**0.4. Zaczynamy kodować w Javie!**

**Typy danych**

W ramach kodu programistycznego głównie operujemy na danych. O jakie dane dokładnie chodzi? Wyobraź sobie, że użytkownik wypełnia formularz internetowy, a w nim pola: "Imię i nazwisko", "Data urodzenia", "Zarobki do dwóch miejsc po przecinku".

Musimy pilnować, aby użytkownik wpisywał w pola odpowiednie typy danych. To znaczy, że jeśli w pole daty urodzenia użytkownik wpisze "24 lata temu", to ciężko będzie nam np. policzyć średnią wieku wszystkich, którzy wypełnili formularz (po zapisaniu do bazy danych na serwerze). Tak samo jest w Excelu — aby można było zsumować pewne pola, muszą one mieć taki sam typ danych.

Podsumowując, jeśli chcesz dokonać pewnych obliczeń w Javie, to zawsze musisz określić typ wartości!

W kodzie:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

System.out.println("Rozpoczynamy naukę w Javie!");

}

}

polecenie System.out.println (w nawiasie) może przyjąć różnego rodzaju wartości do wyświetlenia, np. tekst, liczbę czy typy logiczne (w tym przypadku tekst: "Rozpoczynamy naukę w Javie"). Dlatego omówimy teraz, jakie mamy w Javie typy danych:

* **Tekst (String)** — w Javie, aby napisać dowolny tekst, stosujemy cudzysłów, np. "Hello, my name is Maciej". Będzie on traktowany jako pewien ciąg znaków, który nie ma wartości matematycznej oprócz liczby znaków w tym ciągu. W tym przypadku powyższy tekst lub imię i nazwisko użytkownika wpisanego w formularz www (z wcześniejszego przykładu), będzie trzymane jako typ danych *String*.
* **Liczby (int lub double)** — jeżeli chcemy wyświetlić liczbę przy użyciu naszego nowo poznanego polecenia, wystarczy wpisać po prostu: 9 (bez cudzysłowu), natomiast liczby zmiennoprzecinkowe należy oddzielać kropką: 3.14.
* **Typy logiczne (boolean)** — w Javie wyrażane są jako true i false, czyli prawda i fałsz. Jeżeli chcielibyśmy wyświetlić jeden z nich, wystarczy, że wpiszemy je w nawiasach poznanego polecenia bez apostrofów.
* **Litera (char)** — inaczej reprezentacja pojedynczej litery, np. 'G'. Ona zawsze występuje pomiędzy apostrofami. Jedną z różnic między char a String jest fakt, że char zawsze zawiera jedną literę, natomiast String może zawierać 0 bądź więcej liter.

**Zadanie — pierwsza kompilacja**

W ramach <https://kodilla.com/pl/projects> (edytor wewnętrzny Kodilli) napisz kod, który wyświetli w 3 osobnych liniach: Twoje imię, Twój wiek oraz pierwszą literę nazwiska. Wykorzystaj do tego polecenie System.out.println().

Zapisz a następnie naciśnij Uruchom (kod otrzyma unikalny URL — link), wykona się kompilacja kodu oraz wyświetlenie wyniku w konsoli (stdout — jako konsola).

Jeśli kod będzie niepoprawny, to zwrócony zostanie błąd w sekcji "compilation info".

**Zmienne**

Może z matematyki pamiętasz zmienne. Co one nam dają? Pod zmienną możemy "schować" pewną wartość i użyć w kilku miejscach krótszej nazwy zmiennej. Na przykład możemy ustalić, że X = "bardzo długi tekst" i tego X użyć w wielu innych miejscach, a jego zawartość zmieniać tylko w jednym miejscu.

W kodzie Javy deklaruje się zmienne, nadając im nazwę oraz definiując, jakiego są typu danych. W poniższym kodzie mamy zadeklarowanych 5 zmiennych, a każda ma inny typ danych, które poznaliśmy wcześniej.

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

String myName = "Janusz";

int myAge = 22;

double height = 1.78;

char carClass = 'A';

boolean isGoodProgrammer = true;

}

}

Nazwa zmiennej zawsze musi się zaczynać małą literą. Zasady nadawania nazw:

* przyjmuje się zawsze angielskojęzyczne nazwy;
* jeśli nazwa składa się z 2, 3 słów, to oprócz pierwszego słowa, kolejne pisze się wielką literą, aby łatwiej było to odczytać (tzw. camelCase);
* jeśli używamy zmiennej ponownie w innym miejscu kodu, to musimy pilnować określonych wielkości liter;
* to Ty nadajesz nazwę taką, która będzie zrozumiała w przyszłości dla Ciebie i innych. Używanie w projektach programistycznych nazw typu X, Y... to droga do katastrofy. Potem każdy będzie się pytać, do czego jest ta zmienna. Dobre (logiczne) nazwy zmiennych to oznaka profesjonalizmu.

Jak piszemy deklarację zmiennej?

typ nazwaZmiennej = wartość;

**PAMIĘTAJ:** zawsze średnik na końcu!

**Zadanie — typy zmiennych**

W poniższym kodzie zadeklarowano 5 zmiennych:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

String myName = "Janusz";

int myAge = 22;

double height = 1.78;

char carClass = 'A';

boolean isGoodProgrammer = true;

}

}

1. Wykorzystując przykład, stwórz własne zmienne i zapisz w nich wartości zgodne z typami.
2. Użyj polecenia System.out.println() i wewnątrz okrągłych nawiasów wpisz nazwę zmiennej.
3. Użyj przycisku RUN i ciesz się wynikiem :)

**Instrukcje warunkowe**

Warto teraz zastanowić się, co z takimi zmiennymi można więcej zrobić. Poznajmy instrukcje warunkowe. Polegają one na tym, że jeśli wartość danej zmiennej w kodzie będzie dokładnie = X, to ma wydarzyć się jedno, a jeśli wartość inna niż X, to ma się wydarzyć co innego.

Przykład? Na portalu społecznościowym ktoś zakłada konto i wybiera w formularzu płeć. Taki serwis internetowy można zaprogramować dodając warunek, że:

* jeśli wybierze "Kobieta", to w tle strony wyświetl kwiaty (grafika),
* jeśli wybierze "Mężczyzna", to w tle strony wyświetl samochody.

Tak wygląda przykładowa instrukcja warunkowa: <https://kodilla.com/pl/project-java/38803>

Ale rozwińmy ją. Przypominasz sobie niedawno poznany typ logiczny — boolean? On sprawdza się tutaj idealnie. Na pewno nasuwa Ci się pytanie: a co jeśli, ktoś nie jest dobrym programistą, co wtedy?!

Na szczęście ten przypadek też możemy sprawnie obsłużyć za pomocą "else":

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

boolean isGoodProgrammer = true;

if(isGoodProgrammer) {

} else {

}

}

}

Teraz rozbierzemy instrukcję na części pierwsze i pokażemy raz jeszcze, co każdy element oznacza:

* Deklaracja zmiennej — zadeklarowaliśmy zmienną o nazwie isGoodProgrammer o typie boolean. Następnie zapisaliśmy w zmiennej isGoodProgrammer wartość true.
* Wywołanie instrukcji if() — wywołanie tej instrukcji daje nam możliwość wpisania warunku w okrągłe nawiasy. Domyślnie instrukcja uzyskuje wartość true, aby wykonać polecenie zawarte w klamrach. W naszym wypadku zmienna isGoodProgrammer jest true, więc to, co zostanie napisane w klamrach pod instrukcją if(), zostanie wykonane.
* Wywołanie instrukcji else — instrukcja obejmuje przypadek, w którym warunek zawarty w instrukcji if() nie będzie spełniać naszych oczekiwań. W naszym przykładzie: wtedy i tylko wtedy kiedy isGoodProgrammer nie będzie true. Wtedy wykona się kod zawarty w klamrach pod instrukcją else.

Poznaliśmy już polecenie System.out.println(); — spróbujmy je teraz wykorzystać!

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

boolean isGoodProgrammer = true;

if(isGoodProgrammer) {

System.out.println("Java, java, java!!!");

} else {

System.out.println("Musze się jeszcze pouczyć!");

}

}

}

[Sprawdź wynik powyższego programu.](https://kodilla.com/pl/project-java/38806)

W powyższym przypadku, jeżeli zmienna isGoodProgrammer, będzie równa true, w konsoli pojawi się tekst: "Java, java, java!". W przypadku, gdy do zmiennej isGoodProgrammer została przypisana wartość false, w konsoli ukaże się informacja: "Muszę się jeszcze pouczyć!".

Zastanawiasz się pewnie, w jaki sposób użytkownik wybiera, czy programista jest dobry czy zły. Jest to bardzo dobre spostrzeżenie! Jednakże na ten moment nie skupiamy się na wprowadzaniu danych do programu przez użytkowników — to my je deklarujemy i na tę chwilę będziemy się tego trzymać, ponieważ chcemy mieć pełną kontrolę nad programem,który piszemy.

**Zadanie — instrukcje warunkowe**

Wykorzystując poznaną wiedzę na temat zmiennych i instrukcji if() i else, stwórz swój własny warunek wykorzystując do tego zadania edytor Kodilli (<https://kodilla.com/pl/projects>).

Wewnątrz metody stwórz swoją własną instrukcję, np. isGoodWeather = false; i wyświetl z wykorzystaniem polecenia "System.out.println();" informację dla dobrej i złej pogody.

**Podstawowe operacje na zmiennych**

Wyobraź sobie kalkulator. On też na samym początku prosi Cię o liczby, które potem dla Ciebie policzy. Tak samo Ty tworząc oprogramowanie musisz znać dane wejściowe do wykonania konkretnych zachowań. Można to także zobrazować na zasadzie formularza WWW, który ma wyliczyć Twoją zdolność kredytową. Użytkownik wprowadza w formularzu liczby, a program je przelicza.

