

'Zapiski uczestnika Bootcampu Zajavka.pl w 12 tygodni' by Bartek Borowczyk aka Samuraj Programowania. **Dopiski na czerwono** od Karola Rogowskiego.
Więcej na <https://zajavka.pl>

Dzień 1 Bootcampu

Miłe wprowadzenie :)

Ten bootcamp to był dobry wybór ;)

Dzień 2 Bootcampu

Java jest językiem

- **przenośnym** - a więc działającym niezależnie od sprzętu (procesora) i systemu, w oparciu o maszynę wirtualną - czym jest maszyna wirtualna (**Java Virtual Machine**) piszę dalszej części notatek :). Java w wyniku kompilacji kodu źródłowego napisanego w Javie tworzy kod pośredni (inaczej zwany **kodem bajtowym**), który może być wykonany na dowolnym procesorze/systemie, na którym jest zainstalowana **wirtualna maszyna Javy**. Nie trzeba tworzyć osobnego kodu dla różnych systemów/procesorów. Pamiętajmy na tym etapie, że przenośność oznacza niezależność od systemu czy procesora.
- **szybkim** - kod pośredni (bajtowy) generowany przez kompilator Javy jest zoptymalizowany, podobnie zresztą jak maszyna wirtualna. Dlatego kod może wykonywać się szybko, więc mimo że mamy tutaj potrzebę interpretacji przez maszynę wirtualną kodu bajtowego, to działa to szybko jak błyskawica.
- **bezpiecznym** - za sprawą m.in. ograniczania programu do środowiska uruchomieniowego (inaczej zwanego wykonawczym). Program nie może zrobić nic więcej niż pozwala mu właśnie maszyna wirtualna. Tak więc nie może np. uzyskać nieograniczonego dostępu do systemu. **Inaczej taką sytuację można nazwać sandboxem (dla dociekliwych [Link](#)).**

A ponadto możemy o Javie powiedzieć, że jest językiem:

- **ogólnego przeznaczenia** - to znaczy, że sprawdza się w licznych, zróżnicowanych zastosowaniach, nie tylko w serwisach webowych.
- **językiem wysokiego poziomu** (w przeciwieństwie do języków niskopoziomowych) - nie chodzi tu o poziom rozwoju języka (choć rzeczywiście tu poziom Javy jest wysoki ;), tylko poziom abstrakcji względem kodu maszynowego (czyli 0 i 1, które rozumie procesor). Wszystkie współczesne języki (Java, PHP, Python, C++, Ruby, JavaScript, a nawet C) są językami wysokiego poziomu.
- **językiem zorientowanym obiektowo** - program w Javie jest tworzony i reprezentowany za pomocą obiektów tworzonych na podstawie klas. Obiekty składają się z danych i metod do obsługi tych danych. Tym zagadnieniem w bootcampie poświęcimy sporo czasu w 4. i 5. tygodniu nauki.

Co warto wiedzieć z historii Javy

- Opracowana przez Sun Microsystem w 1991 r. Nazwa Java funkcjonuje jednak od 1995 r.
- Bezpośredni przodkowie Javy (z których czerpie ona wiele rozwiązań) to języki C i C++. Składnia oraz model programowania obiektowego są oparte właśnie o C (składnia) i C++ (obiektywność). Natomiast językiem, który czerpał wiele z Javy był (i jest) C# (czyli Csharp).
- W 2010 r. firma Oracle kupiła Sun Microsystem i została właścicielem Javy.
- Java obecnie jest dostarczana przez Oracle w dwóch wersjach: wersji płatnej (SE - Standard Edition) i bezpłatnej (OpenJDK). Są to najbardziej popularne, ale nie jedyne dostępne dystrybucje Javy (specyfikacja Javy może być wdrażana także przez innych dostawców (**vendorów - nawiązuję do nomenklatury w odcinku**)).
- Java jest językiem mocno i systematycznie rozwijającym. Dwa razy w roku pojawia się nowa wersja. Obecnie najpopularniejszą wersją jest Java 11 (z oznaczeniem LTS - czyli długotrwałe wsparcie - **Long Term Support**). Najnowszą wersją Javy jest natomiast wydana w marcu 2021 wersja z numer 16.

Java jest językiem kompilowanym, ale także interpretowanym.

Kod napisany w Javie (a więc kod źródłowy) jest kompilowany przez kompilator (o tym w dalszej części) do formatu zrozumiałego dla maszyny wirtualnej Javy (zapamiętać skrót:

JVM). Taki kod nazywamy **kodem pośrednim** lub **kodem bajtowym** (w odróżnieniu od kodu binarnego czy kodu maszynowego, który jest wersją już zrozumiałą dla procesora).

Ten wygenerowany w wyniku kompilacji kod jest zoptymalizowany do użycia przez JVM.

Kod bajtowy (pośredni) jest następnie wykonywany przez interpreter. Interpreter dostarcza właśnie JVM (tak, tak, ta maszyna wirtualna właśnie pełni rolę interpretera). Interpretowanie i wykonanie kodu ma miejsce w wyizolowanym środowisku (a więc jest 'całkiem' bezpiecznie - o czym już wspominałem wskazując cechy Javy). Kod bajtowy (przy okazji, po angielsku to bytecode) zapewnia także przenośność, wystarczy bowiem zainstalowana maszyna wirtualna na danej platformie i można uruchomić program.

Wróćmy jeszcze do szybkości - kod w Javie jest interpretowany w chwili wykonania (kod bajtowy nie jest zrozumiały bez JVM dla procesora), co oznacza, że Java jest potencjalnie językiem trochę wolniejszym niż języki, których kod jest od razu kompilowany do kodu wykonywalnego (maszynowego). Jednak w tym przypadku sposób optymalizacji zarówno kodu bajtowego jak i samej JVM zapewnia, że Java jest uważana za język szybki. *(Kiedyś była uznawana za wolny, natomiast wszystko idzie do przodu)*

Uwaga: Temat kompilacji i optymalizacji jest ciekawy ale zaawansowany. Pojawia się tutaj wiele pojęć takich jak kompilacja just-in-time czy ahead-of-time, ale na tym etapie (i często późniejszym) nie jest nam ta wiedza do niczego potrzebna ;) (Chyba że będziesz potrzebował(a) zejść naprawdę głęboko, ale w pracy na co dzień raczej się nad tymi pojęciami nie będziesz zastanawiać)

Kompilator - kompiluje kod Javy na kod pośredni (kod bajtowy). Kod bajtowy nie jest zrozumiały dla człowieka. Nie jest też zrozumiały dla procesora, dlatego określamy go kodem pośrednim. Dla kogo jest więc zrozumiały? Dokładnie! Dla wirtualnej maszyny Javy :) Kompilator jest dostarczony razem z JDK. No więc przejdźmy do JDK.

JDK - Java Development Kit - narzędzia developerskie Javy / środowisko programowania w Javie

Jawę najczęściej utożsamia się z językiem programowania. Potocznie, oczywiście, Java to język programowania, ale, patrząc całościowo, Java to całe JDK. Dobrze to widać, gdy mówimy o wersji Javy. Numer (wersja) Javy odnosi się właśnie do JDK.

JDK należy traktować jako pakiet narzędzi dla programistów Javy, który udostępnia przede wszystkim dwa programy:

- **kompilator** - nazywa się on **javac** - zamienia kod napisany w Javie na kod bajtowy
- **interpreter** - o nazwie **java** - który interpretuje kod bajtowy, a więc w praktyce uruchamia aplikacje

Kod napisany w Javie umieszczamy w plikach z rozszerzeniem **.java** (pliki źródłowe). Po kompilacji (pliki wynikowe) mają już rozszerzenie **.class**.

krok 1 - uruchomienie w konsoli kompilatora

```
C:\Users\48512>javac app.java
```

krok 2 - uruchomienie w konsoli interpretera (tutaj już bez rozszerzenia .class wpisujemy)

```
C:\Users\48512>java app
```

Zanim przejdziemy dalej, zastanówmy się, co obejmuje rdzeń zasad (specyfikacji) Javy. Są to przede wszystkim:

- maszyna wirtualna
- zdefiniowany język programowania (czyli m.in. składnia)
- API Javy czyli zbiór kilkuset gotowych, dostarczonych przez Jawę klas do których mamy dostęp, kiedy programujemy w Javie.

I właśnie te rzeczy są wdrażane potem w konkretnym JDK. JDK z danym numerem jest zgodne ze specyfikacją dla danego numeru.

Występują dwie wersje Javy dostarczane przez Oracle:

- Java z literkami SE oznaczającymi Standard Edition np. Java SE 11 (11 oznacza, że oparta o specyfikację JDK 11)
- Java OpenJDK, która również ma swoje numeryczne oznaczenie, a od wersji SE różni się tym, że jest open source i można jej używać za darmo.

JRE - Java Runtime Environment - środowisko uruchomieniowe Javy

Odpowiedzialne za uruchomienie programu. Zawiera zarówno maszynę wirtualną Javy, jak i zestaw użytecznych klas (klas podstawowych).

JVM - Java Virtual Machine - maszyna wirtualna Javy

Jest elementem JRE. Interpretuje i wykonuje kod bajtowy.

Jak wygląda cały proces od kodu źródłowego do wykonania programu:

1. kod źródłowy w Javie (rozszerzenie plików .java)
2. proces kompilacji kodu źródłowego [kompilator]
3. wygenerowanie kodu bajtowego (rozszerzenie plików .class)
4. proces interpretacji kodu bajtowego [interpreter - maszyna wirtualna]
5. wyjście - działanie programu

Dzień 3 Bootcampu

Skąd pobrać Javę?

<https://openjdk.java.net/>

Jak sprawdzić, czy wszystko poszło dobrze i mamy już Javę u siebie?

