

# JAVA

*Jebne kawusie i będzie git*



Wersja 1.02  
13.10.2025

## Spis treści

<b>DZIAŁ 1 - TYPY ZMIENNYCH/STAŁYCH I ICH ZASIĘGI, BUDOWA KODU W JAVIE .....</b>	<b>3</b>
W JAVIE MAMY KILKA RODZAJÓW ZMIENNYCH:.....	3
ZASIĘGI ZMIENNYCH.....	4
RZUTOWANIE (KONWERTOWANIE) TYPÓW (TODO) .....	5
BUDOWA KODU W JAVIE .....	6
<b>DZIAŁ 2 – OPERATORY ARYTMETYCZNE I INNE TAKIE .....</b>	<b>7</b>
OPERATORY ARYTMETYCZNE.....	7
OPERATORY PRZYPISANIA .....	7
OPERATORY PORÓWNANIA .....	8
OPERATORY LOGICZNE (TABELKA P I Q JOINED THE CHAT) .....	8
<b>DZIAŁ 3 – INSTRUKCJE WARUNKOWE I PĘTLE .....</b>	<b>9</b>
IF .....	9
SWITCH.....	10
PĘTLA WHILE ORAZ DOWHILE .....	10
PĘTLA FOR .....	11
<b>DZIAŁ 4 – TABLICE, TABLICE WIELOWYMIAROWE.....</b>	<b>12</b>
TABLICE (JEDNOWYMIAROWE) .....	12
TABLICE WIELOWYMIAROWE .....	13
PĘTLA FOR-EACH – WARIACJA PĘTLI FOR .....	14
<b>DZIAŁ 5 – FUNKCJE W JAVIE .....</b>	<b>15</b>
PO CO ROBIMY FUNKCJE?.....	15
NO TO ROBIMY FUNKCJE.....	15
PRZECIĄŻENIA FUNKCJI .....	17
<b>DZIAŁ 6 – BIBLIOTEKI (I INNE TAKIE) W JAVIE .....</b>	<b>18</b>
KRÓTKO O BIBLIOTEKACH .....	18
SCANNER .....	18
MATH – ZAAWANSOWANE OBLICZENIA MATEMATYCZNE .....	19
MATH.RANDOM() – LICZBY PSEUDOLOSOWE I CO TO W OGÓLE ZNACZY .....	19
KLASA STRING – OPERACJE NA ZMIENNYCH TEKSTOWYCH.....	20
<b>DZIAŁ 7 – OPERACJE NA PLIKACH .....</b>	<b>21</b>
<i>Odczytywanie danych z pliku.....</i>	<i>21</i>
<b>DZIAŁ 8 – WYJĄTKI – CZYLI SYSTEMY IDIOTO-ODPORNE .....</b>	<b>24</b>
TRY-CATCH .....	24
RZUCANIE WYJĄTKÓW (THROW) .....	26
<b>DZIAŁ 9 – OPERACJE BINARNE I NA LICZBACH BINARNYCH.....</b>	<b>27</b>
<b>DZIAŁ 10 – KLASY W JAVIE.....</b>	<b>28</b>
PEAK PROGRAMOWANIA OBIEKTOWEGO .....	28
PUBLIC, PROTECTED, PRIVATE – MODYFIKATORY DOSTĘPU .....	29
KONSTRUKTOR KLASY .....	31
METODY STATYCZNE .....	32

DZIAŁ 11 – DZIEDZICZENIE KLAS (TODO) .....	33
DZIAŁ 12 – KLASY ABSTRAKCYJNE (TODO).....	34

*Może i nie zrobi z ciebie programisty, ale za to żarty w środku też są chujowe :DD*

# Dział 1 - Typy zmiennych/statycznych i ich zasięgi, budowa kodu w Javie

W Javie mamy kilka rodzajów zmiennych:

*boolean* – zmienna *true/false*, w przeciwieństwie do innych języków programowania w Javie nie mamy opcji przypisania wartości 0 albo 1 – musimy *true/false*

*Dopiero zacząłem a już mamy powody, żeby jebać ten język*

*byte* – zmienna liczbowa o długości (nie zgadniesz) jednego bajta (8 bitów). Przyjmuje dane w zakresie od -127 do 127 (8 bitów, czyli  $2^8$ )

*char* – zmienna 2-bajtowa reprezentująca jakiś znak, np. '+'. Określić go możemy również za pomocą kodu z tablicy ASCII (jak i również potem zamienić znak na wartość kodu ASCII)

*ASCII to opracowany standard kodowania znaków, tablice znajdziesz na necie*

*short* – zmienna liczbowa o długości 2 bajtów. Przyjmuje wartości od -32768 do 32768

*int* – klasyczna zmienna liczbowa o długości 4 bajtów. Pozwala przyjmować dane od minus do plus 2147483648

*Wiem, że nie przeczytasz tej liczby, więc ci powiem, że to lekko ponad 2 mld*

*long* – jak sama nazwa wskazuje, jest kurwa długi, dwa razy większy niż *int*. Pamiętaj, że przy użyciu tej zmiennej na końcu liczby musisz dodać L

*float* – liczba zmiennoprzecinkowa. Ta jest mniejsza, pozwala na przechowanie 6-7 liczb po przecinku

*Na końcu liczby musisz dodać „f”. W innym przypadku zostanie potraktowana jako double i otrzymasz błąd typu*

*double* – większa zmiennoprzecinkowa. Pozwala na użycie do 16 cyfr.

A no i jeszcze *void* jest – zmienna, która jest pusta, nie ma żadnej wartości. Void'em oznaczamy funkcje, które nic nie zwracają.

Opisane powyżej typy danych to tak zwane typy prymitywne – najbardziej podstawowe. Na podstawie tych typów będziemy budować bardziej złożone struktury danych.

Za typ prymitywny nie jest uznawany typ *String*

*String* – zmienna tekstowa, jako jedyna typ ma pisany wielką literą

Stałe tworzymy za pomocą dodania *final* przed typem zmiennej

```

Main.java
public class Main {
    public static void main(String[] args) {
        boolean prawda = true;
        byte liczbaMała = 120;
        char znak = 'a';
        char znakZKodu = 97; //97 w kodzie ASCII to literka a
        short liczba1 = 1000;
        int liczba2 = 10000000;
        long liczba3 = 123123123123123L;
        float zmiennaPrzecinkowa = 3.141592f;
        final double zmiennaPrzecinkowa2 = 3.141592; //nie zmienimy wartosci
        String tekst = "Ała ma kota, kot ma ale";
    }
}

```

*Jak coś to te zielone podkreślenia od tego, że Intelij jest po prostu głupi*

## Zasięgi zmiennych

Zmienne które tworzymy mają swój zasięg „istnienia”. W tym zasięgu możesz jej używać. Przykładowe zasięgi zmiennych na screenie poniżej:

```

Main.java
public class Main {

    public static int globalna = 444; //tej zmiennej mozesz uzywac w roznnych funkcjach
                                     //w ramach klasy Main

    public static void main(String[] args) {
        int zmiennaMain = 456; //zmienna lokalna w głównym kodzie programu

        if(zmiennaMain == 456) {
            String lokalna1 = "AAA"; //ta zmienna jest tylko w zasięgu if'a
        }
    }

    public void test(){ no usages
        String lokalna2 = "AAA"; //ta zmienna jest tylko w zasięgu funkcji test
    }
}

```

Miej na uwadze, że zmienna nazwana jako globalna nie jest tak do końca globalna. Java niezbyt pozwala na takie typowe zmienne globalne, a próba implementacji czegoś co je przypomina jest bardziej złożona – pokażę ją później.

