

# WPROWADZENIE DO KLAS, OBIEKTÓW I METOD

# WPROWADZENIE

Klasa stanowi esencję języka Java. Na niej oparto całą definicję języka, ponieważ klasa definiuje naturę obiektu. Stanowi w ten sposób fundament programowania obiektowego w języku Java. Wewnątrz klas zdefiniowane zostają dane i kod, który na nich operuje. Kod ten ma postać metod. Ponieważ klasy, obiekty i metody stanowią fundament Javy. Ich zrozumienie pozwoli nie tylko pisać bardziej zaawansowane programy w Javie, ale umożliwi również poznanie elementów – języka Java - które będziemy omawiać na kolejnych wykładach.

# POJĘCIE KLASY

Klasa jest szablonem definiującym postać obiektu. Określa ona zarówno dane obiektu, jak i kod, który działa na tych danych. Java używa specyfikacji klasy podczas tworzenia **obiektu**. Obiekty są **instancjami** klasy. Innymi słowy, klasa jest zestawem planów określających sposób konstrukcji **obiektu**. Fizyczna reprezentacja klasy w pamięci komputera powstaje dopiero na skutek utworzenie obiektu klasy.

Zmienne oraz metody tworzące klasę nazywamy **składowymi** klasy.

# OGÓLNA POSTAĆ KLASY

Definiując klasę, deklarujemy jej dokładną postać i zachowanie. W tym celu określamy zmienne, które zawiera klasa, i metody, które działają na tych zmiennych. Oczywiście pewne klasy mogą zawierać tylko same metody bądź same zmienne składowe, ale większość klas zawiera jedno i drugie. Klasę tworzymy przy użyciu słowa *class*. Ogólna definicja klasy:

```
class nazwaKlasy {  
    //deklaracja zmiennych składowych  
    typ zmienna 1;  
    ...  
    typ zmiennaN;  
    //deklaracja metod  
    typ metoda1(parametry){//ciało metody}  
    ...  
    typ metodaN(parametry){//ciało metody}}
```

# OGÓLNA POSTAĆ KLASY

Chociaż nie wymuszają tego żadne reguły składni, poprawnie zaprojektowana klasa powinna definiować jedną i tylko jedną logiczną jednostkę. Na przykład klasa przechowująca nazwiska i numery telefonów nie powinna również przechowywać również o informacji o kursach walut, cenach produktów czy innych niepowiązanych informacji. Poprawne zaprojektowane klasa grupują jedynie te informacje, które są powiązane logicznie. Jeśli umieścimy w klasie niezwiązane ze sobą dane, to dosyć szybko przekonamy się, że odbije się to niekorzystnie na strukturze i czytelności naszego kodu.

Dotychczas używane przez nas klasy miały tylko jedną metodę: *main()*, której składnie omówiliśmy sobie na pierwszych ćwiczeniach. Teraz zaczniemy tworzyć własne metody.

# DEFINIOWANIE KLASY

Do omawiania tego zagadnienia posłużą nam klasy hermetyzujące informacje o samochodach takich jak osobowe, vany i dostawcze. Klasą nadrzędną nazwiemy *Vehicle* i umieścimy w niej trzy informacje o pojeździe: liczbę pasażerów, pojemność zbiornika oraz średni zużycie paliwa.

Klasa definiuje nowy typ danych. W naszym przypadku nazwą tego typu jest *Vehicle*. Nazwy tej będziemy używać, aby deklarować obiekty typu *Vehicle*. Pamiętajmy o tym, o czym wspomniałem wcześniej, że klasa stanowi jedynie opis typu i nie tworzy żadnego obiektu. Aby utworzyć obiekt typu *Vehicle* należy użyć następującej instrukcji:

*Vehicle minivan = new Vehicle(); // utworzenie obiektu o nazwie minivan*

Po wykonaniu tej instrukcji *minivan* będzie **instancją** klasy *Vehicle* i będzie posiadać fizyczną reprezentację w pamięci komputera.

# DEFINIOWANIE KLASY

Za każdym razem, gdy tworzymy instancję klasy, powstaje obiekt zawierający własną kopię każdej zmiennej składowej zdefiniowanej przez klasę. Zatem każdy obiekt typu *Vehicle* będzie zawierać własną kopię zmiennych składowych *passengers*, *fuelcap* i *lkm*. Dostęp do tych zmiennych odbywa się za pomocą operatora zapisywanego za pomocą kropki (.). Operator ten łączy nazwę obiektu z nazwą zmiennej składowej:

*obiekt.składowa;*

Na przykład żeby przypisać wartość 65 składowej *fuelcap* obiektu *minivan*, użyjemy takiej instrukcji:

*minivan.fuelcap = 65;*

Operatora w postaci kropki używamy zarówno do dostępu do zmiennych składowych, jak i metod.

# PRZYKŁAD DEFINICJI KLASY

```
2
3 class Vehicle {
4     int passengers;
5     int fuelcap;
6     double lkm; //zużycie paliwa na 100km
7 }
8
9 //klasa służąca zadeklarowaniu obiektu typu Vehicle
10 class VehicleDemo {
11     public static void main(String[] args) {
12         int range;
13
14         Vehicle minivan = new Vehicle();
15
16         //przypisanie wartości składowym obiektu minivan
17         minivan.passengers = 7;
18         minivan.fuelcap = 65;
19         minivan.lkm = 9.1;
20
21         //obliczanie zasięgu pojazdu przy pełnym zbiorniku
22         range = (int) ( minivan.fuelcap / minivan.lkm * 100);
23
24         System.out.println("Minivan przewozi: " + minivan.passengers +
25                             " osób na odległość " + range + " kilometrów.");
26     }
27 }
```

Minivan przewozi: 7 osób na odległość 714 kilometrów.

