



# Wprowadzenie do języka JAVA

Marek Sontag  
[mareksontag@gmail.com](mailto:mareksontag@gmail.com)

Autor: Marek Sontag

Prawa do korzystania z materiałów posiada Software Development Academy

# AGENDA



1. Powtórka, omówienie zadań domowych
2. Inkrementacja, dekrementacja
3. Tablice
4. Pętle
5. String
6. Object
7. equals() i hashCode()
8. Instrukcje warunkowe
9. Elementy statyczne i finalne
10. Interfejs
11. Klasa abstrakcyjna
12. Typ wyliczeniowy



```
int licznik = 0;
```

```
// ...
```

```
licznik = licznik + 1;  $\longleftrightarrow$  licznik += 1;
```



```
int licznik = 0;  
// ...  
licznik = licznik + 1;  $\longleftrightarrow$  licznik++;
```



```
int licznik = 100;
```

```
// ...
```

```
licznik = licznik - 1;  $\longleftrightarrow$  licznik -= 1;
```



```
int licznik = 100;
```

```
// ...
```

```
licznik = licznik - 1;  $\longleftrightarrow$  licznik--;
```



## WIELE ZMIENNYCH TEGO SAMEGO TYPU

```
int a0;  
int a1;  
int a2;  
int a3;  
int a4;  
int a5;  
int a6;
```



## WIELE ZMIENNYCH TEGO SAMEGO TYPU

```
int a0;
```

```
int a1;
```

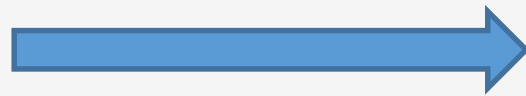
```
int a2;
```

```
int a3;
```

```
int a4;
```

```
int a5;
```

```
int a6;
```

