Podstawy programowania obiektowego

Klasy i obiekty

- Każdy program w Javie składa się z jednej lub wielu *klas*.

- Wiemy już, że np. zmienna typu int może przechowywać liczby całkowite z przedziału $[-2147483648, 2147483647]$.

- Z obiektami jest podobnie: zmienna obiektowa klasy Punkt może przechowywać obiekty klasy (typu) Punkt. Klasa to zatem nic innego jak definicja nowego typu danych.

Tworzenie klasy

```
class nazwa_klasy
{
   treść klasy
}
```

- W treści klasy definiuje się *pola* i *metody*.
- **Pola** służą do przechowywania danych.
- **Metody** służą do wykonywania różnych operacji.
- Pola i metody nazywane są składowymi klasy.
- Każdą klasę zapisuje się w oddzielnym pliku o nazwie zgodnej z jej nazwą oraz rozszerzeniu *java*, np. klasę Main zapiszemy w pliku o nazwie *Main.java*, a klasę Punkt w pliku o nazwie *Punkt.java*.

Klasy i obiekty

Do opisu położenia punktu na płaszczyźnie potrzebujemy dwóch zmiennych. Załóżmy, że chodzi np. o punkty ekranowe (piksele) i wystarczy, jeśli będą mogły przyjmować tylko współrzędne całkowite.

```
class Punkt{
  int x;
  int y;
}
```

Po zdefiniowaniu klasy Punkt można zadeklarować zmienną typu Punkt.
Robi się to w taki sam sposób, w jaki deklarowane były zmienne typów prostych (np. short, int, char), pisząc:

typ_zmiennej nazwa_zmiennej;

Ponieważ typem zmiennej jest nazwa klasy, deklaracja ta przyjmie postać:

Tworzenie zmiennej referencyjnej (odnośnikowej/obiektowej)

nazwa_klasy nazwa_zmiennej;

Zatem

```
Punkt przykladowyPunkt;
```

Klasy i obiekty

```
Punkt przykladowyPunkt;
```

Zmienna o nazwie przykladowyPunkt domyślnie jest pusta, tzn. nie zawiera żadnych danych. Dokładniej, zmienna taka zawiera wartość specjalną *null*, która określa, że zmienna nie zawiera odniesienia do żadnego obiektu. Trzeba więc samodzielnie utworzyć obiekt klasy Punkt i powiązać go z tą zmienną:

Tworzenie obiektu (instancji/egzemplarza) klasy

```
nazwa_klasy nazwa_zmiennej = new nazwa_klasy();
```

W przypadku klasy Punkt:

```
Punkt przykladowyPunkt = new Punkt();
```

lub

```
Punkt przykladowyPunkt;
przykladowyPunkt = new Punkt();
```

Zmienna przykladowyPunkt zawiera referencję/odniesienie do przypisanego jej obiektu klasy `Punkt` i tylko poprzez nią można się do tego obiektu odwoływać.

Pola klasy

Aby odwołać się do danego pola klasy, należy skorzystać z operatora . (kropka):

Dostęp do pól obiektu

nazwa_zmiennej_referencyjnej.nazwa_pola_obiektu

Przykład (przypisanie do obu pól obiektu przykladowyPunkt wartości 100):

```
przykladowyPunkt.x = 100;
przykladowyPunkt.y = 100;
```

W pliku Punkt.java mamy:

```
public class Punkt{
  int x;
  int y;
}
```

W pliku Main.java mamy:

```
public class Main {
  public static void main(String[] args) {
    Punkt A = new Punkt();
    A.x = 100;
    A.y = 100;
    System.out.println("Współrzędna x punktu A: " + A.x);
    System.out.println("Współrzędna y punktu A: " + A.y);
  }
}
```

Po wywołaniu Run dla pliku *Main.java* otrzymujemy:

```
Współrzędna x punktu A: 100
Współrzędna y punktu A: 100
```

Nie trzeba oddzielnie kompilować klas Main i Punkt. Ponieważ w klasie Main wykorzystywana jest klasa Punkt, wywołanie polecenia *Run* dla *Main.java* spowoduje kompilację zarówno pliku *Main.java*, jak i *Punkt.java*. Obie klasy muszą znajdować się w tym samym katalogu.