Poznaliśmy już podstawowe typy danych i instrukcję "if". Nadszedł czas, żeby pobawić się trochę ze zmiennymi i wkroczyć jedną nogą w codzienność programisty, czyli dokonać operacji na informacjach.

Za chwilę nauczymy się wykonywać podstawowe operacje matematyczne i **konkatenację**, czyli łączyć ze sobą ciągi znaków. Brzmi strasznie? Nie martw się, gwarantujemy, że zaraz najdzie Cię ochota na policzenie jakiejś macierzy :)

**Dodawanie i odejmowanie**

Dodawać i odejmować można liczby, a poznaliśmy już 2 typy liczb w Javie: **double (liczby zmiennoprzecinkowe)** i **int (liczby całkowite)**.

Na szczęście w Javie dodawanie i odejmowanie jest takie samo, jak w rzeczywistości! Zacznijmy od dodawania liczb całkowitych:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

int numberOne = 30;

int numberTwo = 50;

int result = numberOne + numberTwo;

System.out.println(result);

}

}

Jak widzisz, zadeklarowaliśmy dwie zmienne o typie int, następnie korzystając z operatora dodawania "+" dodaliśmy je do siebie i zapisaliśmy do zmiennej wynikowej result, również o typie int. Po obliczeniu wykonaliśmy znane nam polecenie wyświetlania i przy jego pomocy w konsoli pojawił się wynik operacji dodawania. Zapisanie wyniku do innej zmiennej jest dobrą praktyką kiedy wiemy, że będziemy korzystać z niej w dalszej części programu np. po dodaniu do siebie 2 liczb będziemy chcieli tę liczbę przemnożyć jeszcze przez 3.

Analogicznie możemy wykonać operacje odejmowania przy użyciu operatora "-":

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

int numberOne = 30;

int numberTwo = 50;

int result = numberOne - numberTwo;

System.out.println(result);

}

}

Wszystko przebiega sprawnie, ale co jeśli będziemy chcieli dodać 2 różne typy liczb? Czy będzie to działać? Spróbujmy:

<https://kodilla.com/pl/project-java/38807>

Na powyższym przykładzie wyświetla nam się błąd związany z operacją na niekompatybilnych typach danych, ponieważ operacje mogą być wykonywane tylko na tych samych typach. W tym przypadku tylko na liczbach całkowitych (int). Dzieje się tak dlatego, że wynik odejmowania: 30 - 5.5 = 24.5 jest liczbą zmiennoprzecinkową (double) Poprawmy to:

<https://kodilla.com/pl/project-java/38808>

Udało się! Deklaracja wyniku jako liczby zmiennoprzecinkowej pozwoliła nam na wykonanie działania.

**Mnożenie i dzielenie**

Operatorem mnożenia jest symbol \*, natomiast dzielenie wykonujemy przy użyciu / . Popatrzmy na kod:

Mnożenie liczb całkowitych:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

int numberOne = 3;

int numberTwo = 5;

int result = numberOne \* numberTwo;

System.out.println(result);

}

}

Mnożenie liczb zmiennoprzecinkowych:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

double numberOne = 3.5;

double numberTwo = 2.5;

double result = numberOne \* numberTwo;

System.out.println(result);

}

}

Jak widzisz, jest to bardzo proste! Zobaczmy jak wykonać dzielenie w Javie:

Dzielenie liczb całkowitych

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

int numberOne = 15;

int numberTwo = 5;

int result = numberOne / numberTwo;

System.out.println(result);

}

}

Dzielenie liczb zmiennoprzecinkowych:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

double numberOne = 15.90;

double numberTwo = 5.30;

double result = numberOne / numberTwo;

System.out.println(result);

}

}

**Modulo, czyli otrzymanie reszty z dzielenia**

W trakcie Twoich przygód z Javą może przydać się operator o tajemniczej nazwie modulo, czyli znak %. Operator ten pozwala na otrzymanie reszty z dzielenia. Spójrzmy na poniższy przykład:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

int numberOne = 25;

int numberTwo = 6;

int modulo = numberOne % numberTwo;

System.out.println(modulo);

}

}

Czym jest reszta z dzielenia? Zobaczmy na przykład — ile razy liczba 5 zmieści się w 25? Dokładnie — 5 razy. Dla tej operacji modulo zwróci wynik równy 0, co oznacza, że w wypadku działania 25 / 5 nie otrzymamy żadnej dodatkowej liczby po przecinku.

Mówimy wówczas, że liczba a daje resztę r przy dzieleniu przez b. W szczególności, jeśli r=0, to mówimy, że liczba a jest podzielna przez b.

**Konkatenacja**

Konkatenacja jest połączeniem ze sobą zmiennych typu String, np. "Gosia ma" i "psa o imieniu Jerzy" za pomocą operatora dodawania, czyli +. Jak to zrobić? Załóżmy, że chcemy przedstawić co ma Gosia, ale w osobnych zdaniach, np. Gosia ma psa, Gosia ma 3 koty, Gosia ma brata o imieniu Stefan. Przedstawmy to w formie kodu:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

String stringOne = "Gosia ma";

String stringTwo = "psa";

String stringThree = "3 koty";

String stringFour = "brata o imieniu Stefan";

}

}

Zadeklarowaliśmy cztery różne teksty i zapisaliśmy je do zmiennych. Wykorzystując konkatenacje i polecenie System.out.println() chce wyświetlić trzy zdania. Spróbujmy!

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

String stringOne = "Gosia ma";

String stringTwo = "psa";

String stringThree = "3 koty";

String stringFour = "brata o imieniu Stefan";

System.out.println(stringOne + stringTwo);

System.out.println(stringOne + stringThree);

System.out.println(stringOne + stringFour);

}

}

W wyniku kompilacji otrzymujemy wynik:

<https://kodilla.com/pl/project-java/38809>

Wygląda na to, że brakuje nam spacji... jak ją dodać?

<https://kodilla.com/pl/project-java/38810>

W celu dodania spacji zadeklarowaliśmy nową zmienną, którą nazwaliśmy "space" i dodaliśmy w niej odstęp. Następnie przy użyciu operatorów dodawania, czyli **konkatenacji**, połączyliśmy wszystkie teksty typu String.

**Równości i operatory relacyjne**

Równości i operatory relacyjne przydadzą się nam w celu tworzenia jeszcze bardziej zaawansowanych instrukcji warunkowych. Na przykład na stronie internetowej serwis pyta: ile masz lat? Jeżeli wpiszesz w pole 22, to serwis może zastosować takie zachowanie: jeśli wiek użytkownika jest większy niż 18 lat, to można pokazać na stronie reklamę dotyczącą alkoholu.

Pamiętasz instrukcję IF? Teraz ją wykorzystamy w całości! Poniżej znajdziesz najbardziej znane i popularne operatory przydatne w instrukcji warunkowej IF:

== równa się

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

String name = "Krzysiek";

if(name == "Krzysiek") {

System.out.println("On ma na imie " + name + " to pewne!");

}

}

}

!= różny od

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

String name = "Janek";

if(name != "Krzysiek") {

System.out.println("On na pewno nie ma na imię Krzysiek, tylko " + name + " to pewne!!!");

}

}

}

> większy od

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

int age = 11;

if(age > 10) {

System.out.println("Janek na pewno ma więcej niż 10 lat");

}

}

}

>= większy lub równy

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

int age = 10;

if(age >= 10) {

System.out.println("Janek na pewno ma 10 lat lub więcej");

}

}

}

< mniejszy od

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

int age = 9;

if(age < 10) {

System.out.println("Janek ma na pewno mniej niż 10 lat");

}

}

}

<= mniejszy od lub równy

import java.util.\*;

import java.lang.\*;

import java.io.\*;

/\* Name of the class has to be "Main" only if the class is public. \*/

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

int age = 10;

if(age <= 10) {

System.out.println("Janek na pewno ma mniej lub dokładnie 10 lat");

}

}

}

**Zadanie — operacje na zmiennych**

Za pomocą edytora Kodilla stwórz kod, który:

1. będzie sprawdzał, czy liczba X jest podzielna przez liczbę 7 bez reszty;
2. jeżeli tak, to w konsoli wyświetl zdanie "Liczba 7 jest dzielnikiem liczby X :)" (zamiast X ma się wyświetlać wartość zmiennej);
3. jeżeli nie, to w konsoli ma się pojawić zdanie "Liczba 7 NIE jest dzielnikiem liczby X :(" (tu też zamiast X ma się wyświetlać wartość zmiennej).

Udało się! :)

**Tablice jednowymiarowe**

Tablica w programowaniu to struktura stworzona po to, aby przechowywać większą ilość informacji. Możesz sobie to wyobrazić jako miejsce na przechowywanie wielu zmiennych, np. imion koleżanek/kolegów.

Aby stworzyć tablicę, należy zastanowić się nad:

* typem danych, jakie będzie przechowywać;
* liczbą elementów: czy to jest 2, czy może 100 wartości?

Stwórzmy więc tablicę przechowującą imiona 5 najlepszych koleżanek i kolegów. Wiemy, że imiona są tekstem, czyli na pewno są typu **String**, a rozmiar tablicy to **5**.

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

String names[] = new String[5];

}

}

Zadeklarowaliśmy właśnie 5-elementową tablicę przechowującą dane typu **String** i zapisaliśmy ją do zmiennej names o typie **String**. Dobrym pytaniem będzie, co robią kwadratowe nawiasy przy zmiennej names. Otóż nawiasy te tłumaczą kompilatorowi, że zmienna names jest tablicą.