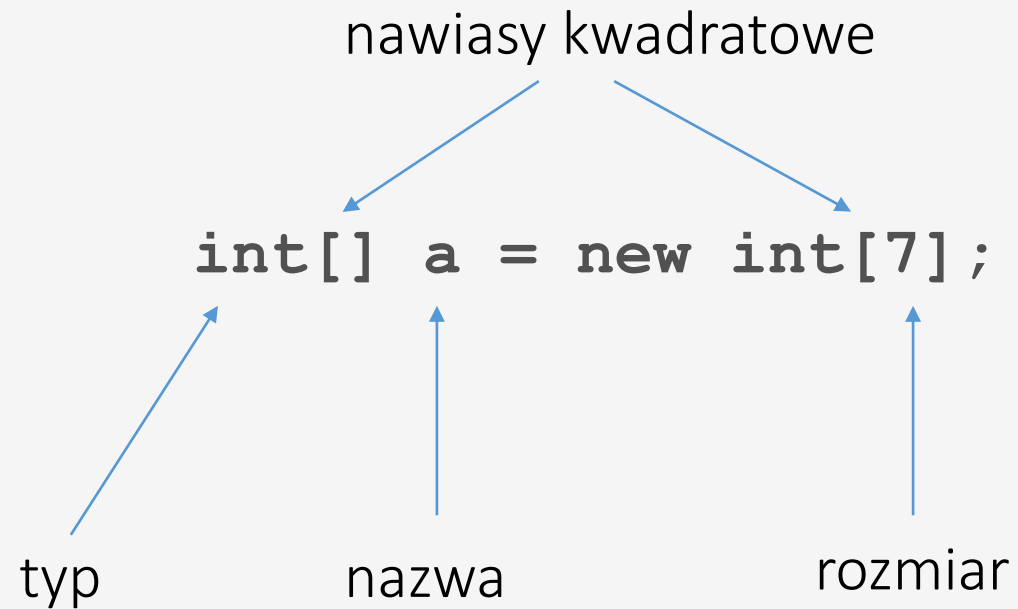


```
int[] a = new int[7];
```



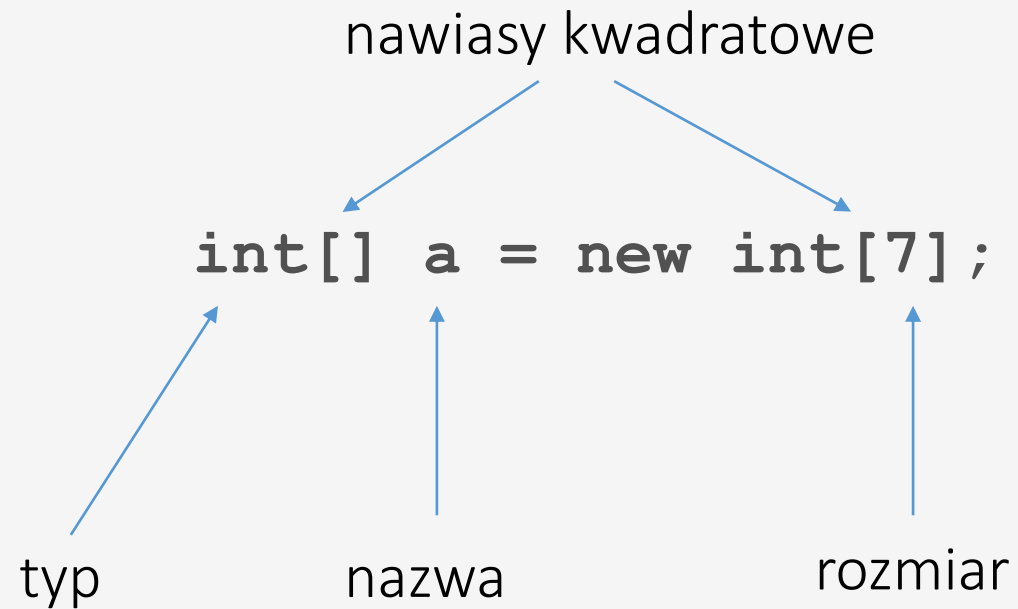


## DEKLARACJA / INICJALIZACJA





## DEKLARACJA / INICJALIZACJA



*Elementy mają wartości domyślne*



## DEKLARACJA / INICJALIZACJA

```
int[] a = new int[] {2, 4, 6, 8, 10, 12, 0};
```



Inicjalizacja wartościami  
(nie podajemy rozmiaru!)



## DEKLARACJA / INICJALIZACJA

```
int[] a = {2, 4, 6, 8, 10, 12, 0};
```



Inicjalizacja wartościami  
(wersja skrócona – bez new)



## ODCZYT / ZAPIS ELEMENTÓW

```
int[] a = {2, 4, 6, 8, 10, 12, 0};
```



element o indeksie 0  
a[0]



element o indeksie 6  
a[6]



## ODCZYT / ZAPIS ELEMENTÓW

```
int[] a = {2, 4, 6, 8, 10, 12, 0};
```

```
System.out.println(a[0]);
```

← wypisze „2” (indeksowanie od 0)

```
System.out.println(a[1]);
```

```
System.out.println(a[5]);
```

← wypisze „12”

```
System.out.println(a[6]);
```



## ODCZYT / ZAPIS ELEMENTÓW

```
int[] a = {2, 4, 6, 8, 10, 12, 0};  
a[0] = 7;  
System.out.println(a[0]);
```

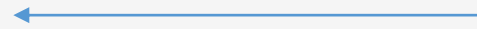


## ODCZYT / ZAPIS ELEMENTÓW

```
int[] a = {2, 4, 6, 8, 10, 12, 0};
```

```
a[0] = 7;
```

```
System.out.println(a[0]);
```



wypisze „7”

*aktualny stan tablicy a:*

**{7, 4, 6, 8, 10, 12, 0}**





## ODCZYT / ZAPIS ELEMENTÓW

```
int[] a = {2, 4, 6, 8, 10, 12, 0};
```

```
int zmienna = 100;
```

```
a[6] = zmienna;
```

```
System.out.println(a[6]);
```

 wypisze „100”

*aktualny stan tablicy a:*

`{2, 4, 6, 8, 10, 12, 100}`



## ROZMIAR TABLICY

```
int[] a = {2, 4, 6, 8, 10, 12, 0};  
System.out.println(a.length);
```

← wypisze „7”



## TABLICA JAKO PARAMETR

```
void methodWithArrayAsParam(int[] array) {  
    // ...  
}
```

```
// ...  
int[] a = new int[5];  
methodWithArrayAsParam(a);
```



1. Utworzyć nowy projekt Maven o nazwie `arrays-example`.
2. Stworzyć klasę `ArrayExample`.
3. W klasie `ArrayExample` stworzyć tablicę liczb całkowitych z pięcioma elementami (na czwartej pozycji ustawić wartość 8).
4. Stworzyć test dla klasy `ArrayExample` sprawdzający rozmiar tablicy.
5. Stworzyć test dla klasy `ArrayExample` sprawdzający czy element na pozycji czwartej to wartość 8.
6. W klasie `ArrayExample` zadeklarować tablicę (o nazwie `tabWithoutValues`) liczb całkowitych o rozmiarze pięć.
7. W klasie `ArrayExample` zadeklarować tablicę (o nazwie `stringsWithoutValues`) `String` o rozmiarze pięć.
8. Stworzyć test dla klasy `ArrayExample` sprawdzający element na indeksie 0 z tablicy `tabWithoutValues`.
9. Stworzyć test dla klasy `ArrayExample` sprawdzający element na indeksie 1 z tablicy `stringsWithoutValues`.



## POWTÓRZENIE TEGO SAMEGO KODU WIELOKROTNIIE

```
for (int i = 0; i < 10; i++) {  
    System.out.println(„Hello For!”);  
}
```



## POWTÓRZENIE TEGO SAMEGO KODU WIELOKROTNIIE

```
for (int i = 0; i < 10; i++) {  
    System.out.println("Hello For!");  
}
```

*„Hello For!” zostanie wypisany 10 razy*



## FOR

wyrażenie początkowe

```
for (int i = 0; i < 10; i++) {  
    System.out.println("Hello For!");  
}
```



## FOR

warunek kontynuowania pętli

```
for (int i = 0; i < 10; i++) {  
    System.out.println("Hello For!");  
}
```





## FOR

Instrukcja wykonywana po każdej iteracji

```
for (int i = 0; i < 10; i++) {  
    System.out.println("Hello For!");  
}
```



## PĘTLE I TABLICE

```
void printArray(int[] array) {  
    System.out.println(array[0]);  
    System.out.println(array[1]);  
    System.out.println(array[2]);  
    System.out.println(array[3]);  
    System.out.println(array[4]);  
}
```



## PĘTLE I TABLICE

```
void printArray(int[] array) {  
    System.out.println(array[0]);  
    System.out.println(array[1]);  
    System.out.println(array[2]);  
    System.out.println(array[3]);  
    System.out.println(array[4]);  
}
```

Musimy znać rozmiar tablicy,  
powtórzyć ten sam kod wielokrotnie  
- mało elastyczne rozwiązanie!



