nextLine()
  - nextInt()
  - nextDouble()

Java SE
Ćwiczenie

- wyświetl informację o dostępnych opcjach (enum)
- pobierz opcję z klawiatury
- ustaw odpowiednią wartość enum w zależności od podanej liczby
- jeżeli błędna wartość to wyświetl informację i spróbuj pobrać ponownie
- podanie wartości 0 przerywa działanie

Czy obiekty są równe?

Java SE
== vs equals

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

```
String tekstA = "tekst";
String tekstB = "tekst";

if (tekstA == tekstB) {
    System.out.println("warunek == prawdziwy");
}

if (tekstA.equals(tekstB)) {
    System.out.println("warunek equals prawdziwy");
}
```

Java SE
== vs equals

- equals() to metoda klasy Object
- wykorzystuje hashCode obiektu

- *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()

Java SE
Ćwiczenie

- stwórz 2 stringi o takiej samej wartości
- porównaj je za pomocą instrukcji *if* i operatorów:
  - ==
  - equals()
- wypisz ich hashCode

Podstawowe kolekcje

Java SE
set

- elementy nie mają przyporządkowanego indeksu
- dostęp za pomocą iteratora
- obiekty w zbiorze **nie mogą** się powtarzać

- **HashSet** – podstawowa implementacja, wykorzystuje hashCode()
- **TreeSet** – przechowuje elementy w postaci drzewa

Java SE
set

```
Set<String> zbior = new HashSet<String>();
zbior.add("pierwszy");
zbior.add("drugi");
for (String ciagZnakow : zbior) {
    System.out.println(ciagZnakow);
}
```

Java SE
Ćwiczenie

- stwórz kilka obiektów, w tym 2 równe
- dodaj je do kolekcji Set
- wypisz wszystkie elementy tej kolekcji

Java SE
lista

- każdy element ma przyporządkowany indeks
- obiekty mogą się powtarzać
- możemy odwołać się do konkretnego elementu po indeksie
- podstawowe operacje:
  add()
  get(indeks)

Java SE
lista

- **ArrayList** – przechowuje dane wewnątrz tablicy, wydajna gdy znamy ilość elementów lub wykonujemy mało operacji dodawania
- **LinkedList** – przchowuje dane w postaci powiązanej, wydajniejsza gdy dodajemy dużo elementów

```
List<String> lista = new ArrayList<String>();
lista.add("pierwszy");
lista.add("drugi");
System.out.println(lista.get(1)); //wypisze "drugi"
```

Java SE
Ćwiczenie

- stwórz kilka obiektów, w tym 2 równe
- dodaj je do kolekcji List
- wypisz wszystkie elementy tej kolekcji

Java SE
mapa

- formalnie nie są kolekcjami (nie są typu Collection)
- przechowują parę klucz-wartość
- do elementów odwołujemy się po kluczu
- .. który wskazuje na wartość
- klucz jest obiektem
- klucze muszą być unikalne

Java SE
mapa

- **HashMap** – właściwości podobne do HashSet, kolejność i przechowywanie wynika z implementacji hashCode()
- **TreeMap** – elementy przechowywane w formie posortowanej (wg klucza)

```
Map<String, Integer> mapa = new HashMap<String, Integer>();
mapa.put("pierwszy", 1);
mapa.put("drugi", 2);
System.out.println(map.get("pierwszy")); //wypisze "1"
```

Java SE
Ćwiczenie

- stwórz kilka obiektów (Integer), w tym 2 równe (klucze)
- stwórz kilka obiektów (String), w tym 2 równe (wartości)
- wypisz wszystkie elementy tej kolekcji:
  - klucze
  - wartości
  - pary klucz - wartość

Modyfikatory dostępu

Java SE
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 *java*, który znajduje się w pakiecie *pl*

  package pl.java

Java SE
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
- dobra praktyka – wszystkie pola prywatne


GOT A NEW
ERROR
PROGRESS
memegenerator.net

Java SE
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

Java SE
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;
    }
```

Java SE
interfejs

- nie chodzi o interfejs użytkownika
- szablon klasy
- definiuje metody, które klasa musi implementować (wszystkie metody interfejsu)
- klasa może implementować wiele interfejsów

```
public interface Pojazd {
    public void jazda(int predkosc);
    public void stop();
}
```

Java SE
interfejs

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

Java SE
Ćwiczenie

- refaktor klasy Menu na MainMenu
- stwórz interfejs Menu
- dodaj metody:
  - show()
  - close()
  - runOption(int option)
- niech klasa MainMenu implementuje Menu

Java SE
klasa abstrakcyjna

- drugi sposób tworzenia abstrakcji
- bardzo podobne do interfejsów
- mogą posiadać metody abstrakcyjne (*abstract*), które nie posiadają implementacji
- może zawierać zwykłe metody
- klasy rozszerzające muszą implementować wszystkie abstrakcyjne metody
- nie można tworzyć instancji takiej klasy

Java SE
klasa abstrakcyjna

