Szerszy wstęp do Javy, nauka o krok dalej niż twoja leniwa dupa chce

Opracowane przez: Jakub Jakubowski

Spis treści

Adnotacja: Wszystkie kody zamieszczone na screenach znajdują się na moim githubie: https://github.com/Rolaski

Opracowane przez: Jakub Jakubowski

Wyskakujące okno dialogowe

Metoda do wpisywania danych w nowym oknie dialogowym. Zmienne typu int, double muszą być konwertowane(parse) ponieważ wszystkie są standardowo w Stringu (trochę jak w JavaScriptcie 😊). Przyjemna metoda wprowadzania

Pamiętaj o komendzie importującej!

```
import javax.swing.JOptionPane;
no usages
public class Main
{
    no usages
    public static void main(String[] args)
    {
        String name = JOptionPane.showInputDialog("Podaj swoje imie");
        JOptionPane.showMessageDialog( parentComponent: null,  message: "Hello "+name);

        int age  = Integer.parseInt(JOptionPane.showInputDialog("Podaj wiek"));
        JOptionPane.showMessageDialog( parentComponent: null,  message: "Masz "+age+" lat");

        double height = Double.parseDouble(JOptionPane.showInputDialog("Podaj wzrost"));
        JOptionPane.showMessageDialog( parentComponent: null,  message: "Masz "+height+"cm wzrostu");

        System.out.println(name+" "+age+" "+height);
    }
}
```


Opracowane przez: Jakub Jakubowski

Funkcja Random, losowe liczby

Randomowe cyfry wytłumaczenie jak to działa.
W nawiasie masz zakres cyfry np. 6 cyfr łącznie, standardowo pokaże nam od 0 do 5(bo to w sumie 6 cyfry), +10 czy -10 to operacja dodawania, odejmowania od 0 i wtedy zmieniamy zakres początkowy bo jak napisane wyżej zawsze zaczyna od 0

Pamiętaj o komendzie importującej!

```java
import java.util.Random;
no usages
public class RandomNumbers
{
    no usages
    public static void main(String[] args)
    {
        Random randomNumber = new Random();
        int x;
        double y = randomNumber.nextDouble();
        boolean z = randomNumber.nextBoolean();
        for(int i=0; i<=10; i++)
        {
            System.out.print(randomNumber.nextInt( bound: 6)+" ");
            // wyświetli od 0 do 5;  6 to ilość cyfr
        }
        System.out.println();
        for(int i=0; i<=10; i++)
        {
            System.out.print(randomNumber.nextInt( bound: 10)+11+" ");
            //10 to zakres, standardowo od 0, dodajemy 11 żeby zaczął od 10
        }
        System.out.println();
        for(int i=0; i<=10; i++)
        {
            System.out.print(randomNumber.nextInt( bound: 21)-10+" ");
            //<-10,10> leci od -10 i zakres to 21 cyfr
        }
    }
}
```


Opracowane przez: Jakub Jakubowski

Metody pracy na Stringach

Praca na Stringach, porównywanie, długość i inne przydatne operacje

Pamiętaj o komendzie importującej!

```
no usages
public class StringMethods
{
    no usages
    public static void main(String[] args)
    {
        String name = "Jacob";
        String name1 = "jacob";
        String name2 = "";
        String name3 = "     Maciek          ";
        boolean result = name.equals("Frajer");
        boolean result1 = name.equalsIgnoreCase(name1);
        //equalsIgnoreCase - polecenie ma w dupie duże i małe znaki

        int WordLength = name1.length();        //długość słowa
        char ThirdLetter = name.charAt(2);      //wyświetla np. 2 znak Stringu name
        int SearchingSign = name.indexOf("o");  //wyświetla nr indeksu konkretnego znaku np. o
        boolean isEmpty = name2.isEmpty();      //sprawdza czy String name2 jest pusty
        String Uppercase = name.toUpperCase();  //zmienia znaki na duże, analogicznie .toLowerCase
        String Trim = name3.trim();             //usuwa spacje, ucina je
        String Replace = Trim.replace( target: "a",  replacement: "o");    //zamiana znaku a na o



        System.out.println(result);
        System.out.println(result1 +" "+WordLength);
        System.out.println("Pierwsza litera słowa "+name+ " to: "+ThirdLetter);
        System.out.println("Litera 'o' ma indeks nr."+SearchingSign);
        System.out.println(isEmpty);
        System.out.println("słowo "+name+" ma teraz duże litery -> "+Uppercase);
        System.out.println("Teraz pozbyliśmy się niepotrzebnych miejsc: "+Trim);
        System.out.println("Maciek po przejściach: "+Replace);
    }
}
```


Opracowane przez: Jakub Jakubowski

ArrayList lepsza tablica

Sposób na uzyskanie tablicy, listy tzw. ArrayList. Funkcja która daje nam nowe możliwości i przyjemne do pracy z nią komendy.