## PĘTLE I TABLICE

```
void printArray(int[] array) {  
    System.out.println(array[0]);  
    System.out.println(array[1]);  
    System.out.println(array[2]);  
    System.out.println(array[3]);  
    System.out.println(array[4]);  
}
```

Musimy znać rozmiar tablicy,  
powtórzyć ten sam kod wielokrotnie  
- mało elastyczne rozwiązanie!

*Wyobraź sobie, że `array.length = 1000...`*



## PĘTLE I TABLICE

```
void printArray(int[] array) {  
    // wypisz wartość wszystkich elementów  
    for (int i = 0; i < array.length; i++) {  
        System.out.println(array[i]);  
    }  
}
```



## PĘTLE I TABLICE

```
void printArray(int[] array) {  
    // wypisz wartość wszystkich elementów  
    for (int i = 0; i < array.length; i++) {  
        System.out.println(array[i]);  
    }  
}
```

Na początku ustawiamy indeks `i` na 0  
(rozpoczynamy wypisywanie od pierwszego elementu)



## PĘTLE I TABLICE

```
void printArray(int[] array) {  
    // wypisz wartość wszystkich elementów  
    for (int i = 0; i < array.length; i++) {  
        System.out.println(array[i]);  
    }  
}
```

Po każdym przejściu pętli zwiększamy indeks o 1



## PĘTLE I TABLICE

```
void printArray(int[] array) {  
    // wypisz wartość wszystkich elementów  
    for (int i = 0; i < array.length; i++) {  
        System.out.println(array[i]);  
    }  
}
```

Pętle powtarzamy dopóki indeks `i` jest mniejszy od długości tablicy  
(nie możemy iterować poza rozmiarem tablicy!)





## WHILE

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



## WHILE

```
int i = 0; ← Licznik ustawiamy przed pętlą  
while (i < array.length) {  
    System.out.println(array[i]);  
    i++;  
}
```



## WHILE

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

Warunek zakończenia pętli



## WHILE

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

Samodzielnie zwiększamy indeks na końcu pętli



## DO WHILE

```
int i = 0;  
do {  
    System.out.println(array[i]);  
    i++;  
} while (i < array.length)
```



## DO WHILE

```
int i = 0;  
do {  
    System.out.println(array[i]);  
    i++;  
} while (i < array.length)
```



Warunek jest sprawdzany na końcu pętli, a nie na początku



## DO WHILE

```
int i = 0;  
do {  
    System.out.println(array[i]);  
    i++;  
} while (i < array.length)
```

*Pętla **do while** zawsze wykona się chociaż raz!*

Warunek jest sprawdzany na końcu pętli, a nie na początku



## FOR EACH – ODCZYT ELEMENTÓW TABLICY

```
int[] array = { 9, 8, 7, 6, 5, 4, 3 };  
  
for (int element : array) {  
    System.out.println(element);  
}
```





## FOR EACH

```
int[] array = { 9, 8, 7, 6, 5, 4, 3 };
```

```
for (int element : array) {  
    System.out.println(element);  
}
```

Typ elementów tablicy



## FOR EACH

```
int[] array = { 9, 8, 7, 6, 5, 4, 3 };
```

```
for (int element : array) {  
    System.out.println(element);  
}
```

Nazwa zmiennej (dowolna), do której przy każdym przejściu pętli przypisywany jest kolejny element tablicy



## FOR EACH

```
int[] array = { 9, 8, 7, 6, 5, 4, 3 };
```

```
for (int element : array) {  
    System.out.println(element);  
}
```

Nazwa tablicy, po której iterujemy



## PRZYKŁAD: zwiększ każdy element tablicy o 10

```
int[] array = { 9, 8, 7, 6, 5, 4, 3 };  
  
for (int i = 0; i < array.length; i++) {  
    array[i] += 10;  
}
```



Wykonaj zadania ze stron 41 i 42.




## METODA ZE ZMIENNĄ LICZBĄ ARGUMENTÓW

```
public void wypiszNazwy(String... nazwy) {  
    for (String nazwa : nazwy) {  
        System.out.println(nazwa);  
    }  
}
```



## METODA ZE ZMIENNĄ LICZBĄ ARGUMENTÓW

```
public void wypiszNazwy(String... nazwy) {  
    for (String nazwa : nazwy) {  
        System.out.println(nazwa);  
    }  
}
```



Metoda przyjmuje zmienną liczbę parametrów (od 0)



## METODA ZE ZMIENNĄ LICZBĄ ARGUMENTÓW

```
public void wypiszNazwy(String... nazwy) {  
    for (String nazwa : nazwy) {  
        System.out.println(nazwa);  
    }  
}
```

Parametr varargs (**nazwy**)  
jest tablicą





## METODA ZE ZMIENNĄ LICZBĄ ARGUMENTÓW

```
public void wypiszNazwy(String... nazwy) {  
    for (String nazwa : nazwy) {  
        System.out.println(nazwa);  
    }  
}  
  
// ...  
wypiszNazwy();  
wypiszNazwy("Katowice");  
wypiszNazwy("Katowice", "Mikołów", "Gliwice");
```