Nasza 5-elementowa tablica na ten moment jest pusta, ale w prosty sposób możemy dodać do niej nowe elementy za pomocą kwadratowych nawiasów:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

String names[] = new String[5];

names[0] = "Ania";

names[1] = "Krzysiek";

names[2] = "Paula";

names[3] = "Janusz";

names[4] = "Wiesław";

}

}

Dziwna sprawa! Mamy 5 imion, ale zaczęliśmy dodawać elementy od 0. Wynika to z tego, że każda tablica w Javie zapisuje dane od indeksu 0 i należy o tym zawsze pamiętać!

Wiemy już, jak stworzyć tablicę i jak przypisać do niej wartości. Teraz zobaczmy, jak wyciągnąć konkretną wartość i zapisać ją do zmiennej:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class Kodilla

{

public static void main (String[] args) throws java.lang.Exception

{

String names[] = new String[5];

names[0] = "Ania";

names[1] = "Krzysiek";

names[2] = "Paula";

names[3] = "Janusz";

names[4] = "Wiesław";

String name = names[2];

}

}

Powyższy przykład pokazuje, jak wyciągnąć konkretną wartość z tablicy i zapisać ją do zmiennej za pomocą kwadratowych nawiasów i numeru indeksu, dla którego zapisaliśmy imię.

**Zadanie — tablica ulubionych filmów**

Wykorzystując zdobytą wiedzę o tablicy, stwórz tablicę zawierającą tytuły Twoich dziesięciu ulubionych filmów. Skorzystaj z edytora wewnętrznego Kodilla.

Jeśli masz wątpliwości do powyższego materiału, to - zanim zatwierdzisz - zapytaj na czacie :)

Zapoznałe(a)m się!

**0.5. Czym jest obiektowość?**

Pisząc kod można to robić "ciągiem", wypisując w kolejnych liniach instrukcje, deklaracje zmiennych, warunki i inne elementy — takie programowanie nazywamy *programowanie strukturalnym*. Jednak na dłuższą metę to może być bardzo nieczytelne. Dlatego oprócz umieszczania kodu w wielu oddzielnych plikach, dodatkowo w ramach jednego pliku kod dzieli się na pewne kawałki, które mają określone przeznaczenie (funkcję).

Dzięki temu cały kod jest lepiej zorganizowany (czytelny), a przeznaczenie danych bloków kodu jest jasno określone poprzez ich rozdzielenie.

Przy podejściu obiektowym, wszystko jest obiektem. Każdy element kodu jest traktowany jako obiekt posiadający swoje atrybuty i zachowania. Dzięki temu zyskujemy uporządkowaną strukturę kodu naszej aplikacji.

**Klasa**

W Javie, tym wydzielonym kawałkiem kodu jest najczęściej klasa, która w kodzie, w najprostszej postaci, wygląda tak: class nazwa\_klasy {...}

Można to porównać do wzoru umowy na serwerze firmowym. Zamiast tworzyć tę umowę od początku wraz z każdym nowym klientem, używamy "matrycy" (wzoru) i wypełniamy go danymi specyficznymi dla danego klienta.

A teraz zmierzamy do sedna. Klasa to właśnie pewna matryca do tworzenia **obiektów** na bazie tego wzorca. No dobrze, ale czym jest obiekt?

Obiekt to wystąpienie danej klasy. Możemy utworzyć wiele obiektów na bazie jednej klasy — klasa określa pewną konstrukcję obiektu, a sam obiekt jest bytem, który ma pewne specyficzne dane, odróżniające go od innych obiektów.

Aby łatwiej to zrozumieć, wyobraź sobie, że budujemy pewien serwis internetowy. W kodzie serwisu jest klasa, która odpowiada za utworzenie nowego użytkownika (po tym, jak ten dokonał rejestracji). Klasa odpowiadająca za reprezentację użytkownika musi zawierać w sobie e-mail i imię nowego użytkownika.

Każda klasa powinna odpowiadać tylko za jedną rzecz, tzn. powinna realizować tylko takie operacje, które mają ścisłe połączenie, wpisują się w jej przeznaczenie. Jest to tzw. **pojedyncza odpowiedzialność klasy**. Np. tworząc klasę DziennikOcen, możemy umieścić w niej takie operacje, jak wpisanie oceny, czy poprawienie oceny. Załóżmy, że zbliża się wywiadówka i chcemy wydrukować kartki z ocenami. Mogłoby się wydawać, że metoda do drukowania takich raportów pasowałaby idealnie do klasy Dziennika, ale w tym celu powinniśmy stworzyć dodatkową klasę do generowania raportów.

Na bazie tej "matrycy" są tworzeni użytkownicy, z których każdy z osobna ("Krzysiek", "Zosia") są właśnie oddzielnymi obiektami, które utworzyliśmy za pomocą tejże klasy.

Tworzenie szablonu/matrycy dla obiektów w postaci klasy i wykorzystywanie ich w dalszej części tworzenia programu jest nazywane **podejściem obiektowym**. Java funkcjonuje w ramach takiego podejścia jako **język obiektowy** (Object Oriented Programming — OOP).

W założeniu podejście obiektowe ma pozwolić lepiej zrozumieć kod programistyczny, który nawiązuje wtedy do rzeczywistości, tak jak w podanych wcześniej przykładach. Podstawą udanej przygody z Javą, jest uruchomienie "myślenia obiektowego".

**Więcej o klasie**

Wiemy już, że obiekt to pewien byt. Jest nim też np. pilot do telewizora. Konstrukcję takiego pilota możemy przedstawić za pomocą klasy (możemy stworzyć wiele pilotów na bazie tej klasy, np. z różnymi numerami seryjnymi). Pilot ma określone funkcje, np. włączenie i wyłączenie telewizora. Takie funkcje można zobrazować w ramach klasy za pomocą tzw. **METOD**, których używamy w Javie. Przedstawmy to za pomocą kodu:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class Pilot

{

public void turnOn() {

}

public void turnOff() {

}

}

Klasa pilot zawiera 2 metody (turnOn, turnOff), które są puste i nic nie robią. W środku tych metod możemy zaimplementować (stworzyć) kod, który będzie odpowiadał za włączenie i wyłączenie telewizora.

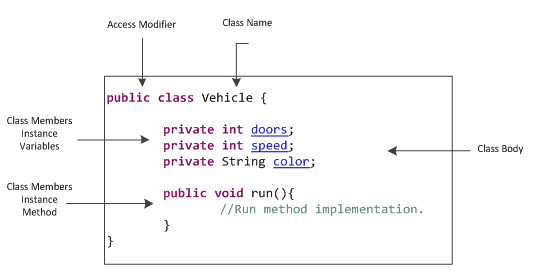
Jak widzisz, kodem Javy można przedstawić rzeczy, które nas otaczają w formie obiektów (konkretnych ich wystąpień, np. jednego z wielu podobnych pilotów). Dlatego spróbuj przejść się po domu/biurze i zastanowić się, jakie obiekty można stworzyć, i jakie metody im przypisać.

**Zadanie — metody**

Za pomocą [edytora Kodilli](https://kodilla.com/pl/projects), stwórz nową klasę Samolot. Zastanów się jakie czynności (metody) mogłaby wykonywać Twoja klasa i dodaj je.

**Budowa klasy i jej składowe**

Spójrzmy na budowę przykładowej klasy w Javie.

[](https://kodilla.com/static/bootcamp-java/java-0_04.png)

Omówmy zatem, jak zbudowana jest klasa według powyższego wzorca:

* **Informacja o dostępności klasy** (access modifier) w obrębie aplikacji, jeżeli jej nie ma, to oznacza, że klasa jest publiczna (public), dlatego często przed nazwami klas nie daje się informacji o dostępności. Ta publiczna dostępność mówi o tym, że można stworzyć obiekt na podstawie tej klasy w całym projekcie.
* **Nazwa klasy** (Class Name) — unikalna nazwa (nie może się powtarzać jako nazwa innych klas) informująca, co konkretna klasa reprezentuje, np. Vehicle, User, Book... to od Ciebie zależy jaką nazwę ustalisz. Warto, aby nazwa klasy była zawsze po angielsku i odzwierciedlała obiekty, które będą tworzone w przyszłości.
* **Zawartość klasy** (Class Body) — jest to nic innego, jak kod (zawartość) klasy znajdujący się między "wąsami" {...}. Znajdziemy tam zarówno atrybuty klasy, jak i jej metody.
* **Atrybuty** (Class Members Instance Variables) — nazywane też polami. Określają cechy klasy (lub tworzonego z niej obiektu), np. klasa Vehicle (samochód) może mieć takie cechy jak liczba drzwi, maksymalna prędkość czy kolor pojazdu.
* **Metody** (Class Members Instance Method) — wydzielone i nazwane fragmenty kodu, które realizują określone zadania napisane przez programistę. Klasa może posiadać wiele metod, które odpowiadają za wykonywanie poszczególnych akcji.

**Metoda**

Metoda jest niczym innym, jak zbiorem możliwych do wykonania akcji przez obiekt stworzony na podstawie klasy, która te metody zawiera. Te akcje to zbiór instrukcji w danej metodzie, które są uruchamiane, kiedy ta metoda zostanie wywołana, np. w innym miejscu w kodzie.

Wróćmy do przykładu pilota do telewizora. Metoda zajmuje się tym, co stanie się po wciśnięciu przycisku na pilocie. Jeżeli wciśniesz przycisk "Volume up" to dany pilot (obiekt) skorzysta z metody (przykładowo nazwanej volumeUp) po to, by podgłośnić telewizor, za pomocą zestawu instrukcji z tej metody.

