Programowanie współbieżne

Język Java – część 1
(streszczenie)

Paweł Rogaliński
Instytut Informatyki, Automatyki i Robotyki
Politechniki Wrocławskiej

pawel.rogalinski@pwr.wroc.pl

Komentarze w Javie

Komentarz wierszowy

```
// Program wypisujący tekst powitania
```

Komentarz blokowy

```
/* Program wypisujący tekst powitania
   Warszawa, 13 listopada 2002 r.
*/
```

Komentarz dokumentacyjny

```
/**
  * Klasa rysująca wykres. Typ wykresu
  * zależy od naciśniętego przycisku.
  * @version 1.0
  */
class Wykres {...
```


Tworzenie dokumentacji

Do opisu fragmentów kodu źródłowego programu używa się komentarzy. Na ich podstawie, używając programu ***javadoc*** można później wygenerować dokumentację. Najczęściej opisuje się elementy takie jak **klasy**, **interfejsy** oraz **metody** i **atrybuty** klas. Komentarze powinny być krótkie, precyzyjne. Należy je umieszczać bezpośrednio przed dokumentowanym elementem programu.

Polecenie wygenerowania dokumentacji ma postać:

```
javadoc nazwa_pliku.java
```

Jego wynikiem jest zbiór plików z opisem w formacie HTML.

Tworzenie dokumentacji

- Aby tekst komentarza został rozpoznany przez ***javadoc***, musi być umieszczony pomiędzy sekwencjami znaków `/**` i `*/`.
- Początkowe znaki `*` w kolejnych wierszach są pomijane.
- W tekście komentarza można umieszczać kod HTML np.

```
<ol>
<li> pierwszy element list
<li> drugi element listy
</ol>
```

- Każdy wiersz zawierający znak @, po którym następuje jeden ze znaczników dokumentacyjnych, powoduje utworzenie w dokumentacji oddzielnego paragrafu.

Znaczniki dokumentacyjne javadoc

@author – informacje o autorze programu,

@version – informacje o wersji programu,

@return – opis wyniku zwracanego przez metodę,

@serial – opis typu danych i możliwych wartości przyjmowanych przez zmienną,

@see – tworzy łącze do innego tematu,

@since – opis wersji, od której zaistniał określony fragment kodu,

@deprecated – informacje o elementach zdeprecjonowanych (które nie są zalecane),

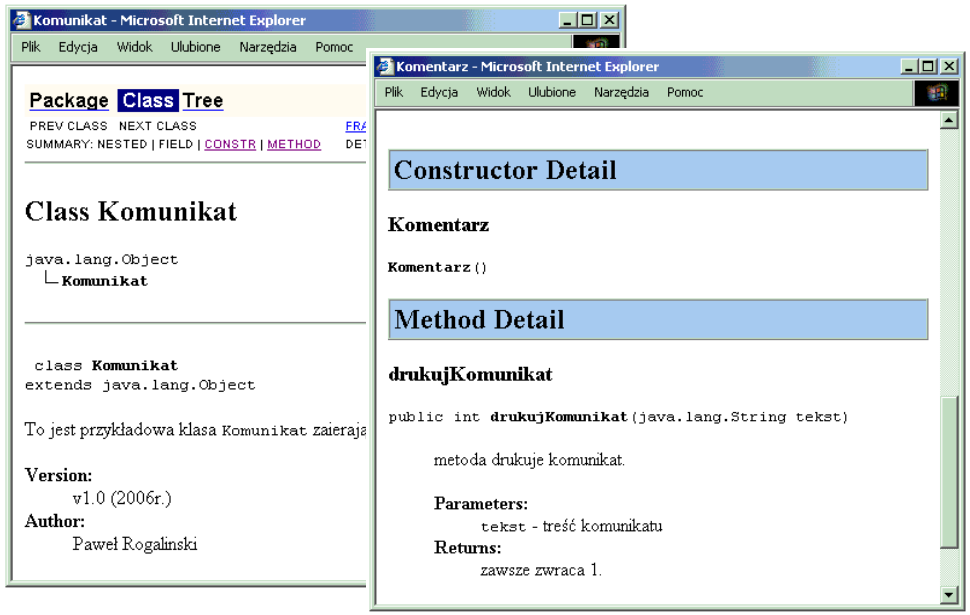
@param – opis parametru wywołania metody,

@exception – identyfikator wyjątku.

Przykład tworzenia dokumentacji

```
/**
 *  To jest przykładowa klasa <code>Komunikat</code>
 *  zawierająca komentarze <i>javadoc</i>.
 *  @author Paweł Rogalinski
 *  @version v1.0 (2006r.)
 */
class Komunikat
{
    /**
     *  metoda drukuje komunikat.
     *  @param tekst treść komunikatu
     *  @return zawsze zwraca 1.
     */
    public int drukujKomunikat(String tekst)
    { System.out.println(tekst);
      return 1;
    }
}
```


Przykład tworzenia dokumentacji



Słowa kluczowe

Słowa kluczowe to słowa, które mają specjalne znaczenie (np. oznaczają instrukcje sterujące) i nie mogą być używane w innym kontekście niż określa składnia języka.

abstract	default	if	package	synchronized
assert	do	implements	private	this
boolean	double	import	protected	throw
break	else	instanceof	public	throws
byte	extends	int	return	transient
case	false	interface	short	true
catch	final	long	static	try
char	finally	native	strictfp	void
class	float	new	super	volatile
const	for	null	switch	while
continue	goto			

Uwagi:

- słowa kluczowe **goto** i **const**, są zarezerwowane ale nie są używane.
- słowa **boolean**, **byte**, **char**, **double**, **float**, **int**, **long**, **short** są nazwami typów podstawowych.
- słowa **true**, **false** i **null** są nazwami stałych.

