## Programowanie obiektowe

# Definiowanie własnych klas

**Paweł Rogaliński**
Instytut Informatyki, Automatyki i Robotyki
Politechniki Wrocławskiej

**pawel.rogalinski@pwr.wroc.pl**

# *Abstrakcja*

**Świat rzeczywisty** jest bardzo złożony i nie jest możliwe dokładne opisanie budowy i działania wszystkich tworów, które się na niego składają

Opis rzeczywistości umożliwia **abstrakcja**, która jest procesem tworzenia pojęć, w którym wychodząc od rzeczy jednostkowych (najczęściej konkretnych) dochodzimy do pojęcia bardziej ogólnego poprzez konstatowanie tego, co dla tych rzeczy wspólne (zazwyczaj własności).

**Abstrakcją w programowaniu** nazywamy pewnego rodzaju uproszczenie rozpatrywanego problemu, polegające na ograniczeniu zakresu cech modelowanych rzeczy wyłącznie do cech kluczowych dla danego problemu, a jednocześnie niezależnych od implementacji.

# *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*

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.

# *Klasy i obiekty cd.*

Definicja klasy określa:

- **zestaw cech (atrybutów)** obiektów klasy,
- **zestaw operacji**, które można wykonywać na obiektach klasy,
- **specjalne operacje**, które pozwalają na inicjowanie obiektów przy ich tworzeniu.

Wspólne cechy (atrybuty) obiektów nazywane są **polami klasy**.

Operacje wykonywane na obiektach nazywane są **metodami**.

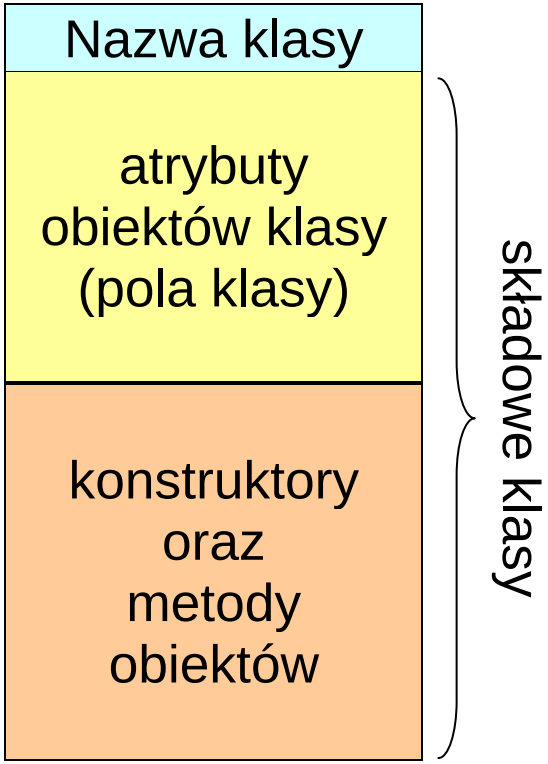
Specjalne operacje inicjalizacji przy tworzeniu obiektów nazywane są **konstruktorami.**

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

# *Klasy i obiekty cd.*

Klasę przedstawia się w formie prostokąta podzielonego na trzy części:

- górna część zawiera nazwę klasy,
- środkowa część przedstawia atrybuty obiektów,
- dolna część przedstawia konstruktory oraz metody obiektów.



# *Klasy i obiekty cd.*

Ogólna postać definicji klasy w języku Java:

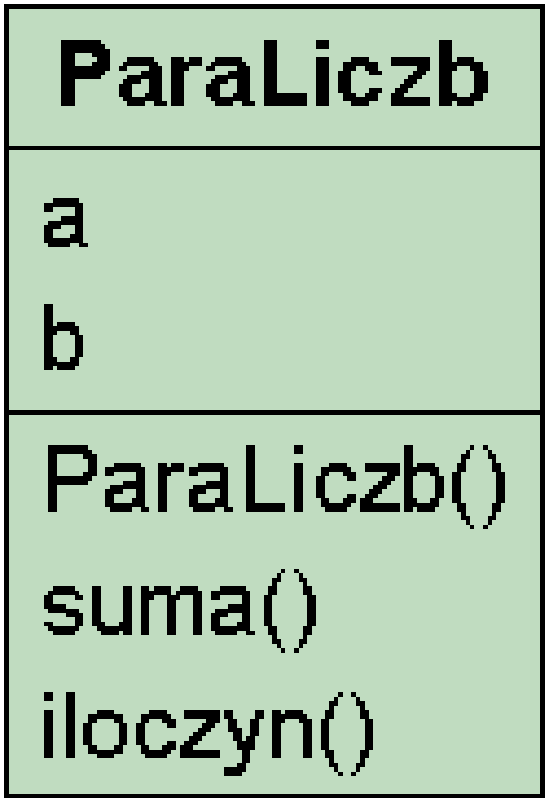
```
public class NazwaKlasy
{
   [spDostępu] typ nazwaPola;
   ...

   [spDostępu] typ nazwaMetody(lista_parametrów)
       {
          definicja_funkcji
       }
  ...
}
```