Pamiętaj o komendzie importującej!

Tutaj operacje na Stringach:

```
import java.util.ArrayList;
no usages
public class Array_List
{
    no usages
    public static void main(String[] args)
    {
        //nawiasy <> deklarują typ zmiennych i z góry to musi być String
        ArrayList<String> food = new ArrayList<String>();
        food.add("zapiekanka");
        food.add("zapiekanka z dodatkami");
        food.add("hamburger");
        food.add("hot-dog");
        food.add("hot-dog z dodatkami");
        food.set(0, "zapiekanka bez dodatków");     //zamiana indeksu 0 na cos nowego
        food.remove( index: 2);                     //usuniecie pozycji nr2
        food.clear();                              //wyczyszczenie listy food

        for(int i=0; i< food.size(); i++)
        {
            System.out.print(food.get(i)+",  ");
        }
    }
}
```

Warto tutaj dodać że pętla for jest **niepotrzebna**!

Komenda *System.out.println(„Tutaj nazwa listy”)* wyświetli nam wszystkie elementy listy

Opracowane przez: Jakub Jakubowski

A tutaj przykład jak używać ArrayLista na innych typach niż String, ponieważ została ona stworzona pod Stringi to musimy ją skonwertować do innego typu, poniżej przykład zamiany ArrayLista na inty (Integer), dla zmiennych double to będzie po prostu <Double>

```
import java.util.ArrayList;
no usages
public class Array_List
{
    no usages
    public static void main(String[] args)
    {
        ArrayList<Integer> Age = new ArrayList<>();
        Age.add(56);
        Age.add(19);
        Age.add(34);
        Age.set(2, 79);
    }
}
```

Jak widać ArrayList jest świetnym przykładem pracy na tablicach za pomocą tej sprytnej funkcji, która znacznie ułatwia nam pracę

ArrayList dwuwymiarowy

Świetny przykład zastosowania ArrayList dla tabel dwuwymiarowych. Komendy i ich łatwe użycie jest wygodną możliwością korzystania z nich. Nie musimy tworzyć wielu pętli do tworzenia tablicy czy ich wypisywania co jest idealnym przykładem optymalizacji naszego kodu. Tutaj zamiast dwóch importów wystarczy skorzystać z gwiazdki(*), która pozwala na import wszystkich potrzebnych tam bibliotek.

```java
import java.util.*;
no usages
public class Array_List_2D
{
    no usages
    public static void main(String[] args)
    {
        ArrayList<String> bakeryList = new ArrayList<String>();
        bakeryList.add("Mąka");
        bakeryList.add("Pszenica");
        bakeryList.add("Chleb");
        ArrayList<String> produceList = new ArrayList<String>();
        produceList.add("pomidory");
        produceList.add("ogórki");
        produceList.add("kukurydza");
        ArrayList<String> drinksList = new ArrayList<String>();
        drinksList.add("kawa");
        drinksList.add("cappuccino");
        drinksList.add("tymbark");

        ArrayList<ArrayList<String>> groceryList = new ArrayList<>();   //na końcu <> musi być puste!
        groceryList.add(bakeryList);
        groceryList.add(produceList);
        groceryList.add(drinksList);

        System.out.println(bakeryList);         //wyświetla całą listę
        System.out.println(bakeryList.get(2));  //wyświetla element o indeksie nr2
        System.out.println(groceryList);
        System.out.println(groceryList.get(1).get(0)); //pierwszy get to nr listy a drugi get to element
    }
}
```

Warto zauważyć że kiedy tworzymy tablice dwuwymiarową po przypisaniu tj. = new ArrayList<>(); specjalnie zostawiamy nawiasy <> puste aby program działał poprawnie, w innym wypadku pokazują się błędy!

Opracowane przez: Jakub Jakubowski

Pętla for-each

Jedną z nowych pętli którą warto się nauczyć z powodu jej prostoty jest pętla for-each, która na początku nie wydaje się taka intuicyjna z czasem jest świetnym zastosowaniem za naszego dobrego starego fora.