Uruchomić konsolę/terminal (np. cmd albo powershell, jeśli chodzi o Windowsa)

Kompilator:

```
C:\Users\48512>javac --version
javac 16
```

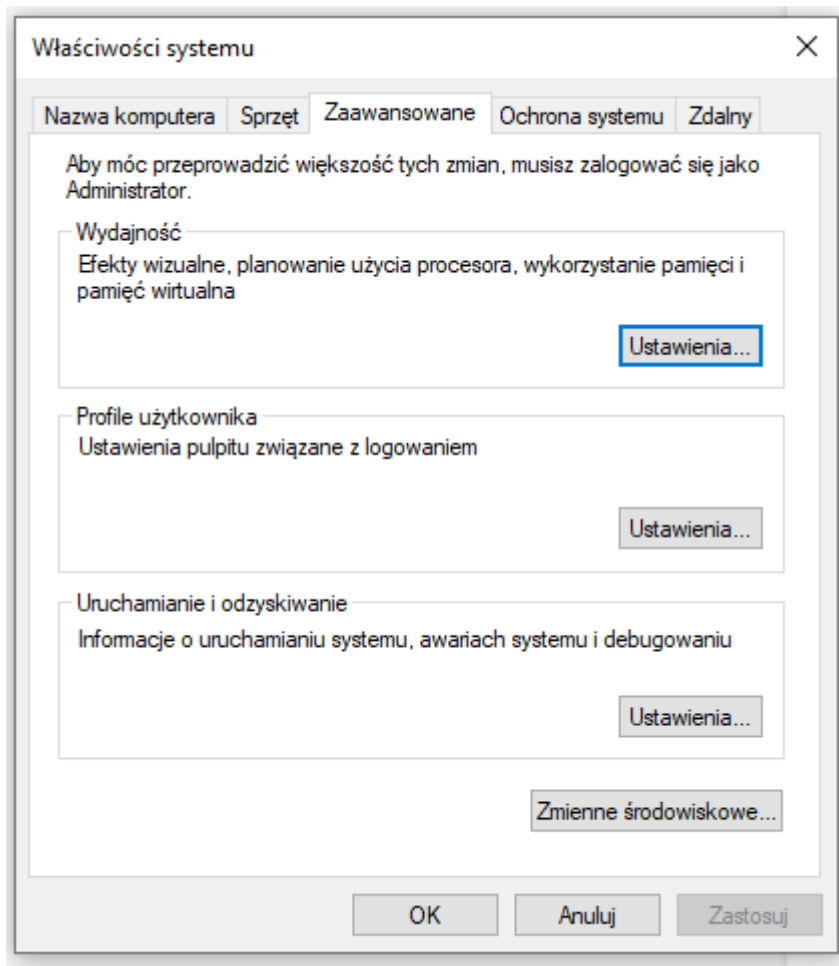
Interpreter:

```
C:\Users\48512>java --version
java 16 2021-03-16
Java(TM) SE Runtime Environment (build 16+36-2231)
Java HotSpot(TM) 64-Bit Server VM (build 16+36-2231, mixed mode, sharing)
```

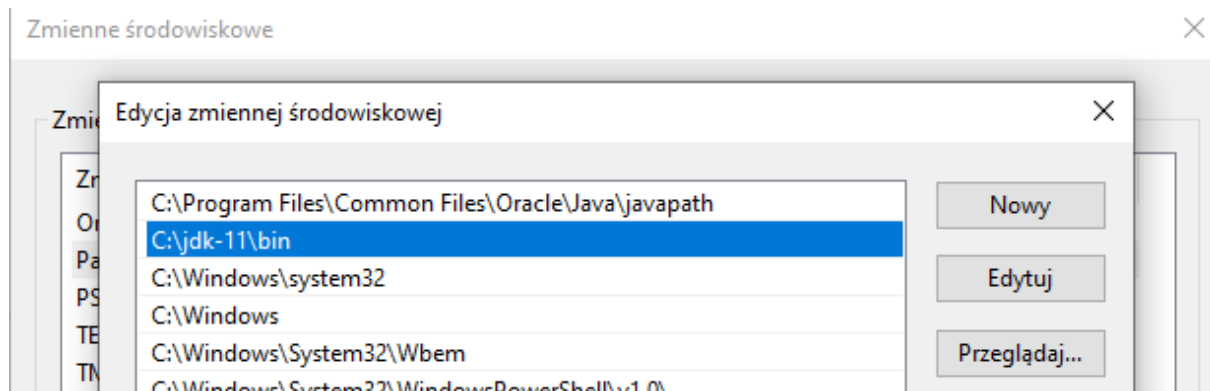
Zmienne środowiskowe

Jeśli nie chodzą polecenia javac i java w konsoli, to należy dodać ścieżkę do zmiennych środowiskowych wg przykładu poniżej.

[kliknij w "Zmienne środowiskowe..."]



Umieszczamy ścieżkę do miejsca, w którym zainstalowaliśmy (rozpakowaliśmy) Javę. W przykładzie poniżej ta ścieżka już jest (a Java znajduje się akurat w folderze C:\jdk-11). Jeśli brakuje ścieżki u Ciebie, użyj przycisku 'Nowy' i wprowadź ścieżkę do folderu bin.



Dzień 4 Bootcampu

Omawialiśmy **IntelliJ IDEA** - na teraz bez jakiś dodatkowych notatek. Ale trzeba zrobić listę podstawowych skrótów klawiszowych do funkcji IntelliJ i do generowania kodu. Karol pewnie podrzuci jakąś :)

Proszę [Link](#), tylko nie uczcie się na pamięć, to samo wejdzie do głowy

Dzień 5 Bootcampu

W tym dniu Karol mówił nam o tym, że:

- program w Javie zaczyna się od **wywołania metody main** (metoda main jest punktem wejściowym aplikacji)
- czym są i przede wszystkim, jak używać **komentarzy (mówił, by nie nadużywać)**
- poznaliśmy wykaz **słów zastrzeżonych** przez Javę, czyli słów, które coś w Javie robią. Karol wspominał, by nie uczyć się ich na pamięć, bo one z czasem i tak nam wejdą do głowy, jak będziemy ich wielokrotnie i codziennie używać.
- **'przypominajki'** - todo and fixme - Karol ich nie lubi. Ale jest i niektórzy używają. **Nie lubię ich, bo potem ludzie o nich zapominają i często 'wala się' dużo zapomnianych w kodzie**

```
// plik Example.java
```

```
class Example {  
    public static void main(String[] args) {  
        // kod metody  
    }  
}
```

Na tym etapie jeszcze nie przejmuj się składnią: czym jest class, public, static, void, String[] i args, dowiesz się już wkrótce (nawet trochę w tych notatkach, ale przede wszystkim w kolejnych dniach).

Zapewniam, że nie ma tu wielkiej filozofii i krok po kroku to zrozumiesz.

Na co na tym etapie zwrócić uwagę i co zapamiętać:

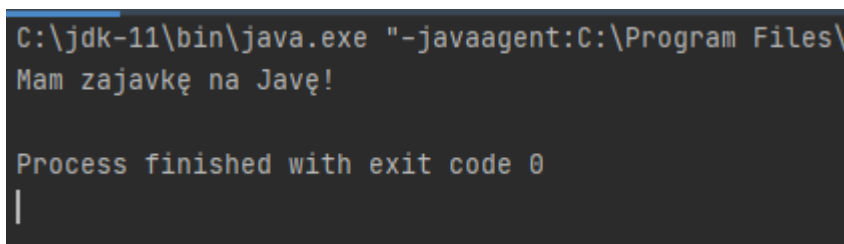
- **Nazwa pliku musi być taka sama jak nazwa klasy w pliku**, np. Example.java i nazwa klasy w pliku (po słowie kluczowym class), również Example. Wielkość liter też ma znaczenie! Czyli jak mamy w pliku *class Main*, to plik nazwiemy Main.java
- **Jedna metoda main jest wymagana!** Jest wymagana, żeby program w jakikolwiek sposób uruchomić. Możesz napisać też program bez metody main, ale jak go wtedy uruchomić, dowiesz się na dalszych etapach. Dlatego na dzień dzisiejszy przyjmijmy, że programu bez metody main nie uruchomisz ;)
- Od metody main rozpoczyna się wykonanie programu. Jest to tzw. **punkt wejściowy aplikacji**.

Popatrzmy jeszcze raz na nasz kod źródłowy, tym razem jest trochę bardziej rozbudowany. Przede wszystkim pojawia się zawartość metody.

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Mam zajawkę na Javę!");  
    }  
}
```


Zawartość metody zostanie wykonana w chwili uruchomienia programu, ponieważ metoda `main` jest punktem startowym programu. W tym konkretnym wypadku wyświetlony zostanie (wydrukowany) tekst w naszej konsoli. Oczywiście, najpierw kod zostanie skompilowany do kodu bajtowego (w IntelliJ pojawi się katalog `/out` a w nim plik `Main.class`), który potem będzie wykonany przez maszynę wirtualną javy.

Efekt w konsoli:



```
C:\jdk-11\bin\java.exe "-javaagent:C:\Program Files\
Mam zajawkę na Javę!

Process finished with exit code 0
|
```

ps. code 0 na wyjściu znaczy, że wszystko poszło dobrze.

Przypomnę tylko, że ten sam efekt uzyskamy, gdy wpisujemy w wierszu poleceń (oczywiście, we właściwej ścieżce prowadzącej do danego pliku):

```
> javac Main.java
> java Main
```

Tak jak to widać na dwóch screenach poniżej.

Na screenie pierwszym kod w pliku `Main.java` + użycie kompilatora `javac` (w terminalu), który stworzył plik `Main.class` (kod bajtowy) - do którego zazaczyłem niebieską linię. Na screenie drugim wywołujemy już ten plik (klasę) za pomocą polecenia **java**. Przy okazji zobaczcie na drugim screenie, jak edytor widzi kod bajtowy - otworzyłem bowiem plik `Main.class`. Efektem wywołania `Main.class` jest wywołanie metody `main` a w niej wywołanie metody **`println`**, która drukuje przekazany tekst w konsoli/terminalu.

```
1 public class Main {  
2  
3     Run | Debug  
4     public static void main(String[] args) {  
5         System.out.println("Mam zajawkę na Javę!");  
6     }  
}
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
PS C:\Users\48512\Desktop\java> javac Main.java  
PS C:\Users\48512\Desktop\java>
```

```
1  
2  
3  
4  
5  
6
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
PS C:\Users\48512\Desktop\java> javac Main.java  
PS C:\Users\48512\Desktop\java> java Main  
Mam zajawkę na Javę!  
PS C:\Users\48512\Desktop\java>
```

Powróćmy ponownie do tego kodu:

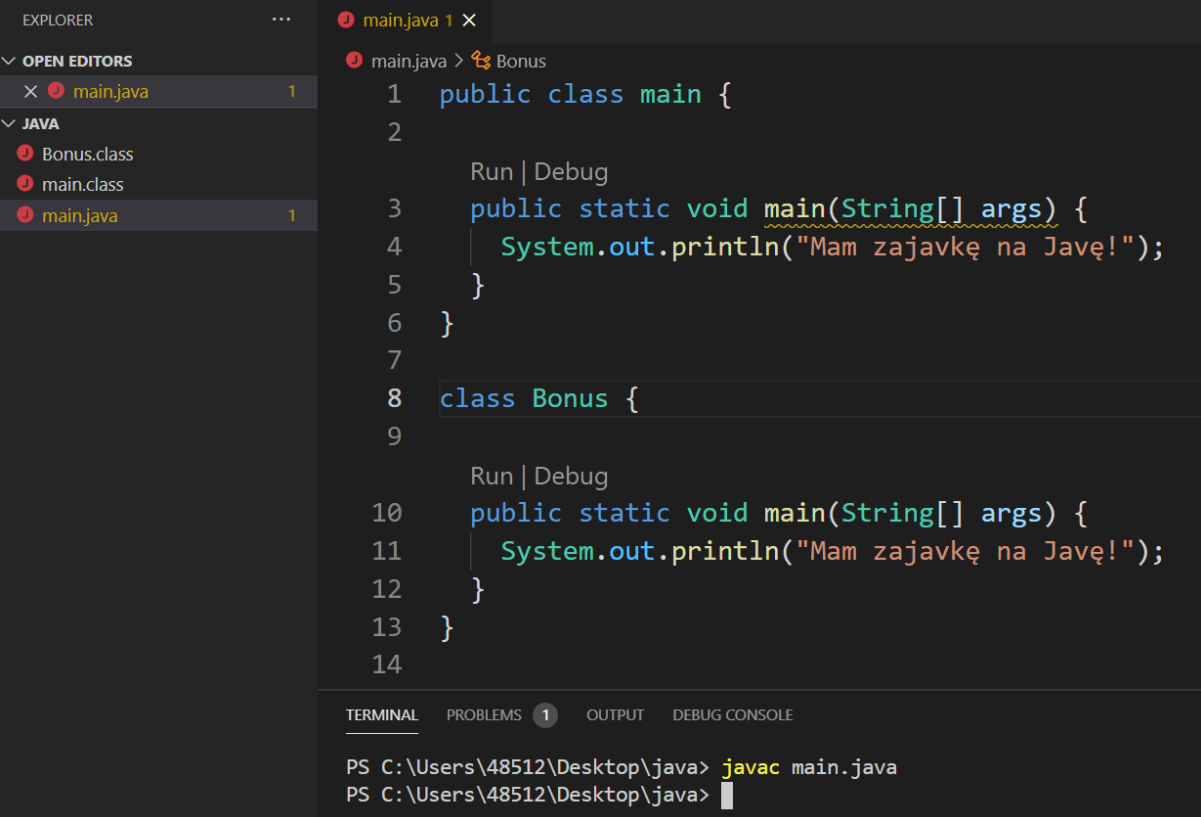
```
public class Main {  
  