Process finished with exit code 0



# TWORZENIE OBIEKTÓW

W poprzednim przykładzie obiekt typu *Vehicle* deklarowaliśmy za pomocą instrukcji:

```
Vehicle minivan = new Vehicle();
```

Taka deklaracja spełnia dwie funkcje. Po pierwsze, deklaruje zmienną *minivan* typu *Vehicle*. Sama zmienna nie definiuje jeszcze obiektu, a jedynie może zawierać jego **referencję**. Po drugie, tworzy instancję obiektu i przypisuje zmiennej *minivan* referencję do tego obiektu. Odbywa się to przy użyciu operatora *new*.

Operator *new* dynamicznie (czyli w trakcie działania programu) przydziela pamięć dla obiektu i zwraca jego referencję. Referencja ta jest de facto adresem tego obiektu utworzonego w pamięci za pomocą instrukcji *new*. Zostaje ona przypisana zmiennej. W języku Java wszystkie obiekty są tworzone dynamicznie.

# TWORZENIE OBIEKTÓW

W poprzednim przykładzie obiekt typu *Vehicle* deklarowaliśmy za pomocą instrukcji:

```
Vehicle minivan = new Vehicle();
```

Taka deklaracja spełnia dwie funkcje. Po pierwsze, deklaruje zmienną *minivan* typu *Vehicle*. Sama zmienna nie definiuje jeszcze obiektu, a jedynie może zawierać jego **referencję**. Po drugie, tworzy instancję obiektu i przypisuje zmiennej *minivan* referencję do tego obiektu. Odbywa się to przy użyciu operatora *new*.

Operator *new* dynamicznie (czyli w trakcie działania programu) przydziela pamięć dla obiektu i zwraca jego referencję. Referencja ta jest de facto adresem tego obiektu utworzonego w pamięci za pomocą instrukcji *new*. Zostaje ona przypisana zmiennej. W języku Java wszystkie obiekty są tworzone dynamicznie.

# REFERENCJE OBIEKTÓW I OPERACJE PRZYPISANIA

W operacjach przypisania zmienne stanowiące referencje obiektów zachowują się inaczej niż zmienne typów prostych takich jak *int*. Przypisanie wartości zmiennej typu prostego innej zmiennej działa w oczywisty sposób. Zmienna po lewej stronie operatora przypisania otrzymuje **kopię wartości** zmiennej znajdującej się po prawej stronie. Podczas przypisania zmiennych będących referencjami obiektów sytuacja jest nieco bardziej skomplikowana, ponieważ powodują zmianę obiektu, którego dotyczy zmienna po lewej stronie operatora przypisania. Różnica ta może czasami powodować, że wynik przypisania nie jest zgodny z intuicją.

# REFERENCJE OBIEKTÓW I OPERACJE PRZYPISANIA

Przykład:

```
Vehicle car1 = new Vehicle();
```

```
Vehicle car2 = car1;
```

Na pierwszy rzut oka możemy mieć wrażenie, że zmienne *car1* i *car2* odnoszą się do różnych obiektów, ale rzeczywistość jest inna. Zmienne *car1* i *car2* będą stanowiły referencję tego samego obiektu. Przypisanie zmiennej *car1* zmiennej *car2* sprawia, że *car2* staje się referencją tego samego obiektu co *car1*. Zatem do obiektu tego możemy odwoływać się zarówno za pomocą zmiennej *car1*, jak i *car2*. Na przykład po wykonaniu poniższej instrukcji przypisania:

```
car1.lkm = 9.5;
```

Obie poniższe instrukcje `println()`

```
System.out.println(car1.lkm);
```

```
System.out.println(car2.lkm);
```

Spowodują wyświetlenie tej samej wartości 9.5.

# REFERENCJE OBIEKTÓW I OPERACJE PRZYPISANIA

Chociaż *car1* i *car2* odnoszą się do tego samego obiektu, nie są powiązane w żaden inny sposób. Na przykład kolejna instrukcja przypisania na zmiennej *car2* może zmienić obiekt, którego referencję stanowi ta zmienna. Przykład:

```
Vehicle car1 = new Vehicle();
```

```
Vehicle car2 = car1;
```

```
Vehicle car3 = new Vehicle();
```

```
car2 = car3; //teraz car2 i car3 są referencjami tego samego obiektu
```

Po wykonaniu instrukcji zmienna *car2* odnosi się do tego samego obiektu co zmienna *car3*. Natomiast obiekt, którego referencją jest zmienna *car1*, nie zmienił się.

# METODY

Klasy składają się ze zmiennych i metod. Jad dotąd nasza klasa Vehicle zawiera jedynie dane, ale nie ma żadnych metod. Choć dane zawierając jedynie dane są zupełnie poprawne, to w praktyce zdecydowana większość klas dysponuje również metodami. Metody są programami działającymi na danych zdefiniowanych w klasie oraz w wielu przypadkach udostępniającymi te dane. W większości przypadków pozostałe części programu komunikują się z klasą za pośrednictwem jej metod.