Tutaj przykład zastosowania for-eacha do wypisywania danych z zwykłej tablicy oraz naszego ulubionego ArrayLista

```
import java.util.ArrayList;

no usages
public class foreachLoop
{
    no usages
    public static void main(String[] args)
    {
        String[] animals = {"kot", "pies", "bocian","karp"};

        for(String i: animals)              //w petli foreach elementy to
        {                                   //String - po jakich elementach chodzimy
            System.out.print(i+" ");        //i - to nasze typowe i z fora
        }                                   //animals - tablica która używamy
        System.out.println();

        ArrayList<String> food = new ArrayList<String>();
        food.add("Pomarańcza");
        food.add("Pieczarka");
        food.add("Kiwi");
        food.add("Ogórek");

        for(String i: food)
        {
            System.out.print(i+" ");
        }
    }
}
```


Opracowane przez: Jakub Jakubowski

Dziedziczenie w klasach

Pojęcie dziedziczenia będzie dotyczyło naszych klas, przydaje nam się żeby zabrać wszystkie obiekty/elementy z klasy A do klasy B i dodajemy jedynie w niej jeden lub parę nowych rzeczy np. wypłata jak w naszym przykładzie. Jeżeli chcemy przekazać obiekty do innej klasy np. obiekt imię, nazwisko możemy do tego wykorzystać dziedziczenie, wykonujemy wtedy rozszerzenie klasy tj. *public class BookKeeping extends Employee* , nasza nowa klasa BookKeeping zawiera teraz obiekty z klasy Employee. Kiedy klasa A posiada obiekty B i nic poza tym mówimy wówczas o **kompozycj**i. Zaczynamy od podstawowych obiektów które będą zawierać się wszędzie, czyli od naszego pracownika, w nim tworzymy konstruktor

```
package Inheritance;
2 usages  2 inheritors
public class Employee
{
    8 usages
    String name, surname, profession;
    8 usages
    int paycheck;

    2 usages
    public Employee()
    {
        name = "";
        surname = "";
        profession = "";
        paycheck = 0;
    }

    2 usages
    public Employee(String name, String surname, String profession, int paycheck)
    {
        this.name = name;
        this.surname = surname;
        this.profession = profession;
        this.paycheck = paycheck;
    }
}
```


Opracowane przez: Jakub Jakubowski

Następnym naszym krokiem po stworzeniu bazowej klasy jest utworzenie nowej która będzie dziedziczyła elementy. Do tego służą nam dwa polecenia: *extends* i *super*. Tworzymy metodę konstruktora do której przekazujemy wraz z podaniem typu zmiennej obiektów, w środku metody wpisujemy komendę super(obiekt1, obiekt2, …), gdzie obiekt to nazwa obiektu którą dziedziczymy z góry.

Nie zapomnij o dodaniu w konstruktorze odwołania do nowych obiektów jeśli takie stworzysz! (this.obiekt = obiekt).

```java
package Inheritance;
import javax.swing.*;

2 usages
public class Boss extends Employee
{
    4 usages
    int bonus;
    4 usages
    int monthlyTax;
    1 usage
    public Boss()
    {
        bonus = 0;
        monthlyTax = 0;
    }
    no usages
    public Boss(String name, String surname, String profession, int paycheck, int bonus, int monthlyTax)
    {
        super(name, surname, profession, paycheck);
        this.bonus = bonus;
        this.monthlyTax = monthlyTax;
    }
```

```java
package Inheritance;
import javax.swing.*;

2 usages
public class BookKeeping extends Employee
{
    8 usages
    int bonus;
    1 usage
    public BookKeeping()
    {
        bonus = 0;
    }
    no usages
    public BookKeeping(String name, String surname, String profession, int paycheck)
    {
        super(name, surname, profession, paycheck);
        this.bonus = bonus;
    }
```


Opracowane przez: Jakub Jakubowski

Klasy, metody abstrakcyjne

Klasa abstrakcyjna to jak taka klasa nad innymi klasami która jest bardziej ogólna, bez żadnych konkretów np. klasa abstrakcyjna vehicle i klasa car, motorbike.

Klasy abstrakcyjnej nie używa się do tworzenia obiektów! Może jedynie służyć jako obiekt do dziedziczenia dla innych klas.

Metody abstrakcyjne tworzy się aby klasy dziedziczone musiały, jest to wymuszone aby użyły tej metody, kiedy tworzymy metodę abstrakcyjną nie może zawierać się ona z żadnego ciała! **Ciało musi być puste**, bez żadnej implementacji. Przykład poniżej:

```
no usages   1 implementation
abstract void go(); //prawidłowe zastosowanie metody abstrakcyjnej
```

Nieprawidłowe zastosowanie metody abstrakcyjnej, kiedy chcemy dodać do niej ciało:

```
no usages
abstract void stop()
{
    System.out.println("The driver is stopping the vehicle");
};
```

W skrócie klasy abstrakcyjne zostały stworzone do deklarowania metod, atrybuty i stałe, które następnie mają, muszą zostać użyte później np. w innych klasach poprzez dziedziczenie. Warto dodać że możemy ustawiać im metody o wszelkiej widoczności w zależności od potrzeb.