    public static void main(String[] args) {  
        System.out.println("Mam zajawkę na Javę!");  
    }  
}
```

Na tym etapie zwróć uwagę, że mamy tutaj coś takiego jak klasa (słowo kluczowe **class**). W Javie cały kod musi znajdować się w klasach (w dużych projektach mogą być ich setki a nawet tysiące).

Zwróćmy uwagę że plik wynikowy (czyli ten tworzony w wyniku kompilacji z kodem bajtowym i rozszerzeniem .class) jest też tworzony z nazwą klasy (i jednocześnie pliku). Tak więc w pliku wynikowym kompilacji **każda klasa uzyskuje swój plik** z rozszerzeniem .class.

Widać to dobrze w przykładzie poniżej, gdzie mamy dwie niemal identyczne klasy, ale, rzecz jasna, z różnymi nazwami (nie można mieć dwóch klas o tych samych nazwach oczywiście).

Kompilując plik main.java (celowo plik i klasa małą literą, byśmy zobaczyli wynik), który zawiera dwie klasy (najczęściej najlepszym rozwiązaniem jest jeden plik-jedna klasa, ale teraz to tylko przykład), uzyskaliśmy dwa pliki wyjściowe. W tym przykładzie main.class (mała litera na początku, bo taka była nazwa klasy) i Bonus.class (wielka litera na początku, bo taka była nazwa klasy).



The screenshot shows an IDE with a file explorer on the left and a code editor on the right. The file explorer shows a project with two Java files: `Bonus.class` and `main.class`, and a source file `main.java`. The code editor shows the content of `main.java`, which contains two classes: `main` and `Bonus`. Both classes have a `main` method that prints "Mam zajawkę na Javę!". The terminal at the bottom shows the command `javac main.java` being executed, resulting in two class files.

```
1 public class main {
2
3     Run | Debug
4     public static void main(String[] args) {
5         System.out.println("Mam zajawkę na Javę!");
6     }
7 }
8 class Bonus {
9
10    Run | Debug
11    public static void main(String[] args) {
12        System.out.println("Mam zajawkę na Javę!");
13    }
14 }
```

TERMINAL PROBLEMS 1 OUTPUT DEBUG CONSOLE

```
PS C:\Users\48512\Desktop\java> javac main.java
PS C:\Users\48512\Desktop\java>
```

!Pamiętaj: nazwy klas pisz wielkimi literami, podobnie jak nazwy plików. Staraj się, by jeden plik zawierał jedną klasę.

Słowa kluczowe, które poznaliśmy w dzisiejszych zajęciach:

public -- określa dostępność danego elementu dla innych elementów programu - w naszych przykładach widzimy już public przy klasie i przy metodzie. Na teraz taka wiedza jest wystarczająca, ale jeśli chcesz zapamiętać coś więcej, to to, że są to tzw. **modyfikatory dostępu** (jeden z kilku, mamy też np. private, protected - na spokojnie je poznasz).

class -- słowo kluczowe, które służy do utworzenia klasy. Klasa jest podstawowym elementem każdego popularnego języka programowania zorientowanego obiektowo (będziemy się o programowaniu zorientowanym obiektowo uczyć w 4. i 5. tygodniu). Java wręcz wymusza na nas używanie klas i to jest świetny sposób na uczenie się programowania zorientowanego obiektowo (skrót z angielskiego **OOP - Object Oriented Programming**), które jest fundamentem współczesnego programowania.

Zwróć uwagę, że po słowie class mamy zawsze nazwę (**identyfikator klasy**). Dzięki temu, możemy odwołać się do tworzonej klasy. np. class Dog, class Main, class User czy class ElectricCar itd.

!!! Uspokajam, na tym etapie nawet nie mówimy, co robi klasa i czym jest. Przejdziemy do tego i szczegółowo wytłumaczymy, tak byś zrozumiał(a). Obiecuję!

Identyfikator klasy (jej nazwę) zgodnie z przyjętą konwencją piszemy wielką literą, dlatego plik też będzie nazywany z użyciem wielkiej litery na początku, bo nazwa pliku musi być identyczna jak nazwa klasy. Wiem, że się powtarzam, już nie będę :)

{ } -- Kolejny element składni to nawiasy klamrowe, w kręgach programistów pamiętających lata 90-te zwane także wąsami. Nawiasy klamrowe oznaczają ciało (zawartość) zarówno klasy jak i metod, czyli ich zawartość. Są wymagane w Javie.

class Identyfikator { } - podstawowa składnia klas

Przejdźmy teraz do wiersza z definicją metody main. Wygląda on następująco:

```
public static void main(String[] args) { }
```

Metoda to mini program, którego instrukcje będą wywołane w chwili wywołania metody.
Metoda main jest wywoływana domyślnie.

public - to modyfikator dostępu. Na teraz warto wiedzieć, że takie określenie przy metodzie oznacza, że jest ona... publiczna :) Czyli dostępna także spoza klasy (co to dokładnie znaczy, jeszcze się dowiemy). Metoda main musi być metodą publiczną. Przekonamy się w przyszłości (Karol mi mówił ;)), że znacznie częściej występuje przy metodach modyfikator **private**, który oznacza, że dana metoda nie może być wywołana poza klasą.

// Jeszcze raz uspokajam, że to bardzo powierzchowne tłumaczenie, podczas bootcampu dowiesz się szczegółowo i dobrze, co to oznacza i jak korzystać z danych metod.

static - to słowo kluczowe, które umożliwia wywołanie metody bez tworzenia obiektu. O Panie, brzmi strasznie. Ale takie się nie okaże w praktyce. Klasy będą służyły nam do tworzenia obiektów. Trochę tak jak plan domu (klasa) pozwala zbudować wiele domów (obiektów). Sama klasa nie jest obiektem! Używając **static** mamy możliwość coś zrobić bez zbudowania domu, np. **static** **ZmierzPokoje** może być metodą, która pozwala zmierzyć wymiary pokoju na podstawie projektu (klasy), bo obiektu jeszcze nie ma. Gdyby **ZmierzPokoje** była metodą nieposiadającą słowa kluczowego **static**, to można by użyć tej funkcji tylko poprzez odwołanie się do istniejącego (na podstawie klasy) obiektu. Na teraz nie ma co bardziej kombinować ;)

void - jest słowem kluczowym, które oznacza, że dana metoda nic nie zwraca. Bo, o czym się przekonasz wkrótce, metody bardzo często zwracają jakąś wartość. Wyobraź sobie dwie metody w pseudokodzie:

```
metoda nicNieZwraca dodaj(a, b) {  
    wydrukujNaEkranie(a + b)  
}
```

**nicNieZwraca zastępuje tu void właśnie*

```
metoda zwracaLiczbe odejmij(a,b) {  
    wydrukujNaEkranie(a - b)  
    zwróć a - b  
}
```

*zamiast void wskazujemy, co zostanie zwrócone, np. liczba (int) - o typach dowiemy się wkrótce.

Pierwsza z metod nic nie zwraca (więc używamy void) - wykonuje po prostu jakieś zadanie. Druga wykonuje jakieś zadanie, ale oprócz tego coś zwraca (nie użyjemy więc void a np. int).

Coś, co zostanie zwrócone z metody, może zostać przekazane do innej metody czy zapisane i użyte w innym miejscu programu.

(String[] args) - wreszcie w nawiasach metody wskazujemy, jakie parametry przyjmie metoda w chwili wywołania. Na metodach z pseudokodem, gdzie metody pozwoliły nam dodać i odejmować, nazwaliśmy je (parametry) a i b. W przypadku metody main musimy użyć String[].

Oczywiście może się zdarzyć, że metoda nie przyjmie żadnych parametrów.

Jeśli szukasz innego przykładu parametrów, to spójrz na metodę, której użyliśmy do wydrukowania w konsoli:

System.out.println("Mam zajawkę na Javę!");

W tym wypadku metoda println otrzymuje parametr, który jest tekstem, a implementacja metody (jest to metoda wbudowana, której tylko używamy, ale nie potrzebujemy wiedzieć, jak to jest zaimplementowane), sprawia, że przekazana w parametrze wartość pojawia się w konsoli.

String[] args - które jest użyte w nawiasach, oznacza tablicę obiektów typu String (wartości tekstowe), na teraz nie jest to istotne, dowiesz się o stringach i innych obiektach w kolejnych dniach. Natomiast args jest identyfikatorem tej tablicy, przy czym nazwa może być inna (nie musi to być args, ale to konwencja, w praktyce nic nie zmieniamy).

Po co więc ta tablica stringów w metodzie main? Jeśli byśmy chcieli przekazać do programu jakieś wartości w chwili wywołania, to moglibyśmy to zrobić, póki co nie jest ona nam do niczego potrzebna, ale nie możemy jej usuwać.

tłumacząc na ludzki ten zapis: `public static void main(String args[]) { }`

otrzymamy:

```
dostępnaPozaKlasą niepotrzebującaDoWykonaniaObiektu nicNieZwracająca
metoda o identyfikatorze main (Przekazująca tablicę o nazwie args
zawierająca potencjalnie wartości tekstowe) {
    // tutaj kod metody;
}
```

Pamiętaj tylko, że od tej metody zaczyna się wykonanie naszego programu. I tyle ;)

Komentarze w Javie

Po pierwsze pamiętajmy, że komentarze są ignorowane przez kompilator. Komentarze są dla nas i innych programistów.

