

# | Podstawy Java SE



# Hello!

**Tomasz Lisowski**

Software developer, JIT Solutions  
IT trainer

# Agenda

- wprowadzenie
- typy danych
- instrukcje sterujące
- pętle
- równość obiektów



# Java SE

## Wprowadzenie

- **JVM** – wirtualna maszyna javy
  - “procesor” wykonujący skompilowany kod Javy
- **JRE** – Java Runtime Environment
  - zawiera JVM oraz klasy niezbędne do uruchomienia programów Java
- **JDK** – Java Development Kit
  - zawiera JRE oraz narzędzia do implementacji i kompilacji

# Java SE

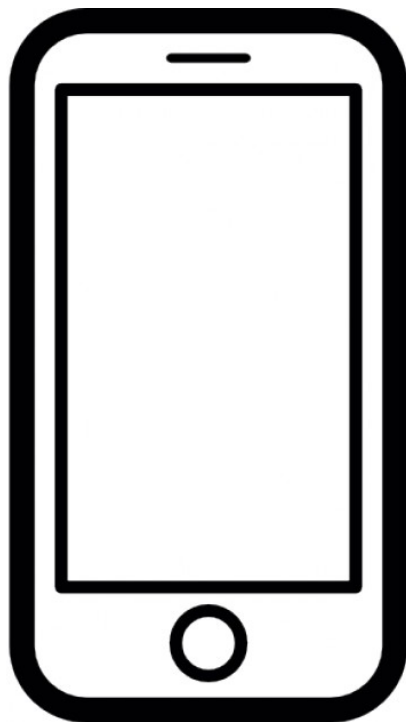
## Klasa

- podstawowy element składowy aplikacji
- typ danych
- konkretna **definicja** pewnego 'bytu'
- .. zawierająca pola i metody

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

# Java SE

## Klasa



<i>właściwości / pola</i>	<i>czynności / metody</i>
<b>marka</b>	<b>zadzwon()</b>
<b>model</b>	<b>odbierz()</b>
<b>numer</b>	<b>wyslijSMS()</b>
<b>lista kontaktów</b>	<b>odbierzSMS()</b>
<b>waga</b>	<b>zrobZdjecie()</b>

# Java SE

## main()

- metoda *main()* to główna metoda, od której rozpoczyna się uruchamianie programu przez JVM

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

# Java SE

## Obiekt

- instancja klasy
- konkretny obiekt na podstawie definicji klasy

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

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



# Java SE

## Konstruktor

- metoda tworząca obiekt
- domyślny konstruktor
- konstruktor parametrowy
- słowo kluczowe *this*
  - zwraca aktualny obiekt


```
public class Car {  
    public String name;  
    public int maxSpeed;  
  
    public Car() {  
        name = "default name";  
        maxSpeed = 150;  
    }  
}
```

# Java SE

## Konstruktor

```
public class Car {  
    public String name;  
    public int maxSpeed;  
  
    public Car() {  
        name = "default name";  
        maxSpeed = 150;  
    }  
  
    public Car(String name) {  
        this.name = name;  
    }  
}
```

przypisanie do pola name  
(pole klasy) wartości  
parametru



# Java SE

## Konstruktor

```
int number;  
String text;
```

```
//domyślny konstruktor, nie trzeba go pisać jawnie  
Menu() {  
}
```

```
//parametrowy konstruktor  
public Menu(int number, String text) {  
    this.number = number;  
    this.text = text;  
}
```

# Java SE

## Metody

- funkcje, które dana klasa udostępnia
- prywatne lub publiczne
- mogą posiadać parametry

```
public class Methods {  
  
    public void method1() {  
        System.out.println("Ta metoda nie zwraca nic!");  
    }  
  
    public int getNumberTwo() {  
        return 2; //ta metoda zwraca liczbę całkowitą 2  
    }  
  
    public int sum(int a, int b) {  
        return a + b; //ta metoda zwraca sumę dwóch parametrów  
    }  
}
```