Identyfikatory

Identyfikatory to tworzone przez programistę nazwy **klas**, **pól** i **metod** klasy oraz **stałych** i **zmiennych**.

Identyfikator musi zaczynać się od litery lub podkreślenia i może składać się z dowolnych znaków alfanumerycznych (liter i cyfr) oraz znaków podkreślenia.

Java rozróżnia wielkie i małe litery w identyfikatorach

Identyfikator nie może pokrywać się ze słowami kluczowymi.

Zalecenia:

- Nazwy klas: wszystkie słowa w nazwie rozpoczynać dużą literą, np.: `ObiektGraficzny`
- Nazwy metod i pól publicznych: pierwsze słowo rozpoczynać małą literą, a kolejne wyrazy dużą literą, np.: `rysujTlo`, `kolorWypelnienia`.
- Nazwy metod i pól prywatnych: pisać wyłącznie małymi literami, a wyrazy łączyć podkreśleniem, np.: `kierunek_ruchu`.
- Nazwy zmiennych niemodyfikowalnych (stałych): pisać wyłącznie dużymi literami, a wyrazy łączyć podkreśleniem, np.: `ROZMIAR_TABLICY`.

Typy danych

Typ danej to zbiór jej możliwych wartości oraz zestaw operacji, które można na nich wykonywać. Jednocześnie określa on rozmiar pamięci potrzebny do przechowywania danej oraz sposób zapisu danej w pamięci komputera.

Język Java zawiera następujące typy danych:

- **typy proste**:
  - typy całkowite: `byte`, `short`, `int`, `long`,
  - typy rzeczywiste: `float`, `double`,
  - typ znakowy: `char`,
  - typ logiczny: `boolean`,
- **typ wyliczeniowy**
- **typ referencyjny** – nazwy typu referencyjnego pochodzą od nazwy klasy lub interfejsu. Wartością zmiennej typu referencyjnego jest referencja (odniesienie) do obiektu.

Dane w programie przedstawiamy za pomocą literałów, zmiennych oraz stałych.

Typy proste

Typy proste reprezentują pojedyncze wartości – nie są one złożonymi obiektami. Zapewnia to bardzo dużą wydajność przy wykonywaniu obliczeń.

Rozmiar i zakres wartości typów prostych

nazwa typu	zajętość pamięci	zakres wartości	wartość domyślna	znaczenie
`byte`	1	od -128 do 127	0	liczby całkowite
`short`	2	od -32768 do 32767	0	
`int`	4	od ok. -2×10^9 do ok. 2×10^9	0	
`long`	8	od ok. -9×10^{18} do ok. 9×10^{18}	0	
`float`	4	od ok. -3.4×10^{38} do ok. 3.4×10^{38}	0.0F	liczby rzeczywiste
`double`	8	od ok. -1.7×10^{308} do ok. 1.7×10^{308}	0.0D	
`char`	2	od 0 do 65535	‘x0’	znaki *unicode*
`boolean`	1	`false`,`true`	`false`	wartości logiczne

Opakowane typy proste

Każdy typ prosty posiada w języku Java swój odpowiednik w postaci klasy opakowującej, która umożliwia reprezentację zmiennej typu prostego w postaci obiektu. Klasy opakowujące mają nazwę taką jak typ prosty, ale pisaną z dużej litery np. `Byte`, `Long`, `Float`, `Double`.

Wyjątkiem jest typ `int`, który posiada klasę opakowującą `Integer`.

W wersji 1.5 do języka Java zostały dodane mechanizmy automatycznego pakowania i odpakowywania (ang. **autoboxing** i **auto-unboxing**), które umożliwiają bezpośrednią niejawną konwersję typu prostego do typu opakowanego i na odwrót.

```
float prostaLiczba;      // deklaracja zmiennej typu prostego
Float obiektLiczbowy;    // deklaracja zmiennej opakowanej
```

Dozwolone są np. instrukcje przypisania:

```
obiektLiczbowy = 3.14f;
prostaLiczba = obiektLiczbowy;
```

Opakowane typy podstawowe c.d.

Różnice między typami prostymi a typami opakowanymi:

- Typy proste mają wyłącznie swoją wartość,
- Opakowane typy proste nie są rozróżniane przez ich wartość (tzn. dwa różne obiekty opakowanych typów prostych mogą mieć tą samą wartość).

```
int z1 = 10;
int z2 = 10;

Integer o1 = new Integer(10);
Integer o2 = new Integer(10);
```

`z1==z2` zwróci wartość `true`

`o1==o2` zwróci wartość `false` (pomimo, że obiekty `o1` i `o2` posiadają tą samą wartość opakowaną)

`o1==z2` zwróci wartość `true`

Opakowane typy podstawowe c.d.

Różnice między typami prostymi a typami opakowanymi:

- Typy proste mają wyłącznie wartości funkcjonalne,
- Opakowane typy proste oprócz wartości funkcjonalnych mogą posiadać wartość niefunkcjonalną `null`.

```
int z1, z2;
Integer o1, o2;

z1 = 10;
z2 = null;

o1 = 10;
o2 = null;
```

Instrukcja błędna – zmiennej typu prostego nie można przypisać wartości `null`

Instrukcja dozwolona – zmiennej typu opakowanego prostego można przypisać wartość `null`

Opakowane typy podstawowe c.d.