Metoda zawiera jedną lub więcej instrukcji. W poprawnym, obiektowym kodzie metoda powinna wykonywać jedno, ściśle określone zadanie. Każda metoda ma nazwę, za pomocą której ją wywołujemy. Ogólna postać metody:

```
typ-zwracany nazwa (lista-parametrów){  
//ciało metody  
}
```

*Typ-zwracany* określa typ danych zwracanych przez metodę. Może to być dowolny typ, w tym także określony typ zdefiniowanej przez nas klasy. Jeśli metoda nie zwraca żadnych danych, *Typ-zwracany* musi być określony jako *void*. Nazwa metody może być dowolnym identyfikatorem różnym od już używanych w bieżącym zasięgu. *lista-parametrów* jest sekwencją typ-identyfikator rozdzielonych przecinkami. *Parametry* metody są właściwie zmiennymi, które otrzymują wartość *argumentów* przekazywanych metodzie podczas jej wywoływania. Jeśli metoda nie ma parametrów, to *lista-parametrów* jest pusta.

# KLASA VEHICLE Z DODANĄ METODĄ

```
3  class Vehicle {
4      int passengers;
5      int fuelcap;
6      double lkm; //zużycie paliwa na 100km
7
8      //metoda wyświetlająca zasięg
9      void range(){
10         System.out.println("Zasięg (w km): " + (int)(fuelcap/lkm*100));
11     }
12 }
13
14 //klasa służąca zadeklarowaniu obiektu typu Vehicle
15 class VehicleDemo {
16     public static void main(String[] args) {
17         int range;
18
19         Vehicle minivan = new Vehicle();
20         Vehicle sportscar = new Vehicle();
21
22         //przypisanie wartości składowym obiektu minivan
23         minivan.passengers = 7;
24         minivan.fuelcap = 65;
25         minivan.lkm = 9.1;
26         //przypisanie wartości składowym obiektu sportscar
27         sportscar.passengers = 0;
28         sportscar.fuelcap = 55;
29         sportscar.lkm = 12.5;
30
31         System.out.println("Minivan przewozi: " + minivan.passengers +
32             " osób");
33
34         minivan.range(); //wywołanie metody do wyświetlenia zasięgu
35         System.out.println("Auto sportowe przewozi: " + sportscar.passengers +
36             " osób");
37         sportscar.range(); //wywołanie metody do wyświetlenia zasięgu
38     }
39 }
```

Zasięg (w km): 714  
Auto sportowe przewozi: 0 osób  
Zasięg (w km): 440

Process finished with exit code 0

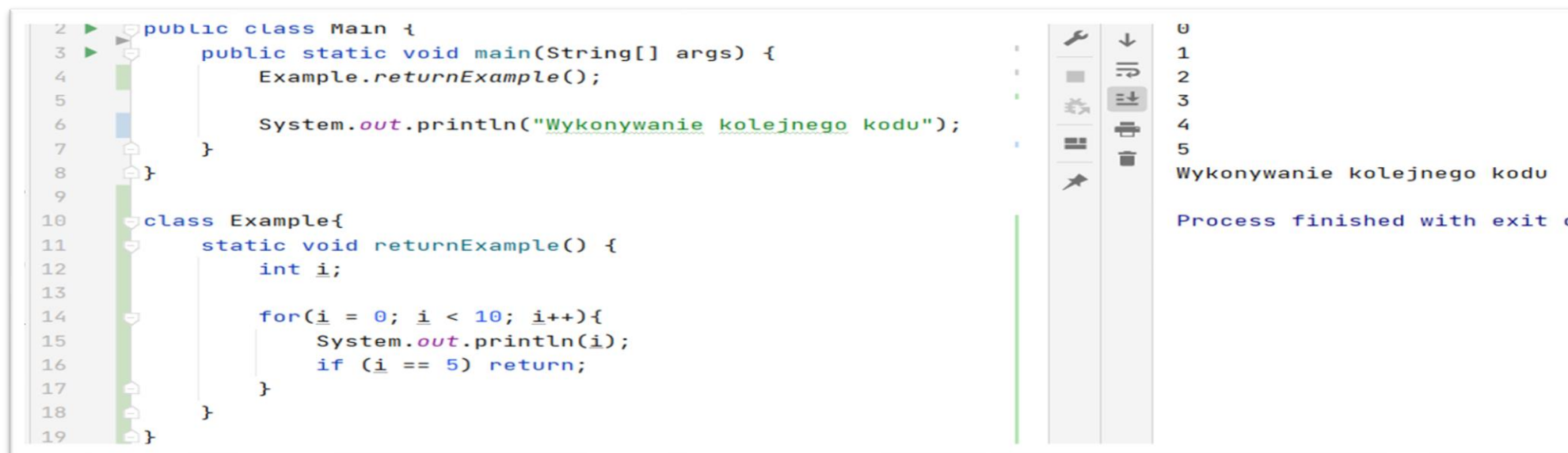
# POWRÓT Z METODY

W ogólnym przypadku metoda zwraca sterowanie na skutek wystąpienia jednej z dwóch przyczyn. Pierwszą z nich, co ilustruje *range()* w poprzednim przykładzie, jest napotkanie nawiasu klamrowego zamykającego ciało metody. Drugą przyczyną jest wykonanie instrukcji *return*. Istnieją dwie postacie instrukcji *return*: jedna stosowana w metodach *void* (nie zwracających) i druga służąca do zwrócenia pewnej wartości. Pierwsza służy do natychmiastowego zakończenia działania metody *void*. Przykład na kolejnym slajdzie:



# POWRÓT Z METODY

W tym przypadku pętla *for* zostanie wykonana jedynie dla wartości od 0 do 5, ponieważ gdy *i* jest równe 5, następuje powrót z metody. Metoda może zawierać więcej instrukcji *return*, zwłaszcza gdy jej kod zawiera wiele rozgałęzień.