Skoro zmienna np. wewnątrz pętli działa tylko w tym jednym miejscu, to nic nie stoi na przeszkodzie, aby w dwóch różnych pętlach utworzyć zmienną o tej samej nazwie ;DD

```
Main.java
public static void main(String[] args) {
    int zmiennaMain = 456; //zmienna lokalna w głównym kodzie programu

    if(zmiennaMain == 456) {
        String lokalna1 = "AAA"; //ta zmienna jest tylko w zasięgu if'a
    }

    if(zmiennaMain == 111){
        String lokalna1 = "BBB"; //zmienna nazwana tak samo
    }
}
```

Staraj się jednak używać tego z głową, nadużywanie tego może doprowadzić do tego, że kod będzie nieczytelny.

## Rzutowanie (konwertowanie) typów (TODO)

## Budowa kodu w Javie

Ja piszę to z perspektywy osoby, dla której pisanie o tym może wydawać się aż nader oczywiste, ale napisać nie zaszkodzi a może kogoś oświecę.

Budowa kodu jest oparta o klasy, każda z klas ma swoje funkcje (metody), a w nich są już instrukcje zawarte w klamrach {}

Każdy program zamiera główną klasę *Main*, a w niej znajduje się funkcja *main()*. W dalszej części dowiesz się, że każda funkcja musi zwracać jakąś wartość (tak jak w matematyce). Funkcja *main()* to wyjątek, nie zwraca ona żadnej wartości, nie posiada więc ona na końcu polecenia „return”, ale jeżeli chcesz, to możesz go użyć. Program czyta kod domyślnie tylko z funkcji *main()*, z góry do dołu. Inne funkcje program czyta dopiero po ich wywołaniu.

```
public class Main {  
  
    public static void test(){ 1 usage  
        String lokalna2 = "AAA"; //to sie wykona dopiero przy wywołaniu tej funkcji z funkcji main  
    }  
  
    public static void main(String[] args) {  
  
        //jakies instrukcje  
  
        test(); //wywołanie funkcji test  
  
        return; //niekonieczne, ale można  
    }  
  
}
```

## Dział 2 – Operatory arytmetyczne i inne takie

### Operatory arytmetyczne

Dobra to jest tak banalne że nie będę się tu za bardzo rozwodził.

+ - \* / te 4 znasz na pewno.

Z takich ciekawszych mamy jeszcze:

% - modulo – zwraca resztę z dzielenia: np.  $5 \% 2$  da nam 1

++ - inkrementacja – zwiększenie zmiennej o 1, np.  $i++$  lub  $++i$

*Zamiast pisać  $x = x + 1$ , możemy krócej  $x++$*

-- - to się chyba dekrementacja nazywa czy coś nie wiem, ale to obniża wartość zmiennej o 1

UWAGA: inkrementacje/dekrementacje możemy przeprowadzać na dwa sposoby. Gdy przekazujemy gdzieś zmienną jako argument jednocześnie zmieniając ją o 1, zachodzą pewne różnice między inkrementacją z plusami z przodu (preinkrementacją) a przypadkiem kiedy plusy mamy z tyłu (postinkrementacją).

```
Main.java
public static void main(String[] args) {
    int x = 5;
    System.out.println(x++); //najpierw przekaże do wypisania, potem zwiększy o 1
    x = 5;
    System.out.println(++x); //najpierw zwiększy, potem przekaże do wypisania
}
```

Za pierwszym razem wypisze 5, za drugim 6.

### Operatory przypisania

= - no to po prostu przypisujemy do zmiennej jakąś wartość

Do tego dochodzą nam jeszcze operatory typu += -= \*= /=, które skracają zapis działania na tej samej zmiennej:

$x += 3$  to to samo co  $x = x + 3$

$x *= 5$  to to samo co  $x = x * 5$

Do takich operatorów zaliczają się również operatory działań bitowych, ale to na razie zostawiamy w spokoju.



## Operatory porównania

Używane do tworzenia warunków np. do instrukcji *if*

Do operatorów porównania należą:

$X == Y$  - jest równe

$X != Y$  nie jest równe

$X < > Y$  mniejsze/większe

$X >= <= Y$  większe lub równe/ mniejsze lub równe

## Operatory logiczne (tabelka p i q joined the chat)

Również przydatne do różnych warunków, takich bardziej złożonych

Mamy 3 takie operatory

$\text{warunek1} \ \&\& \ \text{warunek2}$  – logiczne AND

$\text{warunek1} \ || \ \text{warunek2}$  – logiczne OR

$!\text{warunek1}$  – logiczne NOT – po prostu odwraca stan logiczny z *true* na *false* i odwrotnie

Operatory logiczne będą ci służyć przede wszystkim do łączenia ze sobą warunków np. w instrukcjach warunkowych

Oczywiście wszystkie operatory tak samo jak w matematyce tak i tu mają swoją hierarchię.

- `()` - Parentheses
- `*`, `/`, `%` - Multiplication, Division, Modulus
- `+`, `-` - Addition, Subtraction
- `>`, `<`, `>=`, `<=` - Comparison
- `==`, `!=` - Equality
- `&&` - Logical AND
- `||` - Logical OR
- `=` - Assignment

Tutaj masz screena bo mi się przepisywać nie chciało. Czas nałaził z pisaniem tego ;-;

## Dział 3 – Instrukcje warunkowe i pętle

Aby program nie działał prostoliniowo – wykonywał wszystkie instrukcje po kolei nie zważając na dane, mamy coś takiego jak instrukcje sterujące, które jak sama nazwa wskazuje sterują programem.

### If

```
● ● ● Main.java
public static void main(String[] args) {
    int zmienna1 = 12;
    int zmienna2 = 12;
    if(zmienna1 == zmienna2){
        //jeżeli zmienne mają te same wartości (czyli warunek w nawiasie zwraca prawdę),
        // wykonane zostaną instrukcje z tej klamry
        //jeżeli warunek zwraca fałsz, instrukcje zostaną pominięte
    }
}
```

Instrukcję warunkową `if` możemy rozszerzyć o polecenia „`else`” i „`elseif`”. Pierwszego używamy, gdy chcemy, aby dana część kodu została wykonana WTEDY I TYLKO WTEDY, gdy nasz warunek okaże się fałszywy. Przykład poniżej:

```
● ● ● Main.java
public static void main(String[] args) {
    int zmienna1 = 12;
    int zmienna2 = 12;
    if(zmienna1 == zmienna2){
        //jeżeli zmienne mają te same wartości (czyli warunek w nawiasie zwraca prawdę),
        // wykonane zostaną instrukcje z tej klamry
        //jeżeli warunek zwraca fałsz, instrukcje zostaną pominięte
    } else if (zmienna1 * 2 == zmienna2) {
        //jesli pierwszy if to fałsz, sprawdzany jest kolejny warunek
    }
    else {
        //jesli wszystkie if'y są fałszywe, wykonywany jest else
    }
}
```