Różnice między typami prostymi a typami opakowanymi:

- Typy proste są bardziej wydajne czasowo i pamięciowo w porównaniu do opakowanych typów prostych.

```
class WolnyProgram
{
   public static void main(String[] args)
      {
        Long sum = 0L;

        for(long i = 0;  i<Integer.MAX_VALUE; i++)
           {
             sum+=i;
           }
        System.out.println(sum);
      }
}
```

Dlaczego ten program jest powolny ?

Opakowane typy podstawowe c.d.

Różnice między typami prostymi a typami opakowanymi:

- Typy proste są bardziej wydajne czasowo i pamięciowo w porównaniu do opakowanych typów prostych.

```
class WolnyProgram
{
   public static void main(String[] args)
      {
        Long sum = 0L;

        for(long i = 0;  i<Integer.MAX_VALUE; i++)
           {
             sum+=i;
           }
        System.out.println(sum);
      }
}
```

Zmienna `sum` jest typu opakowanego, co powoduje w każdej iteracji pętli for konieczność wielokrotnego pakowania i rozpakowywania wartości

Ten program będzie działać kilkakrotnie szybciej jeśli zmienna `sum` będzie typu prostego tzn: `long sum = 0L;`

Dlaczego ten program jest powolny ?

Typy wyliczeniowe

- Wyliczenia tworzy się za pomocą słowa kluczowego **enum,** np.:

```
enum Kolor
{ Zielony, Zolty, Czerwony
}
```

- Identyfikatory **Zielony**, **Zolty**, **Czerwony** nazywany stałymi wyliczeniowymi.
  Są one publicznymi statycznymi składowymi wyliczenia i posiadają taki sam typ jak wyliczenie

- W programie można deklarować zmienne wyliczeniowe, którym można przypisywać stałe wyliczenia, np.:

```
Kolor kol;
kol = Kolor.Zielony.
```


Typy wyliczeniowe cd.

- Stałe wyliczeniowe można wykorzystywać w instrukcji warunkowej **if** oraz w instrukcji wyboru **switch**, np.:

```
if (kol==Kolor.Czerwony){ ... }

switch(kol)
   { case Zielony: System.out.print("GREEN");  break;
     case Zolty:   System.out.print("YELLOW"); break;
     case Czerwony: System.out.print("RED");   break;
   }
```

- Wszystkie wyliczenia automatycznie zawierają dwie predefiniowane metody:

```
public static typ-wyliczeniowy[] values()
public static typ-wyliczeniowy   valueOf(String tekst)
```

Metoda **values()** zwraca tablicę zawierającą listę stałych wyliczeniowych.
Metoda **valueOf()** zwraca stałą wyliczeniową, której odpowiada tekst przekazany jako argument.

Typy wyliczeniowe - przykład

```
class KoloryDemo
{
    // deklaracja typu wyliczeniowego
    enum Kolor
    { Zielony, Zolty, Czerwony
    }

    public static void main(String[ ] args)
    { // deklaracja zmniennej typu wyliczeniowego
      Kolor kolor;

      // użycie metody values()
      System.out.println("Oto wszystkie kolory");
      Kolor[] tab = Kolor.values();
      for(Kolor k : tab) System.out.println(k);

      // użycie metody valueOf()
      kolor = Kolor.valueOf("Zielony");

      // użycie stałej wyliczeniowej w instrukcji if
      if (kolor==Kolor.Zielony)  System.out.println("\n GREEN  \n");
      if (kolor==Kolor.Zolty)    System.out.println("\n YELLOW \n");
      if (kolor==Kolor.Czerwony) System.out.println("\n RED    \n");

      // użycie stałych wyliczeniowych w instrukcji switch
      switch(kolor)
      { case Zielony:
            System.out.println("kolor zielony - droga wolna");  break;
        case Zolty:
            System.out.println("kolor żółty - uwaga"); break;
        case Czerwony:
            System.out.println("kolor czerwony - stop"); break;
}   } }
```

```
Wiersz polecenia
C:\Testjava>java KoloryDemo
Oto wszystkie kolory
Zielony
Zolty
Czerwony

GREEN

kolor zielony - droga wolna
```


Operatory

Operatory są to specjalne symbole stosowane do wykonywania działań arytmetycznych, przypisań, porównań i innych operacji na danych.

Dane, na których są wykonywane operacje są nazywane argumentami. Operatory są jedno, dwu lub trzyargumentowe.
Uwaga: Niektóre operatory mogą być stosowane zarówno jako jednoargumentowe jak i dwuargumentowe np. **+** .

Każdy operator może być stosowany wyłącznie do danych określonego typu. Wynik działania operatora jest określonego typu.
Uwaga: Dla niektórych operatorów typ wyniku zależy od typu argumentów.

Wyrażenia tworzy się za pomocą operatorów i nawiasów ze zmiennych, stałych, literałów oraz wywołań metod. Wyrażenia są opracowywane (wyliczane), a ich wyniki mogą być w różny sposób wykorzystane np. w przypisaniach, jako argumenty innych operatorów, w instrukcjach sterujących wykonaniem programu, w wywołaniach metod, itd.

Operatory cd.

Kolejność opracowywania (wyliczania) wyrażeń zależy od priorytetów i wiązań operatorów użytych w tych wyrażeniach.

Priorytety mówią o tym, w jakiej kolejności będą wykonywane różne operacje w tym samym wyrażeniu.

Przykład: W wyrażeniu `a+b*c` najpierw będzie wykonane mnożenie, a potem dodawanie ponieważ operator `*` ma wyższy priorytet niż operator `+`.
Żeby odwrócić kolejność wykonywania działań trzeba użyć nawiasów: `(a+b)*c`

Wiązania określają kolejność wykonywania operacji o tym samym priorytecie tzn. czy są one wykonywane od lewej strony wyrażenia czy od prawej.