```
public abstract class Emeryt {
    public static final int ILOSC_OCZU = 2; //stałe są ok

    //metoda abstrakcyjna
    public abstract String krzyczNaDzieci();

    //zwykła metoda z implementacją
    public static void biegnijDoSklepu(int odleglosc, int predkosc) {
        double czas = (double)odleglosc/predkosc;
        System.out.println("Biegne po kiełbase bede za "+czas);
    }
}
```

Java SE
Ćwiczenie

- stwórz klasę abstrakcyjną AbstractMenu
- dodaj metody:
  - show()
  - close()
  - runOption(int option)
- niech klasa MainMenu rozszerza AbstractMenu zamiast implementacji interfejsu Menu

Java SE
różnice

- można dziedziczyć tylko z jednej klasy, ale implementować wiele interfejsów
- wszystkie metody interfejsu są publiczne
- interfejs może zawierać tylko deklarację

Java SE
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

Java SE
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 STALA_WARTOSC = "stała";
```

Java SE
Ćwiczenie

- stwórz nową klasę
- dodaj w niej pola i metody statyczne oraz nie statyczne
- w klasie Main odwołaj się do nich

Java SE
typy generyczne

- specjany typ służący do parametryzowania
- umożliwia podawanie typu dopiero w momencie użycia
- zwykle oznaczane jako *T* od *type*

```
public class GeneryczneWalizka<T> {
    private T przedmiot;

    public void set(T przedmiot) { this.przedmiot = przedmiot; }
    public T get() { return przedmiot; }
}
```

Java SE
Ćwiczenie

- stwórz klasę generyczną
- dodaj pole *T object*
- dodaj konstruktor z parametrem T ustawiający to pole
- stwórz 2 obiekty generyczne różnego typu
- stwórz 2 obiekty za pomocą *getObject()*

Java SE
pattern i matcher

- klasy do obsługi wyrażeń regularnych
- matcher posiada metody find() i matches()
- *find()* - zwraca *true* jeśli coś pasuje do wzoru
- *matches()* - zwraca *true* jeśli całość pasuje do wyrażenia

```
Pattern compiledPattern = Pattern.compile("wzór");
Matcher matcher = compiledPattern.matcher( input: "tekst, w którym szukam wzoru");

System.out.println(matcher.find());
System.out.println(matcher.matches());
```

Java SE
data i czas

- LocalTime
- LocalDate
- LocalDateTime
- ZonedDateTime
- Duration

- Date
- SimpleDateFormat
- Calendar


```
LocalDate newDate = LocalDate.parse("2016-08-23");

System.out.println("Parsed date : " + newDate);
```

Java SE
overriding
overloading

- **overloading** – przeciążanie
  kilka metod o tej samej nazwie
- **overriding** – nadpisanie
  klasa dziedzicząca nadpisuje funkcjonalność metody z klasy bazowej

```
void foo(int a)
void foo(int a, float b)
```

```
class Parent {
    void foo(double d) {
        // do something
    }
}
```

```
class Child extends Parent {

    @Override
    void foo(double d){
        // this method is overridden.
    }
}
```

Wyjątki

Java SE
wyjątki

- mechanizm pozwalający wyłapywać błędy
- zamknięcie instrukcji w blok try..catch

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

Java SE
wyjątki

- Checked Exception – metoda musi deklarować możliwość rzucenia wyjątku poprzez try..catch (IOException)
- Runtime Exception – metoda nie musi deklarować możliwości rzucenia wyjątku, np. NullPointerException
- unikaj tworzenia własnych wyjątków
- złapany wyjątek zawsze powinno się obsłużyć
- blok *finally* jest wykonywany zawsze

Java SE
Optional

- kontener na obiekty, które mogą być nullem
- tworzenie obiektu Optional:
  - *Optional.of(T value)* → jeżeli value == null to NullPointer
  - *Optional.ofNullable(T value)*
  - *Optional.empty* – zwraca pusty obiekt Optional

```
Optional<String> gender = Optional.of("MALE");
```

Java SE
Optional

- pobieranie wartości z Optional
  - *isPresent()* - true jeżeli Optional zawiera wartość
  - *get()* - pobiera wartość (wyjątek gdy *null*)
  - *orElse(T other)* – zwraca wartość lub *other*
  - *orElseThrow(Exception)* – zwraca wartość lub rzuca wyjątek

```
Optional.ofNullable(answer2).orElse( other: "zwróć gdy answer1 jest nullem");
```

Strumienie

Java SE
strumienie

- I/O Stream – dowolne wejście lub wyjście
  np. plik na dysku, inny program
- strumienie dziedziczą z klas InputStream i OutputStream
- różne rodzaje strumieni mają to samo API
- operacje na strumieniach znajdują się w bloku try..catch