Metody tworzymy w ramach klasy, aby potem wszystkie obiekty stworzone na jej podstawie mogły z nich korzystać. Trzymając się przykładu pilota: w klasie pilota tworzymy metody odpowiadające za jego funkcje. Wtedy każdy pilot (obiekt), jaki stworzymy, będzie posiadał te funkcje.

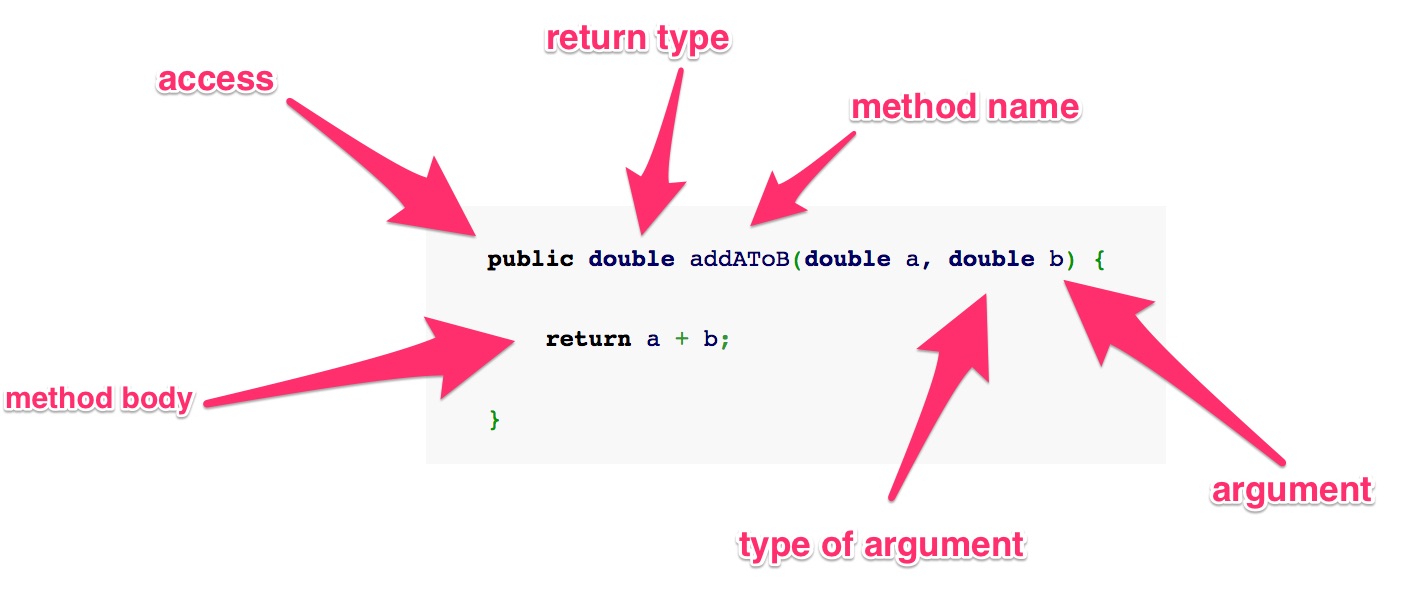
Więcej o metodach:

* nazwa metody zawsze powinna definiować, co dokładnie będzie wykonywać, np. printMyName, showTheBestMovies, volumeUp (nazwy metod zaczynamy małą literą, a kolejne słowa w niej — wielką literą);
* jedna metoda wykonuje jedno konkretne zadanie (może na nie się składać kilka instrukcji w ciele metody), które definiuje jej przeznaczenie.

Wywołując metodę można przekazać do niej tzw. argumenty (jeżeli je przyjmuje). Są to dane wejściowe do metody, które zostaną wykorzystane do wykonania instrukcji utworzonych wewnątrz niej. Np. jeśli w metodzie znajduje się instrukcja (wzór) dodawania dwóch liczb A i B, to mogą być one tam przekazywane przez argumenty metody.

Efektem wywołania (uruchomienia) metody jest wynik jej działania. Może to być zmiana kanału w telewizorze (za pomocą naszego pilota) lub wynik jakichś obliczeń. W tym drugim przypadku, kiedy dostaniemy wynik działania, mówimy, że jest to wartość zwracana przez tę metodę. Metoda może zwrócić wynik np. za pomocą wbudowanej w Javę funkcji return.

Poniższy schemat przedstawia budowę metody:

[](https://kodilla.com/static/bootcamp-java/java-0_05.jpg)

Każda METODA składa się z :

* **Informacji o swojej dostępności** (access) — na ten moment omówimy tylko jeden rodzaj dostępności, a dokładnie: public. Oznacza ona, że metoda jest publiczna i można z niej korzystać po stworzeniu obiektu.
* **Typ danych** (return type — value to be returned), które zwraca za pomocą funkcji return (jeszcze więcej o niej niebawem), np. String, int, double, boolean. Zamiast typu danych może pojawić się również void — oznacza on, że metoda nie zwraca żadnej wartości.
* **Nazwa metody** (method name), którą sobie wymyślisz. Nazwa powinna zawsze opisywać dokładnie to, co w metodzie się wydarzy. Przykład: jeżeli wiemy, że metoda będzie odejmować liczbę "a" od liczby "b" (B - A), to warto rozważyć nazwę subtractAFromB. Ideą jest, abyśmy zawsze mówili nazwą metody to, co dokładnie się w niej stanie, tak samo jak w przypadku nazw klas czy zmiennych.
* **Typ Argumentu** (type of argument) — inaczej typ zmiennej przyjmowanej przez metodę. Jeżeli używając metody przekażemy inny typ argumentu niż ten, który został wskazany przy jej tworzeniu w klasie, w naszym programie wystąpi błąd.
* **Argumenty, Parametry** (tu: a, b), które metoda opcjonalnie może przyjmować. Argumentem jest zmienna o określonym typie (opisanym wyżej), np. String, int, double, boolean. Za pomocą tych argumentów, czyli inaczej informacji dla metody, możemy wykonać konkretne operacje. Tak jak na przykład wspomniane wcześniej odejmowanie liczby "a" od liczby "b".
* **Ciała metody** (method body), czyli zbiory instrukcji zawartych między klamrami {..}, które są w niej wykonywane, gdy metoda zostanie wywołana.

**Zadanie — tworzenie własnej klasy użytkownika**

Używając edytora Kodilla, stwórz swoją własną klasę Użytkownika. Przypisz jej atrybuty i metody.

**Przykład projektu prostej klasy**

Aby móc stworzyć w przyszłości obiekt potrzebny np. do wykonania jakiegoś obliczenia, musimy najpierw utworzyć klasę. Naszą klasę w tym wypadku nazwiemy SimpleCalculator, ponieważ chcemy, aby była odpowiedzialna za dodawanie i odejmowanie liczb całkowitych:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class SimpleCalculator

{

public int addAToB(int a, int b) {

return a + b;

}

public int subtractBFromA(int a, int b) {

return a - b;

}

}

Jak widzisz, powyższa klasa ma 2 metody. Przeanalizujmy je:

Pierwsza metoda public int addAToB(int a, int b):

1. jest **publiczna (public)**, czyli możliwa do wykorzystania poza kodem samej klasy;
2. zwraca wartość o typie int (liczbę całkowitą), czyli że np. wynikiem jej obliczeń jest liczba całkowita;
3. jej nazwa to addAToB;
4. przyjmuje dwa argumenty typu int o nazwach: a i b;
5. zwraca (return) wynik dodawania przekazanych argumentów (a + b).

Druga metoda — public int subtractAFromB(int a, int b) w tym wypadku jest podobnie, również metoda jest **publiczna**, zwraca wartość o typie int, jej nazwa to subtractAFromB, czyli wiemy, że jest odpowiedzialna za odejmowanie przyjmowanych argumentów o typie int, czyli a i b.

W każdej z metod na pewno rozpoznajesz operacje, które były przedstawiane wcześniej w przykładach związanych ze składnią Javy. "Ciało" omawianych metod (pomiędzy klamrami {...} ), zawiera operacje dodawania i odejmowania z wykorzystaniem znaków **+** i **-** .

Tworząc naszą klasę SimpleCalculator, za pomocą zawartych w niej metod, zbudowaliśmy "matryce" do obliczeń, takie jak "...+..." oraz "...-...". Jednak klasa nie jest w stanie sama z siebie wywołać utworzonych przez nas metod, aby np. wykonać jedno z tych działań (chyba, że metody są statyczne, ale do tego wrócimy w przyszłości).

Aby **wykonać konkretne obliczenie** na bazie tego wzorca z klasy, musimy **utworzyć obiekt**, który będzie wystąpieniem kalkulatora mogącego wykonywać obliczenia takie jak, np. "2+3" czy "5-2". To właśnie obiekt jest zdolny do wykonania instrukcji zawartych wewnątrz metod. Stworzenie nowego obiektu pozwoli nam na wywołanie przygotowanej przez nas metody, np. odejmowania na tym konkretnym utworzonym kalkulatorze.