The screenshot shows an IDE with two panes. The left pane displays Java code for a class `Main` and a static method `returnExample()` in a class `Example`. The `returnExample()` method contains a `for` loop from `i = 0` to `i < 10`, printing `i` and returning when `i == 5`. The right pane shows the execution output, which includes the text "Wykonywanie kolejnego kodu" and "Process finished with exit c".

```
2 public class Main {
3     public static void main(String[] args) {
4         Example.returnExample();
5
6         System.out.println("Wykonywanie kolejnego kodu");
7     }
8 }
9
10 class Example{
11     static void returnExample() {
12         int i;
13
14         for(i = 0; i < 10; i++){
15             System.out.println(i);
16             if (i == 5) return;
17         }
18     }
19 }
```

0  
1  
2  
3  
4  
5  
Wykonywanie kolejnego kodu  
Process finished with exit c

Podsumowując: metoda *void* może zakończyć działanie albo na skutek osiągnięcia nawiasu klamrowego zamykającego ciało, albo przez wykonanie instrukcji *return*.

# ZWRACANIE WARTOŚCI

Tak jak ustaliliśmy wcześniej metody *void* nie zwracają żadnej wartości i rzeczywiście takie metody często stosujemy w kodzie, to jednak większość pisanych przez nas metod zwraca jakąś wartość.

Wartości zwracane przez metody spełniają wiele różnych zadań. W niektórych przypadkach, na przykład *random()*, stanowią wynik pewnego algorytmu. W innych mogą informować o powodzeniu operacji wykonywanych przez metodę lub o błędzie bądź też zawierać kod statusu operacji. Tak czy inaczej, zwracanie wartości przez metody jest integralną częścią programowania w języku Java. Metoda zwraca wartość, używając poniższej instrukcji *return*:

*return wartość;*

Tej postaci instrukcji *return* możesz użyć jedynie w przypadku metod, których typ zwracany jest różny od *void*. Co więcej, metody takie muszą użyć instrukcji *return* w tej postaci.

# KLASA VEHICLE Z METODĄ ZWRACAJĄCĄ WARTOŚĆ

```
3 class Vehicle {
4     int passengers;
5     int fuelcap;
6     double lkm; //zużycie paliwa na 100km
7
8     int range(){
9         return (int)(fuelcap/lkm *100);
10    }
11 }
12 //klasa służąca zadeklarowaniu obiektu typu Vehicle
13 class VehicleDemo {
14     public static void main(String[] args) {
15         int range1, range2;
16
17         Vehicle minivan = new Vehicle();
18         Vehicle sportscar = new Vehicle();
19
20         //przypisanie wartości składowym obiektu minivan
21         minivan.passengers = 7;
22         minivan.fuelcap = 65;
23         minivan.lkm = 9.1;
24         //przypisanie wartości składowym obiektu sportscar
25         sportscar.passengers = 2;
26         sportscar.fuelcap = 55;
27         sportscar.lkm = 12.5;
28
29         range1 = minivan.range();
30         range2 = sportscar.range();
31
32         System.out.println("Minivan przewozi: " + minivan.passengers +
33             " osób na odległość do " + range1 + " kilometrów");
34         System.out.println("Auto sportowe przewozi: " + sportscar.passengers +
35             " osób na odległość do " + range2 + " kilometrów");
36     }
37 }
```

"C:\Program Files\Java\jdk-11.0.8\bin\java.exe" "-javaagent:C:\Program Files\Java\jdk-11.0.8\bin\javaagent.jar" -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=9090 -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote.authenticate=false -Djava.rmi.server.hostname=localhost -Djdk.attach.allowAttachSelf=true -Djconsole.port=9090 -Djconsole.ssl=false -Djconsole.authenticate=false -Djconsole.url=service:jmx:rmi:///jndi/rmi://localhost:9090/jmxrmi -jar C:\Program Files\Java\jdk-11.0.8\bin\java.exe

Minivan przewozi: 7 osób na odległość do 714 kilometrów  
Auto sportowe przewozi: 2 osób na odległość do 440 kilometrów

Process finished with exit code 0

# STOSOWANIE PARAMETRÓW

Podczas wywoływania możemy przekazać metodzie jeden lub więcej wartości. Pamiętajmy, że wartości przekazywane metodzie nazywamy *argumentami*. Natomiast zmienne, które wewnątrz metody otrzymują te wartości, nazywamy **parametrami**. Parametry metody deklarujemy wewnątrz nawiasów okrągłych następujących po nazwie metody. Składnie deklaracji parametrów jest taka sama jak w przypadku zwykłych zmiennych. Parametr należy do zasięgu metody i pomijając fakt, że otrzymuje wartości argumentów wywołania metody, poza tym zachowuje się jak każda inna zmienna lokalna.