```
/*
komentarz wielowierszowy
*/
```

```
// komentarz jednowierszowy
```

Istnieje jeszcze jeden typ komentarzy w Javie, tzw. komentarze dokumentujące (javadoc), ale na ten moment nic z nimi nie robimy.

Dzień 6 - 7 Bootcampu

weekend

Dzień 8 Bootcampu

Programowanie obiektowe - tym zajmiemy się w 4. i 5. tygodniu. Jednak w dzisiejszym dniu trochę o ten temat zahaczyliśmy.

Programowanie zorientowane obiektowo zakłada, że program składa się z obiektów składających się z danych i metod. Obiekty pozostają ze sobą w różnych relacjach oraz są hermetyczne, a więc niedostępne dla innych obiektów z wyjątkiem interfejsu, który udostępniają (czyli w praktyce jakichś metod publicznych).

Karol używa przykładu kota. Równie dobrym jest samochód, który staje się metaforą naszego programu.

- silnik, kierownica, fotele, żarówki - to wszystko obiekty
- kierownica jako obiekt ma właściwości: średnicę, materiał wykonania, ale ma też metody, jak obracanie. O danych zdefiniowanych w klasie mówimy, że są atrybutami (właściwościami) a o zachowaniach, że są metodami.
- W kierownicę mogą być wbudowane inne elementy, np. przycisk tempomatu, zestawu głośnomówiącego czy radia.
- Na przykładzie kierownicy widać relacje z innymi elementami. Np. przycisk radia jest wbudowany w kierownicę, kierownica zaś wchodzi w relacje z innymi obiektami, takim elementem jest człowiek (za pomocą metody obracanie może sterować kierownicą, a za jej pomocą kierunkiem jazdy), ale też z elementem zawieszenia, tak by przekazać informacje o kierunku jazdy do kół.
- kierownica ma więc obiekty wewnętrzne, ale i sama jest częścią innego obiektu: układu kierowniczego, który jest znowu częścią obiektu samochód.

Nie wchodźmy w to póki co dalej ;) Zastanówmy się jednak nad obiektem i jego projektem. Projektem (schematem, opisem) jest klasa. Samochód, ale i każdy samodzielny element (obiekt) w samochodzie ma swój projekt, czyli klasę.

Zanim przejdziemy dalej, zastanówmy się jeszcze nad jednym terminem: **interfejs**.

Przekonasz się że obiekt, niech to będzie w naszym przypadku telewizor, ma interfejs, czyli zestaw metod, które są udostępniane innym obiektom. Np. obiekt człowiek może korzystać z telewizora za pomocą interfejsu telewizora, którym jest pilot. Zdecydowana większość obiektu telewizor jest niedostępna dla obiektu człowiek, ponieważ on ich nie potrzebuje. Obiektu człowiek nie interesuje, jak działa telewizor, on chce móc zmienić program czy regulować głośność. Tak też będzie w programowaniu, przekonasz się. To, co jedna klasa/obiekt udostępnia innemu, nazywamy interfejsem klasy/obiektu.

Podczas nauki w tym bootcampie upieczemy dwie pieczenie na jednym ogniu. Nauczymy się Javy i nauczmy się programowania zorientowanego obiektowo.

Jakie pojęcia nas interesują w dzisiejszym dniu:

class (klasa) - projekt, szablon obiektu. Na podstawie danej klasy można stworzyć wiele identycznych czy podobnych obiektów. W Javie cały kod znajduje się w klasach (przynależy do klasy). Przy czym pamiętajmy, że nawet dwa identyczne obiekty tworzone na podstawie klasy to dwa inne obiekty. Tak jak w życiu. Programowanie obiektowe jest bardzo życiowe :)

W klasie mamy atrybuty, które są cechami, właściwościami czy stanem tworzonego obiektu (np. kolor samochodu, prędkość maksymalna, liczba poduszek powietrznych, marka, liczba przejechanych kilometrów). Tworząc obiekt na podstawie klasy, nadajemy wartości indywidualne danemu obiektowi, tzn. możemy przyjąć, że każdy nowy samochód będzie miał 0 km przejechanych, ale będzie to już wartość przypisana do konkretnego obiektu samochodu, a nie do klasy 'samochód'. Czasami tworzy się też wartość współdzieloną przez wszystkie obiekty, w przypadku samochodu może to być marka (skoro wszystkie egzemplarze klasy mają mieć taką samą markę, to może nie ma co tworzyć tej wartości indywidualnie dla obiektu). Powrócimy do tych rozważań, gdy już mocniej przejdziemy do klas.

Co do metod w klasie, to traktuj je jako czynności, które mogą być wykonane przez klasę/obiekt na samym obiekcie, jak i na innym obiekcie (na innym obiekcie to właśnie za pomocą interfejsu czyli udostępnionych publicznie metod). Metody w danej klasie mogą też określać, jak dany obiekt będzie miał komunikować się (wchodzić w interakcje) z obiektami, od których będzie czegoś 'chciał', czy tych, które będą coś chciały od niego.

Wyobraźmy sobie 'pedał gazu' w samochodzie:

- jest on gotowy, by przyjąć dane od stopy - czyli ma metodę przyspiesz udostępnioną dla stopy. Metoda ta przyjmuje moc nacisku na pedał.
- ma też metodę przekazPrzyspieszenieDoSilnika, która komunikuje do silnika (poprzez jego interfejs) - przyspieszamy z taką i taką siłą.

Oczywiście taki pedał gazu ma też właściwości, jak np. opór (czasami pedał gazu wchodzi w podłogę bez oporu, ale to nie jest dobry znak ;)), wielkość, położenie.

Warto jeszcze pamiętać że kod, który znajduje się pomiędzy klamrami { }, np. nazwaMetody() { }, określamy **ciałem metody**.

Na teraz więcej nie potrzeba! :) Chyba, że Karol uważa, że jednak warto coś dodać. Karol? Myślę, że jest gitara. I tak będziemy wchodzić w opisywanie świata przez obiekty w tygodniach 4-5 ;)

object (obiekt) - element programu zaprojektowany do wykonywania określonych zadań. Obiekt jest tworzony na podstawie klasy. O obiekcie często mówimy, że jest instancją (a więc egzemplarzem) klasy.

variable (zmienna) - o tym będziemy się uczyć przez dwa kolejne dni.

method (metoda) - metodę w wielu innych językach określamy także funkcją (czasami wymiennie). W Javie nie ma funkcji umieszczonych poza klasami (a często, mówiąc funkcja, mamy na myśli taki samodzielny byt, niepowiązany z klasą), tak więc w Javie mówimy tylko o metodach w klasach, czy po prostu metodach.

Metoda to forma, w jakiej implementujemy zachowania obiektu w klasie.

Metoda to kod, który wykonuje określone zadanie. Technicznie możemy powiedzieć, że to zbiór instrukcji. Metoda może odnosić się do obiektu, w którym jest umieszczona (np. przetwarzać jakieś dane obiektu) lub do innych obiektów (coś udostępniać, coś pobierać, kazać coś zrobić innemu obiektowi).

Warto w tym momencie zauważyć, że istnieją metody, które wykonujemy na obiekcie (i takie są zdecydowanie najczęściej spotykane), ale i metody, które możemy wykonać na klasie (pamiętasz słowo kluczowe static w metodzie main?). Kiedy napiszemy trochę kodu, wszystko stanie się jasne, póki co więcej rozważań nie potrzebujemy, naprawdę.

scope (zakres) - zakres odnosi się do czasu życia i dostępności zmiennej. Jak duży jest zakres, zależy od tego, gdzie zadeklarowano zmienną. Na przykład, jeśli zmienna jest zadeklarowana jako pole w klasie (atrybut), będzie dostępna dla wszystkich metod klasy. Jeśli jest zadeklarowana w metodzie, może być używana tylko w tej metodzie. Wszystko stanie się jasne, jak zaczniemy pisać razem kod.

nazewnictwo - w Javie przyjęła się konwencja, by używać nazewnictwa w oparciu o tzw. notację wielbłądzą (ang. camelCase) - czyli pierwsze słowo nazwy piszemy małą literą a kolejne wielką. Przykładowo:

userName

someNumber

hasCriticalError

Pamiętajmy jednak, że ta konwencja nie dotyczy klas, które nazywamy już w pierwszym wyrazie wielką literą. Przykładowo:

ExampleClass

CarEngine

package (paczki, pakiety) - pakiet w Javie służy do grupowania powiązanych klas. Pomyśl o tym, jak o folderze w katalogu plików. Używamy pakietów, aby uniknąć konfliktów nazw i pisać lepszy w późniejszym utrzymaniu kod. Na czym może polegać konflikt nazw? Dwie takie same nazwy klas :) Wystarczy, że zgrupujemy je za pomocą pakietu i już nic się nie gryzie :)

import (import) - importy w Javie są tylko wskazówkami dla kompilatora, tak aby mógł on wiedzieć, do jakiej klasy się odwołujesz w swoim kodzie i gdzie kompilator może ją znaleźć. Importu można użyć raz na górze klasy i możesz wtedy spokojnie odwoływać się do importowanej klasy, natomiast w sumie to nie jesteś zmuszony ich używać. Możesz zapisywać pełną nazwę klasy (razem z pełną nazwą jej paczki) za każdym razem, gdy jej używasz (spróbuj zrobić to sam). Prowadzi to jednak do dużego nieporządku w kodzie i właśnie po to powstały importy. Pojawia się jednak problem, jeżeli chcemy importować 2 klasy o tej samej nazwie z różnych pakietów. Wtedy tylko jedna klasa może być wskazana na liście importów, druga natomiast (sam możesz wybrać, która) musi być używana razem z pełną nazwą jej paczki. Będziemy w bootcampie wielokrotnie coś importować, więc się nauczysz :)

semicolon (średniki) - średnik służy w Javie do oznaczenia końca instrukcji (czasami Karol mówi linijki, ale technicznie chodzi właśnie o instrukcje).

Średników nie stosujemy jednak zawsze! Są w Javie konstrukcje, które nie pozwalają na to, by na końcu linijki pojawił się średnik.

Chodzi o to, że nie wszystko w Javie jest instrukcją, np. deklaracja metody nie jest instrukcją, przykładowo.

`public void exampleMethod() {}` - nie umieszczasz średnika, bo tylko tworzysz a nie wykonujesz kod (instrukcja to polecenie wykonania kodu). Natomiast wywołanie metody, które wyglądałoby tak `exampleMethod();` - to już instrukcja. Zobacz przykład, który już znasz:

```
System.out.println('a');
```

Instrukcja wywołująca metodę kończy się średnikiem (w przykładzie wyżej wywołujemy metodę `println`).

Z racji, że Java uważa, że koniec linijki jest tam, gdzie jest średnik, to możliwe jest zapisanie czegoś w ten sposób:

```
public static void main(String[] args) {  
    int a = 1; int b = 3; int c = 12;  
    System.out.println("a: " + a + " b: " + b + " c: " + c);  
}
```

I zostanie to zrozumiane jako:

```
public static void main(String[] args) {  
    int a = 1;  
    int b = 3;  
    int c = 12;  
    System.out.println("a: " + a + " b: " + b + " c: " + c);  
}
```

Jeżeli zastanawiasz się, czemu Karol nie podawał żadnych reguł, kiedy należy pisać średniki a kiedy nie, to odpowiedź jest prosta. To w pewnym momencie staje się oczywiste i nawet się nad tym nie zastanawiasz, więc nie ma sensu uczyć się tego na pamięć.

Dzień 9 Bootcampu

Co to jest zmienna?

Zmienna to podstawowa struktura programistyczna we wszystkich popularnych językach programowania.

Zmienna składa się zawsze z nazwy i typu oraz najczęściej przechowuje jakąś wartości lub obiekt. W istocie przechowuje nie samą wartość, a odnośnik do pamięci, gdzie dana wartość się znajduje (ale potocznie możemy powiedzieć, że przechowuje daną wartość).

Dzięki zmiennym możemy przechowywać dane i operować na danych w programie.

Deklaracja zmiennych

Zmienną możemy zadeklarować, wskazując dwa elementy:

1. typ przechowywanych danych (np. któryś z typów prostych, które poznajemy w dzisiejszym dniu bootcampu Zajavka)
2. nazwę, która nie może być nazwą zastrzeżoną (np. niepoprawne jest użycie składni `int int` czy `int class`) i musi zaczynać się od litery (czyli `1value` nie jest prawidłowe, a `value1` już tak)

Przykładowe poprawne deklaracje:

```
int customerNumber;  
boolean hasPermission;
```

Konwencja mówi też, że używamy notacji wielbłądziej, ale możesz używać też zamiast niej np. podkreśleń czyli nazwa `has_permission` jest ok.

Jeśli deklarujemy zmienne tego samego typu, to możemy to zrobić w jednym wierszu:

```
int number1, number2, number3;  
double a,b,c,d;
```

Na końcu deklaracji zawsze umieszczamy średnik!

Inicjalizacja zmiennej

Polega na przypisaniu wartości do zmiennej.

W poprzednim przykładzie stworzyliśmy (zadeklarowaliśmy) 4 zmienne: double a,b,c,d;

Inicjalizujemy je, przypisując im wartość, do czego służy operator przypisania reprezentowany przez znak równości.