Wartości domyślne pól

```
public class Punkt{
  int x;
  int y;
}
```

```
public class Main {
  public static void main (String args[]) {
    Punkt A = new Punkt();
    System.out.println("Współrzędna x punktu A: " + A.x);
    System.out.println("Współrzędna x punktu A: " + A.y);
  }
}
```

```
Współrzędna x punktu A: 0
Współrzędna y punktu A: 0
```

W Javie wartość, jaką przyjmuje niezainicjowane pole obiektu, jest ściśle określona:

Typ	**Wartość domyślna**
byte, short, int, long	0
float, double	0.0
char	\0
boolean	false
referencyjny	null

Metody klasy

- Klasy oprócz pól przechowujących dane zawierają także metody, które wykonują zapisane przez programistę operacje.
- Definiuje się je w ciele (czyli wewnątrz) klasy pomiędzy znakami nawiasu klamrowego.
- Każda metoda może przyjmować argumenty (parametry) oraz zwracać wynik.

Deklaracja metody

```
typ_wyniku  nazwa_metody (argumenty_metody)
{
   instrukcje metody
}
```

- Jeśli metoda nie zwraca żadnego wyniku, jako typ wyniku należy zastosować słowo *void*.
- Jeśli metoda nie przyjmuje żadnych argumentów, pomiędzy znakami nawiasu okrągłego nie należy nic wpisywać.

Metody klasy: metoda typu void

Do klasy Punkt dodamy metodę o nazwie *wyswietl*, która wypisuje na ekranie wartości współrzędnych x i y.
Metoda ta
- nie zwraca żadnego wyniku,
- nie przyjmuje żadnych argumentów.

```
public class Punkt{
  int x;
  int y;
  void wyswietl() {
    System.out.println("Współrzędna x = " + x);
    System.out.println("Współrzędna y = " + y);
  }
}
```

Po utworzeniu obiektu danej klasy można wywołać metodę (czyli wykonać zawarte w niej instrukcje), korzystając z operatora **.** (kropka).

Wywołanie metody

nazwa_zmiennej_referencyjnej.nazwa_metody(argumenty_metody);

Metody klasy: metoda typu void

W pliku Punkt.java mamy:

```
public class Punkt{
  int x;
  int y;
  void wyswietl() {
    System.out.println("Współrzędna x = " + x);
    System.out.println("Współrzędna y = " + y);
  }
}
```

W pliku Main.java mamy:

```
public class Main {
  public static void main (String args[]) {
    Punkt A = new Punkt();
    A.x = 100;
    A.y = 100;
    A.wyswietl();
  }
}
```

Metody klasy: metoda zwracająca wynik (1)

Do klasy Punkt dodamy teraz metody *pobierzX* i *pobierzY*, które będą zwracały odpowiednio wartości współrzędnej *x* i *y*.

Każda z tych metod
- zwraca wynik typu int,
- nie przyjmuje żadnych argumentów.

```
public class Punkt{
   int x;
   int y;
   void wyswietl() {
      System.out.println("Współrzędna x = " + x);
      System.out.println("Współrzędna y = " + y);
   }
   int pobierzX(){
      return x;
   }
   int pobierzY(){
      return y;
   }
}
```

- Przed nazwą metody znajduje się określenie typu zwracanego przez nią wyniku - int.
- Wynik jest zwracany dzięki instrukcji *return*.
- Zapis *return x* oznacza zwrócenie przez metodę wartości zapisanej w polu x.