# METODA Z PARAMETRAMI W KLASIE VEHICLE

```
3 class Vehicle {
4     int passengers;
5     int fuelcap;
6     double lkm; //zużycie paliwa na 100km
7
8     int range(){ return (int)(fuelcap/lkm *100); }
9
10    double fuelneeded (int km){ //oblicza paliwo potrzebne do przejechani km
11        return (double) km / 100 * lkm; }
12 }
13 //klasa służąca zadeklarowaniu obiektu typu Vehicle
14 class VehicleDemo {
15     public static void main(String[] args) {
16         double liters;
17         int dist = 252;
18         Vehicle minivan = new Vehicle();
19         Vehicle sportscar = new Vehicle();
20
21         //przypisanie wartości składowym obiektu minivan
22         minivan.passengers = 7;
23         minivan.fuelcap = 65;
24         minivan.lkm = 9.1;
25         //przypisanie wartości składowym obiektu sportscar
26         sportscar.passengers = 2;
27         sportscar.fuelcap = 55;
28         sportscar.lkm = 12.5;
29
30         liters = minivan.fuelneeded(dist);
31         System.out.println("Aby przejechać " + dist + " kilometrów, minivan" +
32             " potrzebuje " + liters + " litrów paliwa");
33         liters = sportscar.fuelneeded(dist);
34         System.out.println("Aby przejechać " + dist + " kilometrów, autosportowe" +
```

Aby przejechać 252 kilometrów, minivan potrzebuje 22.932 litrów paliwa  
Aby przejechać 252 kilometrów, autosportowe potrzebuje 31.5 litrów paliwa

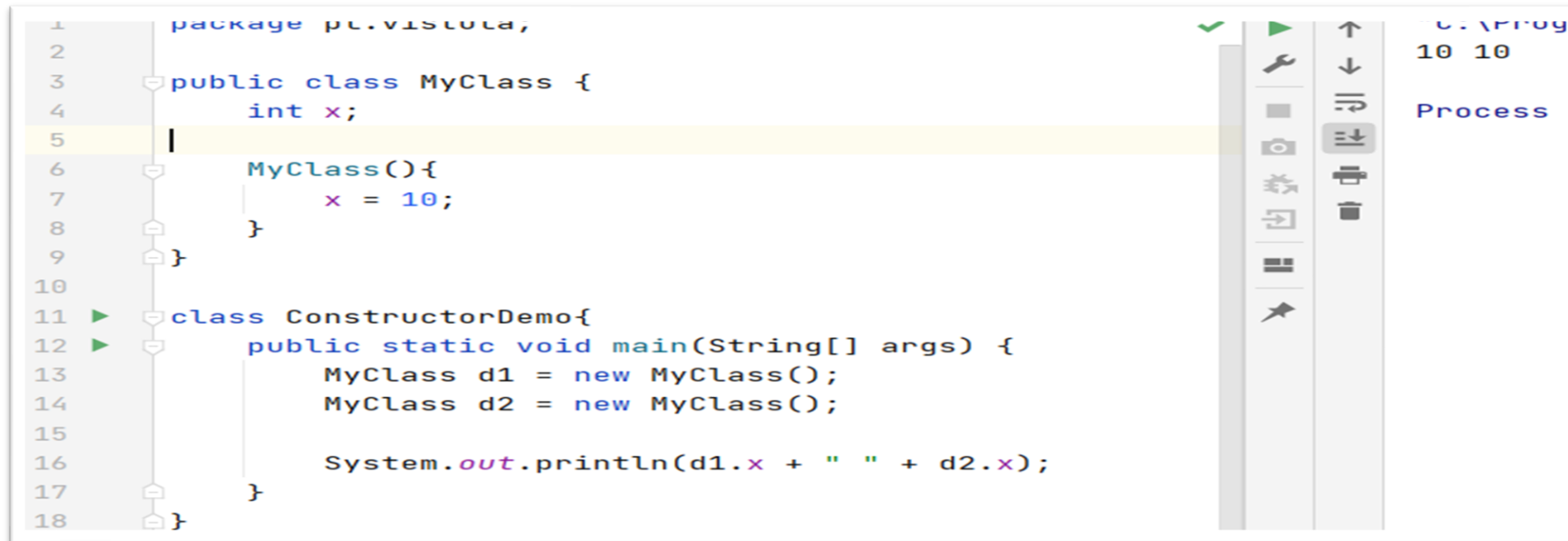
Process finished with exit code 0

# KONSTRUKTORY

Zadaniem **konstruktora** jest inicjalizacja obiektu podczas jego tworzenia. Konstruktor ma taką samą nazwę jak klasa i z punktu widzenia składni jest podobny do metody. Konstruktory nie mają jednak określonego typu zwracanego. Zwykle konstruktor nadaje wartości początkowe zmiennym składowym obiektu lub wykonuje inne czynności wymagane do nadania obiektowi ostatecznej postaci.

Niezależnie od tego czy zdefiniujemy konstruktor czy nie, wszystkie klasy w języku Java mają go, gdyż w razie konieczności Java automatycznie tworzy w klasie konstruktor domyślny. Taki konstruktor inicjalizuje niezainicjowane zmienne składowe domyślnymi wartościami zero, null i false odpowiednio dla typów numerycznych, referencji i typu boolean. Jeśli zdefiniujemy jednak własny konstruktor, konstruktor domyślny przestanie być używany.