```
double a,b,c,d;
```

```
a = 3;
```

```
b = 4.32423;
```

```
c = 1999999.1;
```

```
d = 0;
```

Bardzo często razem z deklaracją robimy też jednocześnie **inicjalizację**, czyli nadajemy wartość danej zmiennej.

```
int age = 56;
```

```
boolean isProgrammer = true;
```

W tym momencie to wszystko, co musimy wiedzieć o zmiennych.

Pamięć a zapisywane w zmiennych wartości

Wartości używane w programie są zapisywane w pamięci. Program korzysta z pamięci RAM (jest to pamięć tymczasowa).

Podstawowa jednostka w pamięci jest reprezentowana przez bajt, a więc 8 bitów.

Polecam ten film o działaniu procesora, pamięci, kodzie maszynowym oraz przekształceniu liczb binarnych na dziesiętne i w drugą stronę. Tym bardziej polecam, że jestem jego autorem :) Ale warto go obejrzeć, bo tłumaczy sporo.

<https://youtu.be/11ToKaoZqHs?t=104>

Typy prymitywne (inaczej proste, podstawowe)

Zacznijmy od typów całkowitych (liczby całkowite, typ całkowitoliczbowy).

byte - składa się z 1 bajta (ang. byte), czyli podstawowej, minimalnej wielkości pamięci. Bajt, jak pokazywał Karol, tworzy 8 bitów, które mogą przyjąć wartość 0 lub 1 (czyli system dwójkowy). Za pomocą byte możemy zapisać wartości od -128 do 127. Np. wartość 3 zostanie zapisana jako 00000011 a wartość 100 jako 01100100.

short - tutaj zawsze wykorzystane będą dwa bajty. Oczywiście, w shorcie, jak i innych typach numerycznych, można przechować także wartość, np. 0, przy czym wtedy zajmiemy i tak dwa bajty (00000000 00000000). Short może pomieścić wartości od -32 768 do 32 767. Ps. nie ucz się tego na pamięć!

int - tworzą go 4 bajty a możliwy zakres przechowywanych liczb wynosi w przybliżeniu od -2.1 mln do 2.1 ml.

long od (w przybliżeniu) - 9.2 tryliarda do 9.2 tryliarda (dokładnie 9 223 372 036 854 775 807). W 99.999 proc przypadków nie potrzebujesz większej liczby niż tryliard :) long korzysta z 8 bajtów. Long wystarczy nawet do zapisania prawdopodobnej liczby gwiazd we wszechświecie widzialnym (tak więc gdyby świat był symulacją, to liczbę gwiazd zmieścilibyśmy w 4 bajtach :))

Typ long wymaga ponadto użycia litery l lub L (małe lub wielkie 'L') - choć możemy użyć obu, to Karol rekomenduje wielkie, bo, jak widać, małe 'l' może się mylić z 1.

Na pierwszy rzut oka wydaje się, że do większości zastosowań nadaje się int, ale biorąc pod uwagę, że 4 bajty to naprawdę mało, to czy nie korzystać z long na wszelki wypadek (byte, short niemal wcale, a int tylko, jeśli na 100% wiemy, że w dalekiej przyszłości potrzeby się nie zmienią)? - To pytanie kieruje do Karola i mam nadzieję, że tu albo w przyszłości usłyszymy odpowiedź :)

Generalnie w praktyce należy się zawsze zastanowić nad możliwym zakresem wartości. Jeżeli już jesteś pewien, że Twoja wartość się nie zmieści, to używaj long. Przykładowo, na co dzień do przechowywania id klienta, na którym operujemy, stosuje się long. W przyszłych filmach będę też mówił o czymś takim, jak BigInteger i BigDecimal, gdyby podane zakresy były za małe, albo potrzebowalibyśmy dużej dokładności obliczeniowej.

Liczby możemy zapisywać także w innych systemach niż najbliższy ludziom dziesiętny, choć, oczywiście, najczęściej używa się systemu dziesiętnego (ang. decimal). Tak więc w Javie prawidłowym będzie zapisanie liczby 10 także w innych systemach.

Liczba 10 zapisana w Javie za pomocą innych systemów liczbowych:

szesnastkowym: 0xA (przedrostek 0x // 0 to zero)

ósemkowym: 012 (przedrostek 0)

dwójkowym (binarnym): 0b1010 (przedrostek 0b)

Przykład deklarowania i inicjalizowania zmiennych typu liczb całkowitych

```
byte number1 = -10;  
short number2 = 12280;  
int number3 = -1240239; // możemy też zapisać -1_240_239  
long number4 = 23948239234L; // możemy też zapisać 23_948_239_234L
```

Typy zmiennoprzecinkowe

Typ zmiennoprzecinkowy pozwala przechowywać wartości zmiennoprzecinkowe (czyli ułamki). W Javie występują dwa takie typy:

float - jego wielkość to 4 bajty;

double - bardziej precyzyjny (double - podwójnie precyzyjny) - przechowuje wartość ułamkową za pomocą 8 bajtów (czyli 64 bitów). Powinien być traktowany jako domyślny typ do pracy z wartościami ułamkowymi.

Jeśli korzystamy z float, to po liczbie powinniśmy użyć znaku 'f' lub 'F'. Przyrostka nie potrzebujemy przy double, choć możemy użyć go także tu i będą to znaki "d" lub "D"

Przykład deklarowania i inicjalizowania zmiennych:

```
float number1 = 2.390723F;  
double number2 = 2.43;
```



```
double number3 = 1 //warto wiedzieć, że i tak zapisywane/odczytywane  
jest jako 1.0
```

Typ boolean

Ja wymawiam 'bulin', Karol widzę podobnie ;) Zdarza się wymowa 'bulen'.

Przyjmuje on jedną z dwóch wartości: prawda lub fałsz, a właściwie **true i false**, pisane małą literą! **W niektórych innych językach można też używać 0 lub 1, tutaj nie! ;)**

```
boolean value1 = true;  
boolean value2 = false;
```

Typ char

char - wymawiamy 'czar' lub 'kar' (wtedy od słowa 'character', czyli znak) :)

Bo typ char to właśnie pojedynczy znak.

Specyfiką typu char jest np. to, że nie możemy tu zastosować cudzysłowu (" "), a musimy posłużyć się apostrofem (' ').

```
char elementB = 'B';  
System.out.println(elementB);
```

Wydrukuję nam B, oczywiście.

Ale możemy użyć też liczby, która reprezentuje dany kod w **ASCII** lub **Unicode**:

```
char elementB = 66;  
System.out.println(elementB);
```

Liczba 66 w tablicy znaków ASCII reprezentuje właśnie znak B. Warto wiedzieć, że w tablicy ASCII/Unicode litery wielkie i małe mają przypisane inne liczby, np. małe b jest reprezentowane przez liczbę 98.

Pokrótko o samym ASCII i Unicodzie. Procesor pracuje na liczbach, ale my chcemy używać też liter i innych znaków. Jak to zrobić? Właśnie za pomocą kodowania, czyli przypisywania

jakiemuś znakowi liczby, która go reprezentuje. Tak powstało ASCII a później Unicode. Unicode jest rozszerzoną wersją ASCII, ASCII bowiem nie zawiera liter i znaków z innych alfabetów. Jeśli chcemy uzyskać taki znak, np małe polskie "ł", to musimy posłużyć się liczbą z tablicy Unicode (czyli większym numerem niż daje ASCII). Java i typ char, oczywiście, sobie z tym poradzą.

```
char character = 322;  
System.out.println(character);
```

zwrócone zostanie "ł"

Nie potrzebujemy w tym momencie wiedzieć nic więcej o ASCII i Unicode, ani tym bardziej o sposobie kodowania UTF-16 ;). Zresztą na co dzień też nie musisz się kompletnie przejmować wartościami znaków w Unicodzie czy ASCII (przy okazji, ASCII wymawiamy jak 'aski'). A i korzystanie z typu char też jest bardzo, bardzo rzadkie, ponieważ korzystamy z czegoś takiego, jak łańcuch (String), o czym na pewno dowiemy się w przyszłości :).

Co jeszcze warto wiedzieć:

Java posiada **ściśłą kontrolę typów**, tzn. wartość przechowywana w zmiennej musi mieć określony, zadeklarowany typ. **Tutaj przypomnę pytanie z testu o typowanie (test nr 2) ;)**

W nazwach można używać liter z tablicy Unicode, a więc poprawne jest użycie w nazwie (np. zmiennej) polskich liter. Natomiast konwencja jest taka, że tego nie robimy. Nazwy piszemy po angielsku.

Liczby zmiennoprzecinkowe (float, double) nie są dobrym rozwiązaniem do zapisywania wartości pieniężnych i operacji finansowych. Sposób zapisu liczby w pamięci (w systemie binarnym) powoduje, że nie ma tu wystarczającej precyzji w systemie dziesiętnym.

```
System.out.println(1.1 - 0.2);  
Wydrukuje: 0.9000000000000001
```

```
System.out.println(1.2 - 0.1);
```

Wydrukuje: 1.0999999999999999

Karol wspominał, że używa się tutaj klasy **BigDecimal**, która z tym 'błędem' sobie radzi - i o tym pewnie dowiemy się w przyszłości.

DZIEŃ 10-12

W tych dniach pojawiło się sporo informacji, ale niemal wszystkie one dotyczyły typów i zmiennych, więc łączę je we wspólnych notatkach za te dni:

final - stałe

W Javie istnieje co prawda takie słowo jako **const** (const od constant - stała), ale nie jest ono do niczego używane. Wspominam o tym (i wspominał Karol), ponieważ zdarza się, że const w innych językach służy do konstruowania stałej (w JavaScript na przykład).

W Javie także możemy tworzyć stałe, ale służy do tego słowo kluczowe **final**.

Słowo kluczowe final pozwala tworzyć stałą. Co to oznacza? Final tworzy trwałe łącze (trwałą **referencję**) z miejscem w pamięci. I to połączenie nie może być już zmieniane w innym miejscu programu. W praktyce przypisujemy do zmiennej typ prosty (jak przypiszemy 20, to już zawsze będzie 20) lub obiekt i taka zmienna zawsze już będzie połączona z tym obiektem, choć sam obiekt może ulec modyfikacji a final nie czyni go 'zamrożonym' (niemutowalnym, niezmiennalnym), co też podkreśla Karol.