## METODA ZE ZMIENNĄ LICZBĄ ARGUMENTÓW

```
public void wypiszNazwy(String... nazwy) { ... }  
public int obliczSumę(int... składniki) { ... }  
public int oblicz(String komentarz, int... elementy) { ... }  
public void błęd(int... elementy, String komentarz)
```



## METODA ZE ZMIENNĄ LICZBĄ ARGUMENTÓW

- ✓ `public void wypiszNazwy(String... nazwy) { ... }`
- ✓ `public int obliczSumę(int... składniki) { ... }`
- ✓ `public int oblicz(String komentarz, int... elementy) { ... }`
- ✗ `public void błąd(int... elementy, String komentarz)`

**parametr varargs musi być ostatni!**



- W klasie `LoopExample` zadeklarować metodę `int manyArgs (String ... strings)` zwracającą ilość przekazanych argumentów.
- W metodzie `int manyArgs (String... strings)` dodaj dowolną pętlę wypisującą przekazane argumenty (metoda w dalszym ciągu ma zwracać ilość argumentów).
- Stworzyć test dla metody `int manyArgs (String ... strings)` sprawdzający ilość przekazanych argumentów dla `manyArgs ()`.
- Stworzyć test dla metody `int manyArgs (String ... strings)` sprawdzający ilość przekazanych argumentów dla `manyArgs ("S", "D", "A")`.



## TYP DO PRZECHOWYWANIA TEKSTÓW (CIĄGÓW ZNAKÓW) - LITERAŁ

```
String text1 = „A long time ago”;
```

*JVM przechowuje łańcuch znaków w osobnej przestrzeni pamięci*



## TYP DO PRZECHOWYWANIA TEKSTÓW (CIĄGÓW ZNAKÓW) - LITERAŁ

```
String text1 = „A long time ago”;  
String text2 = „A long time ago”;
```



Zmienne text1 i text2 odnoszą się  
do tego samego miejsca w pamięci



## TYP DO PRZECHOWYWANIA TEKSTÓW (CIĄGÓW ZNAKÓW) - LITERAŁ

```
String text1 = „A long time ago”;  
String text2 = „A long time ago”;  
String text3 = new String(„A long time ago”);
```



Tworząc przez new alokujemy nowy obszar pamięci



## NIEZMIENNY - IMMUTABLE

```
String text1 = „A long time ago”;  
text1 = text1 + „ in a galaxy far, far away”;
```





## NIEZMIENNY - IMMUTABLE

```
String text1 = „A long time ago”;  
text1 = text1 + „ in a galaxy far, far away”;
```

`text1` ma wartość „A long time ago in a galaxy far, far away”  
(nie został zmodyfikowany ciąg znaków „A long time ago” tylko  
stworzony nowy String „A long time ago in a galaxy far, far away”)



## KONKATENACJA

```
String text1 = „A long time ago” + „ in a galaxy far, far away”;
```

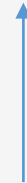


złączenie, czyli konkatencja



## KONKATENACJA

```
Integer liczba = new Integer(10);  
String komunikat = „Liczba” + liczba;
```



Przy konkatencji z obiektem,  
niejawnie wołana jest metoda `toString()`



## KONKATENACJA

```
String a = „Litwo, Ojczyzno moja!”;  
String b = „Ty jesteś jak zdrowie”;  
String c = a.concat(b);
```

↑  
zamiast +



## METODY - valueOf

```
String a = String.valueOf(3.14f);
```

↑  
a = „3.14”;

```
String b = String.valueOf(true);
```

↑  
b = „true”;



## METODY – trim()

```
String a = „    tu są białe znaki    ”;
```

```
String b = a.trim();
```



```
b = „tu są białe znaki”;
```



## METODY – trim()

```
String b = „    tu są białe znaki    “.trim();
```

```
                ↑  
b = „tu są białe znaki”;
```



METODY – toUpperCase(), toLowerCase()

```
String name = „chuck norris”.toUpperCase();
```



```
name = „CHUCK NORRIS”;
```

```
String profession = „TEXAS RANGER”.toLowerCase();
```



```
profession = „texas ranger”;
```





## METODY – substring(from)

from index

```
String a = „Kurs Java w SDA”.substring(5);
```

a = „Java w SDA”



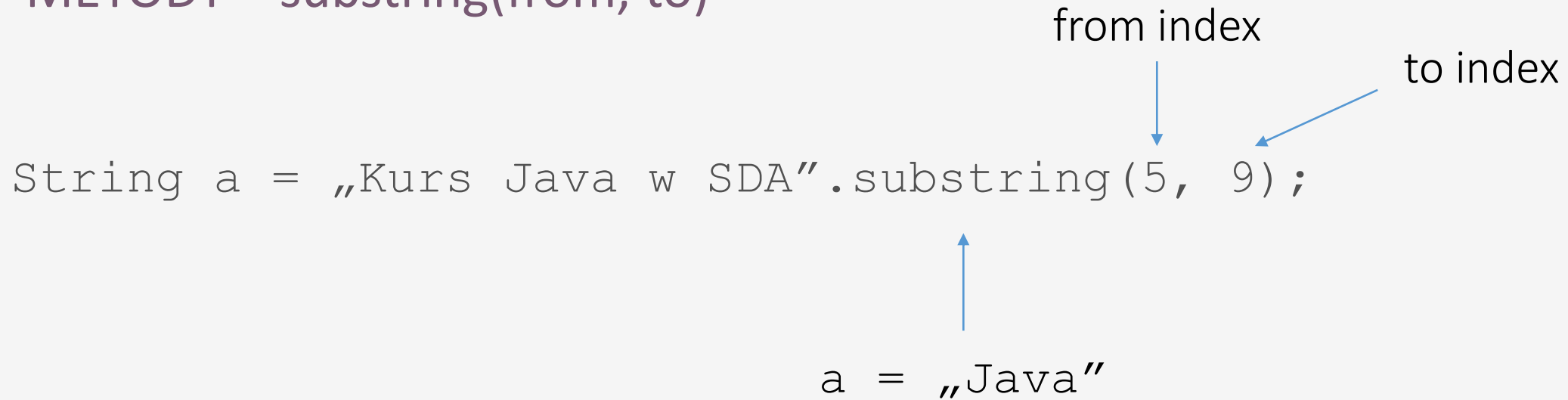
## METODY – substring(from, to)