Java SE
strumienie bajtowe

- traktują dane jako zbiór bajtów
- strumienie zawsze należy zamknąć
- reprezentują niskopoziomowy dostęp do danych

```
FileInputStream in = null;
FileOutputStream out = null;
```

Java SE
strumienie bajtowe

```
try {
    in = new FileInputStream("xanadu.txt");
    out = new FileOutputStream("outagain.txt");
    int c;

    while ((c = in.read()) != -1) {
        out.write(c);
    }
} finally {
    if (in != null) {
        in.close();
    }
    if (out != null) {
        out.close();
    }
}
```

Java SE
strumienie znakowe

- automatycznie konwertują dane tekstowe do Unicode
- lokalizacja kodowania brana z ustawień JVM lub podawana ręcznie przez programistę
- rozszerzają klasy Reader i Writer

```
try {
 in = new FileReader("input.txt");
 out = new FileWriter("output.txt");
```

Java SE
strumienie buforowane

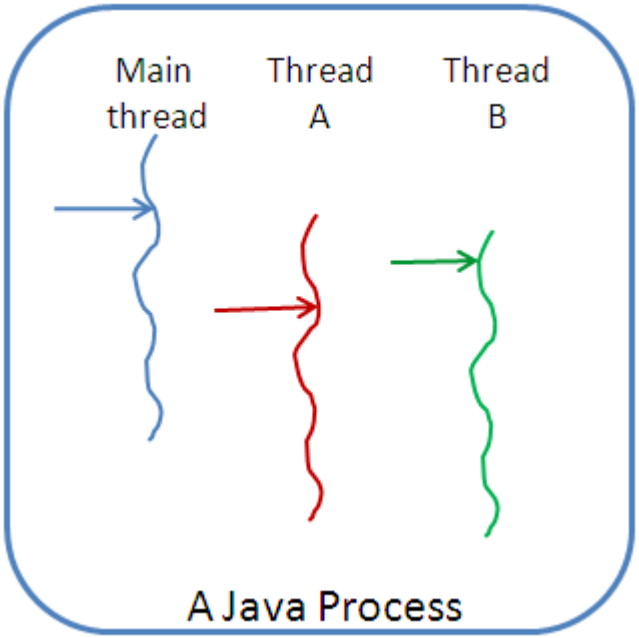
- “owijają” inne strumienie (np. znakowe) w celu optymalizacji
- umożliwiają odczyt linia po linii

```
try {
 in = new BufferedReader(new FileReader("input.txt"));
 out = new PrintWriter(new FileWriter("output.txt"));
 String l;
 while ((l = in.readLine()) != null) {
    out.println(l);
 }
} catch
```

Wątki

Java SE
wątki

- asynchroniczne wykonanie pewnych operacji
- przyspiesza wykonywanie obliczeń
- tworzymy na kilka sposobów
  - rozszerzając klasę Thread
  - implementując interfejs Runnable



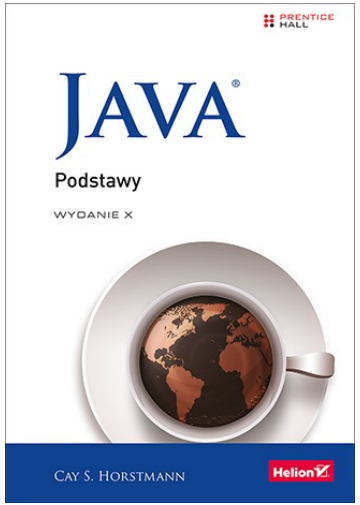
Java SE
wątki

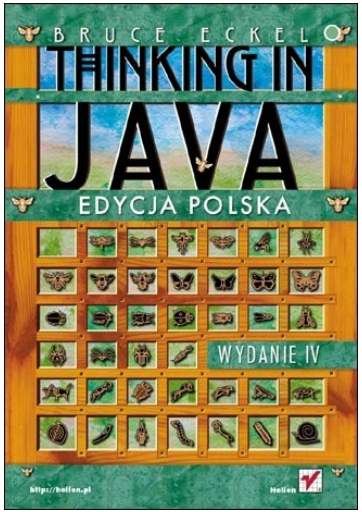
```
public class MyRun implements Runnable {

    private int id;

    public MyRun(int id) {
        this.id = id;
    }

    @Override
    public void run() {
        while(true) {
            System.out.println("Watek "+id);
            try {
                //usypiamy wątek na 100 milisekund
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
```

Materiały do nauki

Java SE
materiały

- Thinking in Java – Bruce Eckel
- Java. Podstawy. - Cay S. Horstmann





Java SE
materiały

- https://docs.oracle.com/javase/tutorial/
- https://javastart.pl/static/darmowy-kurs-java/
- https://kobietydokodu.pl/kurs-javy/
- http://www.samouczekprogramisty.pl/kurs-programowania-java/

Java SE

Thanks!

Q&A

tomasz.lisowski@protonmail.ch