W pliku Punkt.java mamy:

```
public class Punkt{
  int x;
  int y;
  void wyswietl() {
    System.out.println("Współrzędna x = " + x);
    System.out.println("Współrzędna y = " + y);
  }
  int pobierzX(){
    return x;
  }
  int pobierzY(){
    return y;
  }
}
```

W pliku Main.java mamy:

```
public class Main {
  public static void main (String args[]) {
    Punkt A = new Punkt();
    A.x = 100;
    A.y = 100;
    System.out.println("Współrzędna x = " + A.pobierzX());
    System.out.println("Współrzędna y = " + A.pobierzY());
  }
}
```

Metody klasy: metoda zwracająca wynik (2)

Do klasy Punkt dodamy teraz metodę, która zwróci w wyniku nowy obiekt klasy Punkt o takich współrzędnych, jakie zostały zapisane w polach obiektu bieżącego.

Metoda ta
- zwraca wynik typu Punkt,
- nie przyjmuje żadnych argumentów.

```
Punkt pobierzWsp() {
   Punkt nowy = new Punkt();
   nowy.x = x;
   nowy.y = y;
   return nowy;
}
```

- W ciele metody najpierw tworzymy nowy obiekt klasy Punkt, przypisując go zmiennej referencyjnej o nazwie *nowy*, a następnie przypisujemy jego polom wartości pól x i y z obiektu bieżącego.
- Za pomocą instrukcji return powodujemy, że obiekt nowy staje się wartością zwracaną przez metodę.

W pliku Punkt.java mamy:

```
public class Punkt{
  int x, y;
  void wyswietl() {
    System.out.println("Współrzędna x = " + x);
    System.out.println("Współrzędna y = " + y);
  }
  Punkt pobierzWsp() {
    Punkt nowy = new Punkt();
    nowy.x = x;
    nowy.y = y;
    return nowy;
  }
}
```

W pliku Main.java mamy:

```
public class Main {
  public static void main (String args[]) {
    Punkt A = new Punkt();
    A.x = 100;
    A.y = 100;
    Punkt B;
    B = A.pobierzWsp();        //lub Punkt B = A.pobierzWsp();
    B.wyswietl();
  }
}
```

Metody klasy: metoda zwracająca wynik (2)

```
public class Main {
  public static void main (String args[]) {
    Punkt A = new Punkt();
    A.x = 100;
    A.y = 100;
    Punkt B = A.pobierzWsp();
    B.wyswietl();
  }
}
```

```
Współrzędna x = 100
Współrzędna y = 100
```

- Zmiennej referencyjnej *B* przypisujemy obiekt zwrócony przez metodę *pobierzWsp* wywołaną na rzecz obiektu *A*. Zatem zapis:

Punkt B = A.pobierzWsp();

oznacza, że wywoływana jest metoda *pobierzWsp* obiektu *A*, a zwrócony przez nią wynik jest przypisywany zmiennej *B*.

- Wynikiem działania tej metody będzie obiekt (referencja do obiektu) klasy Punkt będący kopią obiektu *A*, czyli zawierający w polach x i y takie same wartości, jakie są zapisane w polach obiektu *A*.

Metody klasy: argumenty

- Argumenty (parametry) metody to inaczej dane, które można jej przekazać.
- Metoda może mieć dowolną liczbę argumentów umieszczonych w nawiasie okrągłym za jej nazwą.
- Poszczególne argumenty oddziela się przecinkiem.

metoda przyjmująca argumenty

```
typ_wyniku  nazwa_metody(typ_parametru_1  nazwa_parametru_1, ...
                                    ..., typ_parametru_n nazwa_parametru_n)
{
   treść metody
}
```

Metody klasy: argumenty (1)

Napiszemy teraz dwie metody *ustawX* i *ustawY*, które będą nadawały wartość polu x i polu y obiektów klasy Punkt.

Każda z tych metod
- nie zwraca żadnego wyniku (jest typu void),
- przyjmuje jeden argument typu int.

```
void ustawX(int wspX){
   x = wspX;
}
void ustawY(int wspY){
   y = wspY;
}
```

Podobnie można napisać metodę, która będzie jednocześnie ustawiała pola x i y obiektów klasy Punkt.