Przykład: W wyrażeniu `a-b+c` najpierw będzie wykonane odejmowanie, a potem dodawanie bo wiązanie operatorów `+` i `–` jest lewostronne.
Żeby odwrócić kolejność wykonywania działań trzeba użyć nawiasów: `a-(b+c)`

Zestawienie operatorów dostępnych w Javie

wiązanie i priorytet		operator	sposób użycia	działanie
lewe	1	`.`	`obiekt.składowa`	wybór składowej klasy
		`[ ]`	`tablica[wyrażenie]`	indeks tablicy
		`( )`	`metoda(lista wyrażeń)`	wywołanie metody
prawe	2	`++`	`zmienna++`	przyrostkowe / przedrostkowe zwiększenie o 1
			`++zmienna`	
		`--`	`zmienna--`	przyrostkowe / przedrostkowe zmniejszenie o 1
			`--zmienna`	
		`+`	`+wyrażenie`	jednoargumentowy plus,
		`-`	`-wyrażenie`	jednoargumentowy minus
		`!`	`!wyrażenie`	negacja logiczna
		`~`	`~wyrażenie`	dopełnienie bitowe
		`(typ)`	`(typ)wyrażenie`	rzutowanie typu
		`new`	`new typ`	tworzenie obiektu
lewe	3	`*`	`wyrażenie*wyrażenie`	mnożenie,
		`/`	`wyrażenie/wyrażenie`	dzielenie,
		`%`	`wyrażenie%wyrażenie`	modulo

wiązanie i priorytet		operator	sposób użycia	działanie
lewe	4	`+`	`wyrażenie+wyrażenie`	dodawanie, łączenie łańcuchów,
		`-`	`wyrażenie-wyrażenie`	odejmowanie
lewe	5	`<<`	`wyrażenie<<wyrażenie`	przesunięcie bitowe w lewo
		`>>`	`wyrażenie>>wyrażenie`	przesunięcie bitowe w prawo
		`>>>`	`wyrażenie>>>wyrażenie`	przes. bitowe w prawo bez znaku
lewe	6	`<`	`wyrażenie<wyrażenie`	mniejsze,
		`<=`	`wyrażenie<=wyrażenie`	mniejsze lub równe,
		`>`	`wyrażenie>wyrażenie`	większe,
		`>=`	`wyrażenie>=wyrażenie`	większe lub równe
		`instanceof`	`obiekt instanceof klasa`	stwierdzenie typu obiektu
	7	`==`	`wyrażenie==wyrażenie`	równość,
		`!=`	`wyrażenie!=wyrażenie`	nierówność
lewe	8	`&`	`wyrażenie&wyrażenie`	bitowe AND
	9	`^`	`wyrażenie^wyrażenie`	bitiwe OR wyłączające
	10	`|`	`wyrażenie|wyrażenie`	bitiwe OR
	11	`&&`	`wyrażenie&&wyrażenie`	logiczne AND
	12	`||`	`wyrażenie||wyrażenie`	logiczne OR
	13	`? :`	`wyraż ? wyraż : wyraż`	operator warunku

wiązanie i priorytet		operator	sposób użycia	działanie
prawe	14	`=`	`zmienna=wyrażenie`	proste przypisanie
		`*=`	`zmienna*=wyrażenie`	pomnóż i przypisz
		`/=`	`zmienna/=wyrażenie`	podziel i przypisz
		`%=`	`zmienna%=wyrażenie`	oblicz modulo i przypisz
		`+=`	`zmienna+=wyrażenie`	dodaj i przypisz
		`-=`	`zmienna-=wyrażenie`	odejmij i przypisz
		`<<=`	`zmienna<<=wyrażenie`	przesuń w lewo i przypisz
		`>>=`	`zmienna>>=wyrażenie`	przesuń w prawo i przypisz
		`>>>=`	`zmienna>>>=wyrażenie`	przesuń w prawo bez znaku i przypisz
		`&=`	`zmienna&=wyrażenie`	koniunkcja bitowa i przypisz
		`^=`	`zmienna^=wyrażenie`	różnica bitowa i przypisz
		`|=`	`zmienna|=wyrażenie`	alternatywa bitowa i przypisz

Operatory przypisania

Operator przypisania **=** oblicza wartość wyrażenia po prawej stronie, a następnie przypisuje obliczoną wartość do zmiennej umieszczonej po lewej stronie.

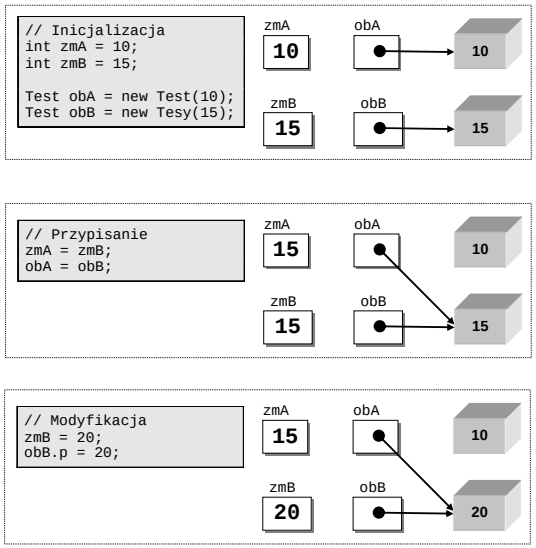
Uwaga: Działanie operatora dla typów prostych jest zgodne z intuicją.
Jeśli **a** i **b** są zmiennymi typu prostego to instrukcja **a=b** powoduje skopiowanie wartości zmiennej **b** do **a**. Późniejsza modyfikacja zmiennej **b** nie wpływa na wartość zmiennej **a**.