## Switch

Switch to kolejna instrukcja warunkowa, podobna do if'a, jednak nastawiony na więcej warunków niż if. Dodatkowo pozwala na realizację kilku warunków na raz, w tym jednego domyślnego.

```
public static void main(String[] args) {
    int zmienna1 = 12;
    switch (zmienna1) {
        case 12: //wykona sie jesli zmienna bedzie rowna 12
            System.out.println("warunek 1");
            break; //break przerywa sprawdzanie kolejnych warunków switcha
        case 11:
            System.out.println("warunek 2");
            zmienna1 = 100;
            //jesli break nie ma, case bedzie dalej sprawdzal warunki
        case 100:
            System.out.println("warunek 3");
            break;
        default: //ten fragment wykona sie zawsze, o ile wcześniejszy nie przerwiemy za pomocą break
            System.out.println("domyślnie");
    }
}
```

## Pętla while oraz doWhile

Teraz pora na dwie pierwsze pętle – *while* oraz bardzo podobną *dowhile*. Polecenia w pętli będą wykonywane w kółko, dopóki warunek pętli jest prawdą. Warunek będzie sprawdzany:

- przed każdym powtórzeniem poleceń pętli – w przypadku pętli *while*
- po każdym wykonaniu poleceń pętli – w przypadku pętli *dowhile*

Dzięki temu polecenia znajdujące się w pętli *dowhile* zostaną wykonane PRZYNAJMNIEJ raz, przed pierwszym sprawdzeniem.

```
byte odliczanie = 10;
while (odliczanie >= 0) {
    System.out.println(odliczanie);
    odliczanie--;
}
```

*Ta pętla będzie wypisywać nam kolejne liczby od 10 do 0 włącznie.*

```

Main.java
byte odliczanie = -10;
do {
    System.out.println(odliczanie);
} while (odliczanie >= 0);
//warunek jest sprawdzany dopiero po wykonaniu instrukcji

```

Warunek tej pętli `doWhile` zwraca fałsz, jednak zanim zostanie to sprawdzone, instrukcje zostaną wykonane

## Pętla `for`

Jest to najbardziej zaawansowana pętla, daje nam najwięcej możliwości.

```

Main.java
public static void main(String[] args) {
    for(int i = 0; i < 10; i++) {
        System.out.println(i);
    }
}

```

Pętla w swoim warunku (w zwykłych nawiasach) ma 3 wyrażenia oddzielone od siebie średnikami.

W pierwszej części tworzymy sobie zmienną pomocniczą `int i = 0;`

*Zmienna pomocnicza w pętli nazywana jest zwyczajowo **iteratorem***

W drugiej części podajemy warunek, dopóki pętla ma być wykonywana.

W trzeciej części podajemy instrukcję, która będzie wykonywana przy każdym „powtórzeniu” (po bożemu mówimy **iteracji**) pętli.

W tym przypadku zmienna `i` będzie co każde przejście pętli zwiększana o 1, dopóki jej wartość nie będzie wynosić 10. Wtedy pętla zostanie przerwana.

Koniec końców pętla zostanie wykonana 10 razy.

## Dział 4 – Tablice, tablice wielowymiarowe

### Tablice (jednowymiarowe)

Założmy scenariusz w którym masz bojowe zadanie utworzyć 15 zmiennych i nadać im jakieś wartości całkowite. Pisanie po kolei...

```
Int x1 = 10;
```

```
Int x2 = 20;
```

```
Int x3 = 2137;
```

...jest dość czasochłonne i męczące. I wtedy wchodzi tablica, czyli grupa jakiś zmiennych. Aby utworzyć tablicę jakiś zmiennych należy napisać:

```
typ_danych[] nazwa_tablicy
```

lub, jeśli wolimy od razu opisać długość tablicy:

```
typ_danych[] nazwa_tablicy = new typ_danych[długosc_tablicy]
```

Czyli wracając do naszego bojowego zadania, tablica 15 zmiennych całkowitych powstanie za pomocą:

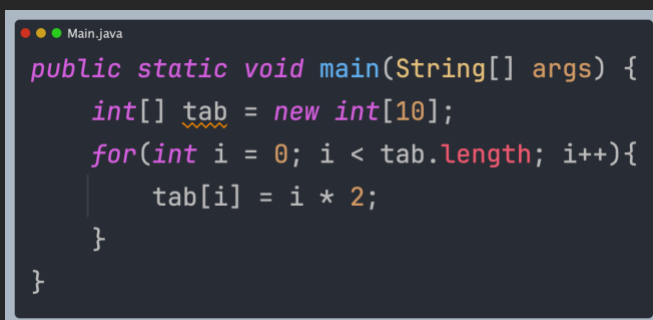
```
int[] tablica = new int[15];
```

Poszczególne elementy (nasze zmienne) w tablicy są numerowane indeksem. Aby odczytać wartość albo przypisać nową, używamy:

```
tablica[5] = 1234;
```

*PAMIĘTAJ! Elementy tablicy są numerowane od 0, czyli ostatnim elementem naszej tablicy będzie tablica[14], a pierwszym tablica[0]. tablica[15] wyrzuci błąd „index out of range” – czyli indeks wychodzi poza zakres tablicy*

Elementy tablicy można wprowadzać za pomocą pętli – to bardzo wygodny sposób:



```
public static void main(String[] args) {  
    int[] tab = new int[10];  
    for(int i = 0; i < tab.length; i++){  
        tab[i] = i * 2;  
    }  
}
```

`tab.length` zwraca nam informacje o długości tablicy (pamiętaj jednak, aby jej wcześniej tę długość nadać)

Ogółem tak jak teraz myślę no to jeśli chodzi o tablice no to teraz na matmie mamy ich oczywisty odpowiednik – macierze. Tablice jednowymiarowe to takie macierze o jednym wierszu XD

Tablice w Javie są statyczne – mają stałą długość której nie można zmienić. Aby zmienić długość, trzeba utworzyć nową – odpowiednio większą – tablice i przepisać wartości z poprzedniej. No ale o tym na razie nie myśl – przyjdzie zabawa na takie kombinowanie.

## Tablice wielowymiarowe

Teraz założmy inny scenariusz: musisz wprowadzić dane 20 uczniów - ich imiona i nazwiska – w osobnych zmiennych. Można wykonać dwie tablice String’ów, jedną na imiona, drugą na nazwiska, ale prościej jest wykonać tablice wielowymiarową. W naszym wypadku będzie to tablica dwuwymiarowa. Jak wykonać takie zadanie?

```
Main.java
String[][] dane_uczniow = new String[20][2];
for(int i = 0; i < 10; i++){
    String imie = scanner.nextLine(); //scanner.nextLine służy do odczytywania z konsoli
    String nazwisko = scanner.nextLine(); //więcej o tym w dalszej części
    dane_uczniow[i][0] = imie;
    dane_uczniow[i][1] = nazwisko;
}
```

W powyższym przykładzie tworzymy sobie dwuwymiarową tablicę zmiennych tekstowych (2 wiersze i 20 kolumn – albo na odwrót w sumie, jeden pies), a następnie w pętli za pomocą Scannera odczytujemy z konsoli imię i nazwisko, i przypisujemy je w odpowiednie miejsca w tablicy

Ważne jest aby pamiętać który „wymiar” tablicy miał 20 zmiennych a który tylko 2

*W powyższym przykładzie znajduje się błąd. Potrafisz go namierzyć?*

No i jeśli chodzi o te tablice wielowymiarowe to tam jakiś limitów nie ma, jak masz potrzebę stworzyć 4-wymiarową to też nie ma z tym problemu.