String a = „Kurs Java w SDA”.substring(5, 9) ;

from index

to index

a = „Java”

The diagram illustrates the substring method call. The text 'String a = „Kurs Java w SDA”.substring(5, 9) ;' is shown. A blue arrow points from the text 'from index' to the number '5' in the method call. Another blue arrow points from the text 'to index' to the number '9'. Below the code, the text 'a = „Java”' is shown, with a blue arrow pointing from it to the '5' in the method call, indicating that the substring starts at index 5.



- W jednym z poprzednio utworzonych projektów dodaj klasę `StringExample`
  - Zaimplementuj metodę `String concatenate(String a, String b)`, która zwróci złączone ciągi `a` i `b`
- Dodaj klasę testową `StringExampleTest`, a w niej test sprawdzający metodę `concatenate`
- Napisz test, w którym przetestujesz metodę `valueOf()`
- Napisz test, w którym przetestujesz metodę `trim()`
- Napisz test, w którym przetestujesz metodę `toUpperCase()`
- Napisz test, w którym przetestujesz metodę `toLowerCase()`
- Napisz test, w którym przetestujesz metodę `toCharArray()` dla słowa "tablica"
  - W teście sprawdź rozmiar zwróconej tablicy
  - W teście sprawdź czy na indeksie 3 znajduje się znak `l`
- Napisz testy, w których przetestujesz metody `substring()` i `substring(from, to)`



## METODY – replace(oldChar, newChar)

```
String imie = „Ola”.replace(`O`, `A`);
```



```
imie = „Ala”
```



## METODY – replace(substring, replacement)

```
String imie = „Ola”.replace(„la”, „lek”);
```



```
imie = „Olek”
```



## METODY – length()

```
int dlugosc = „Ola ma kota”.length();
```



```
dlugosc = 11
```



## METODY – indexOf()

```
int index = „Ola ma kota”.indexOf(`a`);
```



index = 2

(pierwsze wystąpienie znaku `a`)



## METODY – lastIndexOf()

```
int index = „Ola ma kota”.lastIndexOf(`a`);
```



index = 10  
(ostatnie wystąpienie znaku `a`)





## METODY – isEmpty()

```
boolean pusty = "".isEmpty();
```



true



METODY – startsWith(), endsWith()

```
"Bolek".startsWith("Bol");
```

↑  
true

```
"Lolek".endsWith("olek");
```

↑  
true



## METODY – charAt()

```
char znak = "Bolek".charAt(1);
```



znak = `o`



- W klasie `StringExampleTest` napisz testy, w których:
  - przetestujesz metodę `replace()`
  - przetestujesz metodę `length()`
  - przetestujesz metodę `indexOf()`
  - przetestujesz metodę `lastIndexOf()`
  - przetestujesz metodę `isEmpty()`
  - przetestujesz metodę `startsWith()`
  - przetestujesz metodę `endsWith()`
  - przetestujesz metodę `contains()`
  - przetestujesz metodę `charAt()`

# OBJECT



KAŻDA KLASA DZIEDZICZY PO OBJECT

```
class MyClass { }
```

# OBJECT



KAŻDA KLASA DZIEDZICZY PO OBJECT

```
class MyClass { }
```



*podgląd hierarchii klas (ctrl + h)*



## METODY – toString()

Zwraca reprezentację tekstową obiektu

```
System.out.println(new MyClass().toString());
```



```
pl.sda.MyClass@1540e19d
```

*zwykle domyślną implementację nadpisuje się własną*



## METODY – toString()

Zwraca reprezentację tekstową obiektu

```
class Car {  
    String brand;  
    String model;  
  
    @Override  
    public String toString() {  
        return brand  
            + " " + model;  
    }  
}
```

```
Car polo = new Car("VW", "polo");  
System.out.println(polo.toString());
```



„VW polo”





## METODY – toString()

Zwraca reprezentację tekstową obiektu

```
class Car {  
    String brand;  
    String model;  
  
    @Override  
    public String toString() {  
        return brand  
            + " " + model;  
    }  
}
```

```
Car polo = new Car("VW", "polo");  
System.out.println(polo);
```



„VW polo”



## METODY – equals()

Porównuje czy dwa obiekty są sobie równe

```
Object obj1 = new Object();  
boolean areEqual = obj1.equals(obj1);
```



true



## METODY – equals()

Porównuje czy dwa obiekty są sobie równe

```
Object obj1 = new Object();  
Object obj2 = obj1;  
boolean areEqual = obj1.equals(obj2);
```



true



## METODY – equals()

Porównuje czy dwa obiekty są sobie równe

```
Object obj1 = new Object();  
Object obj2 = new Object();  
boolean areEqual = obj1.equals(obj2);
```

↓  
false



## METODY – hashCode()

Zwraca kod obiektu w postaci `int`,  
maksymalnie unikalny  
przy jednoczesnym zachowaniu wydajności metody

*Domyślna implementacja zależy od JVM*



## METODY – getClass()

Zwraca nazwę klasy wraz z pakietem

```
System.out.println(new Car().getClass());
```



„class pl.sda.Car”

# OBJECT



METODY – wait(), notify(), notifyAll()

Metody związane z wielowątkowością.