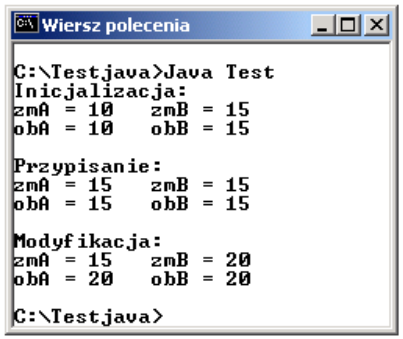
Jeśli zmienne **a** i **b** są typu referencyjnego (zawierają odwołanie do obiektu) to wykonanie instrukcji **a=b** powoduje skopiowanie do zmiennej **a** referencji do obiektu wskazywanego przez zmienną **b**. W efekcie zmienne **a** i **b** wskazują na ten sam obiekt. Późniejsza modyfikacja obiektu wskazywanego przez **b** powoduje również modyfikację obiektu wskazywanego przez **a**.

Operator przypisania - przykład dla typów prostych i referencyjnych

```
class Test
{   int p;

    Test(int p){ this.p=p; }

    public String toString(){ return ""+p; }

    public static void main(String[] args){

        System.out.println("Inicjalizacja:");
        int zmA = 10;
        int zmB = 15;
        Test obA = new Test(10);
        Test obB = new Test(15);
        System.out.println("zmA = " +zmA+ "   zmB = "+zmB);
        System.out.println("obA = " +obA+ "   obB = "+obB);

        System.out.println("\nPrzypisanie:");
        zmA = zmB;
        obA = obB;
        System.out.println("zmA = " +zmA+ "   zmB = "+zmB);
        System.out.println("obA = " +obA+ "   obB = "+obB);

        System.out.println("\nModyfikacja:");
        zmB = 20;
        obB.p = 20;
        System.out.println("zmA = " +zmA+ "   zmB = "+zmB);
        System.out.println("obA = " +obA+ "   obB = "+obB);
}     }
```


Operator przypisania - przykład dla typów prostych i referencyjnych





Rodzaje instrukcji w języku Java

Instrukcja pusta – nie powoduję wykonania żadnych działań np. **;**

Instrukcje wyrażeniowe:

- **przypisanie** np. `a = b;`
- **preinkrementacja** np. `++a;`
- **predekrementacja** np. `--b;`
- **postinckrementacja** np. `a++;`
- **postdekrementacja** np. `b--;`
- **wywołanie metody** np. `x.metoda();`
- **wyrażenie *new*** np. `new Para();`

Uwaga: instrukcja wyrażeniowa jest zawsze zakończona średnikiem.

Instrukcja grupująca – dowolne instrukcje i deklaracje zmiennych ujęte w nawiasy klamrowe np.

```
{ int a,b;
  a = 2*a+b;
}
```

Uwaga: po zamykającym nawiasie nie stawiamy średnika.

Rodzaje instrukcji w języku Java cd.

Instrukcja etykietowana – identyfikator i następujący po nim dwukropek wskazujący instrukcje sterującą `switch`, `for`, `while` lub `do`.

Instrukcja sterująca – umożliwia zmianę sekwencji (kolejności) wykonania innych instrukcji programu. Rozróżniamy instrukcje:

- **warunkowe:** `if`, `if ... else`, `switch`
- **iteracyjne:** `for`, `while`, `do ... while`
- **skoku:** `break`, `continue`, `return`

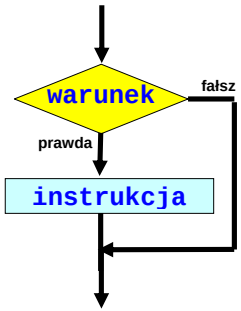
Instrukcja throw – zgłaszanie wyjątku przerywającego normalny tok działania programu.

Instrukcja synchronized – wymuszanie synchronizacji przy współbieżnym wykonywaniu różnych wątków programu

Postać instrukcji warunkowej `if`

Instrukcja warunkowa `if` służy do zapisywania decyzji, gdzie wykonanie *instrukcji* jest uzależnione od spełnienia jakiegoś *warunku*.

```
if (warunek)
   {
    instrukcja;
   }
```

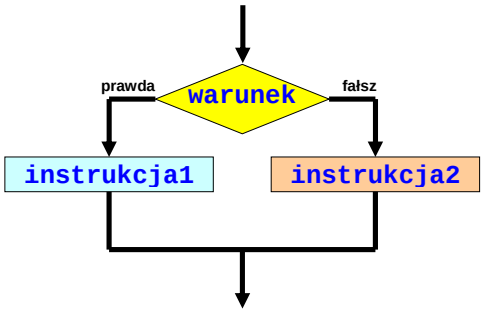


Postać instrukcji warunkowej `if - else`

Instrukcja warunkowa `if ... else` służy do zapisywania decyzji, gdzie wykonanie jednej z alternatywnych instrukcji zależy od spełnienia jakiegoś warunku.