**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.

# *Przykładowa definicja klasy* `ParaLiczb`

```
class ParaLiczb
{
    // definicja pól
    int a;
    int b;

    // definicja konstruktora
    ParaLiczb()
    {
        a = 0;
        b = 0;
    }

    // definicja metody suma
    int suma()
    {
        return a+b;
    }

    // definicja metody iloczn
    int iloczyn()
    {
        return a*b;
    }
}
```



# *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`.

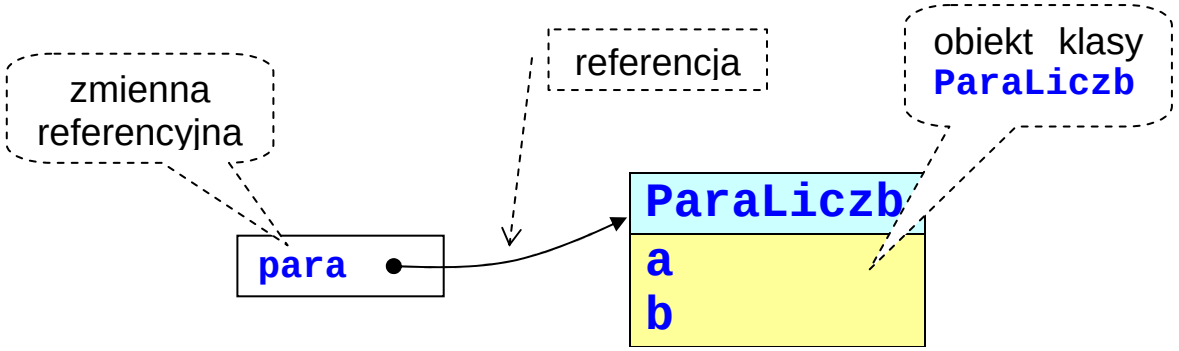
**Uwaga**:

- Zmiennej referencyjnej można przypisywać wyłącznie referencje do obiektu lub wartość `null`.
- Referencje można porównywać wyłącznie za pomocą operatorów `==` lub `!=`.

# *Obiekty i referencje do obiektów*

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();
```



# *Definiowanie pól klasy*

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;
```

**Uwaga:** nazwy pól zwykle piszemy małymi literami.

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 nie zostało przesłonięte, to można odwoływać się z pominięciem zmiennej referencyjnej i operatora selekcji `.` .

```
class ParaLiczb
{ int a, b;

   int getA()
     {  return a;
     }
}
```