**Notatka:** Pamiętaj, powyższy konstrukt to istota myślenia obiektowego, do którego musisz się przyzwyczaić.

**Jak utworzyć obiekt w Javie?**

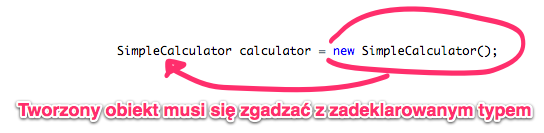
W kontekście stworzonych wcześniej klasy i metod (dla przeprowadzenia pewnych obliczeń), utworzymy obiekt za pomocą operatora new. W tym celu piszemy w programie taką linię kodu:

SimpleCalculator calculator = new SimpleCalculator();

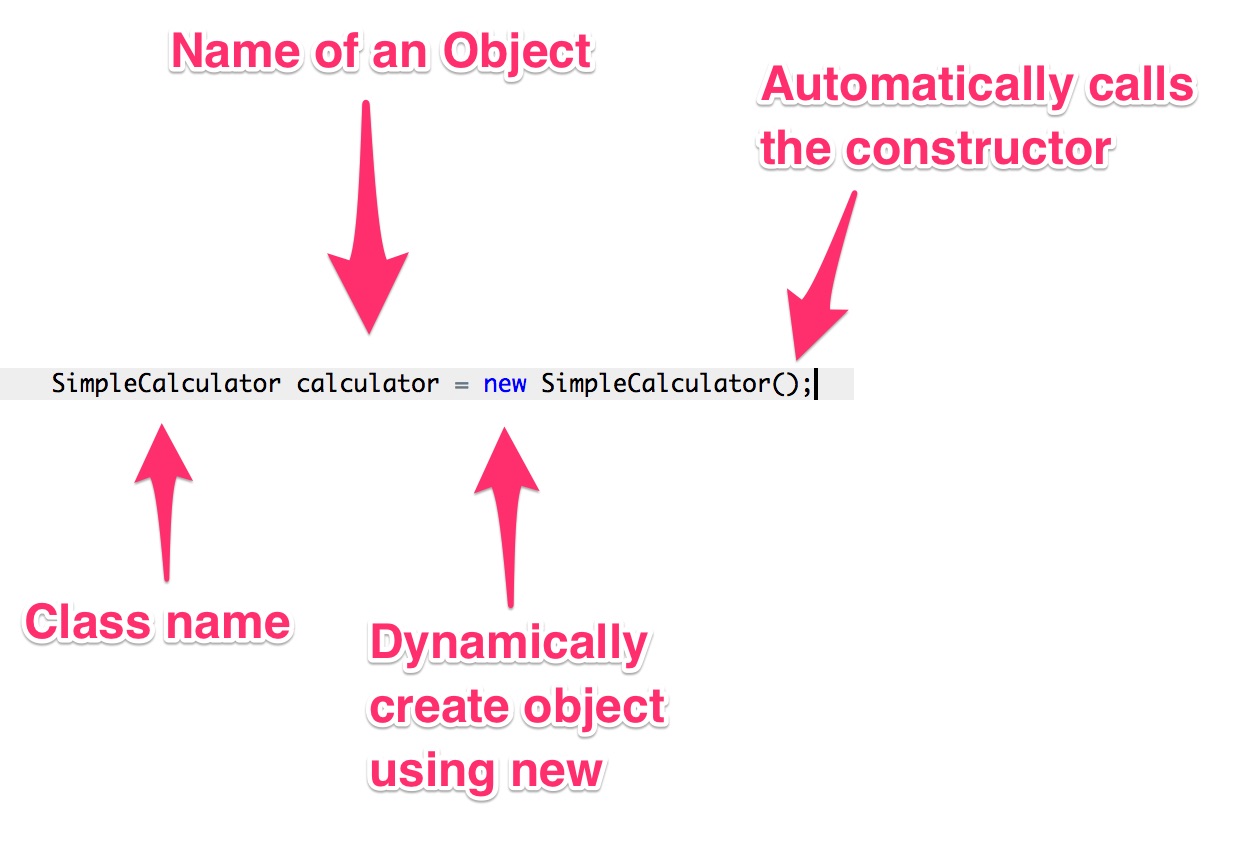
O co w niej chodzi? Po pierwsze, musimy utworzyć zmienną (tu: calculator), do której przypiszemy ten nowy obiekt (ta zmienna będzie reprezentować nasz obiekt). Jak już powiedzieliśmy, każda zmienna w Javie musi mieć określony typ (po swojej lewej stronie, jak np. int). Tutaj, **jako że ta zmienna będzie obiektem, typem tej zmiennej jest klasa**, na bazie której będzie tworzony ten obiekt. Czyli typ zmiennej to: SimpleCalculator.

Po prawej stronie znaku "=" najpierw używamy operatora new, który służy do stworzenia nowego obiektu. Następnie używamy odwołania do klasy, a w zasadzie do jej tzw. konstruktora (tu: SimpleCalculator()), która jest matrycą dla naszego obiektu i kończymy linijkę znakiem średnika. O konstruktorze, który definiuje się jawnie, a nie domyślnie, w ramach klasy (powyżej mamy użycie niejawne, w tle), powiemy więcej przy okazji atrybutów klasy.

Warto zauważyć, że typ zmiennej, do której przypisujemy tworzony obiekt, jest **taki jak nazwa klasy**, na podstawie której ten obiekt tworzymy (konstruktor).

[](https://kodilla.com/static/bootcamp-java/java-0_06.png)

Prześledźmy dodatkowo poniższy przykład tworzenia obiektu, na podstawie wcześniej utworzonej klasy SimpleCalculator:

[](https://kodilla.com/static/bootcamp-java/java-0_07.jpg)

Dla powtórzenia tłumaczymy składowe instrukcji:

* **Class Name** (SimpleCalculator) — typ dla zmiennej calculator, który jest taki sam jak nazwa klasy, z której korzystamy do stworzenia obiektu.
* **Name of an Object** (calculator) — czyli nazwa zmiennej. Może ona być dowolna, jednak warto trzymać się konwencji, by mówiła dokładnie o tym, co w niej jest.
* **Dynamically Create Object using new** (new) — operator służący do tworzenia obiektu.
* **Automatically Calls the Constructor** (SimpleCalculator()) — odwołanie do konstruktora, czyli do naszej klasy, która na bazie tej klasy stworzy obiekt (matryca).

**Notatka:** Warto wspomnieć, że ta zmienna calculator sama w sobie nie jest obiektem, ale elementem, który wskazuje na obiekt, jako tzw. **referencja**.

Wracając do kalkulatora, spójrz na linijkę 20. Jest w niej przedstawiony ponownie przykład tworzenia nowego obiektu:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class SimpleCalculator

{

public int addAToB(int a, int b) {

return a + b;

}

public int subtractAFromB(int a, int b) {

return a - b;

}

}

class Application

{

public static void main(String[] args) {

SimpleCalculator calculator = new SimpleCalculator();

}

}

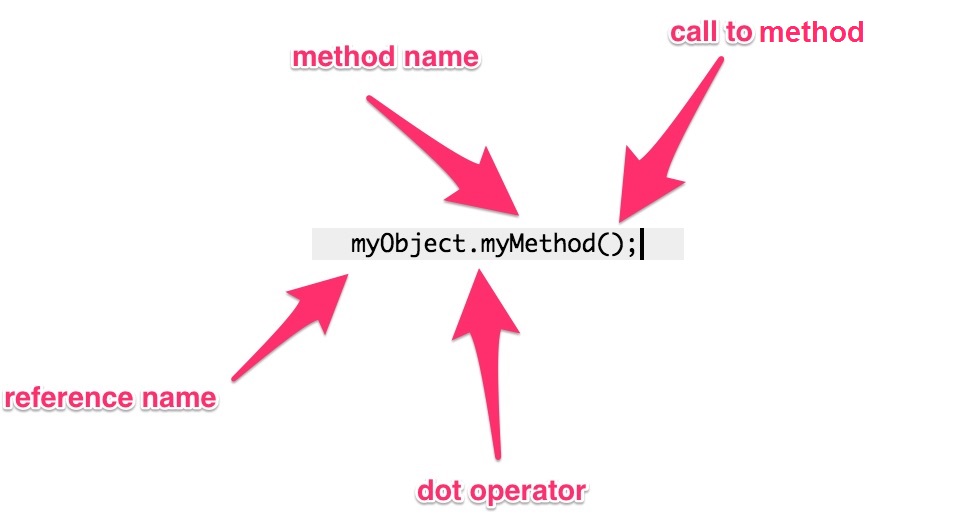
**Notatka:** Patrząc na kod pod linkiem, możesz z zaskoczeniem pomyśleć: "Halo! Dlaczego są tutaj 2 klasy?". Stworzyliśmy klasę Application w celu zobrazowania, jak w rzeczywistości uruchamia się metodę w ramach wcześniej wspomnianej metody main — jest ona "matką" wszystkich metod. Metoda main jest w programach Javowych tą, która jako pierwsza jest uruchamiana w aplikacji (po kliknięciu "Run") — to od niej wszystko się zaczyna, bez niej program nie wykona żadnej operacji. Zazwyczaj metoda main znajduje się w odrębnej klasie odpowiadającej za uruchomienie programu.

**Mamy obiekt! Co teraz?**

Udało nam się stworzyć obiekt, co pozwala teraz wykorzystać metody stworzone w klasie SimpleCalculator. Metody, które posiada nasz obiekt to: dodawanie (addAToB) i odejmowanie (subtractAFromB). Są one w nim dlatego, że utworzyliśmy je wcześniej w klasie (SimpleCalculator).

Metodę można wywołać przy pomocy tak zwanego "operatora kropki".

Spójrz na poniższy schemat przedstawiający wywołanie metody z obiektu:

[](https://kodilla.com/static/bootcamp-java/java-0_08.jpg)

Opis schematu:

* **Reference name** (myObject) — nazwa utworzonego obiektu, do którego chcemy się odwołać.
* **Dot operator** (.) — daje pełen dostęp do metod klasy, z której stworzyliśmy obiekt i pozwala na ich wykorzystanie w obrębie działającego programu, np. w innych klasach, jeśli mają publiczną zawartość.
* **Method name** (myMethod) — nazwa metody, którą chcemy wywołać. Metoda ta musi znajdować się we wcześniej utworzonej klasie, z której powstał obiekt.
* **Call to method ()** () — odwołanie do metody klasy. Jeśli metoda nie przyjmuje żadnych argumentów, nawiasy pozostawiamy puste.