```
final int value = 10;
value = 10 + 1; // Błąd
//The final local variable value cannot be assigned.
```

Często dla stałych używamy nazw zbudowanych z wielkich liter, np.:

```
final double MAX_WIDTH = 500.5;
```

Czy final to więc ciągle zmienna? Tak, to **zmienna, która jest stała** :) I jakby to dziwnie nie brzmiało, to tak należy to rozumieć. Pamiętając, że chodzi tu o trwałe połączenie z określonym miejscem w pamięci.

Rzutowanie

Rzutowanie to jawna konwersja jednego typu na inny. Bez problemu możemy rzutować typ, który ma mniej bajtów, na typ, który ma więcej bajtów np. float na double czy int na long.

Oprócz rzutowania, rozumianego jako **jawna konwersja** (za pomocą jasnej deklaracji), występuje też **konwersja automatyczna**.

Przyjrzyjmy się takim instrukcjom

```
int value1 = 2; //wartość typu int
double value2 = value1; //wartość konwertowana automatycznie na
double i przypisana do nowej zmiennej
```

Wartość przypisana do value2, a znajdująca się w value2, zostanie zamieniona na double (mimo, że przekazujemy int). Gdybyśmy chcieli ją odczytać, to byłoby to 2.0

W drugą stronę taka automatyczna konwersja nie zadziała.

```
double value1 = 2;
int value2 = value1;
// błąd: cannot convert from double to int
```

Typ double ma większą liczbę bajtów w pamięci niż int (jeśli nie pamiętasz ile, to znajdziesz to we wcześniejszych notatkach :)). Dlatego kompilator odmawia automatycznej konwersji. Podobnie będzie, gdy spróbujemy konwertować long na int.

```
long value1 = 2L;
int value2 = value1;
// błąd: cannot convert from long to int
```

Takiej konwersji możemy jednak dokonać jawnie (co nazywamy **rzutowaniem**, z ang. **casting**). Oczywiście, istnieje tutaj ryzyko utraty precyzji (a więc prawidłowych informacji).

Zobaczmy przykłady.

```
long value1 = 29843L;  
int value2 = (int) value1;
```

W ten sposób uda nam się zapisać wartość ze zmiennej value1 (typu long) w zmiennej value2 (typu int) i to, w tym wypadku, bez straty, ponieważ wartość przechowywana w long da się zamieścić także w int. Ale zobaczmy ten przykład:

```
long value1 = 9843495834905833L;  
int value2 = (int) value1;
```

Otrzymamy taką wartość w value2: 2023124201 - czyli na pewno nie taką, jakiej oczekujemy :)

Możemy, oczywiście, rzutować także typ zmiennoprzecinkowy na typ całkowitoliczbowy.

```
double value1 = 33.6421;  
int value2 = (int) value1;
```

W tym wypadku otrzymamy wartość całkowitą 33, po prostu ułamek zostanie odrzucony. **Zwróć uwagę, że nie dokonało się zaokrąglenie, Javka zwyczajnie odrzuca to, co było po przecinku.**

String

String to typ, który przechowuje ciąg znaków (można go traktować jako zbiór znaków Unicode). Nie jest to jednak typ prosty.

String jest klasą, która, jak wiele innych klas i metod, jest dostarczana przez samą Javę. Mówimy tu więc o rozwiązaniu, które jest dostarczane przez standardową bibliotekę Javy (zwaną też API Javy) w ramach JDK. **API Javy** dostarcza wiele klas i metod i jest dostępne dla programu napisanego w Javie (np. metoda println jest częścią tego API).

```
String bootcampTitle = "Zajavka";
```

String zawiera znaki i jest **obejmowany cudzysłowem** (ale nie apostrofem! Apostrof jest przy typie char).

Zwróć uwagę na nazwę typu. Jest to **String**. I, jak już wiesz, nie należy on do typów prostych, co sugeruje już sama nazwa pisana dużą literą. To chyba pierwszy przykład w tym bootcampie, gdzie tworzymy wartość, która jest obiektem, a nie typem prostym.

Czy dwa stringi o tej samej zawartości są takie same? I to jest bardzo dobre pytanie, szczególnie jeśli uczymy się Javy ;) Takie pytania też padają na rekrutacji, dlatego Karol wprowadził takie pojęcia jak String pool i metoda intern(), ale o tym na koniec. Wróćmy do pytania, czy stringi o tej samej zawartości są takie same?

Zacznijmy od porównania za pomocą dwóch znaków równości czyli '==' (co nie jest dobrym rozwiązaniem, jeśli chodzi o stringi).

Tak więc, jeśli dwa stringi powstały za pomocą **literału***, to ich porównanie za pomocą operatora == zwróci nam true.

* literał to zrozumiała dla kompilatora konstrukcja znaków, w tym wypadku (stringów), zaczynająca się i kończąca cudzysłowem, która wprost definiuje jakąś wartość. Inne literały to literał liczbowy, np. 42 czy true (literał tworzący typ bool).

```
// Przykład 1
String txt1 = "tekst";
String txt2 = "tekst";
System.out.println(txt1 == txt2); //true
```

wydrukuje: **true**

Możemy też tworzyć stringa jak inne obiekty na podstawie odwołania do klasy i **słowa kluczowego new**. Ale tu ciekawostka. Porównanie tych dwóch wartości (obiektów) da już false. Dlaczego? Bo wartości tych dwóch obiektów, mimo, że takie same, trzymane są w różnych miejscach pamięci. W przypadku literałów Java uznaje, że jeśli tworzysz string o takiej samej wartości (zawierający te same znaki), który już był tworzony, to optymalizuje to i przypisuje do tego samego miejsca w pamięci. W przykładzie powyżej powstała więc jedna wartość w pamięci, do której prowadzi zarówno pierwsza, jak i druga zmienna. W drugim przykładzie, tym poniżej, mamy też dwa obiekty, ale już dwie różne pozycje w pamięci.

```
// Przykład 2
String txt1 = new String("tekst");
String txt2 = new String("tekst");
System.out.println(txt1 == txt2); //false
```

wydrukuję: **false**

Pamiętaj, że za pomocą dwóch znaków równości sprawdzamy nie to, czy wartości stringów są takie same, ale czy prowadzą do tego samego miejsca w pamięci (czyli czy zmienne mają referencje do tego samego miejsca pamięci!) W praktyce więc, mimo że mogą mieć taką samą zawartość, to mogą prowadzić do różnych miejsc pamięci, w zależności, jak zostały utworzone. Można pomyśleć, że to głupie! No cóż ;)

No dobrze, a co się dzieje tutaj (w przykładzie 3)? Stworzyliśmy nowy obiekt String. Następnie przypisujemy zmienną txt1 (zawierającą referencje do obiektu) do nowej zmiennej typu String. I mamy zaskoczenie (albo i nie ;)), są takie same, tzn. wskazują na ten sam obiekt. Jest tylko jeden obiekt i zarówno txt1 i txt2 wskazują na niego.

```
// Przykład 3
String txt1 = new String("tekst");
String txt2 = txt1;
System.out.println(txt1 == txt2); //true
```

wydrukuję: **true**

Częściowo stanie się to zrozumiałe, jeśli potraktujesz zmienną nie jako pojemnik na dane, tylko link do miejsca w pamięci, tzw. referencję. Kiedy tworzysz dwa obiekty (dwie instancje, dwa egzemplarze), nawet identyczne, to tworzysz dwa różne obiekty. Dwa różne obiekty nie są tymi samymi obiektami a to właśnie jest sprawdzane poprzez dwa znaki równości. Stąd false w przykładzie drugim. Natomiast w przykładzie pierwszym najpierw stworzyliśmy obiekt, ale kiedy już jeden obiekt w pamięci istnieje, a tworzymy taką samą wartość za pomocą literału, to wskazujemy na tę wartość, która już istnieje. **Dodam mały komentarz, że cały czas rozmawiamy o stringach, żeby ktoś nie pomyślał, że tak jest w przypadku każdego innego rodzaju danych.**

```
String txt1 = "tekst"; // stwórz obiekt String z wartości 'tekst'
String txt2 = "tekst"; // wiesz co, mam już taki obiekt z taką wartością w pamięci
                        // więc skieruje Cię do tej wartości zamiast tworzyć nową.
```

W przykładzie trzecim widzimy jeszcze jedną ważną rzecz. Otóż w pierwszej instrukcji tworzymy obiekt i referencję do niego przypisujemy do zmiennej txt1. Natomiast w drugiej instrukcji tworzymy zmienną typu String i nie kopiujemy do niej obiektu (nie tworzymy nowej instancji!) a jedynie zapisujemy w zmiennej referencję do miejsca w pamięci, gdzie jest ten obiekt. Obie zmienne są więc referencjami do tego samego obiektu!

Zapamiętaj więc, że sposób z dwoma znakami równości nie jest dobrym sposobem na porównanie stringów, bo w zależności od tego, jak powstały (a nie wymieniliśmy wszystkich możliwości, np. różnych metod, które przykładowo tworzą z jednego stringa inny string), rezultat może być inny. Do porównania stringów najlepszym rozwiązaniem jest **metoda equals**.

```
String txt1 = new String("tekst");
String txt2 = new String("tekst");
System.out.println(txt1 == txt2); //false
System.out.println(txt1.equals(txt2)); //true
```

Zapamiętaj więc konstrukcję:

String1.equals(String2);

np.

```
"A".equals('a'); // zwróciłoby false

String numberTXT = "120";
"120".equals(numberTXT); // zwróciłoby true

"A".equals("A"); // zwróciłoby true
```

Equals porównuje nie miejsce w pamięci, a rzeczywistą zawartość obiektu. W przypadku Stringa liczy się więc tutaj, co zawiera, a nie czy jego wartość jest trzymana w tej samej czy innej zmiennej. Przy czym pamiętajmy, że metoda equals sprawdza, czy obiekty (stringi) są takie same, a nie te same!

Warto wiedzieć, że string może przyjąć też takie wartości jak:

- **pusty string**, czyli jedynie cudzysłów "" niezawierający nic w środku.
- wartość **null**, co oznacza, że nie jest do niego przypisane żadne miejsce w pamięci.

Co warto wiedzieć o stringach jeszcze:

- są niezmiennicze (**niemutowalne**), tzn. jeśli przekształcamy string, to tworzymy tak naprawdę nowy.
- **String pool** to miejsce w pamięci, do którego trafiają niektóre stringi. Jest to spowodowane optymalizacją Javy. Przy tworzeniu nowego stringa za pomocą literału Java sprawdza, czy dany element jest już w String pool, czyli czy ma dany string (zawierający dane znaki) w pamięci określonej jako pool (stringi stworzone za pomocą `new` nie są trzymane w pool). Jeśli tak, to przypisuje zmienną (referencja) do tej wartości.

