# Programowanie obiektowe

* Programowanie obiektowe jest stylem programowania, w którym do tworzenia programów używa się obiektów.

*obiekt = dane + metody*

* Styl taki powstał w wyniku postrzegania rzeczywistości jako zbioru obiektów różnego typu, które mogą wykonywać określone czynności, potrafią się ze sobą komunikować i na siebie wzajemnie oddziaływać.
* Obiekty w programie często odzwierciedlają cechy i umiejętności swoich odpowiedników ze świata rzeczywistego.

# Paradygmaty programowania obiektowego

* **Abstrakcja**

Każdy obiekt w systemie służy jako model abstrakcyjnego "wykonawcy", który może wykonywać pracę, opisywać i zmieniać swój stan, oraz komunikować się z innymi obiektami w systemie, bez ujawniania, w jaki sposób zaimplementowano dane cechy.

* **Enkapsulacja (hermetyzacja)**

Ukrywanie implementacji. Zapewnia, że obiekt nie może zmieniać stanu wewnętrznego innych obiektów w nieoczekiwany sposób. Tylko wewnętrzne metody klasy mogą zmieniać jego stanu. Każda klasa obiektu prezentuje swój "interfejs", który określa dopuszczalne metody współpracy.

* **Dziedziczenie**

Definiowanie i tworzenie specjalizowanych klas obiektów na podstawie bardziej ogólnych. Dla klas specjalizowanych nie trzeba redefiniować całej funkcjonalności, lecz tylko tą, której nie ma klasa ogólniejsza.

* **Polimorfizm**

Referencje i kolekcje obiektów mogą dotyczyć obiektów różnego typu, a wywołanie metody dla referencji spowoduje zachowanie odpowiednie dla pełnego typu obiektu wywoływanego.

# Klasy i obiekty

1. **Pola klasy** – zestaw cech (atrybutów) obiektów klasy,
2. **Metody** ­– zestaw operacji, które można wykonywać na obiektach klasy,
3. **Konstruktory** – specjalne operacje, które pozwalają na inicjowanie obiektów przy ich tworzeniu.

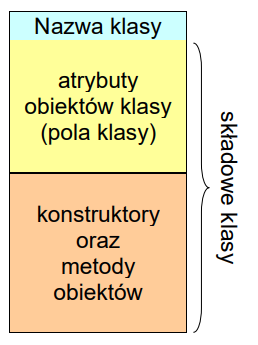
Java jest językiem obiektowym. Języki obiektowe posługują się pojęciem obiektu i klasy.

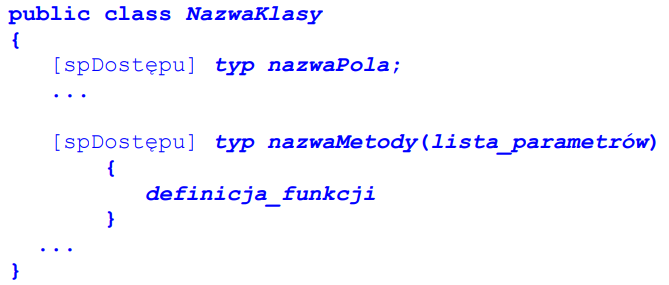
**Obiekt** to **konkretny** **lub** **abstrakcyjny** **byt**, wyróżnialny w modelowanej rzeczywistości, posiadający określone właściwości (atrybuty) oraz mogący świadczyć określone usługi (metody), czyli wykonywać określone działania lub przejawiać określone zachowania.   
Obiekty współdziałają ze sobą wymieniając komunikaty, które żądają wykonania określonych usług (metod).

**Klasa** to mający nazwę **opis pewnego rodzaju bytów** posiadających takie same cechy (byty te nazywamy obiektami lub instancjami klasy). Wspólne cechy to atrybuty (pola) poszczególnych obiektów oraz operacje (metody), które można na obiektach wykonywać.

Definicja klasy określa:

Pola i metody (wraz z konstruktorami) nazywane **składowymi klasy**.

Ogólna postać definicji klasy w języku Java:



*Uwagi:*

* modyfikator dostępu public przed słowem class może nie występować,
* modyfikatory [spDostępu] określają dostępność pól i metod.
* nagłówek i definicja metody w całości muszą znajdować się w klasie.
* definicja klasy nie jest zakończona średnikiem.