## METODY – clone()

Tworzy duplikat obiektu. Domyślnie niezaimplementowana.





## METODY – finalize()

Wywoływana w momencie usunięcia obiektu z pamięci.

# ZADANIA



- Wykonaj zadania ze stron 49 i 50.



## equals()

- Operator == sprawdza czy obiekty są w tym samym miejscu w pamięci
- equals() sprawdza czy dwa obiekty są takie same



equals()

- zwrotna

```
x.equals(x) == true
```



## equals()

- zwrotna

```
x.equals(x) == true
```

- symetryczna

Jeśli `x.equals(y) == true`, to `y.equals(x) == true`



## equals()

- zwrotna  $x.equals(x) == true$
- symetryczna Jeśli  $x.equals(y) == true$ , to  $y.equals(x) == true$
- przechodnia Jeśli  $x.equals(y) == true$  i  $y.equals(z) == true$   
to  $x.equals(z) == true$



## equals()

- zwrotna `x.equals(x) == true`
- symetryczna  
Jeśli `x.equals(y) == true`, to `y.equals(x) == true`
- przechodnia  
Jeśli `x.equals(y) == true` i `y.equals(z) == true`  
to `x.equals(z) == true`
- spójna  
Wielokrotnie wywołana zwróci zawsze ten sam wynik (chyba, że zmienią się obiekty)



## equals()

- zwrotna `x.equals(x) == true`
- symetryczna  
Jeśli `x.equals(y) == true`, to `y.equals(x) == true`
- przechodnia  
Jeśli `x.equals(y) == true` i `y.equals(z) == true`  
to `x.equals(z) == true`
- spójna  
Wielokrotnie wywołana zwróci zawsze ten sam wynik (chyba, że zmienią się obiekty)
- przy porównaniu z `null` zawsze zwróci `false`



# EQUALS I HASHCODE



equals()

```
public class Car {
```

```
    String model;
```

```
    @Override
```

```
    public boolean equals(Object o) {
```

```
        if (this == o) return true;
```

```
        if (o == null || getClass() != o.getClass()) return false;
```

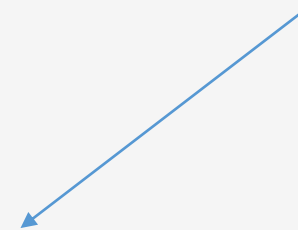
```
        Car car = (Car) o;
```

```
        return Objects.equals(model, car.model);
```

```
    }
```

```
}
```

Metoda wygenerowana przez IntelliJ





## METODY – hashCode()

Ten sam obiekt zawsze zwróci ten sam hashCode (dopóki nie zmieni się jego stan)



## METODY – kontrakt pomiędzy equals() i hashCode()

- hashCode obiektu może się zmienić tylko wtedy, gdy zmieni się pole zawarte w equals()
- Ten sam obiekt (equals() = true) zawsze zwróci ten sam hashCode
- Różne obiekty (equals() = false) mogą zwrócić ten sam hashCode



**ZAWSZE** tworzymy/modyfikujemy **OBIE METODY**: equals() i hashCode()

Do wygenerowania equals() i hashCode() najłatwiej użyć IntelliJ (alt + Ins)

# ZADANIA



- Wykonaj zadania ze strony 53.



```
class Car {  
    //...  
    public void startEngine() {  
        engine.start();  
    }  
}
```

# INSTRUKCJE WARUNKOWE



```
class Car {  
    //...  
    public void startEngine() {  
        engine.start();  
    }  
}
```



A co jeśli pasy nie są zapięte!?



# INSTRUKCJE WARUNKOWE



```
class Car {  
    //...  
    public void startEngine() {  
        if (seatBelt.isFasten()) {  
            engine.start();  
        }  
    }  
}
```

warunek



# INSTRUKCJE WARUNKOWE



```
class Car {  
    //...  
    public void startEngine() {  
        if (seatBelt.isFasten()) {  
            engine.start(); ← Jeśli warunek jest spełniony  
        }  
    }  
}
```

# INSTRUKCJE WARUNKOWE



```
class Car {  
    //...  
    public void startEngine() {  
        if (seatBelt.isFasten()) {  
            engine.start();  
        } else {  
            System.out.println("Fasten your seatbelt!");  
        }  
    }  
}
```

# INSTRUKCJE WARUNKOWE



```
class Car {  
    //...  
    public void startEngine() {  
        if (seatBelt.isFasten()) {  
            engine.start();  
        } else {  
            System.out.println("Fasten your seatbelt!");  
        }  
    }  
}
```

warunek

# INSTRUKCJE WARUNKOWE



```
class Car {  
    //...  
    public void startEngine() {  
        if (seatBelt.isFasten()) {  
            engine.start(); ← Jeśli warunek jest spełniony  
        } else {  
            System.out.println("Fasten your seatbelt!");  
        }  
    }  
}
```