```
String txt1 = "tekst";  
String txt2 = "tekst"; // mechanizm String pool  
String txt3 = "tekst"; // mechanizm String pool  
String txt4 = new String("tekst"); // nowy obszar pamięci -  
domyślnie nie korzysta ze String pool
```

Zapamiętaj: Java przy tworzeniu stringów z literałów (przypominam, że to też obiekty), sprawdza, czy w pamięci nie przechowuje już danego tekstu, a jeśli tak, to powoduje współdzielenie tego tekstu z innym obiektem (optymalizuje w ten sposób pamięć). Jeśli natomiast dodalibyśmy string za pomocą konstrukcji `new`, zawsze tworzone jest nowe miejsce w pamięci, nawet z taką samą wartością.

- String intern - **metoda intern** służy do wymuszania zapisu wartości tworzonego stringa w String pool - chodzi, oczywiście, o stringi stworzone z operatorem `new`.

```
String txt1 = new String("string");  
String txt2 = "string";  
String txt3 = txt1.intern();  
  
System.out.println(txt1 == txt2); //false, bo zapisane w  
innym miejscu pamięci  
System.out.println(txt2 == txt3); //true, bo wymusiliśmy  
zapisanie w pool, więc prowadzi do tego samego miejsca w  
pamięci, w którym znajduje się już txt2
```

Tutaj może to napiszę, bo Bartek zadzwonił do mnie przy pisaniu tych notatek, żeby mi powiedzieć, że chyba za mocno dowaliłem tą String poolą i metodą `intern()` ;). Chciałbym

tutaj napisać, czemu ten temat wjechał. Na rozmowach rekrutacyjnych potrafi paść takie pytanie: wytłumacz, czemu w kodzie poniżej obserwujemy takie zachowanie, jak widać i powiedz co zrobić, żeby zamiast false było drukowane true:

```
String txt1 = "tekst";
String txt2 = new String("tekst");
System.out.println(txt1 == txt2); //false
System.out.println(txt1.equals(txt2)); //true
```

Bez posiadania wiedzy nt. String pooli i metody intern, nie ma opcji, że to wytłumaczysz ;)

Konkatenacja

Łączenie stringów czy innych typów ze stringami - to oznacza właśnie pojęcie konkatenacji. Dokonujemy jej za pomocą operatora +. String jest zwracany, jeżeli choć jedna wartość łączona jest stringiem.

Zobaczmy przykład:

```
System.out.println("A" + "B"); // Wynik AB
System.out.println("A" + 2); //Wynik A2
System.out.println(2 + 3 + "A" + 2); //Wynik 5A2
```

Widać, że mamy tutaj kolejność od lewej do prawej (można ją zaburzyć, dodając nawiasy, to co w nawiasach ma pierwszeństwo). Zaskakujący może być przykład trzeci. Widzimy, że łączymy tutaj 2 i 3, czyli w istocie nie łączymy a za pomocą operatora "+" dodajemy dwie liczby, stąd 5 a nie "23". Dopiero w kolejnym działaniu mamy połączenie z "A". **Czyli jak pojawi się już String (idąc od lewej), to reszta już jest dodawana jako String.**

Do konkatenacji możemy też użyć **metody concat** (dostępnej dla stringów, bo dostarczanej z klasą String).

```
String txt1 = "Mam ";
String txt2 = "chęć ";
String txt3 = "Na Javę!";

String newtxt = txt1.concat(txt2).concat(txt3);
```

```
System.out.println(newtxt); // Mam chęć na javę
```

Metody z klasy String

String jest więc klasą. Co nam to daje? Możliwość wykonania na nim różnych metod, czyli działań. Razem z klasą String dostarczone jest bowiem kilkadziesiąt metod, które można wykonać na tekście (czyli na obiekcie typu String).

.length()

Metoda length zwraca długość stringa, czyli liczbę znaków w stringu, oczywiście, uwzględnia białe znaki. Zwracana wartość ma typ int.

```
String name = "Władysław";  
System.out.println(name.length()); //wydrukuje 9
```

```
String name = "";  
System.out.println(name.length()); //wydrukuje 0
```

.charAt()

Metoda charAt zwraca element stringa ze wskazanej pozycji. Zwracany jest typ char, więc jeśli chcielibyśmy go przypisać do zmiennej, to musi być to zmienna char.

```
String name = "Dzień dobry";  
System.out.println(name.charAt(4)); //wydrukuje ń  
char a = name.charAt(4); przypisze 'ń'
```

.repeat()

Metoda repeat zwraca wartość String, który jest nowym stringiem składającym się z wielokrotności wartości stringa, na którym jest wykonywana metoda.

```
String name = "Dzień dobry";
```

```
System.out.println(name.repeat(3)); // Wydrukuj "Dzień
dobryDzień dobryDzień dobry"
String a = name.repeat(2); // Przypisze do zmiennej stringa
"Dzień dobryDzień dobry"
```

.substring()

Metoda wykonywana na stringu. Zwraca nowy string wg podanych parametrów. Możemy podać jeden lub dwa parametry. Jeśli jeden, to od którego indeksu i domyślnie do końca stringa. Jeśli dwa, to pierwszy parametr mówi, od którego indeksu a drugi, do którego (ale bez tego). Pamiętaj, że indeks liczymy do 0 a nie 1.

```
String name = "Jan Maria";
System.out.println(name.substring(3)); // wydrukuj " Maria"
String a = name.substring(1, 7); // zwróci "an Mar"
System.out.println("Zdanie do przerobienia".substring(3, 5));
// wydrukuj "ni"
System.out.println(a); // wydrukuj "an Mar"
```

.concat()

Metoda dokonująca konkatencji (połączenia) stringów. Zwraca nowy string będący połączeniem dwóch stringów.

```
String name = "Jan";
String name2 = "Roman";
System.out.println(name.concat(name2)); // wydrukuj
"JanRoman"
String a = name.concat(name2); // zwróci "JanRoman" i zapisze
w zmiennej a
System.out.println(name); // wydrukuj "Roman"
System.out.println(a); // wydrukuj "JanRoman"
```

.contains()

Zwraca true lub false (a więc typ boolean!), sprawdzając, czy string, na którym jest wykonana metoda, zawiera przekazany do metody string.

```
String name = "Dzień dobry";  
System.out.println(name.contains("obry")); // zwróci true  
boolean a = name.contains("ziń"); //zwróci false  
System.out.println(a); //wydrukuj false
```

.replace()

Zamienia fragment (pierwszy pasujący) w stringu, na którym jest wykonywana (jeśli, oczywiście, taki fragment znajdzie)

```
String name = "Hej Javowcy!";  
System.out.println(name.replace("hej", "cześć")); // wyświetli  
stringa pierwotnego, ponieważ nie ma w Stringu name tekstu  
"hej" (wielkość liter ma znaczenie). Metoda replace zwraca za  
każdym razem nowego stringa, nawet jeśli jest taki sam jak ten  
na którym pracuje.  
String a = name.replace("Hej", "Cześć"); // tutaj zadziała i do  
zmiennej a zostanie przypisany nowy string "Cześć Javowcy!"  
System.out.println(name); // wyświetli Hej Javowcy!  
System.out.println(a); // wyświetli Cześć Javowcy!
```

.toLowerCase() i toUpperCase()

Metoda wykonana na stringu zwraca nowy string pisany małymi literami (toLowerCase) lub wielkimi (toUpperCase). Pamiętajmy, że string pierwotny nie jest zmieniany (bo jest niemutowalny) i nie jest przypisywany nowy. Te dane, na których pracujemy metodami, nie są zmieniane.

```
String name = "ESTONIA";  
System.out.println(name.toLowerCase()); // wydrukuj estonia
```

```
String a = "Tomasz Nowak".toUpperCase() // wydrukuj TOMASZ NOWAK
System.out.println(name); // wydrukuj ESTONIA
System.out.println(a); // wydrukuj TOMASZ NOWAK
```

.trim()

Zwraca nowy string, usuwając z niego spacje z początku i końca.

```
String name = "  aa bb cc  ";
System.out.println(name.trim()); // wydrukuj "aa bb cc"
String a = "Tomasz Nowak ".trim(); // zwróci "Tomasz Nowak"
System.out.println(name); // wydrukuj "  aa bb cc  "
System.out.println(a); // wydrukuj "Tomasz Nowak"
```

var

Gdy nie chcemy wskazywać typu podczas deklarowania zmiennej, to od wersji 10 nie musimy. W takim wypadku Java automatycznie stworzy typ (bo typ musi być zawsze! Przecież typ nadawany jest dynamicznie przy var).

Pamiętajmy też, że var można stosować tylko w **zmiennych lokalnych**, a więc zmiennych tworzonych w metodach, ale już nie w polach obiektów (ale czym są te pola, to już w programowaniu obiektowym).

```
public class zajavka {
    var age = 20; // błąd: 'var' is not allowed here
    public static void main() {
        // metoda - tutaj powstają zmienne lokalne
        var age2 = 20; // tutaj jest ok
    }
}
```

// oczywiście w if, for też mogą być zmienne z var bo nadal to zmienne lokalne, bo są w metodzie.

Przykład działania (pamiętaj, że chodzi o zmienne lokalne)

```
var a = 1; // stworzy inta
var b = "2"; // stworzy stringa
var c = 58345908350834L; // stworzy longa
```

Są zwolennicy i są przeciwnicy tego rozwiązania :)

Zasięg i cykl życia zmiennych

Wprowadzamy też pojęcie **scope** (zasięg, zakres) zmiennej.

Java pozwala nam deklarować zmienne w dowolnym **bloku kodu**. Blok kodu to przestrzeń ograniczona nawiasami klamrowymi {}. Jeden blok kodu może być, oczywiście, zagnieżdżony w innym, co dobrze widzimy na tym przykładzie.

```
public class main {
// 1 blok kodu
    public static void main(String[] args) {
// 2 blok kodu
        {
//3 blok kodu
        }
    }
}
```

W przyszłości poznasz wiele takich struktur: klasy, metody, pętle, instrukcje warunkowe - to najpopularniejsze z nich.

Blok kodu tworzy zasięg naszej zmiennej. Zmienna stworzona w danym bloku jest widoczna tylko w danym bloku i w blokach potomnych (zagnieżdżonych), ale nie tych nadrzędnych. Druga sprawa to czas życia takiej zmiennej. Otóż zmienna 'żyje' w pamięci, dopóki istnieje blok.

Kolejna sprawa to kwestia deklaracji. Zmienna żyje (jest widoczna i dostępna w kodzie) dopiero od momentu deklaracji.

Przyjrzyjmy się przykładowi:

```

public class main {
    public static void main(String[] args) {
        int a = 1;
        if (a < 10) {
            int b = 2;
            {
                int c = 3;
                System.out.println(a); // wydrukuje 1
                System.out.println(b); // wydrukuje 2
                System.out.println(c); // wydrukuje 3
            }
        }
    }
}

```

Ale to już nie zadziała:

```

public class main {
    public static void main(String[] args) {
        int a = 1;
        if (a < 10) {
            int b = 2;
            {
                int c = 3;
            }
            System.out.println(a);
            System.out.println(b);
            System.out.println(c); // Błąd! nie ma dostępu (nie widzi)
            takiej zmiennej
        }
    }
}

```

Pamiętajmy też, że mimo innego zakresu nie możemy używać tych samych nazw w różnych blokach.