# Obiekty i referencje do obiektów

**Obiekty** są **instancjami** (egzemplarzami) klasy. Do obiektów można odwoływać się w programie za pomocą referencji.

**Referencja** to wartość, która oznacza lokalizację (**adres**) obiektu w pamięci.

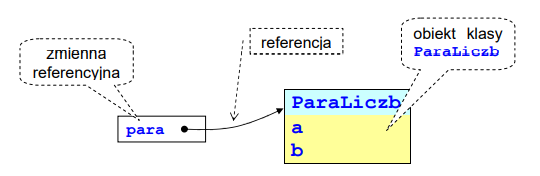
Referencje mogą być pamiętane w zmiennych referencyjnych, np.:

ParaLiczb para;

Zmienne referencyjne mogą zawierać referencje do obiektów lub nie zawierać żadnej referencji (nie wskazywać na żaden obiekt). Zmienna, która nie zawiera referencji do obiektu ma wartość null.

**Deklaracja zmiennej referencyjnej nie tworzy obiektu** tzn. nie wydziela pamięci do przechowywania obiektu klasy. **Obiekt musi być jawnie utworzony** za pomocą operatora new, który zwraca referencję do obiektu. Ta referencja może zostać przypisana zmiennej referencyjnej, np.:

para = new ParaLiczb();



Pola (atrybuty) klasy deklarujemy jako zmienne wewnątrz klasy. Deklaracja może zawierać modyfikator dostępu (np. private, protected lub public), oraz wyrażenie inicjujące, np.:

private float wartość = 100.0f;

Pola ustalone zawierają w deklaracji dodatkowy modyfikator final, np.:

final int ROZMIAR\_CZCIONKI = 14;

*Uwaga:*

Nazwy pól ustalonych zwykle piszemy DUŻYMI\_LITERAMI.

Pola klasy, które nie mają przypisanej wartości początkowej będą miały wartości domyślne:

* pola typu całkowitego (np. typu int) – liczbę 0,
* pola typu rzeczywistego (np. typu float) – liczbę 0.0
* pola typu logicznego – wartość false,
* pola typu referencyjnego – wartość null.

# Odwołania do pól klasy

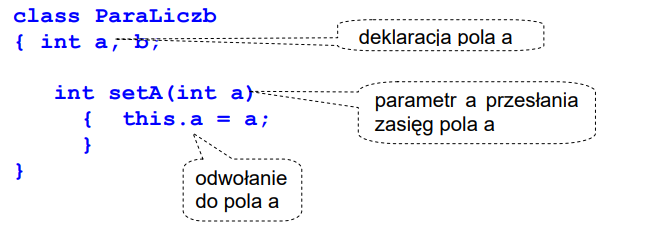
Do pól klasy odwołujemy się za pomocą operatora selekcji .

referencja\_do\_obiektu.nazwa\_pola

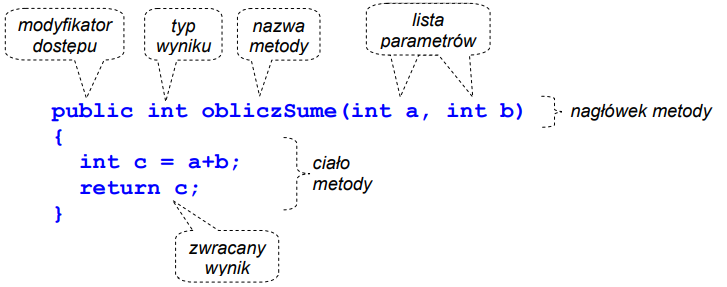
np. para.a

*Uwaga:*

Jeśli odwołujemy się do pola bieżącego obiektu (np. w metodzie wywołanej na rzecz tego obiektu), które zostało przesłonięte przez zmienną lokalną, to do pola można odwoływać się za pomocą słowa this np.:



# Definiowane metod w klasie



* nagłówek i ciało metody w całości muszą znajdować się w klasie.
* nazwę metody zaczynamy od małej litery i dalej stosujemy notację węgierską np. dodaj, obliczSume.
* modyfikator dostępu określa czy metoda może być wywoływana spoza klasy, w której jest zdefiniowana.
* typ wyniku określa typ danych zwracanych przez metodę. Jeśli metoda nic nie zwraca to zapisujemy void.
* Jeśli metoda zwraca wynik to zakończenie działania metody powinno następować na skutek instrukcji return.
* lista parametrów zawiera deklaracje parametrów, które są przekazywane do metody przy wywołaniu. Lista ta może być pusta (metoda bezparametrowa).