# Java SE

## Metody statyczne

- mogą istnieć metody statyczne (*static*)
- można je wywołać bezpośrednio na klasie
- nie wymagają stworzenia instancji obiektu

```
public class Menu {  
    public static void staticMethod() {  
        System.out.println("This is static method!");  
    }  
  
    public void nonStaticMethod() {  
        System.out.println("This is NON static method!");  
    }  
}
```

# Java SE

## Metody statyczne

```
public static void main(String[] args) {  
    Menu menu = new Menu();  
    menu.nonStaticMethod();  
  
    Menu.staticMethod();  
}
```

# Java SE

## Ćwiczenie

- stwórz dwie metody w klasie StaticExample
- obydwie niech będą typu *void*
- jedna z nich niech będzie metodą statyczną
- każda z nich niech wypisze czy jest statyczna czy nie
- wywołaj obydwie metody w klasie Main



# Java SE

## Zasięg widzenia pól

- pola klasy dla wszystkich metod
- ..lub na zewnątrz
- pola w metodzie widoczne tylko w niej
- tworzony obiekt wewnątrz metody “żyje” tylko w niej



# Java SE

## Zasięg widzenia pól

```
int number;
```

```
String text;
```



pola klasy

```
public void method() {
```

```
    int otherNumber;
```



pole metody

```
    number = 1;
```

```
    otherNumber = 2;
```

```
}
```

```
public void otherMethod() {
```

```
    number = 2;
```

```
    otherNumber = 3;
```



błąd - *otherNumber* nie jest  
widoczne w tej metodzie

```
}
```

# Java SE

## getter i setter

- dostęp do wartości pól powinien odbywać się poprzez metody, tzw. getter i setter
- metody *set(Type value)* do ustawiania wartości
- metody *get()* do odczytania

```
private int number;
```

```
public int getNumber() {  
    return number;  
}
```

```
public void setNumber(int number) {  
    this.number = number;  
}
```

# Typy danych

# Java SE

## Co to są typy?

- *wszystko jest obiektem*
- reprezentują różne wartości, przechowywane w zmiennych
- typy tekstowe, liczbowe, zmiennoprzecinkowe, logiczne
- różne formaty dat
- każda klasa jest typem

# Java SE

## Typy proste

- typy proste (*primitive*) nie są instancjami obiektów
- reprezentują podstawowe typy danych
- zawsze mają jakąś wartość

```
int liczbaCalkowita;  
long duzaLiczbaCalkowita;  
double liczbaZmiennoPrzecinkowa; //64bit  
float kolejnaLiczbaZmiennoPrzecinkowa; //32bit  
boolean prawdaFalsz;  
char znak;
```

# Java SE

## Typy proste

Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

# Java SE

## Typy obiektowe

- konkretne instancje
- możemy tworzyć swoje typy
- mogą mieć dowolne zachowanie (metody)
- mogą nie mieć wartości -> NULL
- każda stworzona przez nas klasa to też typ!

```
Integer liczbaCalkowita;  
Long duzaLiczbaCalkowita;  
Double liczbaZmienniePrzecinkowa;  
Float kolejnaLiczbaZmienniePrzecinkowa;  
Boolean prawdaFalsz;  
String napis;
```

# Java SE

## Ćwiczenie

- stwórz dwie zmienne liczbowe
  - typu **int**
  - typu **Integer**
- wypisz ich domyślne wartości
- porównaj metody, które te dwie zmienne udostępniają:  
*zmiennaInt. ?*  
*zmiennaInteger. ?*





# Java SE

## Typy wyliczeniowe

- ENUM
- konkretny (ograniczony) zbiór możliwych wartości

```
public enum Colour {  
    RED,  
    GREEN,  
    BLUE  
}
```

```
Colour myColour = Colour.GREEN;
```

# Java SE

## Ćwiczenie

- stwórz własny typ wyliczeniowy
- dodaj pole tego typu do klasy Menu
- wypisz domyślną wartość pola
- dodaj nowy konstruktor Menu, który uwzględni nowe pole