```

public class main {
    public static void main(String[] args) {
        int a = 1;
        if (a < 10) {
            int a = 2; // błąd!
            {
                int a = 3; // błąd!
            }
        }
    }
}

```

O czym warto jeszcze pamiętać:

- zmienna zdefiniowana w bloku nazywana jest zmienną lokalną danego bloku.
- kiedy dany blok się wykona, dana zmienna przestaje istnieć (jest usuwana z pamięci)
- zasięg jest zagnieżdżony. Coś w zasięgu wyższym jest dostępne w zasięgu zagnieżdżonym, ale nie odwrotnie.

```

{
    String name = "a";
    {
        System.out.println(name); //zmienna  jest dostępna
    }
}

```

a tutaj przykład odwrotny:

```

{
    {
        String name = "a";
    }
    System.out.println(name); // nie jest dostępna
}

```

Operatory

Zaczęliśmy naukę operatorów. Mam nadzieję, że te notatki pomogą Wam uzupełnić proces nauki rozpoczęty w lekcjach Karola :)

Operatory arytmetyczne

No co tu pisać :) Podstawy matematyki.

Dodawanie, odejmowanie, mnożenie, dzielenie. Ale nie tylko...

Najpierw zwróćmy uwagę na ciekawe przykłady Karola z filmów.

```
double d = 4 / 3; //wyrażenie 4/3 zwraca 1 (bo zwraca int gdy
dzielimy dwie wartości int!) i zapisze je w zmiennej d jako
typ double (a więc 1.0)
```

```
double e = (double) 4 / 3; // rzutujemy pierwszą liczbę double
(czyli 4.0) i mamy w związku z tym wynik który jest doublem
czyli 1.3333333333333333 i tyle zapisujemy do zmiennej.
```

Warto jeszcze poznać **modulo**, który oznacza resztę z dzielenia. Jaka jest reszta z dzielenia 8 przez 2 (czyli w programowaniu $8/2$ a w matematyce $8:2$)? Reszta, oczywiście, wynosi 0. A z 9 przez 2? Tu reszta wynosi jeden.

```
int d = 4 % 3; //wynik to 1
int e = 900 % 400; //wynik to 100
int f = 1001 % 1001; // wynik to 0
```

Promocja numeryczna - Pamiętaj, że przy dodawaniu dwóch liczb wynik dodawania jest podnoszony do wyższego typu.

Zerknij na przykłady

```
byte d = 10;
short e = 200;
short f = d + e; // błąd! Pamiętaj, że w wyniku dodawania byte
do short zwracany jest typ int, nawet jeśli taki typ nie jest
potrzebny, by przechować powstałą wartość.
```

teraz dobrze:

```
byte d = 10;
short e = 200;
int f = d + e; //użyliśmy int
```

Albo teraz:

```
byte d = 10;
short e = 200;
short f = (short) (d + e); //wynikiem dodawania będzie int ale
rzutujemy go na short.
```

Podobna sytuacja ma miejsce, gdy dodajemy int do long. Wtedy zwracana wartość jest longiem.

```
int d = 10;
long e = 200;
int f = d + e; błąd! cannot convert from long to int
```

W takim wypadku mamy dwie możliwości:

- zamiast int f, tworzymy long f

```
int d = 10;
long e = 200;
long f = d + e;
```

- rzutujemy wynik d + e na int, jak poniżej

```
int d = 10;
long e = 200;
int f = (int) (d + e);
```

Promocja numeryczna dotyczy też typów zmiennoprzecinkowych.

Karol wrzucił ściągawkę, jednak nie musisz się jej uczyć na pamięć, bo z czasem będziesz to czuć.

Jeśli 2 wartości są innego typu, Java automatycznie wypromuje jedną z nich do wyższego typu.

Jeśli jedna z nich nie jest zmiennoprzecinkowa a druga jest, to ta pierwsza zostanie wypromowana do typu zmiennoprzecinkowego.

Mniejsze typy danych, a mianowicie bajt, short i char, są najpierw promowane do wartości int za każdym razem, gdy są używane z binarnym operatorem arytmetycznym Java (np. dodawanie), nawet jeśli żaden z operandów nie jest int.

Po przeprowadzeniu promocji oraz jeśli operandy mają ten sam typ danych, wynikowa wartość będzie miała ten sam typ danych, co jej promowane operandy.

Przy tej okazji warto poznać operatory, które możemy połączyć z operatorem przypisania, a więc =

```
a = a + 1; // możemy też zapisać jako:
```

```
a += 1; //co oznacza weź wartość po lewej (a) i dodaj ją do prawej  
(1) a potem przypisz do a.
```

```
a = a * 3; //możemy zapisać jako:
```

```
a *= 3;
```

By dodać lub odjąć jedynkę od wartości, możemy też użyć **inkrementacji** lub **dekrementacji**.

```
a++; //oznacza dodaj jeden do a, czyli a = a + 1; jest to  
inkrementacja
```

```
a--; // oznacza odejmij 1 od a, czyli a = a - 1; Mamy tutaj  
dekrementację.
```

Operatory porównania (relacji, operatory relacyjne)

Zwracają typ bool, czyli true lub false. (Dla jasności, "bool" to taka forma skrócona w mowie, "typ bool", tak się mówi, ale pełna nazwa to boolean.)

== **równe** //nie używaj ich do stringów, tam używaj metody equals.

!= **różne od (nierówne)**

> **większe**

< **mniejsze**

>= większe lub równe
<= mniejsze lub równe

```
System.out.println(2 == 2.2); // false
System.out.println(5 > 4); // true
System.out.println(100 >= 100); // true
System.out.println(1 != 1); // false
```

Operatory logiczne

! negacja

Najprościej to odwrócenie wartości logicznej na przeciwną (mamy tylko true i false) i tak: !true zwróci false, a !false zwróci true.

```
System.out.println(!(2 > 3)); // 2>3 zwróci false, negacja
zwróci true, więc wydrukowane będzie true
System.out.println(!true); //false
int a = 10;
long b = 20;
System.out.println(!(a != b)); //! (true) czyli false
```

& koniunkcja (inaczej iloczyn logiczny)

&& koniunkcja zoptymalizowana (zwana "na skróty")

Wartości z obu stron muszą być prawdziwe (true), by zwrócone zostało true. W każdym innym warunku zwrócone zostanie false.

```
System.out.println(true & true); // true
System.out.println(true & false); // false
System.out.println(false & true); // false
System.out.println(false & false); // false
```

O co chodzi z tym “na skróty” przy operatorze &&. Jest to proces optymalizacji. Przy && oba warunki muszą być prawdziwe. Co oznacza, że jak pierwszy warunek byłby false, to sprawdzanie drugiej wartości nie jest konieczne. W zdecydowanej większości przypadków właśnie o to nam chodzi. Proces optymalizacji nie wpływa na zwracany wynik! Po prostu nie wykona kolejnego sprawdzenia, jeśli nie ma to już sensu.

Dlaczego więc zawsze nie używać ‘na skróty’? Bo czasami chcemy, by drugi warunek (który może być np. wykonaniem metody) był wykonany tak czy siak, bo coś robi. Przekonasz się w praktyce! Na teraz spokojnie, nie przejmuj się, jeśli jest to zbyt teoretyczne, to jedna z tych rzeczy, z którą z czasem się zaprzyjaźnisz.

```
System.out.println(true && true); // true
System.out.println(true && false); // false
System.out.println(false && true); // false - TUTAJ NIE
ZOSTANIE SPRAWDZONA DRUGA WARTOŚĆ
System.out.println(false && false); // false - TUTAJ NIE
ZOSTANIE SPRAWDZONA DRUGA WARTOŚĆ
```

Przykład z życia :)

By jeździć samochodem, trzeba mieć samochód i prawo jazdy (tak, wiem, że niektórzy nie mają a jeżdżą ;))

Czy może jeździć samochodem?

Ma prawo jazdy & ma samochód // sprawdza oba i zwraca true

Ma prawo jazdy && ma samochód // sprawdza oba i zwraca true

Ma prawo jazdy & nie ma samochód // sprawdza oba i zwraca false, bo drugi warunek jest niespełniony (drugi warunek to false)

Ma prawo jazdy && nie ma samochód // sprawdza oba i zwraca false, bo drugi warunek jest niespełniony (drugi warunek to false)

Nie ma prawa jazdy & ma samochód // sprawdza oba i zwraca false, bo pierwszy warunek jest niespełniony (drugi warunek to false)

Nie ma prawa jazdy && ma samochód // sprawdza w tym wypadku pierwszy i widzi false.
Ponieważ warunek nie może być już prawdziwy, nie sprawdza drugiej wartości i zwraca false (czyli optymalizuje).

Nie ma prawa jazdy & nie ma samochód // sprawdza oba i zwraca false
Nie ma prawa jazdy && nie ma samochód // sprawdza w tym wypadku pierwszy i widzi false.
Ponieważ warunek nie może być już prawdziwy, nie sprawdza drugiej wartości i zwraca false (czyli optymalizuje).

Pamiętaj że, pisząc '**sprawdza**', mam na myśli 'wykonanie kodu' (to może być jakaś metoda), która zwraca true lub false.

| **alternatywa (inaczej suma logiczna)**

|| **alternatywa zoptymalizowana (zwana "na skróty")**

Alternatywa oznacza, że tylko jedna z wartości, obojętnie która, musi zwracać true, by całość była prawdziwa. Oczywiście, warunek jest prawdziwy także, gdy obie wartości są prawdziwe.

```
System.out.println(true | true); // true - sprawdza oba
System.out.println(true | false); // true - sprawdza oba
System.out.println(false | true); // true - sprawdza oba
System.out.println(false | false); // false - sprawdza oba
```

```
System.out.println(true || true); // true - sprawdza pierwszy
tylko
System.out.println(true || false); // true - sprawdza tylko
pierwszy
System.out.println(false || true); // true - sprawdza oba, bo
pierwszy jest false dlatego by sprawdzić warunek musi sprawdzić
drugą wartość
System.out.println(false || false); // false - sprawdza obie
```

^ alternatywa rozłączna (inaczej wykluczająca czy xor)

Taki rzadko stosowany twór (operator). Definiuje się go jako albo jedno, albo drugie, czyli jeśli jedna wartość jest prawdziwa, a druga nie (nieważne, z której strony), to zwracane jest true. Gdy obie wartości są nieprawdziwe lub obie są prawdziwe, zwracane jest false. Może zastanawiasz się, czemu nie ma ^^ alternatywy rozłącznej 'na skróty'. To wynika to z tego, że zawsze trzeba sprawdzić oba, nie ma więc tu co optymalizować.

```
System.out.println(true ^ true); //false
System.out.println(true ^ false); //true
System.out.println(false ^ true); //true
System.out.println(false ^ false); //false
```

Operatory bitowe

Tylko na typach numerycznych (tylko całkowitych). Daje on dostęp do bitów, które tworzą daną liczbę.

Nie będą nam potrzebne, o czym Karol mówi po kilkunastu minutach tłumaczenia, czym są i jak działają ;)