odwołanie do pola a

# *Odwołania do pól klasy cd.*

**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.:

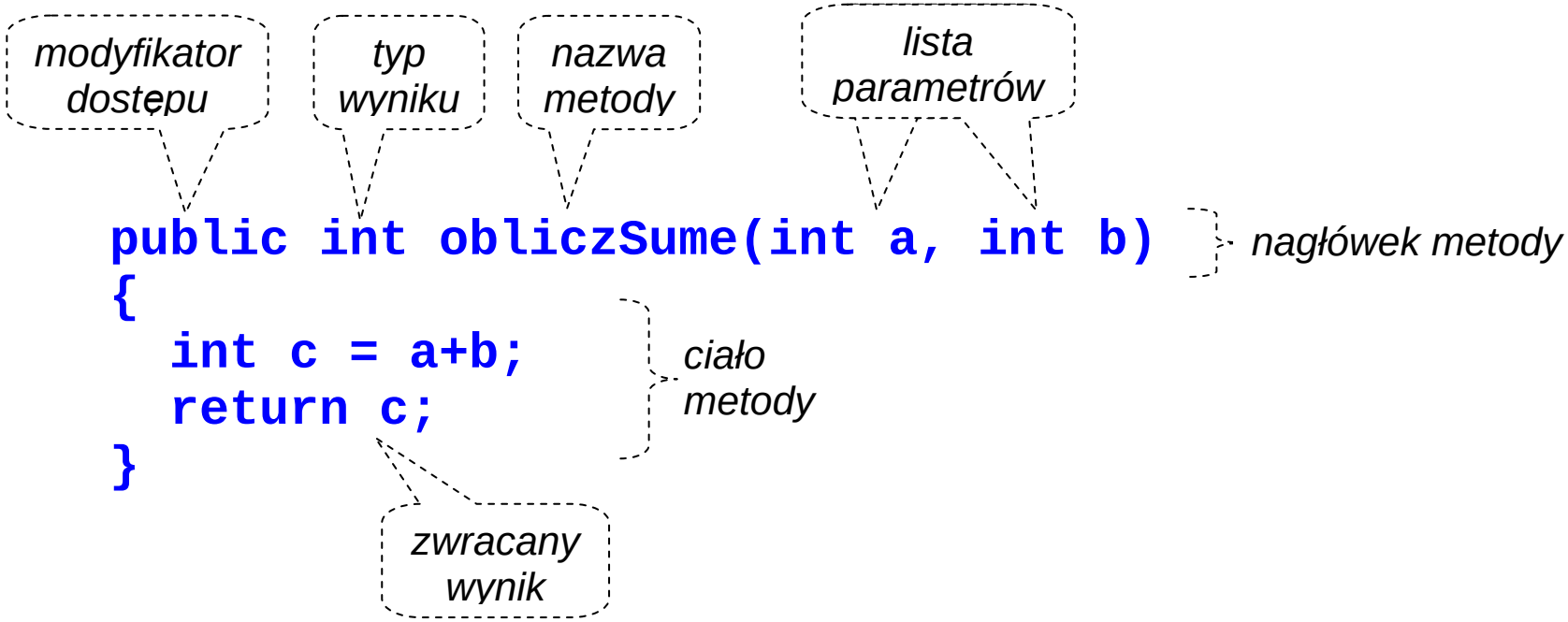
```
class ParaLiczb
{ int a, b;

   int setA(int a)
     {  this.a = a;
     }
}
```

deklaracja pola a

parametr a przesłania zasięg pola a

odwołanie do pola a

# 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.

# *Definiowane metod w klasie cd.*

- ***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.

# Konstruktory – przykład

```
class Towar {
    private String nazwa;
    private double  cena;
    private int    ilosc;

    public Towar()
    {   nazwa = "nieznany";
        cena  = 0.0;
        ilosc = 0;
    }

    public Towar(String nazwa)
    {    this();
         this.nazwa = nazwa;
    }

    public Towar(String nazwa, double cena, int ilosc)
    {   this(nazwa);
        this.cena  = cena;
        this.ilosc = ilosc;
    }

    public static void main(String [] args)
    { Towar t1, t2, t3, t4;

      t1 = new Towar();

      t2 = new Towar("Zeszyt");

      t3 = new Towar("Blok rysunkowy", 2.50, 5);
    }
}
```

wywołanie konstruktora bezparametrowego

wywołanie konstruktora z jednym parametrem

wywołanie konstruktora bezparametrowego

wywołanie konstruktora z jednym parametrem

wywołanie konstruktora z trzema parametrami

# *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 stanowią właściwości całej klasy,**

**a nie poszczególnych obiektów.**

Składowe statyczne (pola i metody):

