Wstęp do Java

Spis treści

Zapiski uczestnika Bootcampu Zajavka.pl w 12 tygodni by Bartek Borowczyk aka Samuraj Programowania. Dopiski na zielono od Karola Rogowskiego.

Wprowadzenie

Wprowadzenie było przyjemne ☺. Ten Bootcamp to był dobry wybór ☺.

Początkowe informacje

Cechy Javy

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 tworzony na podstawie klas. Obiekty składają się z danych i metod do obsługi tych danych. Tym zagadnieniem w Bootcampie poświęcimy sporo czasu, gdy przejdziemy do programowania obiektowego.

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++ (obiektowość). 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 rozwijanym. 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.

Język kompilowany a interpretowany

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)

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.

Javę 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 kompilatora w konsoli,

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

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

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

Specyfikacja Javy

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 Javę 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. Do tematu "za darmo" wrócimy w przyszłości, na razie się tym nie przejmuj.

JRE

JRE - Java Runtime Environment - środowisko uruchomieniowe Javy. Odpowiedzialne za uruchomienie programu. Zawiera zarówno maszynę wirtualną Javy, jak i zestaw użytecznych klas (klas podstawowych).

JVM

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.

Pierwsze komendy w terminalu

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
```

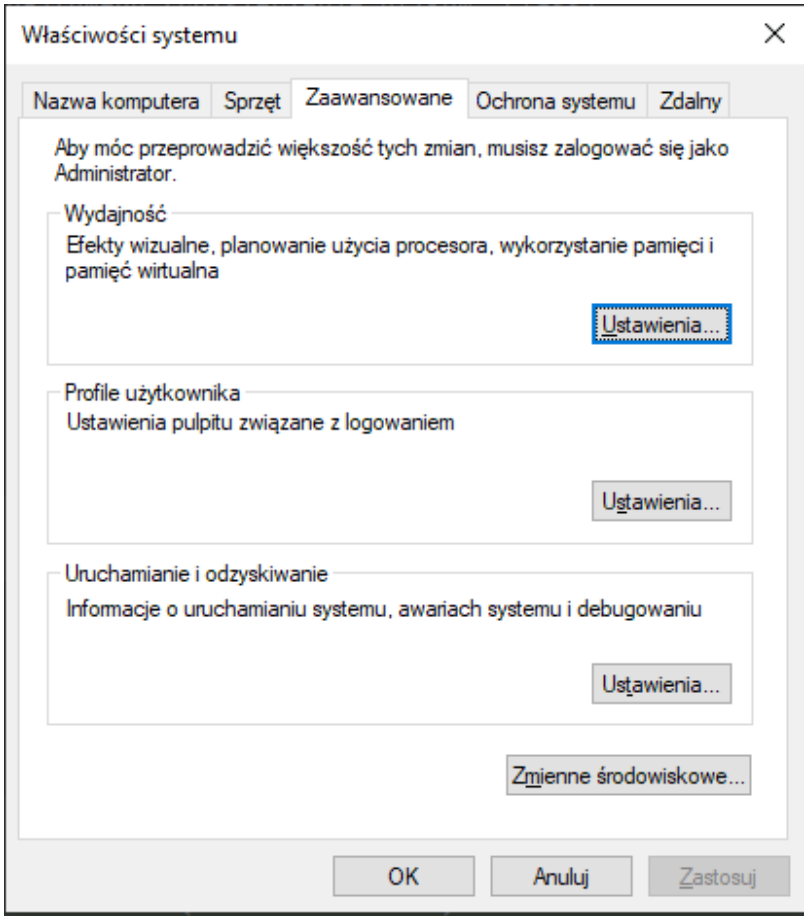
Interpreter:

```
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..."]



Obraz 1. Okienko zmiennych środowiskowych na Windows

Umieszczamy ścieżkę do miejsca, w którym zainstalowaliśmy (rozpakowaliśmy) Javę. Jeśli brakuje ścieżki u Ciebie, użyj przycisku "Nowy" i wprowadź ścieżkę do folderu bin.

Intellij IDEA

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.

Kod źródłowy

Omówiliśmy już dotychczas:

- 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` oraz `FIXME` - Karol ich nie lubi. Ale jest i niektórzy używają. Nie lubię ich, bo potem ludzie o nich zapominają i często zostaje dużo zapomnianych komentarzy w kodzie, które utrudniają pracę.

Klasa Example, 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 klasę `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.

Klasa Main

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Mam zajavkę 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

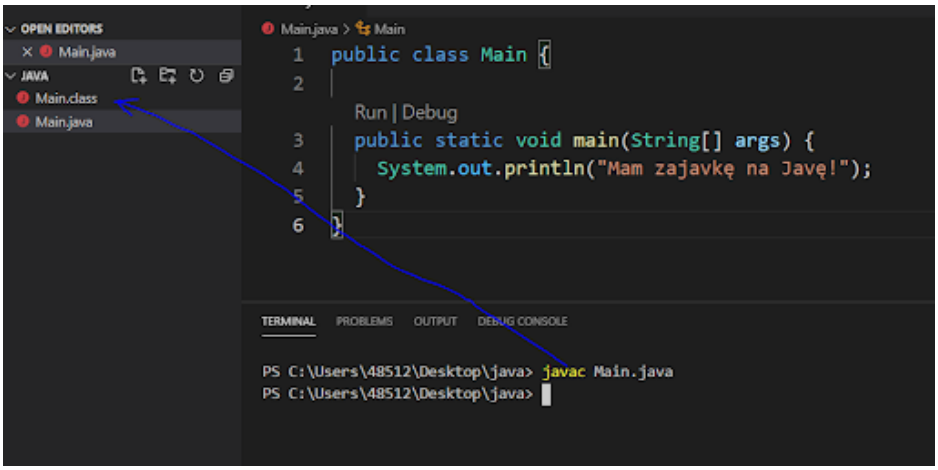
```
C:\jdk-11\bin\java.exe "-javaagent:C:\ProgramFiles\..."
Mam zajavkę na Javę!
Process finished with exit code 0
```