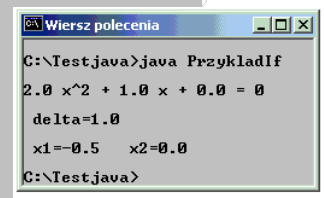
Jeśli *warunek* jest prawdziwy to wykonywana jest *instrukcja1*, w przeciwnym wypadku wykonywana jest *instrukcja2*.

```
if (warunek)
   {
    instrukcja1;
   }
   else
   {
    instrukcja2;
   }
```



Przykład - instrukcja `if ... else`

```
import javax.swing.JOptionPane;

class PrzykladIf
{
    public static void main(String[] args)
    {  double a, b, c, delta, x1, x2;

       a = Double.parseDouble(JOptionPane.showInputDialog(null, "Podaj a:"));
       b = Double.parseDouble(JOptionPane.showInputDialog(null, "Podaj b:"));
       c = Double.parseDouble(JOptionPane.showInputDialog(null, "Podaj c:"));
       System.out.println("\n" + a + " x^2 + " + b + " x + " + c + " = 0");

       if (a<0)
          { System.out.println("\n To nie jest równanie kwadratowe");
            System.exit(0);
          }

      delta = b*b-4*a*c;
      System.out.println("\n delta=" + delta);

      if (delta>0){
          x1 = (-b - Math.sqrt(delta))/(2*a);
          x2 = (-b + Math.sqrt(delta))/(2*a);
          System.out.println("\n x1=" + x1 + "   x2=" + x2);
        }
      else if (delta==0){
          x1 = -b/(2*a);
          System.out.println("\n x1=" + x1);
        }
      else System.out.println("\n To równanie nie ma pierwiastków");
}   }
```

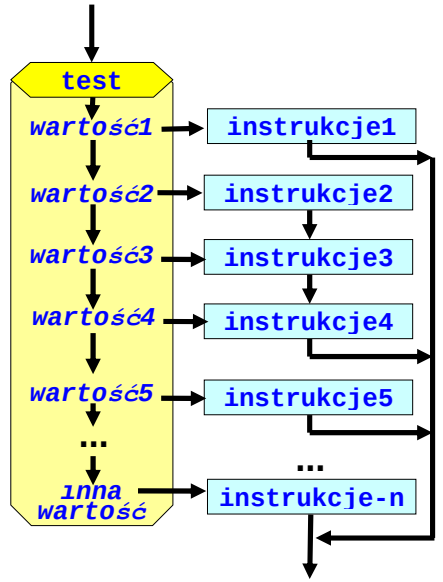


Postać instrukcji wyboru `switch`

Instrukcja wyboru **`switch`** pozwala zapisywać decyzje, kiedy to o wyborze jednej z alternatyw decyduje wartość skalarna jakiegoś *wyrażenia testowego*. Wyrażenie to musi być typu **całkowitego**, **znakowego** lub **wyliczeniowego**. Jego wynik jest porównywany z *wyrażeniami stałymi* (np. literałami) występującymi po słowie kluczowym **`case`**. W przypadku zgodności wykonywana jest odpowiednia *instrukcja po dwukropku* i następujące po niej kolejne instrukcje aż do napotkania instrukcji **`break`** lub **`return`**. Jeśli żadne z wyrażeń stałych po słowie **`case`** nie jest zgodne z wartością *wyrażenia testowego* to wykonywana jest *instrukcja po klauzuli* **`default`**.

Postać instrukcji wyboru `switch`

```
switch (test)
   { case wartość1:
             instrulcje1;
             break;
     case wartość2:
             instrukcje2;
     case wartość3:
             instrukcje3;
     case wartość4:
             instrukcje4;
             break;
     case wartość5:
             instrukcje5;
             break;
       ...     ...
     default:
             instrukcje-n;
   }
```



Przykład – instrukcja `switch`

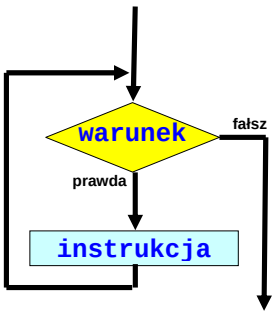
```
import  javax.swing.JOptionPane;

class PrzykladSwitch{
    public static void main(String[] args){
         double a, b;
         char oper;

         a=Double.parseDouble(JOptionPane.showInputDialog(null, "Podaj a"));
         b=Double.parseDouble(JOptionPane.showInputDialog(null, "Podaj b"));
         oper=JOptionPane.showInputDialog(null, "Podaj działanie").charAt(0);
         System.out.println("a=" + a + "   b=" + b + "    oper=" + oper);

         switch(oper){
            case '+': System.out.println("   Suma wynosi " + (a+b));
                      break;
            case '-': System.out.println("Roznica wynosi " + (a-b));
                      break;
            case '*': System.out.println("Iloczyn wynosi " + (a*b));
                      break;
            case '/': System.out.println(" Iloraz wynosi " + (a/b));
                      break;
            default: System.out.println("Nieznana operacja");
         }
         System.out.println("Koniec \n");
    }
}
```


Postać pętli `while`

W nagłówku pętli **`while`** zapisywany jest *warunek*, który jest testowany przed wykonaniem każdej iteracji. Dopóki ten *warunek* jest prawdziwy, powtarzane jest wykonanie *instrukcji*. Gdy *warunek* nie jest spełniony wykonanie pętli kończy się.

Uwaga: Jeśli *warunek* nie będzie spełniony już na wstępie, to *instrukcja* w pętli **`while`** nie będzie wykonana ani razu

```
while (warunek)
    {
     instrukcja;
    }
```

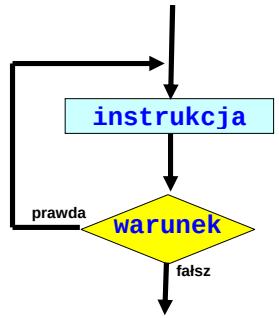


Postać pętli `do ... while`

Pętla **`do ... while`** służy do zapisywania iteracji wykonywanej przynajmniej raz. *Instrukcja* w pętli jest wykonywana, po czym sprawdzany jest *warunek powtórzenia*. Jeśli *warunek* jest spełniony to *instrukcja* w pętli jest wykonywana ponownie. W przeciwnym razie wykonanie pętli kończy się.