W skrócie chodzi o to, że jeśli stworzymy obiekt, to przy jego wywołaniu, dodając kropkę, możemy po tej kropce podać nazwę metody wywołując jej wykonanie. Oczywiście jest to możliwe, jeśli taka metoda istnieje w klasie, na bazie której powstał ten obiekt.

W poniższym przykładzie, w ciele metody main (gdzie finalnie uruchamiamy nasz program), zobaczysz, jak wygląda to na prawdziwym kodzie programu, linijka 22:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class SimpleCalculator

{

public int addAToB(int a, int b) {

return a + b;

}

public int subtractAFromB(int a, int b) {

return a - b;

}

}

class Application

{

public static void main(String[] args) {

SimpleCalculator calculator = new SimpleCalculator();

calculator.addAToB();

}

}

[Wynik kompilacji powyższego programu.](https://kodilla.com/pl/project-java/38811)

Jak widzisz, wykorzystując operator kropki wywołaliśmy metodę dodawania, jednak zamiast wyniku pojawia się błąd. Dzieje się tak, ponieważ nie podaliśmy żadnych argumentów (parametrów), a ta metoda wymaga podania dokładnie 2 argumentów. Wcześniej wspominaliśmy o tym, że metoda, którą chcemy wywołać, ma 2 argumenty (dla dodania 2 liczb). Dokładniej mówiąc — dwie liczby całkowite.

Teraz, w przykładzie poniżej, w linii 22 użyjemy metody addAToB, żeby dodać dwie liczby całkowite, wstawiając w nawiasie wywołanej metody dwie liczby po przecinku. Następnie wynik przypisaliśmy do zmiennej result o typie int.

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class SimpleCalculator

{

public int addAToB(int a, int b) {

return a + b;

}

public int subtractAFromB(int a, int b) {

return a - b;

}

}

class Application

{

public static void main(String[] args) {

SimpleCalculator simpleCalculator = new SimpleCalculator();

int result = simpleCalculator.addAToB(5, 5);

}

}

Dodaliśmy argumenty po przecinku (5, 5) i zapisaliśmy wynik do zmiennej o typie int o nazwie result, ponieważ w metodzie addAToB zadeklarowaliśmy zwracaną wartość o typie int, w klasie SimpleCalculator. W skrócie, jeżeli metoda zwraca wynik o typie np. int, to zmienna, w której ten wynik zapisujemy, musi być tego samego typu (w tym przykładzie int).

Wygląda na to, że mamy już wszystko, ale warto upewnić się, czy dobrze jest wykonywana funkcja dodawania. Wyświetlamy wynik naszego działania (linia 24)!

<https://kodilla.com/pl/project-java/38812>

Sprawdź w sekcji stdout (konsola). Działa! Nasze wyliczenia wykonały się pomyślnie. Wynik to 10, więc możemy uznać, że napisaliśmy swój pierwszy w życiu algorytm liczący w Javie!

**Zadanie — odejmowanie liczb zmiennoprzecinkowych**

Stwórz własną klasę i w niej metodę zajmującą się odejmowaniem liczb zmiennoprzecinkowych. (Nie zapomnij aby nadać też typ zwracanych wartości w metodzie). Następnie stwórz obiekt na podstawie swojej klasy i spraw, by wynik odejmowania pojawił się w konsoli (stdout).

**Przykład tworzenia klasy z atrybutami**

Wiesz już, że metody to pewne możliwe akcje w ramach klasy (obiekt-użytkownik może się zarejestrować w portalu, dodać zdjęcie, wylogować się). Co jeśli jednak chcemy opisać dodatkowo ten obiekt za pomocą pewnych cech, tzw. atrybutów? Np. użytkownika portalu opiszemy za pomocą atrybutów: płeć, rok urodzenia, obywatelstwo. Podobnie dla innych obiektów możemy wyróżnić możliwe metody i cechy:

* Pilot
  + metody: turnOn, volumeUp
  + atrybuty: serialNumber, color, model
* Samochód
  + metody: showGasolineLevel
  + atrybuty: model, engine
* Kot
  + metody: jumpOnTable, runOnSand
  + atrybuty: name, color, weight

A więc obok metod możemy też wymienić jakie cechy chcemy przechowywać na temat obiektu. Takie cechy nazywamy zamiennie: **polami** (fields) lub **atrybutami** (attributes), nie mylić z argumentami (parametrami metody).

Nazwa atrybutu w klasie to np. "color". Natomiast aby przypisać dany kolor obiektowi stworzonemu z tej klasy, należy przypisać takiemu atrybutowi **wartość**, podobnie jak zmienna może posiadać wartość.

Aby przypisać wartość do atrybutu klasy, musimy wykorzystać do tego odpowiedni **konstruktor** w klasie, przystosowany do naszych potrzeb.

Wcześniej nie deklarowaliśmy konstruktora w klasie i korzystaliśmy z domyślnego. Każda klasa posiada taki domyślny konstruktor, który nie jest reprezentowany jawnie kodem w kodzie klasy. W takim przypadku przy tworzeniu obiektu wystarczyło napisać new SimpleCalculator();.

Tym razem chcemy, by nasz obiekt posiadał pewne właściwości i sami chcemy zdecydować o tym, do jakich atrybutów będą one przypisane. Dlatego musimy stworzyć jawnie taki konstruktor w ramach klasy, który będzie potem wywołany przy pomocy operatora new (identycznie jak ten domyślny, niejawny konstruktor), w trakcie tworzenia nowego obiektu.

**Tworzenie jawnego konstruktora**

Konstruktor w Javie zawsze jest tworzony na podstawie nazwy klasy, a więc nazwa konstruktora jest taka, jak nazwa klasy. Spójrz na linijkę 7:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class User

{

public User() {

}

}

Jak widzisz, wygląda on podobnie do metody, jednakże nie określa się typu zwracanych informacji (np. public ~~int~~ User()). Działanie tego konstruktora można przedstawić następująco: "Wywołam się zawsze wtedy, kiedy stworzysz nowy obiekt na podstawie mojej klasy". Możemy to rozumieć jako sposób konstruowania obiektu w trakcie jego tworzenia.

Notatka: Możesz zapytać... a dlaczego po prostu nie możemy napisać String *name = "John";* albo *String sex = "female";*? Z początku może to wydawać się prostsze, ale pewna złożoność obiektów pozwala lepiej opisywać bardziej skomplikowane byty, a możliwość wielokrotnego tworzenia i wykorzystywania obiektów pozwoli zaoszczędzić masę czasu, jeśli musimy np. stworzyć dziesiątki tysięcy użytkowników portalu.

Zatem dzięki konstruktorowi możemy nadać np. użytkownikowi portalu (obiektowi) jakieś właściwości i zarządzać sposobem przypisywania wartości do nich. Jednak nim nasz użytkownik będzie miał konkretną płeć i imię, musimy jeszcze trochę rozbudować naszą klasę:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class User

{

private String name;

private String sex;

public User() {

}

}

Już wcześniej wspominaliśmy o atrybutach klasy, omawiając jej budowę. W powyższym kodzie klasa właśnie uzyskała 2 nowe atrybuty: name i sex (płeć), które będą wykorzystywane w utworzonym już konstruktorze. To są właśnie cechy przyszłych obiektów. Pojawiło się też nieznane nam do tej pory słowo private przed typem atrybutu (pola).

Private (prywatne) mówi o tym, że atrybut jest dostępny tylko w obrębie obiektu i żaden inny obiekt nie może z niego wprost skorzystać. Może to brzmieć jak ograniczenie, a nie zaleta, ale jest to bardzo przydatne i pozwala dbać o porządek. Będziemy to rozwijać szerzej w temacie związanym z *enkapsulacją*.

Skoro mamy już atrybuty klasy, to poznajmy sposób przypisania wartości do tych atrybutów. Przyda nam się do tego konstruktor, zawarty w ciele klasy. Aby konstruktor był w stanie to zrobić, musimy go rozbudować o argumenty:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class User

{

private String name;

private String sex;

public User(String name, String sex) {

}

}

Jak widzisz, konstruktor w nawiasach po swojej nazwie przyjmuje argumenty (linia 10) w ten sam sposób, jak metody omawiane wcześniej. Określamy w nawiasie typ i nazwę argumentów. Każdy z kolejnych argumentów jest od siebie oddzielony przecinkiem, co pozwala je od siebie odróżnić. Dla tych argumentów użyto m.in. typu String, czyli zwykłego tekstu (ciąg znaków), co jest naturalne np. dla imienia.

Konstruktor został rozbudowany o argumenty, jednak musi się też coś wydarzyć w jego ciele (pomiędzy "wąsami"), aby wartości argumentów zostały zapisane w prywatnych polach (atrybutach) naszego obiektu.

Do tego musimy wykorzystać operator this — jest on swego rodzaju wskaźnikiem. Pamiętasz lekcje geografii, gdy nauczyciel wskazywał swoim drewnianym wskaźnikiem na kraj i mówił: "to jest Hiszpania"? this też jest takim wskaźnikiem w obrębie klasy. Chodzi o to, abyśmy mogli odwołać się do wartości w innym miejscu kodu i wykorzystać go w pożądanym miejscu. Tutaj chcemy, w ramach konstruktora, odwołać się do atrybutów klasy.