# KONSTRUKTOR BEZ PARAMETROWY:



```
1 package pl.vistula;
2
3 public class MyClass {
4     int x;
5
6     MyClass(){
7         x = 10;
8     }
9 }
10
11 class ConstructorDemo{
12     public static void main(String[] args) {
13         MyClass d1 = new MyClass();
14         MyClass d2 = new MyClass();
15
16         System.out.println(d1.x + " " + d2.x);
17     }
18 }
```

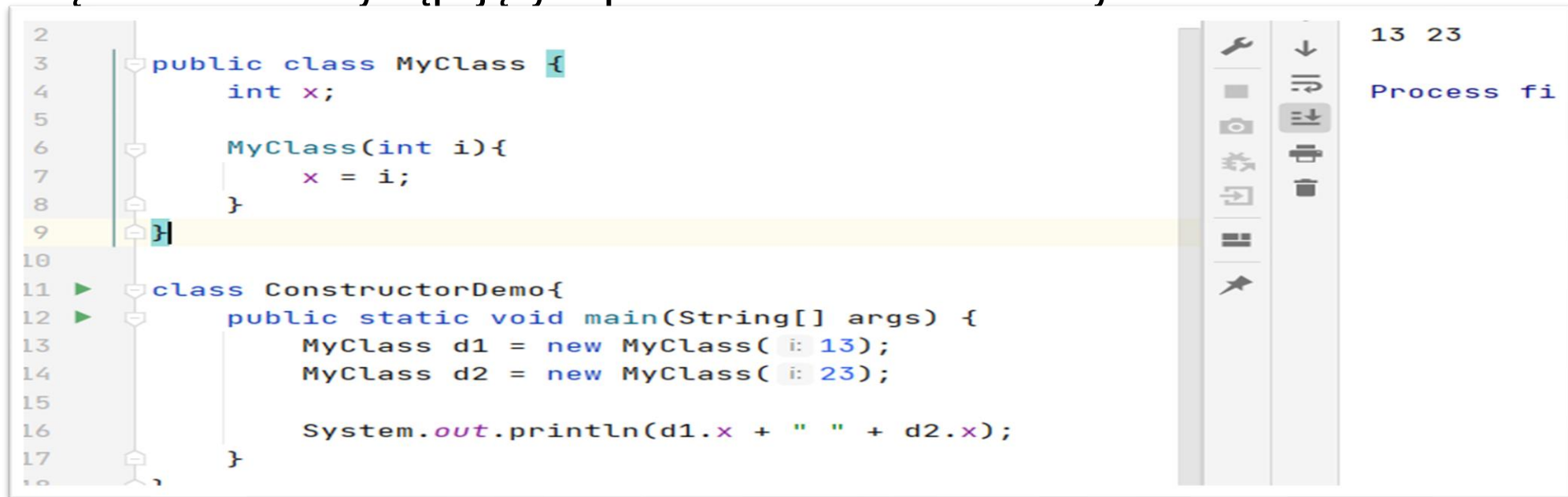
The screenshot shows an IDE with a Java file. The code defines a package `pl.vistula`, a class `MyClass` with an attribute `int x` and a no-argument constructor `MyClass()` that sets `x` to 10. A `ConstructorDemo` class contains a `main` method that creates two instances of `MyClass` and prints their `x` values. The output on the right shows `10 10`.

W tym przykładzie konstruktor przypisuje zmiennej składowej `x` klasy *MyClass* wartość 10. Konstruktor ten zostaje automatycznie wywołany, gdy tworzony jest obiekt klasy za pomocą instrukcji *new*.



# KONSTRUKTOR Z PARAMETRAMI

Konstruktor z poprzedniego przykładu nie miał żadnego parametru. W niektórych sytuacjach to wystarcza ale najczęściej przydatne są konstruktory mające jedno lub więcej parametrów. Parametry dodajemy do konstruktora tak samo jak w przypadku metod: deklarujemy je wewnątrz nawiasów występujących po nazwie konstruktor. Przykład:



```
2
3 public class MyClass {
4     int x;
5
6     MyClass(int i){
7         x = i;
8     }
9 }
10
11 class ConstructorDemo{
12     public static void main(String[] args) {
13         MyClass d1 = new MyClass(13);
14         MyClass d2 = new MyClass(23);
15
16         System.out.println(d1.x + " " + d2.x);
17     }
18 }
```

W tej wersji programu konstruktor *MyClass()* definiuje jeden parametr o nazwie *i*, który służy do inicjalizacji zmiennej składowej *x*. Zatem wykonanie jak w przykładzie powoduje nadanie parametrowi wartość 13, którą następnie konstruktor przypisuje do zmiennej składowej *x*.