Uwaga: *Instrukcja* w pętli **`do ... while`** zawsze wykona się co najmniej jeden raz.

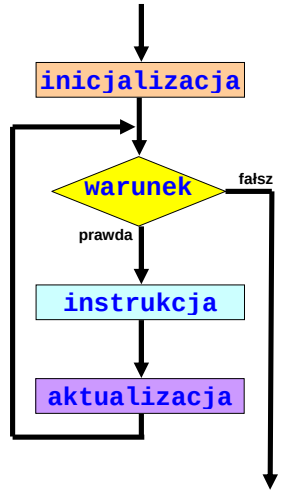
```
do
   {
    instrukcja;
   }
while(warunek);
```



Postać pętli `for`

W nagłówku pętli **`for`** podawane są: *inicjalizacja*, *warunek powtórzenia* oraz *aktualizacja*. *Inicjalizacja* jest wykonywana przed rozpoczęciem wykonywania pętli. *Warunek* jest sprawdzany przed każdą iteracją i jeśli jest spełniony wykonywana jest *instrukcja* wewnątrz pętli i następująca po niej *aktualizacja*. W przeciwnym razie pętla jest przerywana.

```
for(inicjalizacja; warunek; aktualizacja)
   {
    instrukcja;
   }
```



Postać pętli `for-each`

W wersji 1.5 języka Java wprowadzono uproszczoną postać pętli for, która umożliwia szybkie przeglądanie wszystkich elementów tablic oraz kolekcji.

```
for(Element e = elements)
   {
    Instrukcja;
   }
```

`Element` – dowolna klasa reprezentująca obiekty przechowywane w tablicach lub kolekcjach.

`elements` – tablica lub dowolna kolekcja w której są pamiętane obiekty klasy **`Element.`**

`e` – zmienna sterująca pętli for, która przyjmuje kolejno wartości wszystkich elementów pamiętanych w kolekcji **`elements`**.

Postać pętli `for-each`

Przykład: porównanie użycia pętli **`for`** oraz **`for-each`**

```
class Argumenty
 {
   public static void main(String[] args)
   {
      System.out.println("Argumenty: ");

      for (int i=0; i< args.length; ++i)    // zwykła pętla for użyta do
        System.out.println(args[i]+ "\t");  // przeglądania elementów tablicy


      System.out.println("Argumenty: ")

      for (String s : args)                 // pętla for-each
        System.out.println( s + "\t");
   }
 }
```

Porównanie instrukcji iteracyjnych

Pętle **`while`** oraz **`do ...while`** stosujemy zwykle wtedy, gdy kontynuacja działania pętli zależy od jakiegoś warunku, a liczba iteracji nie jest z góry znana lub jest trudna do określenia.

Pętla **`for`** jest stosowana najczęściej przy organizacji pętli iteracyjnych ze znanym zakresem iteracji.

Pętle **`for`** można łatwo przekształcić na pętlę **`while`**. Ilustruje to poniższe zastawienie:

```
for (inicjalizacja; warunek; aktualizacja)
   {
     instrukcja;
   }
```

```
inicjalizacja;
while(warunek)
   {
     instrukcja;
     aktualizacja;
   }
```


Przerywanie pętli – instrukcja `break`

Instrukcja **`break`** powoduje przerwanie wykonywania pętli. W przypadku pętli zagnieżdżonych przerywana jest pętla wewnętrzna, w której bezpośrednio znajduje się instrukcja **`break`**.

Jeśli po instrukcji **`break`** występuje etykieta, to przerywana jest ta pętla lub blok instrukcji, która jest opatrzona tą etykietą.

Uwaga: etykieta musi być umieszczona bezpośrednio przed pętlą lub blokiem instrukcji, które mają być przerwane.

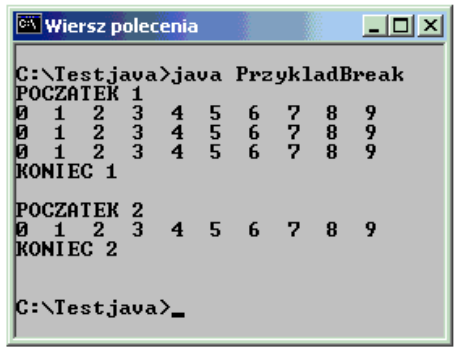
Instrukcja **`break`** stosowana jest również do opuszczania instrukcji **`switch`**.

Przykład – instrukcja `break`

```
class PrzykladBreak{

    public static void main(String[] args)
    {
        System.out.println("POCZATEK 1");
        for(int i=0; i<3; i++)
        {
            for(int j=0; j<100; j++)
            {
                if (j==10) break;
                System.out.print(j + "  ");
            }
            System.out.println();
        }
        System.out.println("KONIEC 1\n");

        System.out.println("POCZATEK 2");
   etykieta:
        for(int i=0; i<3; i++)
        {
            for(int j=0; j<100; j++)
            {
                if (j==10) break etykieta;
                System.out.print(j + "  ");
            }
            System.out.println();
        }
        System.out.println("\nKONIEC 2\n");
    }
}
```


Przykład – instrukcja `break`

```
class PrzykladBreak{

    public static void main(String[] args)
    {
        System.out.println("POCZATEK 1");
        for(int i=0; i<3; i++)
        {
            for(int j=0; j<100; j++)
            {
                if (j==10) break;
                System.out.print(j + "  ");
            }
            System.out.println();
        }
        System.out.println("KONIEC 1\n");

        System.out.println("POCZATEK 2");
   etykieta:
        for(int i=0; i<3; i++)
        {
            for(int j=0; j<100; j++)
            {
                if (j==10) break etykieta;
                System.out.print(j + "  ");
            }
            System.out.println();
        }
        System.out.println("\nKONIEC 2\n");
    }
}
```



Kontynuowanie pętli – instrukcja `continue`

Instrukcja **`continue`** przerywa wykonywanie bieżącego kroku pętli i wznawia wykonanie kolejnej iteracji. W przypadku pętli zagnieżdżonych działanie to dotyczy tej pętli wewnętrznej, w której jest umieszczona instrukcja **`continue`**.