Przykładowe zastosowanie klasy abstrakcyjnej znajduje się na GitHubie 😊.

Opracowane przez: Jakub Jakubowski

Modyfikator dostępu

W javie możemy zarządzać zmiennymi, klasami i obiektami za pomocą modyfikatorów dostępu, pozwalają albo odbierają prawa do widoczności, działania na nich w różnych przestrzeniach.

Public służy do publicznego dostępu z każdego miejsca, *protected* zabezpiecza przed ingerencją z zewnątrz, *brak modyfikatora* czyli nie użycie żadnego z nich zostawia nam dostęp z klasy i pakietów(folderów), a ostatni *private* czyni dostęp tylko z aktualnej klasy

Zostało to przedstawione na tabeli poniżej:

Modyfikator	Klasa	Pakiet	Podklasa	Inni	Poprawny dla klas
`public`	tak	tak	tak	tak	tak
`protected`	tak	tak	tak	nie	nie
brak modyfikatora	tak	tak	nie	nie	tak
`private`	tak	nie	nie	nie	nie

Opracowane przez: Jakub Jakubowski

Oto przykład zastosowania tych modyfikatorów dostępu, z poziomu różnych plików:

- Klasa classC

```
package AccessModifiers.Package2;
2 usages
public class classC
{
    no usages
    String defaultMessage = "This is default message";
    1 usage
    public String publicMessage = "This is public message for everyone!";
}
```

- Klasa classB

```
package AccessModifiers.Package1;
3 usages
public class classB
{
    no usages
    private String privateMessage = "This is private message!";
}
```

- Klasa classA

```
package AccessModifiers.Package1;
import AccessModifiers.Package2.*;
2 usages   1 inheritor
public class classA
{
    public static void main(String[] args)
    {
        classC c = new classC();
        System.out.println(c.publicMessage);

        classB b = new classB();
        System.out.println(classB.privateMessage);
        //nie możemy jej wywołać ponieważ ma modyfikator private
    }
    1 usage
    protected String protectedMessage = "This is protected message!";
}
```

Enkapsulacja

Enkapsulacja (inaczej hermetyzacja) to ukrywanie widoczności pól danej klasy(wartości) dla innych klas, w ten sposób chronimy dane przechowywane w tych polach przed niepowołanym, nieuzasadnionym dostępem, nie możemy w nie ingerować.

Aby dostać się do wartości pól z danej klasy musimy utworzyć Settery, Gettery tylko i wyłącznie wtedy mamy możliwość dokonania zmian.

Należy używać modyfikatorów *private* zawsze dla enkapsulacji, w przypadku ważnego powodu możemy ustawić wyjątkowe zmienne z modyfikatorem *public/protected*

.

```
package Encapsulation;

no usages
public class Main
{
    public static void main(String[] args)
    {
        Car car1 = new Car( mark: "Fiat", model: "Panda", year: 2006);
        //Nie ma możliwości wyświetlenia roku obiektu car1
        //System.out.println(car.year);
        System.out.println(car1.getMark());
        System.out.println(car1.getModel());
        System.out.println(car1.getYear());

        //Nie ma możliwości też ustawiania wartości objektu
        //car1.year = 2010;
        car1.setYear(2010);
    }
}
```

Dla przykładu powyżej została stworzona klasa Car, która posiada konstruktor oraz najważniejsze do możliwości wykonania zmian metody Getters i Setters.

Opracowane przez: Jakub Jakubowski

Interfejsy

Kilka metod bez ich implementacji (bez kodu definiującego zachowanie metody), to właśnie interfejsy. Podobny do dziedziczenia, lecz mówi nam co klasa musi robić/zawierać.

Klasy mogą zawierać w sobie więcej niż jeden interfejs co jest kluczowe gdy spojrzymy na dziedziczenie które jest ograniczone do jednej klasy.

Właściwie można powiedzieć że interfejsy to takie sprytniejsze klasy abstrakcyjne, też zawierają metody bez implementacji i wymuszają dodania ich metod do klas podlegających im, a do jednej klasy można przypisać więcej niż jeden, czy dwa interfejsy!

```java
package Interface;
2 usages  2 implementations
public interface Prey
{
    2 usages  2 implementations
    void flee();
}
```

```java
package Interface;
2 usages  2 implementations
public interface Predator
{
    2 usages  2 implementations
    void hunt();
}
```

```java
package Interface;

no usages
public class Main
{
    public static void main(String[] args)
    {
        Fish fish = new Fish();
        fish.flee();
        fish.hunt();
    }
}
```

Standardowo przykłady do każdego rozdziału znajdują się na moim githubie.