# Java SE

## Rzutowanie

- zmiana typu danych na inny
- np. dzielenie dwóch liczb całkowitych

```
int liczbaA = 10;  
int liczbaB = 3;  
//wynik dzielenia nie jest liczbą całkowitą  
  
int wynik = liczbaA / liczbaB;  
// zmienna wynik = 3
```

# Java SE

## Rzutowanie

```
int liczbaA = 10;  
int liczbaB = 3;  
//wynik dzielenia nie jest liczbą całkowitą  
  
double wynikInt = liczbaA / liczbaB;  
// zmienna wynik = 3.0
```

# Java SE

## Rzutowanie

```
int liczbaA = 10;  
int liczbaB = 3;
```

```
double liczbaC = (double) liczbaA;  
//liczbaC = 10.0  
double liczbaD = (double) liczbaB;  
//liczbaD = 3.0
```

```
double wynikInt = liczbaA / liczbaB;  
//wynikInt = 3.0  
double wynikDouble = liczbaC / liczbaD;  
//wynikDouble = ?
```

# Java SE

## String

- obiektowy typ tekstowy
- *immutable* – nie można go zmienić, zmiana powoduje stworzenie nowej instancji
- posiada zestaw metod do operacji na tekście, np. *compare()*, *concat()*, *split()*, *length()*, *replace()*, *substring()*

```
char[] chars = {'i', 'n', 'f', 'o', 'S', 'h', 'a', 'r', 'e'};  
String s = new String(chars);
```

```
String s2 = "infoShare";
```

# Java SE

## String

- modyfikacje na Stringach wymagają przypisania

```
String s = "infoShare";  
s.concat("Academy");  
System.out.println(s);
```



infoShare

```
String s = "infoShare";  
s = s.concat("Academy");  
System.out.println(s);
```



infoShareAcademy

# Java SE

## Operator

- działania, które można wykonywać na obiektach
- np. operacje matematyczne lub logiczne
- operatory porównania lub przypisania

Operator type	Example
Unary	<i>i++</i> , <i>i--</i> , <i>++i</i> , <i>--i</i> , <i>!</i>
Arithmetic	<i>*</i> , <i>/</i> , <i>%</i> , <i>+</i> , <i>-</i>
Relational	<i>&lt;</i> , <i>&gt;</i> , <i>&lt;=</i> , <i>&gt;=</i> , <i>instanceof</i> , <i>==</i> , <i>!=</i>
Logical	<i>&amp;&amp;</i> , <i>  </i>
Assignment	<i>=</i> , <i>+=</i> , <i>-=</i> , <i>*=</i> , <i>/=</i>



# Java SE

## konwencja

- każdy wyraz 'oddzielamy' dużą literą
- **KLASY** - rzeczownik, zaczynamy dużą literą
- **METODY** - czasownik, zaczynamy małą literą
- **ZMIENNE** - zaczynamy małą literą
- **STAŁE** - duże litery, wyrazy 'oddzielamy' znakiem '\_'

```
class MyClass {  
    Integer myVariable;  
    final Integer MY_CONSTANT = 2;  
  
    void myMethod() {  
        myVariable = MY_CONSTANT + 1;  
    }  
}
```

# Instrukcje sterujące

# Java SE

## 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";  
}
```

# Java SE

## if else

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

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

# Java SE

## Ćwiczenie

- stwórz obiekt menu1 z wartością *number* = 1
- stwórz obiekt menu2 z wartością *number* = 2
- stwórz warunek, przypisujący enuma zależnie od wartości *number*
- wykonaj warunek dla obydwu obiektów
- wypisz wartość enuma dla obydwu obiektów



# Java SE

## switch

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