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

# *Pola i metody statyczne cd.*

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.**

# Pola i metody statyczne – przykład

```
class Towar {

   private static int vat = 0;

   static void ustawVAT(int vat)
   { Towar.vat = vat;
   System.out.printf("\nVAT wynosi %d\n\n", vat);
   }

   private String nazwa = "nieznany";
   private double  cena = 0.0;
   private int    ilosc = 0;

   Towar(String nazwa, double cena, int ilosc)
   {   this.nazwa = nazwa;
       this.cena  = cena;
       this.ilosc = ilosc;
   }

   double obliczWartoscNetto()
   {    return cena * ilosc;
   }

   double obliczVAT()
   {    return cena*ilosc*vat/100;
   }

   double obliczWartoscBrutto()
   {    return obliczWartoscNetto() + obliczVAT();
   }
```

pole statyczne

metoda statyczna

pola niestatyczne

konstruktor

metody niestatyczne

# Pola i metody statyczne – przykład cd.

```
    public String toString()
    {  return String.format("%10s  %7.2f*%d + %2d%% VAT -> %7.2f",
                nazwa, cena, ilosc, vat, obliczWartoscBrutto());
    }
    public void drukuj()
    { System.out.println(this);
    }

    public static void main(String[] args) {
        // nazwa = ”Towar”;
        //  cena = 100.0;
        // ilosc = 1;
        // drukuj();
        Towar t1 = new Towar("Atlas    ", 12.50, 2);
        Towar t2 = new Towar("Zeszyt A4",  2.40, 5);
        ustawVAT(0);
        t1.drukuj();
        t2.drukuj();
        Towar.ustawVAT(7);
        t1.drukuj();
        t2.drukuj();
        t1.ustawVAT(22);
        t1.drukuj();
        t2.drukuj();
    }
}
```

metody niestatyczne

**w metodzie statycznej nie wolno odwoływać się do pól i metod niestatycznych**

wywołania metody statycznej

wywołania metody niestatycznej dla obiektów t1 i t2

# *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 interfejsy programistycznego**

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

- **umożliwienie zmian wewnętrzyj implementacji**

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

# Hermetyzacja – przykład

```
public class Osoba
{
    // pola prywatne
    private String nazwisko;
    private int rokUrodzenia;

    // publiczny konstruktor
    public Osoba(String nazwisko, int rok)
    { this.nazwisko = nazwisko;
      rokUrodzenia = rok;
    }

    // oblicza aktualny wiek osoby
    public int obliczWiek(int rok)
    { return rok - rokUrodzenia;
    }

    // odczytuje nazwisko osoby
    public String podajNazwisko()
    { return nazwisko;
    }

    // odczytuje rok urodzenia
    public int podajRokUrodzenia()
    { return rokUrodzenia;
    }

    // umożliwia zmianę nazwiska
    public void zmienNazwisko(String nazwisko)
    {   this.nazwisko = nazwisko;
    }
}
```

prywatne pola klasy pamiętają dane personalne osoby, którą reprezentuje obiekt

publiczny konstruktor zapisuje dane personalne osoby w chwili urodzenia.

publiczna metoda umożliwia obliczanie wieku osoby w podanym roku

publiczne metody umożliwiają odczytywanie aktualnych danych personalnych osoby.

**metody akcesorowe** – metody umożliwiające publiczny dostęp do prywatnych pól klasy.

publiczna metoda umożliwia zmianę nazwiska osoby np. po zawarciu małżeństwa.
Rok urodzenia osoby nie można zmieniać

**metody modyfikatorów** – metody umożliwiające modyfikacje prywatnych pól klasy

# *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.

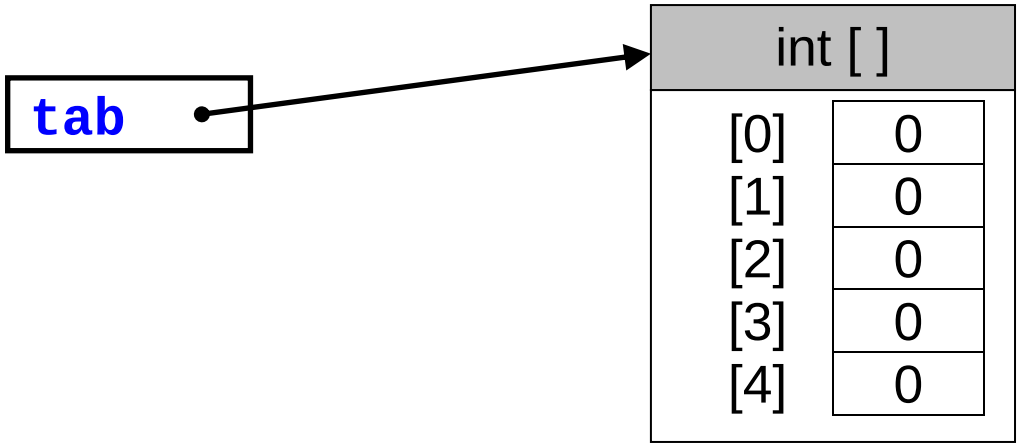
**Tablice zawsze są indeksowane od zera**.