# Konstruktory

**Konstruktor** to specjalna metoda, która służy (głównie) do inicjowania pól obiektów.

Konstruktor:

* zawsze ma nazwę taką samą jak nazwa klasy,
* nie ma żadnego typu wyniku (nawet void),
* ma listę parametrów (w szczególności może być pusta).
* jest zawsze wywoływany za pomocą wyrażenia new

*Uwaga:*

W klasie może być zdefiniowanych wiele przeciążonych konstruktorów, które różnią się listą parametrów.

Jeśli w klasie nie zdefiniowano żadnego konstruktora to jest tworzony **domyślny konstruktor bezparametrowy**, który inicjuje pola obiektu wartościami domyślnymi.

Konstruktor domyślny nie jest dodawany, gdy w klasie zdefiniowano jakikolwiek inny konstruktor.

# Pola i metody statyczne

Wszystkie **pola niestatyczne** istnieją w każdym obiekcie będącym instancją klasy. tzn. każdy obiekt posiada własny indywidualny zestaw atrybutów opisujących jego właściwości.

**Pola statyczne** dotyczą całej klasy, a nie poszczególnych obiektów – są one pamiętane w specjalnym obszarze pamięci wspólnym dla całej klasy.

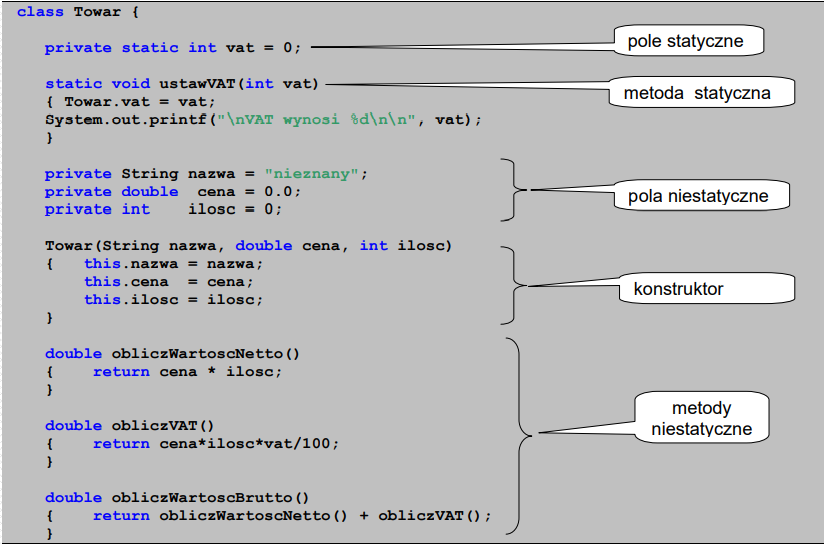
Składowe statyczne (pola i metody):

* są deklarowane przy użyciu specyfikatora static
* mogą być używane nawet wtedy, gdy nie istnieje żaden obiekt klasy.

Do składowych statycznych klasy odwołujemy się za pomocą operatora selekcji .

NazwaKlasy.nazwa\_składowej

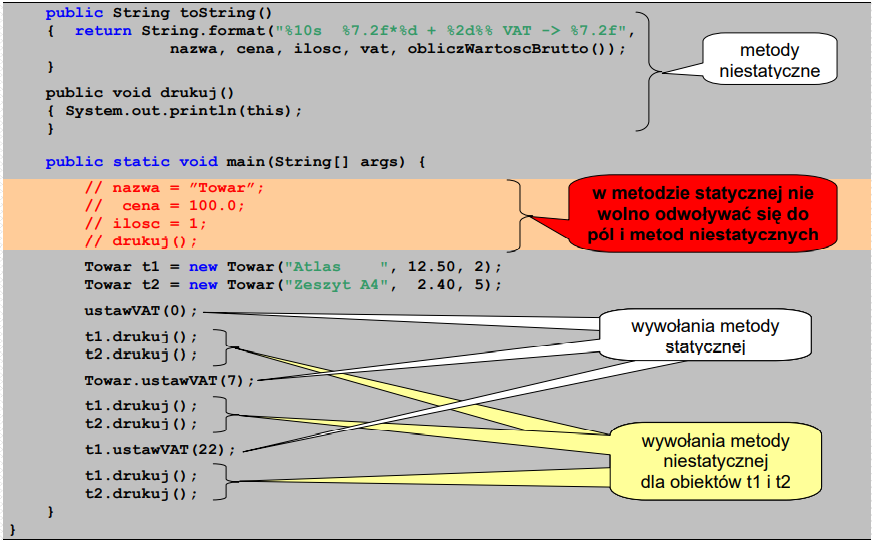
Jeżeli istnieje jakiś obiekt to do składowej statycznej można się również odwoływać tak, jak do zwykłej składowej (tzn. poprzez podanie referencji do obiektu)