```
switch(liczba){  
    case 1:  
        jakiś_instrukcje_1;  
        break;  
    case 2:  
        jakiś_instrukcje_2;  
        break;  
    ...  
    default:  
        instrukcje, gdy nie znaleziono żadnego pasującego przypadku  
}
```

# Java SE

## Ćwiczenie

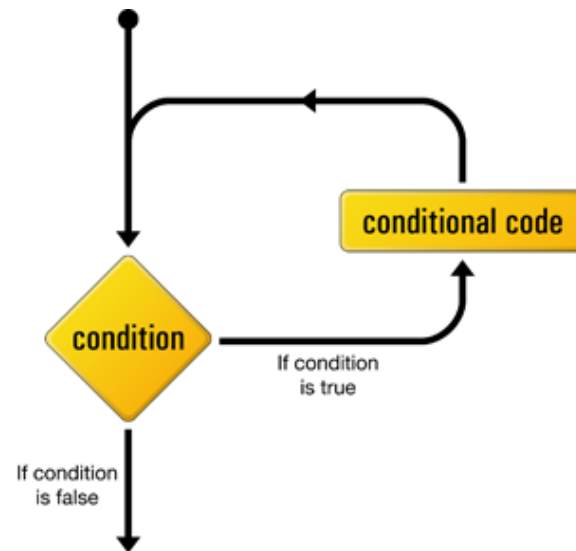
- zmodyfikuj poprzednie zadanie, zamieniając instrukcje warunkowe *if*
- wykorzystaj instrukcję *switch*



# Java SE

## Pętle

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





# Java SE

## pętla 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;
}
}
```

# Java SE

## pętla do..while

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

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

# Java SE

## pętla for

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

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

# Java SE

## break - continue

- instrukcje manipulujące działaniem pętli
- *break*
  - przerwanie pętli
- *continue*
  - ominięcie danej iteracji

```
int liczba = -5;
while(liczba < 0) {
    if (liczba == 2) {
        continue;
    }

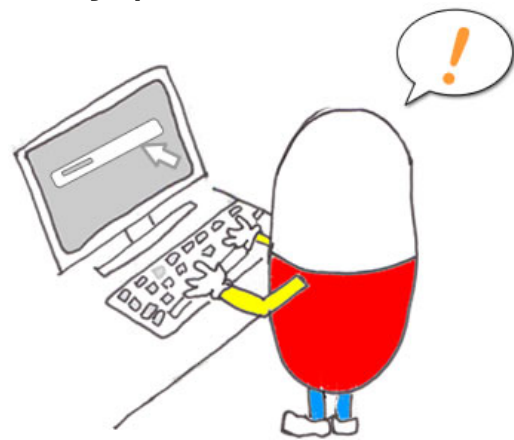
    if (liczba == 3) {
        break;
    }

    liczba++; //liczba = liczba + 1;
}
```

# Java SE

## Ćwiczenie

- napisz metodę, przyjmującą jeden parametr typu *int*
- w metodzie napisz pętlę iterującą od 0 do wartości parametru
- wypisz każdą liczbę nieparzystą
- pominiń każdą liczbę parzystą
- przerwij pętlę, jeśli liczba jest podzielna bez reszty przez 11



# Java SE Petle



```
//Grab odd numbers from array
for(int i=0;i<Array.Length;i++)
{
    if(i == 1){
        Console.Write(i);
    }
    if(i == 3){
        Console.Write(i);
    }
    if(i == 5){
        Console.Write(i);
    }
    if(i == 7){
        Console.Write(i);
    }
    if(i == 9){
        Console.Write(i);
    }
    if(i == 11){
        Console.Write(i);
    }
    if(i == 13){
        Console.Write(i);
    }
    if(i == 15){
        Console.Write(i);
    }
}
```

# Pobranie danych z klawiatury

# 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

- stwórz dwie zmienne typu Double
  - pobierz je za pomocą klasy Scanner
  - dodaj je do siebie
  - wyświetl wynik
- ile wynosi  $0,1 + 0,2$  ?



# Java SE

## Ćwiczenie

- pobierz za pomocą klasy Scanner dwie wartości:
  - tekst
  - liczbęi przypisz je do pól obiektu klasy Menu
- wyświetl wartości



Czy obiekty są  
równe?