Metoda ta
- nie zwraca żadnego wyniku (jest typu void),
- przyjmuje dwa argumenty typu int.

```
void ustawXY(int wspX, int wspY){
   x = wspX;
   y = wspY;
}
```

```java
public class Punkt{
  int x, y;
  void wyswietl() {
    System.out.println("x = " + x + ", y = " + y);
  }
  void ustawX(int wspX){
    x = wspX;
  }
  void ustawY(int wspY){
    y = wspY;
  }
  void ustawXY(int wspX, int wspY){
    x = wspX;
    y = wspY;
  }
}
```

```java
public class Main {
  public static void main (String args[]) {
    Punkt A = new Punkt();
    A.ustawX(100);
    A.ustawY(100);
    Punkt B = new Punkt();
    B.ustawXY(2,5);
  }
}
```

Metody klasy: argumenty (1)

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

    Punkt A = new Punkt();
    A.ustawX(100);
    A.ustawY(100);

    Punkt B = new Punkt();
    B.ustawXY(2,5);

    System.out.print("Współrzędne punktu A: ");
    A.wyswietl();

    System.out.print("Współrzędne punktu B: ");
    B.wyswietl();
  }
}
```

```
Współrzędne punktu A: x = 100, y = 100
Współrzędne punktu B: x = 2, y = 5
```

Metody klasy: argumenty (2)

- Argumentem przekazanym metodzie może być również obiekt.
- Podobnie metoda może zwracać obiekt w wyniku swojego wykonania.
- W obu wymienionych sytuacjach postępowanie jest dokładnie takie samo jak w przypadku typów prostych.
- Np. metoda *ustawXY* w klasie Punkt mogłaby przyjmować jako argument obiekt tej klasy (ściślej: referencję do obiektu), a nie dwie liczby typu int, tak jak zostało to przedstawione na poprzednich slajdach.

Metoda ta
- nie zwraca żadnego wyniku (jest typu void),
- przyjmuje argument typu Punkt.

```
void ustawXY(Punkt przekazany){
  x = przekazany.x;
  y = przekazany.y;
}
```

- Argumentem jest tu obiekt *przekazany* klasy Punkt.
- W ciele metody następuje skopiowanie wartości pól z obiektu przekazanego jako argument do obiektu bieżącego.

```
public class Punkt{
  int x, y;
  void wyswietl() {
    System.out.println("x = " + x + ", y = " + y);
  }
  void ustawXY(Punkt przekazany){
    x = przekazany.x;
    y = przekazany.y;
  }
}
```

```
public class Main {
  public static void main (String args[]) {
    Punkt A = new Punkt();
    A.x = 100;
    A.y = 100;
    Punkt B = new Punkt();
    B.ustawXY(A);
    B.wyswietl();
  }
}
```

```
x = 100, y = 100
```

Przeciążanie metod

Na poprzednich slajdach pojawiły się dwie metody o takiej samej nazwie, ale różnym kodzie:

```
void ustawXY(int wspX, int wspY){
  x = wspX;
  y = wspY;
}
```

```
void ustawXY(Punkt przekazany){
  x = przekazany.x;
  y = przekazany.y;
}
```

Pytanie: czy takie dwie metody mogłyby współistnieć w klasie Punkt?

- W każdej klasie może istnieć dowolna liczba metod, które mają takie same nazwy, o ile tylko różnią się argumentami (ich liczbą lub typem).
- Mogą one, ale nie muszą, różnić się również typem zwracanego wyniku.
- Taką technikę nazywa się *przeciążaniem* metod (ang. overloading).

Przeciążanie metod

```
public class Punkt{
  int x, y;
  void wyswietl() {
    System.out.println("x = " + x + ", y = " + y);
  }
  void ustawXY(int wspX, int wspY){
    x = wspX;
    y = wspY;
  }
  void ustawXY(Punkt przekazany){
    x = przekazany.x;
    y = przekazany.y;
  }
}
```