# INSTRUKCJE WARUNKOWE



```
class Car {  
    //...  
    public void startEngine() {  
        if (seatBelt.isFasten()) {  
            engine.start();  
        } else {  
            System.out.println("Fasten your seatbelt!");  
        }  
    }  
}
```

Jeśli warunek nie jest spełniony



# INSTRUKCJE WARUNKOWE



```
class Car {  
    //...  
    public void startEngine() {  
        if (!seatBelt.isFasten()) {  
            System.out.println("Fasten your seatbelt!");  
        } else if (!lights.turnOn()) {  
            System.out.println("Turn on lights!");  
        } else {  
            engine.start();  
        }  
    }  
}
```

# INSTRUKCJE WARUNKOWE



```
class Car {  
    //...  
    public void startEngine() {  
        if (!seatBelt.isFasten()) {  
            System.out.println("Fasten your seatbelt!");  
        } else if (!lights.turnOn()) {  
            System.out.println("Turn on lights!");  
        } else {  
            engine.start();  
        }  
    }  
}
```

warunek 1

# INSTRUKCJE WARUNKOWE



```
class Car {  
    //...  
    public void startEngine() {  
        if (!seatBelt.isFasten()) {  
            System.out.println("Fasten your seatbelt!");  
        } else if (!lights.turnOn()) {  
            System.out.println("Turn on lights!");  
        } else {  
            engine.start();  
        }  
    }  
}
```

warunek 2



# INSTRUKCJE WARUNKOWE



```
class Car {  
    //...  
    public void startEngine() {  
        if (!seatBelt.isFasten()) {  
            System.out.println("Fasten your seatbelt!");  
        } else if (!lights.turnOn()) {  
            System.out.println("Turn on lights!");  
        } else {  
            engine.start(); ← Jeżeli warunek 1 i warunek 2  
                                nie są spełnione  
        }  
    }  
}
```

# INSTRUKCJE WARUNKOWE



```
void printCommentsCount(int count) {  
    if (count == 1) {  
        System.out.println(count + „ komentarz”);  
    } else if (count == 2) {  
        System.out.println(count + „ komentarze”);  
    } else if (count == 3) {  
        System.out.println(count + „ komentarze”);  
    } else if (count == 4) {  
        System.out.println(count + „ komentarze”);  
    } else if (count == 5) {  
        System.out.println(count + „ komentarzy”);  
    }  
}
```

# INSTRUKCJE WARUNKOWE



```
void printCommentsCount(int count) {  
    switch (count) {  
        case 1:  
            System.out.println(count + „ komentarz”);  
            break;  
        case 2:  
            System.out.println(count + „ komentarze”);  
            break;  
        case 3:  
            System.out.println(count + „ komentarze”);  
            break;  
        case 4:  
            System.out.println(count + „ komentarze”);  
            break;  
        case 5:  
            System.out.println(count + „ komentarzy”);  
            break;  
        default:  
            break;  
    }  
}
```

Autor: Marek Sontag

Prawa do korzystania z materiałów posiada Software Development Academy

# INSTRUKCJE WARUNKOWE



```
void printCommentsCount(int count) {  
    switch (count) {  
        case 1:  
            System.out.println(count + „ komentarz”);  
            break;  
        case 2:  
        case 3:  
        case 4:  
            System.out.println(count + „ komentarze”);  
            break;  
        case 5:  
            System.out.println(count + „ komentarzy”);  
            break;  
        default:  
            break;  
    }  
}
```

# ZADANIA



- Wykonaj zadania ze stron 55 i 56.



## Elementy związane z KLASĄ, a nie INSTANCJĄ KLASY - pole

```
class Circle {  
  
    static double PI = 3.14159;  
  
}
```

↑  
PI jest takie same dla  
każdej instancji klasy Circle



## Elementy związane z KLASĄ, a nie INSTANCJĄ KLASY - pole

```
class Circle {  
  
    static double PI = 3.14159;  
  
}
```

słowo kluczowe

static



## Elementy związane z KLASĄ, a nie INSTANCJĄ KLASY - metoda

```
class Math {  
  
    static double sqrt(double a) {  
        // ...  
    }  
  
}
```

} nie potrzeba obiektu Math, aby obliczyć pierwiastek





## Elementy związane z KLASĄ, a nie INSTANCJĄ KLASY - klasa

```
class Outer {  
  
    static class Inner {  
        // ...  
    }  
  
}
```

} Klasa wewnętrzna



## Statyczny import

```
class Math {  
  
    static double sqrt(double a) {  
        // ...  
    }  
  
}
```

```
import pl.sda.Math;
```

```
class Runner {  
  
    void someMethod(double a) {  
        Math.sqrt(a);  
    }  
  
}
```



## Statyczny import

```
class Math {  
  
    static double sqrt(double a) {  
        // ...  
    }  
  
}
```

```
import static pl.sda.Math.sqrt;  
  
class Runner {  
  
    void someMethod(double a) {  
        sqrt(a);  
    }  
  
}
```



## Klasa finalna – nie można dziedziczyć

```
final class FinalClass { }
```

```
class TryToExtend extends FinalClass { } ❌ Błąd kompilacji!
```