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

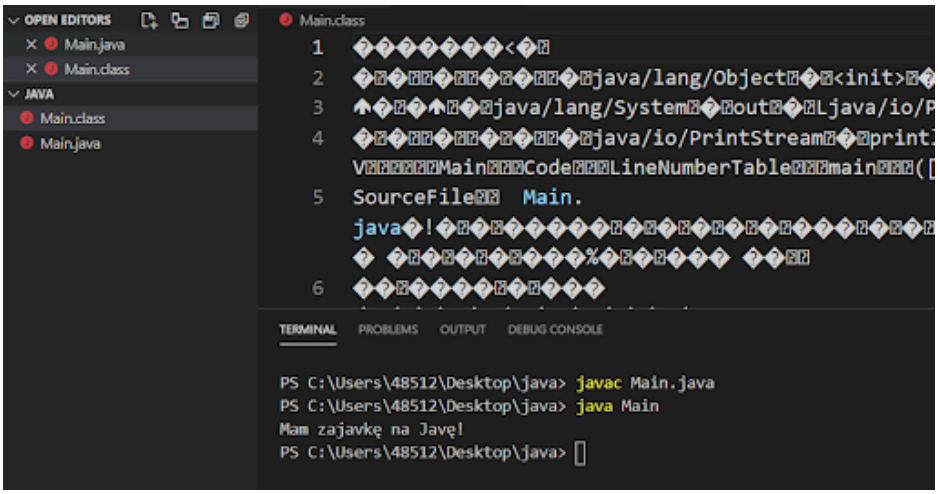
Przypomnę tylko, że ten sam efekt uzyskamy, gdy wpiszemy w wierszu poleceń (oczywiście, we właściwej ścieżce prowadzącej do danego pliku):

```
> 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 zaznaczyłem 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.



Obraz 2. Przykład kodu źródłowego



Obraz 3. Przykład kodu bajtowego

Powróćmy ponownie do tego kodu:

Klasa Main

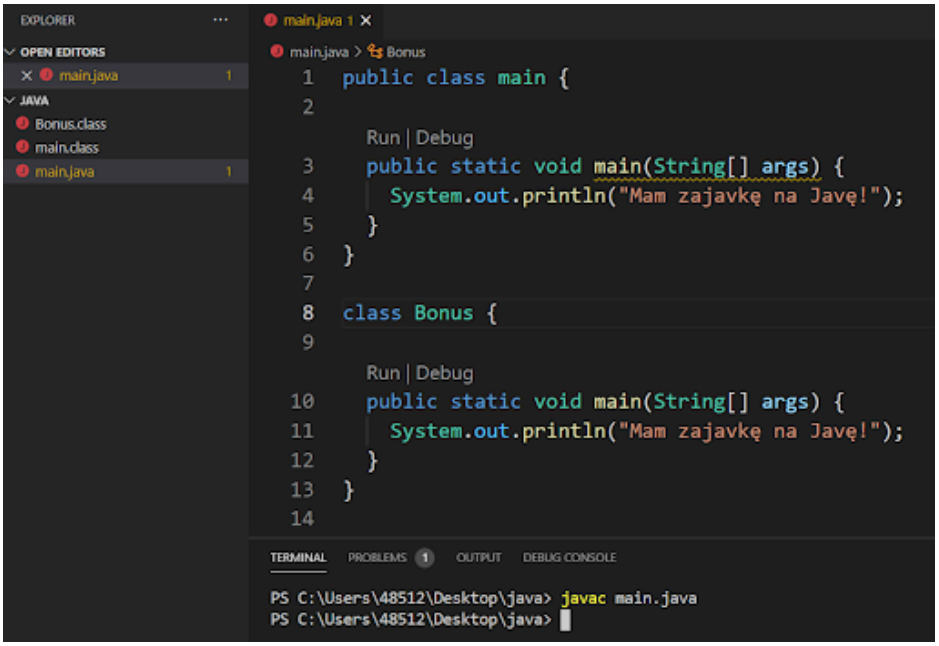
```
public class Main {

    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).



Obraz 4. Pliki otrzymane po kompilacji

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

Słowa kluczowe, które dotychczas poznaliśmy

public

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 później).

class

class - słowo kluczowe, które służy do utworzenia klasy. Klasa jest podstawowym elementem każdego popularnego języka programowania zorientowanego obiektowo (będziemy się o tym uczyć niedługo). Java wręcz wymusza na nasz 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 ☺.

Nawiasy klamrowe

{} — 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.

Podstawowa składnia klas

```
class Identyfikator {}
```

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 (znowu)

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 korzysta się z danych metod.

static

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 zmierzPokoj()` może być metodą, która pozwala zmierzyć wymiary pokoju na podstawie projektu (klasy), bo obiektu jeszcze nie ma. Gdyby `zmierzPokoj()` 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

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 zwracaLiczbę 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

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 zajavkę 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.

Programowanie obiektowe

Do dokładnego omówienia tematu przejdziemy w dalszej części Bootcampu. Jak dotychczas jednak 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 nauczymy się programowania zorientowanego obiektowo.

Jakie pojęcia nas interesują na ten moment:

class

`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() {
    // o tutaj
}
```

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 później ☺.

object

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

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

method

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

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.

naming

nazewnictwo - W Javie przyjęła się konwencja, by używać nazewnictwa w oparciu o tzw. notację wielbłądzią (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

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 (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

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.