# KLASA VEHICLE Z KONSTRUKTOREM

Klasę *Vehicle* możemy ulepszyć, wprowadzając konstruktor, który zajmie się automatyczną inicjalizacją zmiennych *passengers*, *fuelcap* i *lkm* podczas tworzenia nowego obiektu.

```
3 class Vehicle {
4     int passengers;
5     int fuelcap;
6     double lkm; //zużycie paliwa na 100km
7
8     Vehicle(int p, int f, double l){ //Konstruktor klasy Vehicle
9         passengers = p;
10        fuelcap = f;
11        lkm = l;
12    }
13
14    int range(){ return (int)(fuelcap/lkm *100); }
15
16    double fuelneeded (int km){ //oblicza paliwo potrzebne do przejechani km
17        return (double) km / 100 * lkm; }
18 }
19 //klasa służąca zadeklarowaniu obiektu typu Vehicle
20 class VehicleDemo {
21     public static void main(String[] args) {
22         double liters;
23         int dist = 252;
24         Vehicle minivan = new Vehicle( p: 7, f: 65, l: 9.1);
25         Vehicle sportscar = new Vehicle( p: 2, f: 44, l: 12.5);
26
27         liters = minivan.fuelneeded(dist);
28         System.out.println("Aby przejechać " + dist + " kilometrów, minivan" +
29             " potrzebuje " + liters + " litrów paliwa");
30         liters = sportscar.fuelneeded(dist);
31         System.out.println("Aby przejechać " + dist + " kilometrów, autosportowe" +
32             " potrzebuje " + liters + " litrów paliwa");
33     }
34 }
```

```
C:\Program Files\Java\jdk-11.0.8\bin\java.exe -javaagent:C:\Program Fil
Aby przejechać 252 kilometrów, minivan potrzebuje 22.932 litrów paliwa
Aby przejechać 252 kilometrów, autosportowe potrzebuje 31.5 litrów paliwa
Process finished with exit code 0
```

# KLASA VEHICLE Z KONSTRUKTOREM

Zarówno *minivan*, jak i *sportcar* zostają zainicjalizowane przez konstruktor *Vehicle()* podczas ich tworzenia. Odbywa się to zgodnie z argumentami przekazanymi konstruktorowi.

W naszym przykładzie zostały przekazane konstruktorowi *Vehicle()* wartości 7, 65 i 9.1, gdy operator *new* tworzy nowy obiekt. W ten sposób kopie składowe *passengers*, *fuelcap* i *lkm* należące do obiektu *minivan* otrzymają odpowiednio wartości 7, 65 i 9.

# OPERATOR NEW

W tym miejscu należało by powiedzieć trochę więcej o operatorze *new*.

W kontekście operacji przypisania operator *new* przyjmuje postać:

*zmienna = new nazwa-klasy(list-arg);*

W tym przypadku zmienna musi być typu klasy tworzonego obiektu. *Nazwa-klasy* określa właśnie klasę, do której należy tworzony obiekt. Nazwa klasy to, po której następuje lista argumentów umieszczonych w nawiasach (może być pusta), określa konstruktor klasy. Jeśli klasa nie definiuje własnego konstruktora, *new* użyje domyślnego konstruktora dostarczonego przez Javę. Zatem *new* może być używane do tworzenia obiektów każdej klasy. Operator *new* zwraca referencję utworzonego obiektu, która (w powyższym przykładzie) zostaje przypisana zmiennej.

Ponieważ pamięć komputera jest skończonym zasobem, może zdarzyć się, że *new* nie będzie mógł przydzielić pamięci dla nowego obiektu. W takim przypadku podczas działania programu zostanie wyrzucony wyjątek. W naszych uczelnianych programach nie musi oczywiście przejmować się brakiem pamięci, ale jak już będziecie Państwo tworzyć komercyjne aplikacje kwestia pamięci jest istotnym aspektem który należy brać pod uwagę.

# ODZYSKIWANIE PAMIĘCI (GARBAGE COLLECTOR)

Wiemy już że obiekty Javy są dynamiczne tworzone za pomocą operatora *new*. Wiemy również, że pamięć komputera jest skończonym zasobem i wobec tego okaże się, że na skutek jej braku utworzenie obiektów za pomocą operatora *new* nie będzie możliwe. Dlatego też kluczowym komponentem dynamicznego zarządzania pamięcią jest odzyskiwanie pamięci zajmowanej przez nieużywane obiekty. W niektórych innych językach programowania zwolnienie wcześniej przydzielonej pamięci musi odbyć się ręcznie. Na przykład w C++ do zwolnienia pamięci używamy *delete*. Jednak w Javie zastosowano inny bardziej zautomatyzowane podejście.

System odzyskiwania pamięci automatycznie usuwa nieużywane obiekty, bez jakiegokolwiek interwencji programisty. Działa to w uproszczeniu tak: jeśli nie istnieje żadna referencja obiektu, przyjmuje się, że obiekt nie jest już potrzebny, i zajmowana przez niego pamięć zostaje zwolniona. Zwolniona pamięć może być użyta do tworzenia nowych obiektów.

# ODZYSKIWANIE PAMIĘCI (GARBAGE COLLECTOR)

Odzyskiwanie pamięci odbywa się sporadycznie podczas działania programu. Nie następuje natychmiast dlatego, że jeden lub więcej obiektów nie jest już używany. Ze względów efektywnościowych odzyskiwanie pamięci ma zwykle miejsce po spełnieniu dwóch warunków: istnienia nieużywanych obiektów i konieczności odzyskania zajmowanej przez nie pamięci. Ponieważ odzyskiwanie pamięci wiąże się z dodatkowym nakładem, Java wykonuje je tylko wtedy, gdy to konieczne. My jako programiści nie znamy momentu, w którym to się dzieje.