referencja\_do\_obiektu.nazwa\_składowej

Wewnątrz klasy do składowych statycznych można odwoływać się w uproszczony sposób podając tylko ich nazwę.

*Uwaga:*

**Ze statycznych metod nie wolno odwoływać się do niestatycznych składowych klasy** podając ich nazwę (obiekt może nie istnieć). Możliwe są natomiast odwołania do innych składowych statycznych.



# Modyfikatory dostępu do składowych klasy

**Modyfikatory dostępu** pozwalają ukrywać dane i metody przed powszechnym dostępem.

W języku Java występują następujące modyfikatory:

private – składowe prywatne dostępne tylko z danej klasie.

protected – składowe chronione dostępne z danej klasy i wszystkich kasach ją dziedziczących

pubic – składowe publiczne dostępne z każdej klasie

(brak modyfikatora) – składowe zaprzyjaźnione dostępne ze wszystkich klas danego pakietu

# Hermetyzacja

Dane (pola klasy) są traktowane jako nierozdzielna całość z usługami (metodami klasy). Dodatkowe ograniczanie dostępu może znacznie zwiększyć odporność programu na błędy przez:

* **ochronę przed przypadkowym zepsuciem**

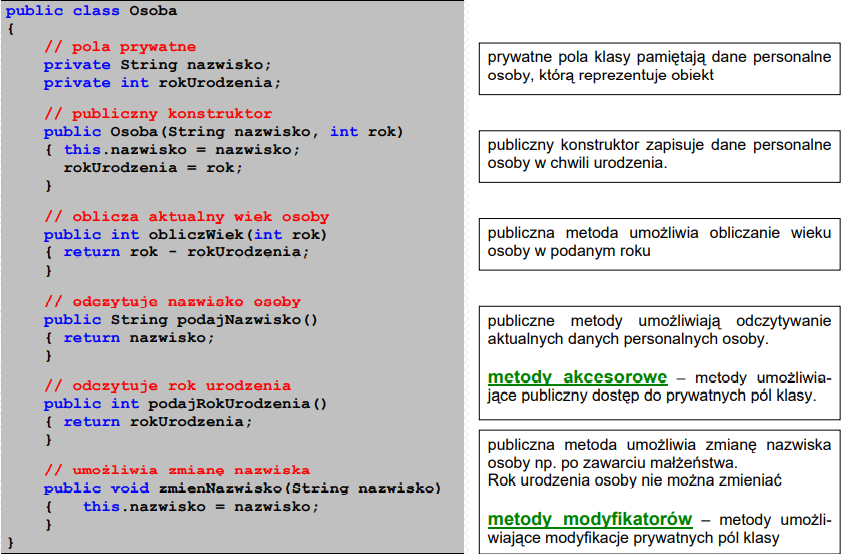
Użytkownik klasy nie ma dostępu do prywatnych pól i tym samym nic nie popsuje nieświadomie.

* **zapewnienie klarownego interfejsu programistycznego**

Użytkownik klasy ma do dyspozycji wyłącznie niezbędne metody, co ułatwia poprawne korzystanie z klasy.

* **umożliwienie zmian wewnątrz implementacji**

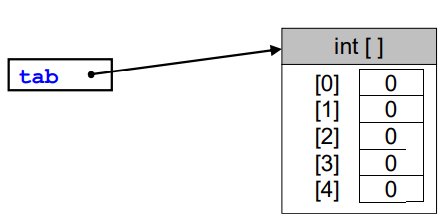
Twórca klasy może bezpiecznie modyfikować wewnętrzną implementację metod prywatnych. Użytkownicy klasy nie będą musieli dokonywać żadnych zmian w swoich programach.