Jeśli chodzi o wypełnianie takich tablic danymi, to robi się to po prostu przez zwykłe zagnieżdżanie pętli. Każda pętla jest odpowiedzialna za jeden wymiar.

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    int[][][] tesseract = new int[3][3][3][3];

    for (int i = 0; i < tesseract.length; i++) {
        for (int j = 0; j < tesseract[i].length; j++) {
            for (int k = 0; k < tesseract[i][j].length; k++) {
                for (int l = 0; l < tesseract[i][j][k].length; l++) {
                    tesseract[i][j][k][l] = scanner.nextInt(); //nextInt odczytuje konkretnie liczbę z konsoli
                }
            }
        }
    }
}
```

*W takich miejscach pamiętaj o zasięgach zmiennych – pętla wewnętrzna nie może mieć już iteratora i – potrzebny jest jakiś inny. Zwyczajowo bierze się kolejne po i litery alfabetu*

Pisałem wyżej że tablice są statyczne – są również wersje tablic dynamicznych, do których możemy sobie swobodnie dopisywać dane i żyć jak nam się podoba, ale nie wiem czy będzie to poruszane na zajęciach – jak coś to zaktualizuje

## Pętla *for-each* – wariacja pętli *for*

Szczerze mówiąc do momentu pisania tego dokumentu nie wiedziałem o istnieniu tej pętli, a wydaje się całkiem przydatna. Jest to pętla która w każdej iteracji przypisuje nam do zmiennej kolejne wartości z naszej tablicy. Dla Pythonowców – jest to odpowiednik pętli `for ... in ...`

```
public static void main(String[] args) {
    int[] arr = {4, 5, 8}; //sposób na nadanie tablicy od razu wartości
    for(int x : arr){
        System.out.println(x);
    }
}
```

## Dział 5 – Funkcje w Javie

Funkcje to odseparowane od głównego przebiegu programu fragmenty kodu. W sumie są one całkiem podobne do tych matematycznych: dla danego argumentu robią one jakąś czarną magię w środku i zwracają jakiś wynik. Są jednak pewne różnice:

Funkcje programistyczne opierają się nie tylko na liczbach (np. tekst albo tablice)

Jeśli chodzi o tą matematyczną definicję (przyporządkowanie), no to tam mówią, że przyporządkowany jest dokładnie jeden element ze zbioru wartości. Tutaj to może być trochę naciągane

### Po co robimy funkcje?

Najczęstsze powody są dwa:

Aby skrócić kod – w funkcje możemy umieścić fragment kodu, który się powtarza w różnych miejscach

Aby zadbać o przejrzystość kodu – można kod tak ładnie porozdzielać i będzie bardziej czytelny

### No to robimy funkcje

To co zwróci nasza funkcja (jaki typ danych) określamy tak samo jak typ danych w zmiennej:

*Int funkcja()* zwróci liczbę całkowitą

*Char zamienNaWielka()* zwróci pojedynczy znak

*Czy pamiętasz system kodowania z którego korzysta java w typie danych char?*



W zwykłych nawiasach po nazwie funkcji podajemy argumenty jakie nasza funkcja będzie przyjmować. Przykładowo funkcja licząca silnie musi przyjąć jako argument liczbę której silnie musi policzyć

```
Main.java
static long silnia(int x) { 1 usage
    long output = 1L;
    for(int i = 2; i <= x; i++) {
        output *= i;
    }
    return output;
}
```

*W przypadku tworzenia funkcji w przestrzeni klasy Main potrzebne może być dodanie słowa static na początku definicji funkcji. O tym czy musisz czy nie dowiesz się w zależności czy program się uruchomi czy nie XD*

Co do argumentów funkcji ważne jest, że jeśli przekazujesz do funkcji wartość z jakiejś zmiennej, to funkcja w środku będzie operować na kopii tej wartości, a oryginalna zmienna zostanie nietknięta. Oczywiście można to obejść – zamiast argumentu zrobić sobie zmienną globalną i modyfikować ją zarówno od strony main'a jak i funkcji. Jednak to dość średnie rozwiązanie i Tabakow może być zły bo kod nieczytelny

Funkcje można używać do tzw. rekurencji. Funkcje rekurencyjne to takie które wywołują same siebie. Algorytmy oparte o rekurencję są czasem szybsze niż tradycyjne iterowanie w pętli.

Spróbujmy zaadaptować funkcję silnia do postaci rekurencyjnej:

```
Main.java
static long silnia(int x) { 2 usages
    if(x == 1){
        return 1; //to jest moment który przerywa nam rekurencję
    }
    else {
        return (long) x * silnia(x - 1);
        //program zwraca wynik mnożenia x i tego co zwróci funkcja silnia(x-1)
        //silnia(x-1) natomiast wynik mnożenia x-1 i wyniku funkcji silnia(x-2)
        //i tak dalej do 1 kiedy to funkcja zostanie zakończona
    }
}
```

Tutaj przepływ działania jest trochę inny, bo w każdym kolejnym wywołaniu funkcji x jest coraz mniejszy

## Przeciążenia funkcji

Nazewnictwo różnych funkcji nie zawsze musi się różnić. Funkcje możemy przeciążyć, nadając jednej nazwie funkcji kilka jej definicji, różniących się przyjmowanymi argumentami oraz tym co tam mają w środku. Mówimy wtedy, że dana funkcja może przyjmować różne argumenty, w zależności od nich program będzie wiedział którą definicję „wybrać”. Może to wydawać się lekko zagmatwane, no ale masz tu screena i zrozumiesz:

```
Main.java
static int aaa(int x){ 1 usage
    System.out.println(x);
    return x;
}

static String aaa(String y){ 1 usage
    System.out.println(y);
    return y;
}

public static void main(String[] args) {
    int output1 = aaa(x: 15);
    String output2 = aaa(y: "aaa");
}
```

Program nie gubi się w nazewnictwie, bo wybiera funkcje na podstawie argumentów. Ważne też aby człowiek się nie pogubił. Dlatego przeciążenia funkcji. Stosujemy wobec funkcji robiących to samo, tylko mogących przyjąć różne rodzaje danych. Przykładem funkcji przeciążonej jest **print**, który może przyjąć int’a, char’a, String’a czy co tam dusza zapagnie.