# Java SE

## == vs equals

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

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

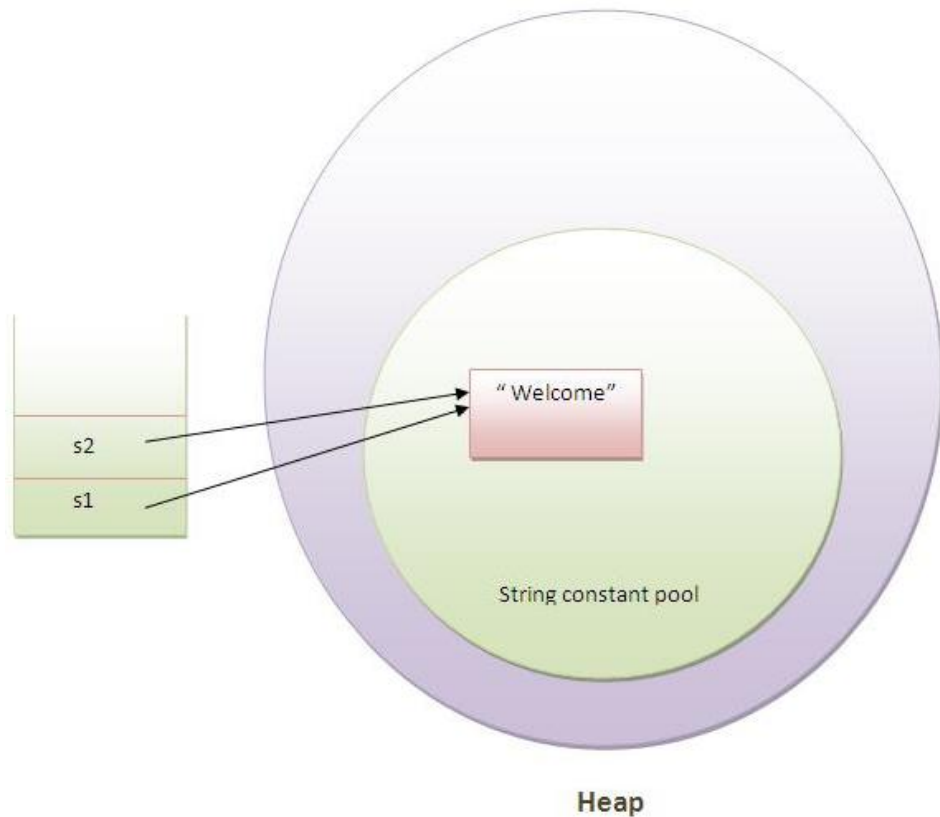
```
String tekstB = "tekst";
```

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

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

# Java SE

## == vs equals



# Java SE

## == vs equals

```
String tekstA = new String( original: "tekst");  
String tekstB = new String( original: "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*





# Java SE

## Ćwiczenie

- stwórz enum, który oznacza dostępne opcje (np. dodawanie)
- wyświetl informację o dostępnych opcjach (enum)
- pobierz opcję z klawiatury (numer opcji)
- ustaw odpowiednią wartość enum w zależności od podanej liczby
- jeżeli błędna wartość to wyświetl informację i ponownie
- podanie wartości 0 przerywa działanie



# Java SE

## Zadanie domowe

- stwórz klasę o nazwie Card
- dodaj dwie klasy enum określające kolor i figurę karty:
  - suits (CLUBS, DIAMONDS, HEARTS, SPADES)
  - ranks (ACE, KING, QUEEN, JACK)
- dodaj gettery i settery
- stwórz metodę getDescription, która zwraca kartę w formacie “figura – kolor”, np. “Ace – Spades”
- ustaw przykładowe wartości dla kilku kart
- \* kolor i figurę pobieraj z klawiatury
- uruchom i przetestuj aplikację



# Thanks!



Q&A

[tomasz.lisowski@protonmail.ch](mailto:tomasz.lisowski@protonmail.ch)