# Tablice

**Tablice** **są zestawami elementów (wartości) tego samego typu**, ułożonych na określonych pozycjach. Do każdego z tych elementów mamy bezpośredni dostęp poprzez nazwę tablicy i indeks (numer) elementu.

Tablice zawsze są indeksowane od zera. Tablica n-elementowa ma indeksy od 0 do n-1.

W Javie **tablice są obiektami**, a **nazwa tablicy jest nazwą zmiennej referencyjnej** **do obiektu-tablicy**. Przykład:

int[] tab = new int[5];

Deklaracja tablicy składa się z:

* nazwy typu elementów tablic,
* pary nawiasów kwadratowych,
* nazwy zmiennej, która identyfikuje tablicę.

*Uwaga:*

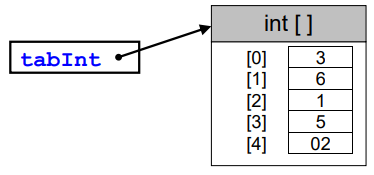
Rozmiar tablicy nie stanowi składnika deklaracji tablicy.

Przykład:

int[] arr; // deklaracja tablicy liczb całkowitych typu int

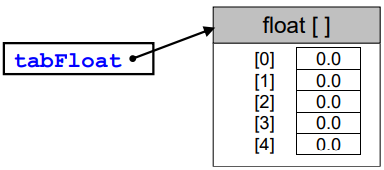
String [] napisy; // deklaracja tablicy referencji do obiektów klasy String

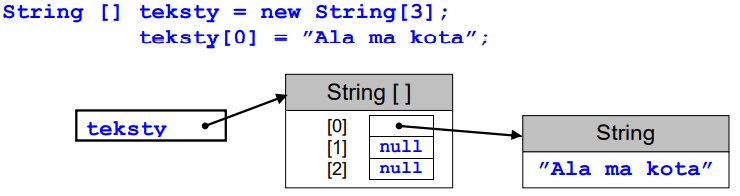
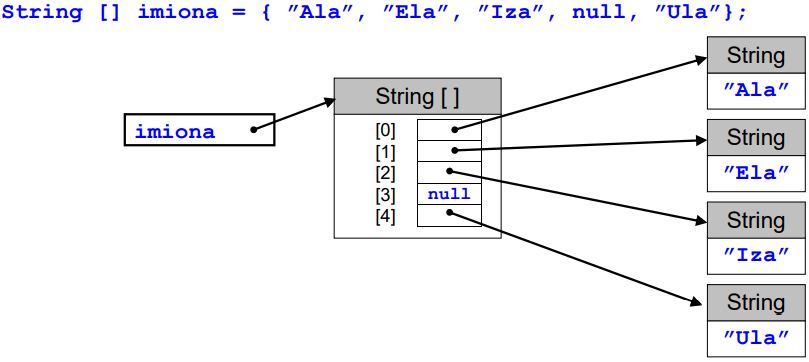
double[][] macierz; // deklaracja dwuwymiarowej tablicy liczb rzeczywistych

Sama deklaracja tablicy tworzy zmienną referencyjną, ale nie alokuje pamięci dla samej tablicy. **Pamięć** **jest alokowana dynamicznie w wyniku inicjacji** **za pomocą nawiasów klamrowych albo** **w wyniku użycia wyrażenia new**.

Przykład:

int[] tabInt = {3, 6, 1, 5, 2};

float [] tabFloat = new float[5];



Tablice w Javie mają **pole length, które pozwala odczytać rozmiar tablicy** za pomocą wywołania:

nazwa\_tablicy.length

**Po utworzeniu obiektu tablicy rozmiar nie może być zmieniany** !!!

Przykład:

String [] imiona = { ”Ala”, ”Ela”, ”Iza”, null, ”Ula”};

for (int i=0; i < imiona.length; i++)

if (imiona[i]!=null) System.out.println(imiona[i]);

Program drukuje wszystkie elementy zapamiętane w tablicy imiona.