```
public class Main {
  public static void main (String args[]) {
    Punkt A = new Punkt();
    A.ustawXY(100,100);
    Punkt B = new Punkt();
    B.ustawXY(A);
  }
}
```

Konstruktory

- Wiemy już, że po utworzeniu obiektu wszystkie jego pola zawierają wartości domyślne (np. pole typu int zostanie domyślnie wypełnione wartością 0).
- Najczęściej jednak oczekuje się, by pola te zawierały jakieś konkretne wartości, np. chcielibyśmy, aby każdy obiekt klasy Punkt otrzymywał: $x = 1$ i $y = 1$.

W tym celu można po każdym utworzeniu obiektu przypisywać wartości tym polom, np.:

```
Punkt A = new Punkt();
A.x = 1;
A.y = 1;
```

Można też dopisać do klasy Punkt dodatkową metodę, na przykład o nazwie inicjuj, postaci:

```
void inicjuj() {
  x = 1;
  y = 1;
}
```

i wywoływać ją po każdym utworzeniu obiektu.

Jednak żadne z tych rozwiązań nie jest wygodne. Przede wszystkim wymagają one, aby programista zawsze pamiętał o ich stosowaniu.

Konstruktory

Konstruktor jest to specjalna metoda, która zostaje wywołana zawsze w trakcie tworzenia obiektu w pamięci.

Metoda będąca konstruktorem

- nigdy nie zwraca żadnego wyniku,
- musi mieć nazwę zgodną z nazwą klasy.

Tworzenie konstruktora

```
class nazwa_klasy
{
  nazwa_klasy()
  {
    kod konstruktora
  }
}
```

Przed definicją nie umieszcza się słowa *void*, tak jak miałoby to miejsce w przypadku zwykłej metody.

Konstruktory

```
public class Punkt
{
  int x, y;
  Punkt ()
  {
    x = 1;
    y = 1;
  }
}
```

```
public class Main {
  public static void main (String args[]) {
    Punkt A = new Punkt ();
    System.out.println("Punkt: x = " + A.x + ", y = " + A.y);
  }
}
```

```
Punkt: x = 1, y = 1
```

Konstruktory

- Konstruktor nie musi być bezargumentowy, może również przyjmować argumenty.
- Argumenty przekazuje się dokładnie tak samo jak w przypadku zwykłych metod.

Tworzenie konstruktora

```
class nazwa_klasy
{
   nazwa_klasy(typ1 argument1, ..., typN argumentN)
   {
      kod konstruktora
   }
}
```

Jeśli konstruktor przyjmuje argumenty, przy tworzeniu obiektu należy je podać, czyli zamiast stosowanej do tej pory konstrukcji:

nazwa_klasy zmienna = **new** nazwa_klasy()

napiszemy:

nazwa_klasy zmienna = **new** nazwa_klasy(argumenty_konstruktora)

Konstruktory

```
public class Punkt
{
   int x, y;
   Punkt (int wspX, int wspY)
   {
      x = wspX;
      y = wspY;
   }
}
```

```
public class Main {
   public static void main (String args[]) {
      Punkt A = new Punkt (100, 100);
      System.out.println("Punkt: x = " + A.x + ", y = " + A.y);
   }
}
```

```
Punkt: x = 100, y = 100
```

Konstruktory, tak jak zwykłe metody, mogą być przeciążane, tzn. każda klasa może mieć kilka konstruktorów, o ile tylko różnią się one przyjmowanymi argumentami.