Aby umieć sobie to lepiej ułożyć, spójrz na ciało konstruktora w linijce 11:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class User

{

private String name;

private String sex;

public User(String name, String sex) {

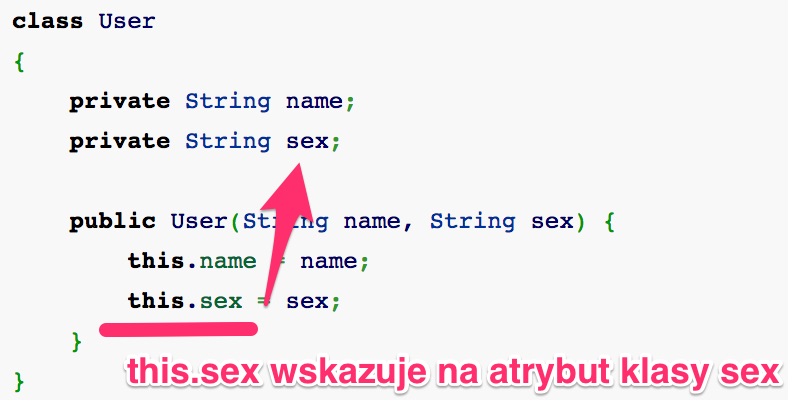
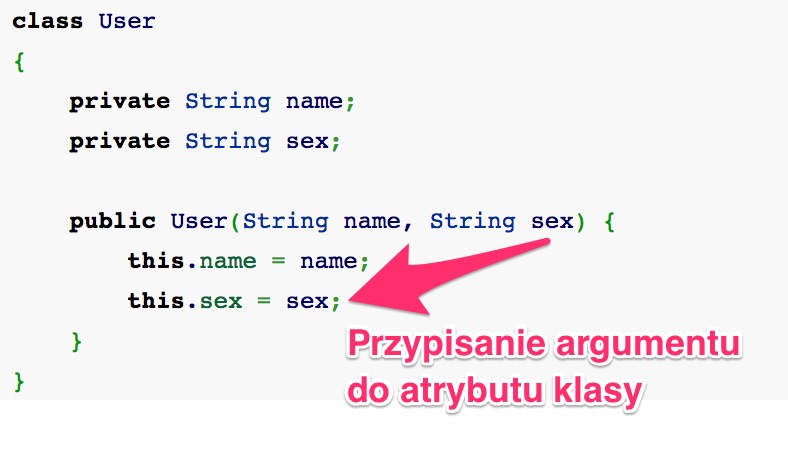
this.name = name;

this.sex = sex;

}

}

W ciele metody konstruktora wykorzystaliśmy operator this, który wskazuje na atrybut klasy, a następnie za pomocą "=" przypisuje argument otrzymany w konstruktorze. Możemy przetłumaczyć sobie tę operację w ten sposób: "Przy tworzeniu obiektu, do mojego atrybutu o nazwie sex (ang. płeć), zapisz argument konstruktora, który nazywa się sex".

[](https://kodilla.com/static/bootcamp-java/java-0_09.jpg)[](https://kodilla.com/static/bootcamp-java/java-0_10.jpg)

Parafrazując: za pomocą powyższego kodu możemy, używając konstruktora z danymi wartościami argumentów, utworzyć obiekty na bazie klasy gdzie argumenty konstruktora trafią do atrybutów klasy, z której tworzymy obiekt, i ten obiekt będzie zawierał te atrybuty (cechy).

Dzięki utworzonej w ten sposób klasie jesteśmy w stanie stworzyć obiekt Użytkownika o imieniu Andrew, wykorzystując operator new i wprowadzając nasze właściwości w okrągłych nawiasach po nazwie klasy. Zerknij na linijkę 20:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class User

{

private String name;

private String sex;

public User(String name, String sex) {

this.name = name;

this.sex = sex;

}

}

class Appliaction {

public static void main(String[] args) {

User andrew = new User("Andrew", "male");

}

}

Zastanówmy się, jak sprawdzić, czy faktycznie te wartości zostały zapisane przez konstruktor do atrybutów klasy.

**Gettery — metody zwracające właściwości**

Na razie stworzyliśmy obiekt z prywatnymi polami (cechami). Co jeśli chcemy wartości tych pól (atrybutów) wyświetlić w konsoli? Najlepszym sposobem jest stworzenie tak zwanych getterów, czyli metod pomocniczych wyciągających wartości z prywatnych pól klasy. Wykorzystują one operator this, który już poznaliśmy. Skupmy się na metodach znajdujących się w linijkach 15 i 19:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class User

{

private String name;

private String sex;

public User(String name, String sex) {

this.name = name;

this.sex = sex;

}

public String getName() {

return this.name;

}

public String getSex() {

return this.sex;

}

}

class Appliaction {

public static void main(String[] args) {

User andrew = new User("Andrew", "male");

}

}

Jak zapewne widzisz, metody te zawierają w sobie operator this w ramach funkcji return, mówiący w wolnym tłumaczeniu: "Klaso, daj mi wartość tego pola". Następnie wartość ta jest zwracana z wykorzystaniem return.

[](https://kodilla.com/static/bootcamp-java/java-0_11.jpg)

Aby poprawnie sprawdzić stworzone metody, należy po utworzeniu naszego obiektu użytkownika wywołać je za pomocą referencji "andrew" (zmienna, do której zapisaliśmy nasz obiekt, jest inaczej referencją/odnośnikiem do obiektu) i wyświetlić w konsoli z wykorzystaniem polecenia "System.out.println()" w linijkach od 31 do 34:

<https://kodilla.com/pl/project-java/38814>

Wynik wyświetla się w stdout, a w wygląda dokładnie tak: "male Andrew"!

**Zadanie — tworzenie obiektu**

Stwórz swoją własną klasę z atrybutami i konstruktorem. Pamiętaj, aby utworzyć klasę z metodą main, w której stworzysz obiekt. W metodzie main zainicjalizuj obiekt wcześniej utworzonej klasy i wyciągnij jej atrybuty za pomocą getterów.

**Dziedziczenie**

**Czym jest?**

Babcia czasem jak nas widzi, to mówi: "Ach, ten kolor włosów to Ty masz po mamie". Każdy z nas ma jakiegoś członka rodziny, po którym coś odziedziczył, np. niebieskie oczy, talent do sportów lub robótek ręcznych. Obiekty, tak jak ludzie, potrafią odziedziczyć pewne właściwości (oczywiście jest to znacznie bardziej kontrolowane, niż w przypadku ludzi :)). Obiekty (dzieci) mogą dziedziczyć po innych obiektach (rodzicach) i przejmować ich umiejętność, np: dodawania, wyświetlania śmiesznych obrazków, wysyłania maili do NASA, itp. W tym temacie poznamy działanie dziedziczenia w programowaniu i jego kluczowe założenia.

**Po co nam dziedziczenie?**

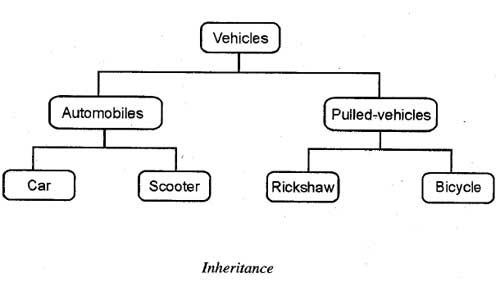
Dziedziczenie w podejściu obiektowym spełnia bardzo ważną rolę: możliwość wydzielenia części wspólnych klas, aby nie powielać wcześniej już napisanego kodu. Inaczej mówiąc, stosujemy tu pewną reużywalność wytworzonych elementów programu. Taka konwencja pozwala na użycie kodu do obsługi nowych przypadków. Dalsze przykłady pozwolą na głębsze zrozumienie zalet dziedziczenia.

**Jak to działa?**

Wyobraźmy sobie sytuację, w której chcemy zbudować portal z ogłoszeniami sprzedaży różnego rodzaju pojazdów (samochody, rowery, traktory). Część cech wszystkich pojazdów jest wspólna (np. każdy ma rok produkcji). Inna część cech jest specyficzna tylko dla konkretnego rodzaju pojazdu — np. samochody mają pojemność silnika, a rowery model siodełka.

Możemy przedstawić powyższe jako pewną hierarchię. Pojazd jest bardziej ogólnym pojęciem, który definiuje cechy opisujące wszelkie rodzaje pojazdów. Natomiast "pod" pojazdami, mamy konkretne rodzaje pojazdów, posiadające cechy występujące tylko u danego z nich. Np. samochody i pojazdy mechaniczne napędzane siłą mięśni są pojazdami. Tak samo jak auto i skuter są automobilami i jednocześnie pojazdami.

Taki model hierarchii możemy przedstawić jak na poniższym diagramie:

[](https://kodilla.com/static/bootcamp-java/java-0_12.jpg)

Tenże diagram nazywamy też drzewem relacji. Chodzi o relacje (powiązania) pomiędzy obiektami, gdzie możemy zacząć rozważać, który obiekt od którego może... dziedziczyć.

**Przykład zastosowania dziedziczenia**

Wyobraź sobie, że w ramach portalu sprzedaży pojazdów, chcesz stworzyć klasy dla kilku różnych rodzajów pojazdów (samochód, skuter, ciężarówka), by móc później tworzyć te pojazdy (obiekty z klas). Te pojazdy mają pewne wspólne cechy (atrybuty), dla przykładu niech będzie to pojemność silnika :)

Żeby nie powielać tworzenia tej cechy w klasie każdego pojazdu (samochód, skuter, ciężarówka), najlepiej stworzyć **klasę nadrzędną** (przykładowo nazwaną właśnie *Vehicle*), która będzie posiadała tę cechę, a klasy poszczególnych pojazdów będą ją po prostu dziedziczyć.