Ciekawostka: W teorii można zasugerować Javie wywołanie odzyskania pamięci używając *System.gc()*, ale praktyce może to nie zadziałać i Java zignoruje to wywołanie.

# SŁOWO KLUCZOWE THIS

Ostatnim elementem tego wykładu, będzie omówienie słowa kluczowego *this*. Podczas wywoływania metody zostaje jej przekazany niejawnie argument stanowiący referencję obiektu, dla którego jest wywoływana metoda. Referencję tę reprezentuje właśnie słowo kluczowe *this*.

Poniższe przykłady powinny pomóc zrozumieć znaczenie tego słowa kluczowego:

# SŁOWO KLUCZOWE THIS

```
2 public class Power {
3     double b;
4     int e;
5     double val;
6
7     Power(double base, int exp){
8         b = base;
9         e = exp;
10
11         val = 1;
12         if(exp== 0) return;
13         for( ; exp>0; exp --) val = val * base;
14     }
15
16     double getPower(){
17         return val;
18     }
19 }
20
21 class DemoPower{
22     public static void main(String[] args) {
23         Power x = new Power( base: 4.0, exp: 2);
24         Power y = new Power( base: 2.5, exp: 1);
25         Power z = new Power( base: 5.7, exp: 0);
26         System.out.println(x.b + " podniesione do potęgi " + x.e +
27             " równa się " + x.getPower());
28         System.out.println(y.b + " podniesione do potęgi " + y.e +
29             " równa się " + y.getPower());
30         System.out.println(z.b + " podniesione do potęgi " + z.e +
31             " równa się " + z.getPower());
32     }
33 }
```

4.0 podniesione do potęgi 2 równa się 16.0  
2.5 podniesione do potęgi 1 równa się 2.5  
5.7 podniesione do potęgi 0 równa się 1.0

Process finished with exit code 0

```
2 public class Power {
3     double b;
4     int e;
5     double val;
6
7     Power(double base, int exp){
8         this.b = base;
9         this.e = exp;
10
11         this.val = 1;
12         if(exp== 0) return;
13         for( ; exp>0; exp --) this.val = this.val * base;
14     }
15
16     double getPower(){
17         return this.val;
18     }
19 }
20
21 class DemoPower{
22     public static void main(String[] args) {
23         Power x = new Power( base: 4.0, exp: 2);
24         Power y = new Power( base: 2.5, exp: 1);
25         Power z = new Power( base: 5.7, exp: 0);
26         System.out.println(x.b + " podniesione do potęgi " + x.e +
27             " równa się " + x.getPower());
28         System.out.println(y.b + " podniesione do potęgi " + y.e +
29             " równa się " + y.getPower());
30         System.out.println(z.b + " podniesione do potęgi " + z.e +
31             " równa się " + z.getPower());
32     }
33 }
```

4.0 podniesione do potęgi 2 równa się 16.0  
2.5 podniesione do potęgi 1 równa się 2.5  
5.7 podniesione do potęgi 0 równa się 1.0

Process finished with exit code 0



# SŁOWO KLUCZOWE THIS

Jak już wiemy wewnątrz metody możemy się odwoływać do się do pozostałych składowych klasy (nie podając nazwy obiektu ani klasy). Zatem wewnątrz metody *getPower()* instrukcja *return val;* dotyczy kopii zmiennej *val* należącej do obiektu, dla którego wywołana została metoda *getPower()*. Tę samą instrukcję możemy również zapisać w ten sposób: *return this.val;* W tym przypadku *this* oznacza referencję obiektu, dla którego wywołano metodę *getPower()*. Zatem *this.val* odnosi się do kopii *val* należącej do obiektu. Jeśli na przykład *getPower()* zostało wywołane dla obiektu *x*, to *this* stanowi referencję obiektu *x*. Zapis bez słowa kluczowego *this* stanowi formę skróconą.

W rzeczywistości pisanie kodu w ten sposób nie ma sensu, ponieważ w powyższych przykładzie użycie *this* niczego nie daje, a stosowanie skróconych odwołań jest łatwiejsze. Ale słowo kluczowe *this* ma też inne ważne zastosowanie.



# SŁOWO KLUCZOWE THIS

To zastosowanie najczęściej stosowane jest przy tworzeniu konstruktora klasy. Java zezwala bowiem, aby nazwy parametrów zmiennych lokalnych były takie same jak nazwy zmiennych składowych. W takim przypadku lokalny identyfikator przesłania zmienną składową. Nadal możemy się odwołać do zmiennej składowej, ale właśnie przez użycie *this*.

```
3 public class Power {  
4     double b;  
5     int e;  
6     double val;  
7  
8     Power(double b, int e){  
9         this.b = b;  
10        this.e = e;  
11  
12        val = 1;  
13        if(e== 0) return;  
14        for( ; e>0; e --) val = val * b;  
15    }
```

W tej wersji nazwy parametrów konstruktora są takie same jak nazwy zmiennych składowych, które w efekcie zostają przesłonięte. Dostęp do przesłoniętych zmiennych składowych jest właśnie możliwy za pomocą słowa kluczowego *this*.

# DZIĘKUJĘ

Więcej na:

[www.vistula.edu.pl](http://www.vistula.edu.pl)



**Akademia Finansów i Biznesu Vistula**  
ul. Stokłosy 3  
02-787 Warszawa  
(obok stacji metro Stokłosy)