```
public class Punkt{
  int x, y;
  Punkt() {
    x = 1;
    y = 1;
  }
  Punkt(int wspX, int wspY){
    x = wspX;
    y = wspY;
  }
  Punkt(Punkt P){
    x = P.x;
    y = P.y;
  }
}
```

```
public class Main {
  public static void main (String args[]) {
    Punkt A = new Punkt();
    Punkt B = new Punkt(100, 100);
    Punkt C = new Punkt(A);
    System.out.println("A: x = " + A.x + ", y = " + A.y);
    System.out.println("B: x = " + B.x + ", y = " + B.y);
    System.out.println("C: x = " + C.x + ", y = " + C.y);
  }
}
```

```
public class Main {
  public static void main (String args[]) {
    Punkt A = new Punkt();
    Punkt B = new Punkt(100, 100);
    Punkt C = new Punkt(A);
    System.out.println("A: x = " + A.x + ", y = " + A.y);
    System.out.println("B: x = " + B.x + ", y = " + B.y);
    System.out.println("C: x = " + C.x + ", y = " + C.y);
  }
}
```

```
A: x = 1, y = 1
B: x = 100, y = 100
C: x = 1, y = 1
```

Słowo kluczowe **this**

Słowo kluczowe *this* to odwołanie do obiektu bieżącego. Można je traktować jako referencję do aktualnego obiektu.

Co by się stało, gdyby w konstruktorze

```
Punkt(int wspX, int wspY)
{
  x = wspX;
  y = wspY;
}
```

zmienić nazwy argumentów *wspX*, *wspY* na *x*, *y*:

```
Punkt(int x, int y)
{
  x = x;
  y = y;
}
```

- Powyższy zapis formalnie jest poprawny, jednak nie ma sensu.
- W jaki sposób kompilator ma rozróżnić, kiedy chodzi o argument konstruktora, a kiedy o pole klasy, jeśli ich nazwy są takie same?

Słowo kluczowe **this**

Poprawna postać konstruktora

```
Punkt(int x, int y)
{
  x = x;
  y = y;
}
```

jest następująca:

```
Punkt(int x, int y)
{
  this.x = x;
  this.y = y;
}
```

Jeśli chcemy zaznaczyć, że chodzi o składową klasy (pole, metodę), korzystamy z odwołania w postaci:

`this.nazwa_pola` lub `this.nazwa_metody`.

Instrukcję `this.x = x` rozumiemy jako: przypisz polu x wartość przekazaną jako argument o nazwie x.

Wywoływanie metod w konstruktorach

Z wnętrza konstruktora (tak jak z wnętrza każdej innej metody) można wywoływać inne metody.

```
public class Punkt{
  int x, y;

  Punkt() {
    ustawXY(1,1);
  }
  Punkt(int wspX, int wspY){
    ustawXY(wspX, wspY);
  }
  Punkt(Punkt P){
    ustawXY(P);
  }
  void ustawXY(int wspX, int wspY){
    x = wspX;
    y = wspY;
  }
  void ustawXY(Punkt P){
    x = P.x;
    y = P.y;
  }
}
```

Wywoływanie metod w konstruktorach

Czy w poniższym konstruktorze nie powinniśmy użyć słowa *this*, aby rozróżnić, czy chodzi o pola, czy o argumenty?

```
public class Punkt{
  int x, y;
  Punkt(int x, int y)
  {
    ustawXY(x, y);
  }
  void ustawXY(int wspX, int wspY)
  {
    x = wspX;
    y = wspY;
  }
}
```

- Konstruktor jest prawidłowy.
- Jeśli gdziekolwiek we wnętrzu metody odwołujemy się do nazwy argumentu, to niezależnie od tego, czy istnieje pole o takiej nazwie, czy nie, komputer zawsze przyjmuje, że chodzi o argument.
- W ciele metody argument ma większą siłę niż nazwa pola.

Wywoływanie konstruktora w konstruktorze

Wywołanie konstruktora z wnętrza innego konstruktora jest przydatne, gdy w klasie znajduje się kilka przeciążonych konstruktorów, a zakres wykonywanego przez nie kodu się pokrywa, tzn. gdy kod jednego z konstruktorów jest podzbiorem kodu innego.

Polecenie `this(argument1, ..., argumentN)`
wywoła konstruktor, którego argumenty pasują do wymienionych.