```
public static void main(String[] args) {
    aaa
    aaa(int x) int
    aaa(String y) String
    Press ^Space to see non-imported classes Next Tip
```

*Tak to wygląda jak wpisujemy nazwę i program nam podpowiada*

## Dział 6 – Biblioteki (i inne takie) w Javie

### Krótko o bibliotekach

Jak w bardzo wielu językach programowania – to co dostajemy na początku to tylko załączek możliwości całego systemu. Biblioteki to zestawy funkcji i instrukcji, które rozszerzają możliwości systemu.

Z pewnością znana jest Ci już biblioteka `System`, w której znajdują się funkcja `print()` służące do wypisywania na konsoli jakiś naszych danych

Java jest o tyle zjebana, że samo wczytywanie danych z konsoli robione jest w całkowicie inny (i pojebany dla zwykłego użytkownika) sposób

### Scanner

Otoż potrzebujemy do tego czegoś co się nazywa *Scanner*. Jest to klasa obecna w bibliotece `Java.util`, zatem aby jej użyć w danym pliku, na jego początku (jeszcze przed klasą `main`) musimy wykonać import.

```
import java.util.Scanner;

public class Main {
```

Następnie w miejscu gdzie chcemy ze *scannera* skorzystać musimy utworzyć obiekt *Scanner*

```
Scanner sc = new Scanner(System.in);
```

O tym co tu się dokładnie dzieje, czym jest to `new` i czemu to tak wygląda, dowiesz się za chwilę w dziale z klasami. Teraz natomiast musisz wiedzieć że tak się tworzy *Scanner* korzystający z domyślnego wejścia systemowego (`System.in` – w naszym przypadku to nasza konsola)

```
import java.util.Scanner;

public class Main {

    static Scanner sc = new Scanner(System.in);

    public static void main(String[] args) {
        int x = sc.nextInt();
        System.out.println(x);
    }
}
```

*Jakbyś się zastanawiał/a na co ten static – jeśli w funkcji static używasz jakiegoś obiektu spoza tej funkcji, to ten obiekt również musi być statyczny*

Tak to mniej więcej wygląda – aby odczytać dane z konsoli używamy nazwy naszego *scannera* i po kropce wybieramy metodę (*nextInt()* odczyta nam *int'a*, *nextDouble()* *double'a*, a *nextLine()* całą linię, wszystkie metody przejrzyś w menu z podpowiedziami po napisaniu *sc.* – wyświetli się ta taka lista fajna i masz opisy).

## *Math* – zaawansowane obliczenia matematyczne

Do bardziej zaawansowanych funkcji matematycznych zbawienna może okazać się klasa *Math*, w której znajduje się mnóstwo funkcji matematycznych. Ta klasa nie wymaga importu, jednak aby skorzystać z jej metody musimy za każdym razem pisać na początku *Math*.

Z ciekawszych rzeczy, które zawiera, no to liczenie wartości bezwzględnej, wszystkie funkcje trygonometryczne i cyklometryczne (i inne których nie znamy XD), potęgowanie, pierwiastkowanie, zaokrąglanie w dół w górę. A do tego najważniejsze...

## *Math.random()* – liczby pseudolosowe i co to w ogóle znaczy

Generowanie liczb losowych to ciekawy problem w informatyce w ogóle, ale tutaj opowiem o nim po krótce.

Ze względu na specyfikę komputerów – ich architekturę i organizację, to że operują na zerach i jedynkach, niemożliwe jest generowanie całkowicie losowych danych – albo coś jest albo czegoś nie ma, 1 albo 0. Intuicja mi podpowiada, że całkowitą losowość mogą zapewnić komputery kwantowe (albo pierdolę farmazony). My jednak mówimy o śmiesznej Javie, zatem mamy do dyspozycji tylko pseudolosowość.

Pseudolosowość to wymysł informatyków mających na celu zastąpienie typowej losowości w najbardziej wierny sposób. Jest to zestaw operacji matematycznych które na podstawie naszego wejścia (ziarna), mają nam dać jakiś (z naszej perspektywy losowy) wynik. Ziarno (seed) może ci się kojarzyć z *Minecraftem* – i to dobra poszlaka, bo działa to praktycznie tak samo. W starszych wersjach w takim przypadku ziarnem generatora jest najczęściej zegar systemowy. Zegar systemowy reprezentuje bardzo dużą liczbą wyznaczoną przez liczbę sekund od początku czasu UNIX'owego, zmienia się dość dynamicznie. Pseudolosowość polega więc na tym – że jeśli jakimś cudem ziarno się powtórzy, to wynik dla dwóch losowań będzie taki sam.

Zwykle jednak się nie powtarza (chyba że zrobimy to celowo), i tego się trzymajmy.

Jak tam więc ta pseudolosowość w Javie wygląda?

Mamy metodę *Math.random()* która zwróci nam pseudolosową liczbę zmiennoprzecinkową (*double*) z zakresu od 0.0 do 1.0.

Możesz się zastanawiać czemu tak dziwnie taki ułamek, ale to jest bardzo sprytne rozwiązanie, bo wystarczy pomnożyć ten ułamek przez liczbę X, aby mieć wartość losową z zakresu od 0 do X.

*Zastanów się, jak zmienić zakres aby pozbyć się 0 i wygenerować liczbę w zakresie np. 50-100? Rozwiązanie na zrzucie poniżej*

```
public static void main(String[] args) {
    double random = Math.random(); //liczba w zakresie 0.0 - 1.0
    double randomTo100 = Math.random() * 100; //w zakresie 0 - 100
    double random50to100 = 50 + Math.random() * 50; //w zakresie 50-100
}
```

*No więc po prostu dodajemy na początku liczbę będącą początkiem naszego zakresu, a nasz mnożnik zmniejszamy o tą właśnie liczbę.*

## Klasa String – operacje na zmiennych tekstowych.

Jak zapewne pamiętasz (jak nie pamiętasz to nie wiem co ty tu w ogóle robisz), String nie jest pierwotnym typem danych – między innymi dlatego pisany jest wielką literą. String to cała klasa, mająca poza typem danych sporo ciekawych metod. Pomijając to że na zmiennych String możemy wykonywać operacje arytmetyczne – dodawać przypisywać i inne takie. To jeszcze mamy takie cuda jak *length* – które zwracają długość łańcucha tekstowego, *indexOf()* szukający miejsca w którym znajduje się w tekście podany znak, albo operacje na samej zmiennej takie jak *toUpperCase()*. Mnóstwo tego i trochę mi się nie chce rozpisywać. Listę metod razem z opisami ich działania masz [tutaj](#). Testuj do woli.

```
public static void main(String[] args) {
    String wiersz1 = "Ała ma kota";
    String wiersz1glosno = wiersz1.toUpperCase();
    String wiersz2 = "Kot ma Alę";
    int alaPos = wiersz2.indexOf("Alę"); //zwróci nam pozycję od której zaczyna się tekst Alę
    //w tym przypadku będzie to 7
    System.out.println("głosno: " + wiersz1glosno);
    System.out.println("Pozycja: " + alaPos);
}
```

A tutaj wynik:

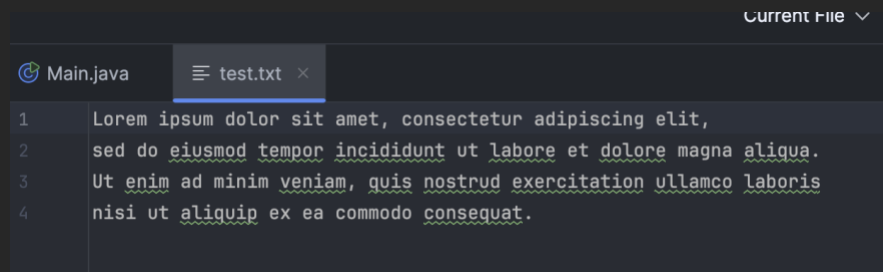
```
/Users/oliwander/Library
głosno: ALA MA KOTA
Pozycja: 7
```

## Dział 7 – Operacje na plikach

Nie jestem pewien czy ten dział przyda się w kontekście zajęć, no ale to dość ważny dział więc o nim napiszę. Tworzy nam to alternatywne źródło danych poza tą nudną konsolą.

### Odczytywanie danych z pliku

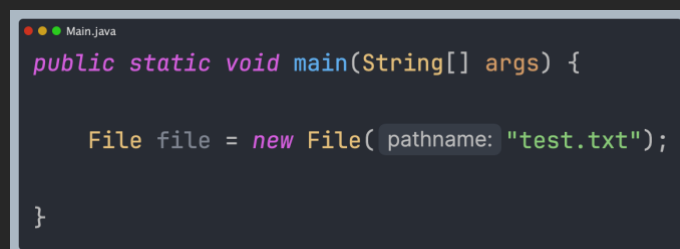
Dla testów utwórzmy sobie testowy plik o podanej zawartości