# Argumenty metod

## Typy proste jako argumenty

Jak wiemy metodom można przekazywać wiele parametrów, poniżej znajdują się przykładowe metody, które odpowiednio:

* wyświetla napis podany jako parametr i go zwraca
* oblicza sumę trzech liczb typu int
* oblicza sumę trzech liczb różnych typów

String metodaNapis(String str){

System.out.println(str);

return str;

}

int sumaLiczb(int a, int b, int c){

return a+b+c;

}

double sumaLiczb2(int a, short b, double c){

return a+b+c;

}

Dodatkowo jak widać, aby zwrócić sumę nie musimy jej pośrednio obliczać w dodatkowej zmiennej, można to zrobić bezpośrednio po słówku return. Przy ostatniej metodzie należy z kolei pamiętać o zwracanym typie. Według zasad konwersji i rzutowania typów wiemy, że w przypadku dodawania różnych typów liczb dokonywana jest automatyczna konwersja do najogólniejszego typu - w naszym przypadku double - i taki musi być zadeklarowany typ metody.

Najważniejsze jednak w tym wszystkim jest to, że zmiany dokonane na argumentach, na przykład:

void wyswietl(int liczba){

liczba++;

System.out.println(liczba);

}

nie wpływają na jego oryginalną wartość! Jeśli przekażemy metodzie wyswietl() jakąś zmienną typu int to jej zmiana będzie dotyczyła wyłącznie wnętrza metody - w powyższym przykładzie wyświetli się liczba powiększona o 1, ale liczba przekazana metodzie pozostanie bez zmian. **W Javie wszystkie wartości przekazywane są przez wartość.**

Przykład:

Utwórzmy klasę Test, która przechowuje tylko jedną metodę zmieniającą argument za pomocą inkrementacji, dodatkowo niech będzie ona statyczna i nie zwraca wyniku (void). W drugiej klasie o nazwie Main utwórzmy zmienną całkowitoliczbową i zainicjujmy ją, następnie przekażmy ją do metody klasy Test i zobaczmy co się stanie.

class Test{

static void zwieksz(int liczba){

liczba++;

}

}

class Main{

public static void main(String[] args) {

int a = 5;

Test.zwieksz(a);

System.out.println(a);

}

}

Ponieważ metoda zwieksz() jest zadeklarowana jako statyczna to nie musimy tworzyć obiektu klasy Test, aby ją wywołać. Ale ważniejsze jest tutaj to, czy zmienna a uległa zmianie. Po skompilowaniu i uruchomieniu klasy Main widać, że nie. Co potwierdza słowa, że **typy proste przekazane jako argument nie są bezpośrednio modyfikowane**.

## Typy Obiektowe

Wcześniej wspomniano o tym, że argumenty w języku Java są przekazywane przez wartość, ale taką wartością są też referencje (jakaś liczba). Prościej mówiąc, jeśli przekażemy jakiś obiekt jako argument metody i zmodyfikujemy go w jej wnętrzu, to obiekt też zostanie zmodyfikowany, ponieważ operujemy na tym samym obszarze pamięci (kopii referencji wskazującej na ten sam obiekt). Utwórzmy podobny przykład do pierwszego, tylko zamiast zmiennej typu int użyjmy obiektu Punkt.

public class Punkt {

int x;

int y;

}

public class Test {

static void zmien(Punkt pkt){

pkt.x++;

pkt.y++;

}

}

public class Main{

public static void main(String args[]){

Punkt punkt = new Punkt();

punkt.x = 5;

punkt.y = 5;

Test.zmien(punkt);

System.out.println("Współrzędne to: "+punkt.x+" "+punkt.y);

}

}

Klasa Punkt przechowuje tylko dwie wartości typu int.

Klasa Test zmieniła się tylko o to, że zamiast liczby przyjmuje obiekt typu Punkt i zwiększa oba pola poprzez inkrementację.

W klasie Main tworzymy obiekt typu Punkt i inicjujemy pola x oraz y liczbami 5. Następnie wywołujemy statyczną metodę zmien() klasy Test. Spodziewamy się jak to było w poprzednim przykładzie, że tak naprawdę nie uległy one zmianie - w końcu to praktycznie to samo.

Po wyświetleniu współrzędnych widzimy jednak, że zostały one zwiększone.

Jest to ważne - **metodom przekazujemy referencję tak jak każdą inną wartość.** Jednak w metodach operujemy tak naprawdę na kopiach oryginalnych referencji - **przekazując jakąś referencję do obiektu jako parametr metody, na dany obiekt wskazują co najmniej 2 referencje**. Zmieniając argument metody (jego wewnętrzną strukturę), zmieniamy też zewnętrzny obiekt, który został tam przekazany, ale zmieniając referencję parametru metody (na przykład przypisując jej nowy obiekt) nie zmieniamy oryginalnej referencji, a tylko jej kopię.