```
public class Liczby{
  int a;
  float b;

  Liczby(int x) {
    a = x;
  }
  Liczby(float x) {
    b = x;
  }
  Liczby(int a, float b) {
    this(a);
    this.b = b;
  }
}
```

Wywoływanie konstruktora w konstruktorze

Poniższego konstruktora

```
Liczby(int a, float b) {
  this(a);
  this.b = b;
}
```

nie możemy zmodyfikować następująco:

```
Liczby(int a, float b) {
  this(a);
  this(b);
}
```

Konstruktor można wywołać jawnie tylko w innym konstruktorze i musi on być pierwszą wykonywaną instrukcją. Oznacza to, że można wywołać tylko jeden konstruktor, a przed nim nie powinna się znaleźć żadna inna instrukcja.

Metoda **finalize***

- Wiemy już, że operator *new* pozwala na utworzenie obiektu, tzn. zarezerwowanie dla niego pamięci operacyjnej. Logicznym założeniem jest, że po jego wykorzystaniu pamięć tę należy zwolnić.
- W Javie za zwalnianie pamięci odpowiada maszyna wirtualna, a programista praktycznie nie ma nad tym procesem kontroli.
- Zajmuje się tym tak zwany odśmiecacz (ang. *garbage collector*), który czuwa nad optymalnym wykorzystaniem pamięci i uruchamia proces jej odzyskiwania w momencie, kiedy wolna ilość oddana do dyspozycji programu zbytnio się zmniejszy.
- Java jest jednak w stanie automatycznie zarządzać wykorzystywaniem pamięci, ale tylko tej, która jest alokowana standardowo, czyli za pomocą operatora *new*.
- Służy do tego metoda *finalize*, która jest wykonywana zawsze, kiedy obiekt jest niszczony/usuwany z pamięci.
- Wystarczy więc, że klasa będzie zawierała taką metodę, a przy niszczeniu obiektu zostanie ona wykonana. Wewnątrz tej metody można wykonać dowolne instrukcje.
- Deklaracja:

```
public void finalize(){
}
```

- Nie ma jednak żadnej gwarancji, że metoda ta zostanie wykonana w trakcie działania programu.
- Proces odśmiecania zaczyna się wtedy, kiedy garbage collector uzna to za stosowne. Może się więc okazać, że pamięć zostanie zwolniona dopiero po zakończeniu pracy aplikacji.

Metoda **finalize***

```
public class Test {
  public void finalize() {
    System.out.println("Niszczenie obiektu.");
  }
}
```

```
public class Main {
  public static void main (String args[]) {
    Test test = null;
    for(int i = 0; i < 5; i++)
    {
      test = new Test();
    }
    System.gc();
  }
}
```

```
Niszczenie obiektu.
Niszczenie obiektu.
Niszczenie obiektu.
Niszczenie obiektu.
```

```
public class Test {
  public void finalize() {
    System.out.println("Niszczenie obiektu.");
  }
}
```

```
public class Main {
  public static void main (String args[]) {
    Test test = null;
    for(int i = 0; i < 5; i++) {
      test = new Test();
    }
    System.gc();
  }
}
```

- W pętli pięć razy tworzymy nowy obiekt klasy Test i przypisujemy referencję do niego zmiennej test.
- Każde takie przypisanie powoduje utracenie poprzedniej wartości zmiennej test (czyli poprzedniej referencji) i oznaczenie wcześniej przypisanego obiektu jako nieużywanego.
- Po wykonaniu całej pętli zmienna test będzie zawierała referencję do ostatnio przypisanego, piątego obiektu, a wszystkie poprzednie obiekty zostaną utracone i będą mogły zostać poddane procesowi odśmiecania.
- Metoda *System.gc()* to sugestia dla maszyny wirtualnej, aby uruchomiła proces odzyskiwania pamięci. Nie ma stuprocentowej gwarancji, że zostanie on uruchomiony dokładnie w chwili wykonania tej instrukcji. Odbędzie się to w najbliższym możliwym