```
Current File ▾  
Main.java test.txt ×  
1 Lorem ipsum dolor sit amet, consectetur adipiscing elit,  
2 sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.  
3 Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris  
4 nisi ut aliquip ex ea commodo consequat.
```

*Ciekawa ciekawostka – podany tekst to tzw. Lorem ipsum – czyli bardzo popularny fragment łacińskiego tekstu utworzony na podstawie traktatu Cyserona „O granicach dobra i zła”. Tekst jest używany przez projektantów wszelakich design’ów do przedstawienia rozłożenia tekstu na stronie, czasem też jako przykład dla kroju czcionki. Jest on używany dlatego, że nie przykuwa uwagi oglądającego, wygląda dość naturalnie*

Aby odczytać dane z pliku należy utworzyć obiekt File z nazwą naszego pliku (pamiętając o odpowiedniej ścieżce – najlepiej jest tworzyć pliki w tym samym miejscu co program)



```
Main.java  
public static void main(String[] args) {  
  
    File file = new File( pathname: "test.txt");  
  
}
```

Następnie tworzymy sobie obiekt *Scanner* (tak jak do odczytywania danych z konsoli), tylko tym razem zamiast *System.in* użyjemy naszego obiektu file. No i na tym etapie to już działamy dalej tak jak na odczycie z konsoli – *nextLine()* czy co tam sobie zechcemy

```

Main.java
public static void main(String[] args) throws FileNotFoundException {

    File file = new File( pathname: "test.txt");

    Scanner input = new Scanner(file);

    while (input.hasNext()) { //hasNext sprawdza czy cos zostalo w pliku do odczytu
        //odczyt pliku jest liniowy, jak sie skonczy to sie nie zapetli
        String line = input.nextLine();
        System.out.println(line);
    }
}

```

Możesz zauważyć że `main` nam się lekko rozrósł – pojawiło się „**throws** **FileNotFoundException**”, co mi przypomniało o tym, że nie napisałem tu nic o obsłudze wyjątków. Więc napiszę o tym w kolejnym dziale a na razie zostawmy to jako „bo tak”

No i tak wygląda odczyt naszego pliku wypisany w konsoli

```

C:\Users\user> java Main.java
Lorem ipsum dolor sit amet, consectetur adipiscing elit,
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat.

Process finished with exit code 0

```

Zapis danych do pliku

Zapis pliku wygląda trochę inaczej. Tym razem zamiast tworzyć obiekt `File`, utworzymy obiekt `PrintWriter`:

```

Main.java
PrintWriter output = new PrintWriter( fileName: "notatki.txt");

```

Tym razem nie potrzebujemy `Scannera`, sam `PrintWriter` posiada odpowiednie metody, takie jak `println()`, `print()`, `append()` i inne pierdoły.

W podanym przykładzie mamy prosty notatnik, który będzie zapisywał do pliku wszystko co zapiszemy w konsoli, dopóki nie napiszemy w niej „*finite*”.

```

Main.java
public static void main(String[] args) throws FileNotFoundException {

    PrintWriter output = new PrintWriter( fileName: "notatki.txt");

    Scanner input = new Scanner(System.in); //odczyt tekstu z konsoli

    boolean finite = false; //czy zakonczylismy wprowadzanie

    while (!finite) { //dopoki wprowadzanie nie jest zakonczone

        String line = input.nextLine(); //odczytaj kolejną linię z konsoli

        if(line.equals("finite")) { //jesli linijka to "finite"
            finite = true; //konczymy wprowadzanie
        }
        else { //jesli nie
            output.println(line); //wypisz do pliku linię
        }
    }
    output.close(); //WAŻNE! trzeba zamknąć plik na końcu
}

```

*Pamiętaj o zamknięciu pliku na końcu. Jeśli o tym zapomnisz, może to powodować problemy z zapisaniem danych.*

No i zapomniałem też wspomnieć, że te wszystkie klasy obiektów trzeba importować (co prawda IntelliJ czy co tam używasz sam podpowiada i importuje co trzeba, no ale mówię żeby nie było że coś nie działa )

```

Main.java
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

```

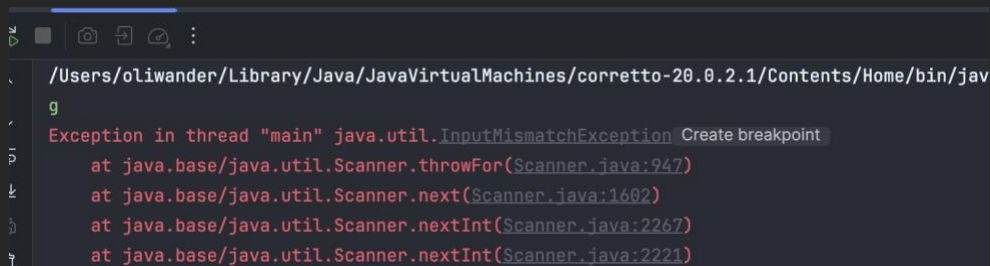


## Dział 8 – Wyjątki – czyli systemy idioto-odporne

Czasami w naszym programie dochodzi do sytuacji wyjątkowych – pojedynczych przypadków które wymagają, aby program zadziałał nieco inaczej niż zwykle. Przykładem takiego wyjątku może być przypadek, kiedy użytkownik poproszony o wprowadzenie liczby wprowadzi tekst, albo jeśli tak jak już zauważyłeś/aś w poprzednim dziale – plik nie zostanie znaleziony w lokacji którą podaliśmy. Albo jeśli piszemy kalkulator i dochodzi do dzielenia przez 0. W wielkim skrócie – wyjątki pozwalają obsłużyć wyjątkową głupotę użytkowników

Część z tych rzeczy można załatwić za pomocą warunków *if*, jednak o wiele ładniej jest dodać tzw. „obsługę wyjątku”.

Rozpatrzmy przykład z niepoprawnymi danymi wprowadzanymi przez konsolę. Jeśli użytkownik wpisze tekst zamiast liczby w `nextInt()`, program się wysypie w następujący sposób:



```
/Users/oliwander/Library/Java/JavaVirtualMachines/corretto-20.0.2.1/Contents/Home/bin/jav
g
Exception in thread "main" java.util.InputMismatchException Create breakpoint
at java.base/java.util.Scanner.throwFor(Scanner.java:947)
at java.base/java.util.Scanner.next(Scanner.java:1602)
at java.base/java.util.Scanner.nextInt(Scanner.java:2267)
at java.base/java.util.Scanner.nextInt(Scanner.java:2221)
```

*Dość ważna może wydawać się nazwa wyjątku – `InputMismatchException` – ale jeśli nie masz pewności albo informacji jaki to dokładnie wyjątek, to możemy po prostu zostać przy `Exception`*

### Try-catch

Aby obsłużyć taki wyjątek, użyjemy instrukcji *try-catch*. Możemy ją interpretować jako:

*spróbuj wykonać to ..., a jeśli się wysypie, wychwycić wyjątek i go obsłużyć w taki sposób...*

Tak to wygląda zaimplementowane w Javie:



```
Main.java
try {
    a = input.nextInt();
}
catch (InputMismatchException e) {
    System.out.println("Niepoprawne dane wejściowe");
    return;
}
```

W tym przypadku znamy typ wyjątku, który wystąpił, więc możemy go tu wypisać. *Try-catch* może mieć kilka klauzuli *catch* do obsługi różnych typów wyjątków w różny sposób. Ponadto możemy też dodać jeden główny z typem *Exception*:

```
Main.java
try {
    a = input.nextInt();
}
catch (InputMismatchException e) {
    System.out.println("Niepoprawne dane wejsciowe");
    return;
}
catch (Exception e) {
    System.out.println("Wystapil nieznany blad");
    return;
}
```

```
/Users/oliwander/Library/Java/JavaVirtua
dudu
Niepoprawne dane wejsciowe
:
:
Process finished with exit code 0
```

*Dzięki takiemu rozwiązaniu nasz program w miejscu wystąpienia wyjątku nie przerwie swojego wykonywania (znaczy w tym przypadku przerwie, bo używamy return, no ale może się wykonywać dalej)*

## Rzucanie wyjątków (*throw*)

Wyjątkami możemy też rzucać sami, a potem je przechwytywać w potrzebnych do tego miejscach. Na przykład mamy funkcję logarytm, liczącą logarytm przy podstawie  $a$  z  $b$  dla argumentów  $a$  i  $b$ . Jeśli  $a$  będzie równe 1 lub mniejsze od 0, to będzie to wyjątek, który musimy obsłużyć. Wyjątek który może wyrzucić nasza funkcja zapisujemy po nawiasie z argumentami

```
public static double logarytm(double a, double b) throws IllegalArgumentException {  
  
    if (a == 1 || a < 0) { //jesli ktorus z tych warunkow nie jest spelniany:  
        throw new IllegalArgumentException(); //rzuc nowy wujatek  
    }  
    else {  
        return Math.log(b) / Math.log(a);  
        //biblioteka math ma tylko logarytm naturalny, lecz mozemy skorzystac ze  
        //wzoru za zamiane podstaw logarytmu  
    }  
}  
  
public static void main(String[] args) throws FileNotFoundException {  
  
    double a = 1, b = 5;  
  
    try {  
        double wynik = logarytm(a, b);  
    }  
    catch (IllegalArgumentException e) {  
        System.out.println("Niepoprawne a");  
    }  
}
```

*Jak coś to tak, pamiętam że warunkiem logarytmu jest jeszcze to że  $b$  nie może być ujemne, no ale to sobie rozpisz sam/a.*

Generalnie to o tych wyjątkach opowiedziałem dość skrótowo – nie zdziwiłbym się jeśli byś nie wszystko do końca zrozumiał/a, więc w razie potrzeby odsyłam do literatury pod [\[tym\]](#) linkiem

## Dział 9 – Operacje binarne i na liczbach binarnych

//TODO

Jak mi się będzie chciało z tym męczyć to zrobię – klasy chyba teraz ważniejsze

# Dział 10 – Klasy w Javie

## Peak programowania obiektowego

Czas na wisienkę na torcie programowania obiektowego – klasy i obiekty

Dotychczas poznane struktury danych, takie jak zmienne liczbowe, tekstowe i inne, są dosyć proste. Nawet biorąc pod uwagę tablice, jest sporo ograniczeń, no a w wielu przypadkach brak konkretnego usystematyzowania danych.

Klasy to swego rodzaju schematy do takich bardziej złożonych danych. Opisują one z czego składają się obiekty (z jakich danych) i jakich funkcji (w przypadku klas mówimy „metod”) możemy używać wobec tych obiektów.

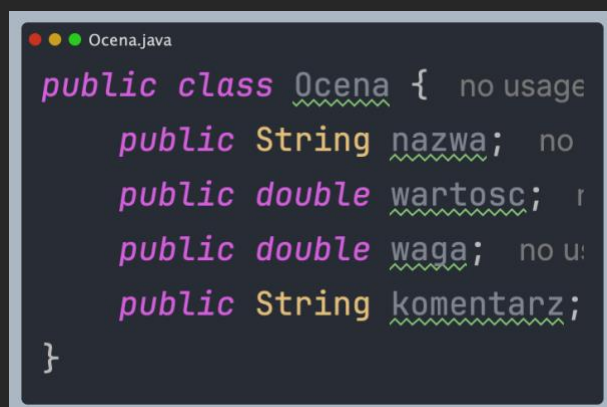
Wiem, że może wydawać się to lekko zagmatwane, więc przejdę do przykładu.

Założmy, że chcemy utworzyć klasę ocena, do przechowywania informacji, o ocenie którą dostał uczeń. Każda ocena ma jakieś swoje informacje: nazwa, wartość oceny, jej waga, no i może jeszcze jakiś komentarz. No i na podstawie tych informacji możemy sobie zrobić klasę, która będzie służyła do tworzenia obiektów ocena, przechowujących określone dane.

Klasy zwykle tworzymy w osobnym pliku, nie wiem z czego korzystasz, ale w intelij prawym klikasz na folder src (ten z dołu nie z góry) i wybierasz *add > java class*

Powstaje ci nowy plik, więc możemy sobie przejść do napisania naszej klasy.

Najpierw rozpiszemy sobie pola w naszej klasie:



```
public class Ocena {  
    public String nazwa;  
    public double wartosc;  
    public double waga;  
    public String komentarz;  
}
```

Wygląda dość prosto, zaraz poznasz też tajemnice tego całego *public*. Teraz wróćmy do głównego pliku i spójrzmy, jak powstaje obiekt naszej klasy:

```
Main.java
public static void main(String[] args) throws FileNotFoundException {
    Ocena kartkowka = new Ocena();
    kartkowka.nazwa = "ułamki zwykłe";
    kartkowka.wartosc = 2;
    kartkowka.komentarz = "";
    kartkowka.waga = 1;
}
```

Aby modyfikować pola naszego obiektu, odnosimy się do danego pola po kropce

Dobra, to o co z tym *public* chodzi?

## Public, protected, private – modyfikatory dostępu

Wszystkie właściwości (pola na dane, metody) naszej klasy mają określony poziom dostępu przez rzeczy z zewnątrz. Poziom dostępu określamy za pomocą modyfikatorów (albo specyfikatorów) zapisywanych przed typem danych.

*Public* oznacza, że do danej metody albo pola możemy dostać się z każdego miejsca w programie – dane te są publiczne

*Private* oznacza, że do danej metody lub pola możemy dostać się tylko i wyłącznie z wnętrza tej klasy

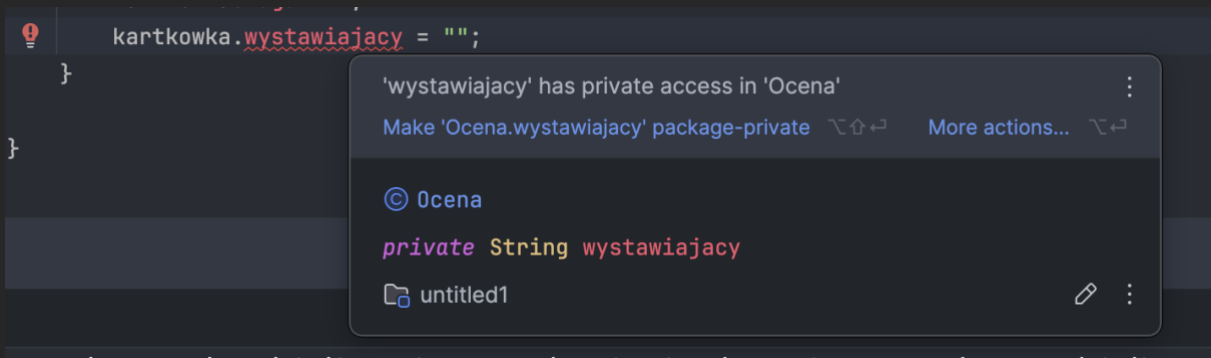
*Protected* oznacza, że do danej metody lub pola możemy dostać się z wnętrza tej klasy oraz klas dziedzicznych (o dziedziczeniu później)

Domyślne jest *package-private*, czyli prywatna w zakresie całej paczki (biblioteki)

Tak oto pole wystawiający możemy ustawić na *private* – RODO i te sprawy

```
Ocena.java
public String nazwa; 2 usages
public double wartosc; 2 usages
public double waga; 2 usages
public String komentarz; 2 u
private String wystawiajacy;
```

W takim przypadku nie w głównym pliku programu nie mamy dostępu do tego pola:



No dobra, ale jak teraz zmodyfikować tą zmienną?

Najprostszym konceptem na rozwiązanie tego projektu są tzw. **getter** i **setter**, czyli metody służące do manipulowania taką zmienną:

```
Ocena.java
private String wystawiajacy; 3 usages

public String getWystawiajacy() { no usages
    return wystawiajacy;
}

public void setWystawiajacy(String wystawiajacy) {
    this.wystawiajacy = wystawiajacy;
}
```

W oczy rzucić Ci się może kolejne niesamowite zaklęcie – **this**

Jeśli dobrze pamiętasz (a powinieneś, to było na poprzedniej stronie), aby modyfikować pole danego obiektu odnosimy się poprzez nazwę tego obiektu (u nas to była kartkowka).

Wewnątrz klasy nie wiemy jak nasz obiekt się będzie nazywał, więc do odniesienia się do pola obiektu używamy **this** (obiekt odnosi się do swojego własnego pola).

Ma to jeszcze jedną zaletę – nazwę zmiennej w argumencie możemy dać taką samą co pola w obiekcie (w obu przypadkach będzie to wystawiający)

## Konstruktor klasy

Konstruktor klasy, to specjalna metoda (funkcja), która jest wywoływana za każdym razem przy tworzeniu obiektu tej klasy. Używamy jej do inicjalizowania obiektu, wykonania jakiś wstępnych działań, nadania wartości zmiennym i inne takie.

W obecnej formie naszej klasy (bez konstruktora), tworzenie oceny odbywałoby się w taki sposób:

```
public static void main(String[] args) throws FileNotFoundException {
    Ocena kartkowka = new Ocena();
    kartkowka.nazwa = "ułamki zwykłe";
    kartkowka.wartosc = 2;
    kartkowka.komentarz = "";
    kartkowka.waga = 1;
    kartkowka.wystawiajacy = "";
    kartkowka.setWystawiajacy("Pan Lusterko");
}
```

Czasochłonne i głupie, takie nadawanie wartości zmiennym może odbywać się w konstruktorze, oto jak wygląda taki konstruktor:

```
public Ocena(String nazwa, double wartosc, double waga, String komentarz, String wystawiajacy) {
    this.nazwa = nazwa;
    this.wartosc = wartosc;
    this.waga = waga;
    this.komentarz = komentarz;
    this.wystawiajacy = wystawiajacy;
}
```

Konstruktor zawsze ma taką samą nazwę jak nazwa klasy, której obiekt konstruuje. Jako argumenty przekazujemy sobie początkowe wartości zmiennych, które wewnątrz są przypisywane do pól.

Jak wrócisz teraz do głównego pliku programu, zauważysz błąd przy tworzeniu obiektu. Konstruktor, który stworzyliśmy to jedyny dostępny konstruktor tej klasy. Fajnie byłoby stworzyć jeszcze drugi, taki bez żadnych argumentów, tworzący pusty obiekt

```
public Ocena(){ 1 usage
    this.nazwa = "";
    this.wartosc = 0;
    this.waga = 0;
    this.komentarz = "";
    this.wystawiajacy = "";
}
```



Teraz możemy wrócić do głównego pliku i znacznie go uprościć:

```
● ● ● Main.java
public static void main(String[] args) {
    Ocena kartkowka = new Ocena( nazwa: "ułamki zwykłe", wartosc: 2, waga: 1, komentarz: "", wystawiający: "" );
}
```

## Metody statyczne

Nadszedł ten wielkopomny moment aby dowiedzieć się o co chodzi z tym pieprzonym static i czemu jest w kompletnie losowych miejscach.

Otóż – nie takich wcale losowych



W naszych klasach możemy tworzyć metody statyczne dodając modyfikator *static*. Metody statyczne nie wymagają tworzenia obiektu (instancji) klasy do ich wywołania. Mogą funkcjonować kompletnie samodzielnie, możemy uruchamiać je pisząc po prostu nazwę klasy i metodę po kropce.

I jak się tak nad tym głębiej zastanowisz (dasz radę, wierzę w Ciebie), to fakt że funkcja main jest statyczna ma sens – kompilator javy nie musi tworzyć obiektu klasy main aby uruchomić nasz program

Pozostaje jeszcze kwestia statycznych zmiennych np. poza funkcją main – tak się niesamowicie składa, że statyczne metody mogą korzystać tylko i wyłącznie ze statycznych zmiennych, które również tak jak metody powstają i mogą zostać użyte bez tworzenia obiektu klasy.

## Dział 11 – Dziedziczenie klas (TODO)

//TODO

## Dział 12 – Klasy abstrakcyjne (TODO)

//TODO

TODO: Rzutowanie typów

Wypociny by Oliwier Popielarczyk, 2025

Mam nadzieję, że komuś jakkolwiek pomogłem

*W coś wierzyć trzeba...*

*...bo żyć się odechciewa*

*Vesemir*