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];
```



# *Tablice cd.*

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:

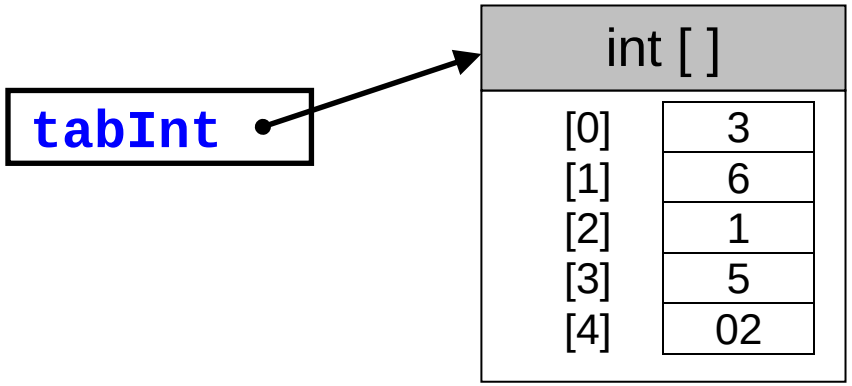
```
int[] arr;              // deklaracja tablicy liczb całkowitych typu int
String [] napisy;       // deklaracja tablicy referencji do obiektów klasy String
double[][] macierz;     // deklaracja dwuwymiarowej tablicy liczb rzeczywistych
```

# *Tablice cd.*

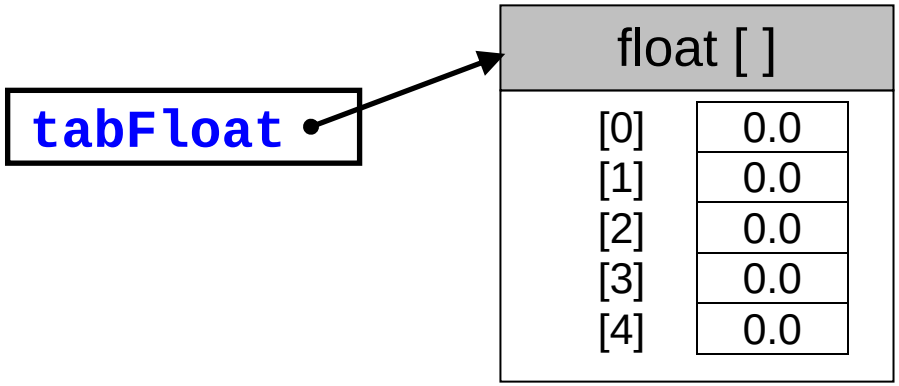
Sama deklaracja tablicy tworzy zmienną referencyjną, ale nie alokuje pamięci dla samej tablicy. Pamięć jest alokowana dynamicznie w wyniku inicjacji za pomocą nawiasów klamrowych albo w wyniku użycia wyrażenia **new**.

Przykład:

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

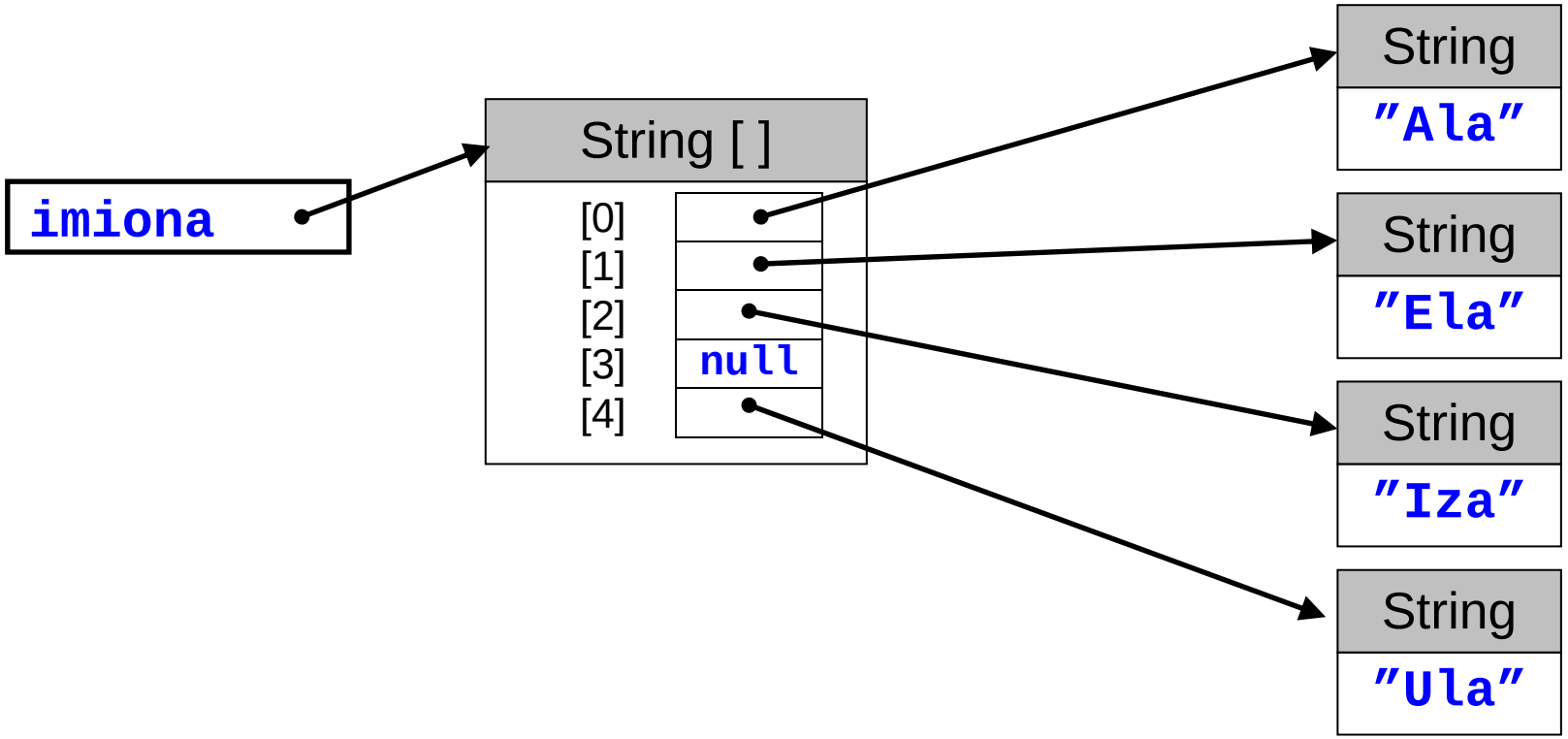