## Metoda finalna – nie można nadpisać

```
class MyClass {  
  
    final void methodA() {  
        // ...  
    }  
  
}
```

```
class AnotherClass extends MyClass {  
  
    @Override  
    void methodA() { ✗ Błąd kompilacji!  
        // ...  
    }  
  
}
```



## Pole finalne – nie można zmodyfikować

```
class MyClass {  
    final int someField = 123;  
}
```

✓ OK – zainicjalizowane przy tworzeniu obiektu



## Pole finalne – nie można zmodyfikować

```
class MyClass {  
    final int someField = 123;  
    final int id;  
  
    MyClass(int id) {  
        this.id = id;  
    }  
}
```



OK – zainicjalizowane przy tworzeniu obiektu  
(w konstruktorze)



## Pole finalne – nie można zmodyfikować

```
class MyClass {  
    final int someField = 123;  
    final int id;  
    final String name; ✗ Błąd kompilacji!  
                                (niezainicjalizowane pole finalne)  
  
    MyClass(int id) {  
        this.id = id;  
    }  
}
```





## Zmienna finalna

```
public static void main (String[] args) {  
    final int a = 123;  
    a = 567;  
}
```



Błąd kompilacji!

(nie można zmodyfikować zmiennej finalnej)



## Stałe

```
public class Circle {  
    static final double PI = 3.14159;  
}
```

Taki sam dla każdej  
instancji `Circle`

Pole `PI` nie może być  
zmodyfikowane

# ZADANIA



- Wykonaj zadania ze strony 64.



## DEFINIUJE KONTRAKT

```
interface MusicPlayer {  
  
    public void playSong(String song);  
  
}
```



## DEFINIUJE KONTRAKT

```
interface MusicPlayer {  
    public void playSong(String song);  
}
```

słowo kluczowe  
interface



## DEFINIUJE KONTRAKT

```
interface MusicPlayer {
```

```
    public void playSong(String song);
```

```
}
```

} Lista metod interfejsu



## DEFINIUJE KONTRAKT

```
interface MusicPlayer {  
  
    public void playSong(String song);  
  
}
```

Klasy **implementujące** ten interfejs definiują swoje wersje metody `playSong`



## IMPLEMENTACJA INTERFEJSU

```
class MP3Player implements MediaPlayer {  
  
    @Override  
    public void playSong(String song) {  
        // ...  
    }  
}
```

nazwa interfejsu

słowo kluczowe implements





## METODA DOMYŚLNA

```
interface MusicPlayer {  
  
    public void playSong(String song);  
  
    default String playerName() {  
        return „Music”;  
    }  
}
```

} Domyślna implementacja metody  
playerName()

słowo kluczowe default



## DZIEDZICZENIE

```
interface MusicPlayer {  
    public void playSong(String song);  
}
```

```
interface VideoPlayer {  
    public void playVideo(String video);  
}
```

```
interface Player extends MusicPlayer, VideoPlayer {  
}
```

Zawiera metody obu interfejsów



## DZIEDZICZENIE

```
class ComboPlayer implements Player {  
    @Override  
    public void playSong(String song) {  
        // ...  
    }  
  
    @Override  
    public void playVideo(String video) {  
        // ...  
    }  
}
```



## STAŁE

```
interface MusicPlayer {  
    int bitRate = 192; ← Automatycznie static final  
}
```

# ZADANIA



- Wykonaj zadania ze stron 68 i 69.

# KLASA ABSTRAKCYJNA



klasa abstrakcyjna



```
abstract class Player {
```

```
    abstract public void playSong(String song);
```

```
    public void playVideo(String video) {
```

```
        // ...
```

```
    }
```

```
}
```



metoda abstrakcyjna

# KLASA ABSTRAKCYJNA



klasę abstrakcyjną się dziedziczy (`extends`), a nie implementuje (`implements`) jak interfejs



```
class WAVPlayer extends Player {  
  
    @Override  
    public void playSong(String song) {  
        // ...  
    }  
}
```



- W Javie można **dziedziczyć** tylko **po jednej klasie**, ale **implementować** wiele interfejsów
- Przed **Java 8** nie było **domyślnych implementacji** w interfejsach



# ZADANIA



- Wykonaj zadania ze strony 71.



## ENUM – dla skończonych zbiorów powiązanych elementów

```
enum TShirtSize {  
    S,  
    M,  
    L,  
    XL  
}
```



słowo kluczowe enum



```
enum TShirtSize {  
    S,  
    M,  
    L,  
    XL  
}
```

*Definiujemy jak klasę – w nowym pliku*



```
enum TShirtSize {
```

```
    S,
```

```
    M,
```

```
    L,
```

```
    XL
```



Nazwy elementów - upper case

```
}
```



```
public boolean isXlSize(TShirtSize size) {  
    return TShirtSize.XL.equals(size);  
}
```

odwołanie do elementu XL

# TYP WYLICZENIOWY



```
enum TShirtSize {  
    S(170),  
    M(178),  
    L(185),  
    XL(195)
```

} inicjalizacja wartości

```
private int maxHeight;
```

← pole w enumeratorze

```
TShirtSize(int maxHeight) {  
    this.maxHeight = maxHeight;  
}
```

} konstruktor ustawiający maxHeight

```
}
```



## ENUM – iteracja po wartościach

metoda zwracająca wszystkie wartości

```
for (TShirtSize size : TShirtSize.values()) {  
    // ...  
}
```

# ZADANIA



- Wykonaj zadania ze stron 73 i 74.