W ten sposób unikniemy powielania kodu :)

**Tworzenie klasy, po której będziemy dziedziczyć**

Tworzymy nadrzędną klasę pojazdu (Vehicle), która zawiera:

* atrybut model (private String model),
* atrybut year (private String year),
* oraz konstruktor (omawiany w poprzedniej sekcji).

Poniżej widzisz przykład takiej klasy:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class Vehicle

{

private String model;

private String year;

public Vehicle(String model, String year) {

this.model = model;

this.year = year;

}

}

**Tworzenie klasy dziedziczącej**

Mamy klasę reprezentującą pojazd, czas na klasę bardziej "konkretną" (piętro niżej) — Car. Będzie ona dziedziczyć od Vehicle. Co wyróżnia wszystkie samochody? Samochody m.in. mają taką cechę (atrybut), jak pojemność silnika oraz funkcje (metody) odpalania i gaszenia silnika. W kodzie będzie to wyglądało tak jak poniżej:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class Vehicle

{

private String model;

private String year;

public Vehicle(String model, String year) {

this.model = model;

this.year = year;

}

}

class Car

{

private double engineCapacity;

public void turnOnEngine() {

}

public void turnOffEngine() {

}

}

**Extends — rozszerzenie klasy**

Dziedziczenie w Javie odbywa się poprzez wykorzystanie funkcji extends (ang. *extend* — rozszerzyć). Mówimy tutaj o rozszerzeniu klasy dziedziczącej (Car) o elementy z klasy nadrzędnej (Vehicle).

Przyjrzyj się linii 16. Po prawej stronie od nazwy klasy dziedziczącej wstawiamy extends i wskazujemy klasę, po której mamy dziedziczyć:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class Vehicle

{

private String model;

private String year;

public Vehicle(String model, String year) {

this.model = model;

this.year = year;

}

}

class Car extends Vehicle

{

private double engineCapacity;

public Car(double engineCapacity, String model, String year) {

super(model, year);

this.engineCapacity = engineCapacity;

}

public void turnOnEngine() {

}

public void turnOffEngine() {

}

}

Podsumowując, *dziedziczenie* oznacza, że obiekty utworzone z klasy Car mają te same pola i metody, co obiekty tworzone z klasy Vehicle. Mogą mieć też dodatkowe, charakterystyczne dla siebie pola i metody, które nie dotyczą wszystkich pojazdów — np. każdy samochód ma pojemność baku, chociaż niektóre pojazdy w ogóle nie mają takiego parametru (np. rowery albo taczki).

**Konstruktor klasy dziedziczącej**

Jako że klasa Car dziedziczy teraz po klasie Vehicle, musimy to także uwzględnić w jej konstruktorze.

**Konstruktor** (kod z ostatniego przykładu, linia 20), który przyjmuje argumenty związane z modelem i rokiem produkcji, czyli z dziedziczonymi atrybutami, musi się jakoś z nimi uporać. Do tego wykorzystamy metodę wbudowaną w język Javy: super();. Odwołuje się ona do **konstruktora** klasy, po której dziedziczymy, bo jak pamiętasz — tam też jest **konstruktor**.

**Metoda super()**

Metoda super(); mówi jednoznacznie: "Użyj konstruktora klasy, po której dziedziczę, do zajęcia się wskazanymi argumentami (model, year)". Pozwala to na uzupełnienie atrybutów klasy, po której dziedziczymy, z poziomu konstruktora klasy dziedziczącej.

Poniższy kod przedstawia cały proces wykorzystania metody super(); :

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class Vehicle

{

private String model;

private String year;

public Vehicle(String model, String year) {

this.model = model;

this.year = year;

}

}

class Car extends Vehicle

{

private double engineCapacity;

public Car(double engineCapacity, String model, String year) {

super(model, year);

this.engineCapacity = engineCapacity;

}

public void turnOnEngine() {

}

public void turnOffEngine() {

}

}

class Application {

public static void main(String[] args) {

Car car = new Car(2.5, "XYZ", "2014");

}

}

Parafrazując: powyższy kod wraz z metodą super() pozwoli wpłynąć na atrybuty klasy nadrzędnej z poziomu konstruktora klasy dziedziczącej.

**Dziedziczenie metod**

Pora urozmaicić odrobinę naszą nadrzędną klasę Pojazdu (Vehicle) o 2 metody. Dorobimy tzw. gettery :)

Zwróć uwagę na linie od 15 do 21:

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class Vehicle

{

private String model;

private String year;

public Vehicle(String model, String year) {

this.model = model;

this.year = year;

}

public String getModel() {

return this.model;

}

public String getYear() {

return this.year;

}

}

class Car extends Vehicle

{

private double engineCapacity;

public Car(double engineCapacity, String model, String year) {

super(model, year);

this.engineCapacity = engineCapacity;

}

public void turnOnEngine() {

}

public void turnOffEngine() {

}

}

Jak widzisz, nasza klasa Vehicle zyskała 2 nowe metody: getModel() i getYear(). Mają one na celu pozwolić wyciągnąć z obiektu ukryte informacje i np. wyświetlić je w konsoli. Dzięki temu zabiegowi klasa Car zyska również te metody! Jak to rozumieć?

Dzięki wykorzystaniu odwołania this zapisaliśmy wartości w ukrytych polach (private) klasy Vehicle. Nie są one dostępne dla innej klasy, jednakże przy użyciu dziedziczenia i stworzonych metod do pobierania danych jak getModel(), jesteśmy w stanie pobrać te dane z klasy, po której dziedziczymy.

Reasumując, dzięki temu, że klasa Car dziedziczy po klasie Vehicle, może korzystać z jej metod.

Dostrzeżesz to w poniższym kodzie, w którym:

* w linii 46 tworzymy nowy samochód i przypisujemy go do referencji car,
* w linii 47 na naszym samochodzie (car) używamy odziedziczonej metody getModel() (odziedziczonej z klasy Vehicle) i przypisujemy zwracaną wartość do zmiennej model,
* w linii 49 sprawdzamy wynik całej operacji poprzez wyświetlenie zmiennej w konsoli.

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class Vehicle

{

private String model;

private String year;

public Vehicle(String model, String year) {

this.model = model;

this.year = year;

}

public String getModel() {

return this.model;

}

public String getYear() {

return this.year;

}

}

class Car extends Vehicle

{

private double engineCapacity;

public Car(double engineCapacity, String model, String year) {

super(model, year);

this.engineCapacity = engineCapacity;

}

public void turnOnEngine() {

}

public void turnOffEngine() {

}

}

class Aplication {

public static void main(String[] args) {

Car car = new Car(2.5, "JWT", "2001");

String model = car.getModel();

System.out.println(model);

}

}

Spójrz na metodę main w linijce 44, która uruchamia nasz program i tworzy obiekt Car. Jak widzisz, mimo że klasa Car nie posiada jawnie zadeklarowanej metody getModel(), można tę metodę wywołać. Wszystko to dzięki dziedziczeniu.

Przeanalizuj kod, który stworzyliśmy, i spróbuj znaleźć inne zastosowanie dla dziedziczenia na przykładach z Twojego otoczenia!

**Notatka:** W kontekście zagadnienia dziedziczenia dowiedzieliśmy się, że możemy ułożyć strukturę klas/obiektów w pewną hierarchię, aby ustalić kto od kogo może dziedziczyć. Ma to na celu wyodrębnienie w klasie nadrzędnej wspólnych atrybutów, które mogą być dziedziczone przez klasy "niżej". Oprócz atrybutów możemy dziedziczyć także metody. Przydadzą nam się wbudowane w Javę metody: extends, super().

**Zadanie — dziedziczenie klas**

Korzystając z edytora Kodilla:

1. Utwórz 2 nowe klasy (Dog i Animal), gdzie klasa Dog będzie dziedziczyć po klasie Animal. W klasie, którą będziesz dziedziczyć, zaimplementuj publiczne metody (np. picie i jedzenie).
2. Stwórz klasę Application z metodą main, a następnie stwórz w niej obiekt klasy Dog.
3. Wywołaj zaimplementowane metody.

Jeśli masz wątpliwości do powyższego materiału, to - zanim zatwierdzisz - zapytaj na czacie :)

Zapoznałe(a)m się!

**0.6. Sprawdź swoja wiedzę!**

Jak wiadomo, praktyka czyni mistrza i pomaga nam upewnić się, że wszystko prawidłowo rozumiemy. Przed Tobą nieco większe zadanie mające na celu podsumować wiedzę zdobytą do tej pory. Wykonaj je za pomocą edytora Kodilla i pochwal się wynikiem na grupie swojej edycji.

**KROK 1** Stwórz klasę

1. Nazwij ją User.
2. Dodaj atrybuty użytkownika takie jak: imię, nazwisko, płeć, wiek, mail.
3. Utwórz konstruktor dla tej klasy.
4. Stwórz metody charakterystyczne dla klasy użytkownika, tj.: logowanie, usunięcie konta.
5. Stwórz gettery dla atrybutów klasy.

**KROK 2** Stwórz klasę podrzędną (dziedziczącą po klasie User).

1. Nazwij ją ForumUser.
2. Dodaj atrybuty charakterystyczne dla użytkownika forum np. nazwa użytkownika, liczba postów, zalogowany (true/false).
3. Utwórz konstruktor dla użytkownika forum.
4. Stwórz metody charakterystyczne dla użytkownika forum, np. opublikowanie postu na forum czy komentowanie postu oraz taką, która będzie przedstawiała użytkownika w konsoli.
5. Odziedzicz właściwości klasy User.
6. Rozbuduj konstruktor o metodę super().

**KROK 3** Stwórz klasę Application:

1. Stwórz w niej metodę main().
2. Utwórz nowy obiekt ForumUser.
3. Wywołaj metodą która przedstawi nam użytkownika forum w konsoli.