```
float [] tabFloat = new float[5];
```

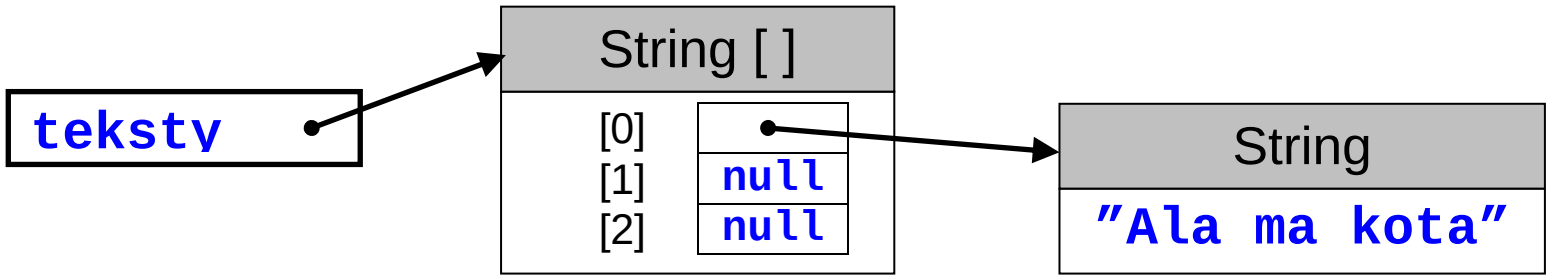


# Tablice cd.

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




```
String [] teksty = new String[3];
          teksty[0] = "Ala ma kota";
```



# *Tablice cd.*

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`**.