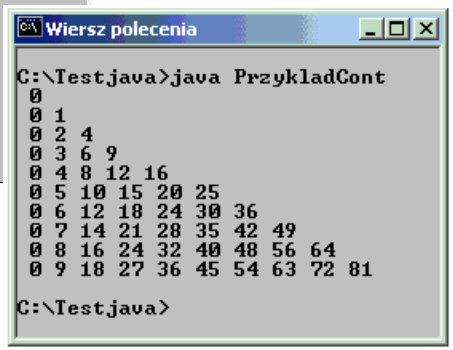
Jeśli po instrukcji **`continue`** występuje etykieta, to wznawiana jest iteracja tej pętli, która jest opatrzona tą etykietą.

Przykład – instrukcja `continue`

```
class PrzykladBreak{

class PrzykladCont{

    public static void main(String[] args)
    {
    etykieta:
        for(int i=0; i<10; i++)
           {
            for(int j=0; j<10; j++)
               {
                if (j>i)
                   {
                    System.out.println();
                    continue etykieta;
                   }
                System.out.print(" " + (i*j));
               }
           }
        System.out.println();
    }
}
```


Przykład – instrukcja `continue`

```
class PrzykladBreak{

class PrzykladCont{

    public static void main(String[] args)
    {
    etykieta:
        for(int i=0; i<10; i++)
           {
            for(int j=0; j<10; j++)
               {
                if (j>i)
                   {
                    System.out.println();
                    continue etykieta;
                   }
                System.out.print(" " + (i*j));
               }
           }
        System.out.println();
    }
}
```



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.

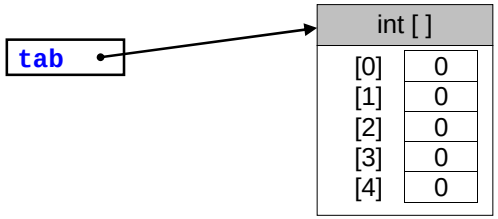
Tablice zawsze są indeksowane od zera.

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:

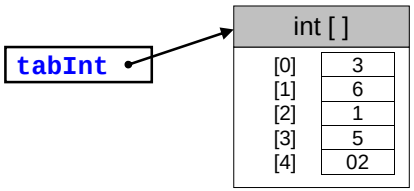
```
int[] arr;            // deklaracja tablicy liczb całkowitych typu int
String [] napisy;     // deklaracja tablicy referencji do obiektów klasy String
double[][] macierz;   // deklaracja dwuwymiarowej tablicy liczb rzeczywistych
```


Tablice cd.

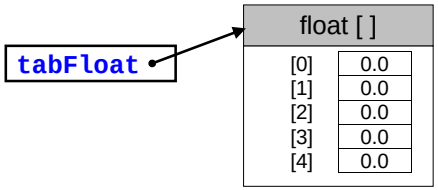
Sama deklaracja tablicy tworzy zmienną referencyjną, ale nie alokuje pamięci dla samej tablicy. Pamięć jest alokowana dynamicznie w wyniku inicjacji za pomocą nawiasów klamrowych albo w wyniku użycia wyrażenia **new**.

Przykład:

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

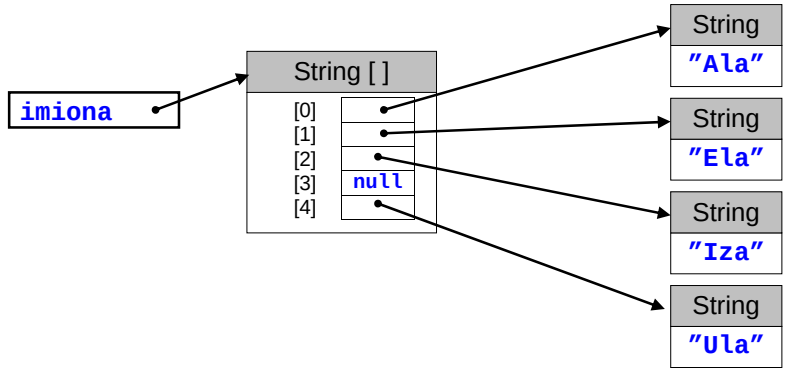



```
float [] tabFloat = new float[5];
```

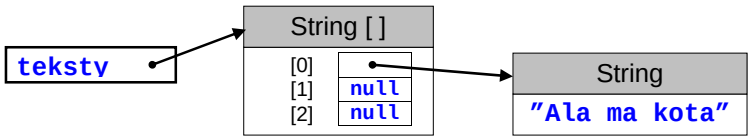


Tablice cd.

```
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**.

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**.

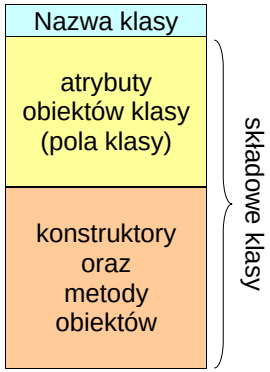
Specjalne operacje inicjalizacji przy tworzeniu obiektów nazywane są **konstruktorami.**

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:

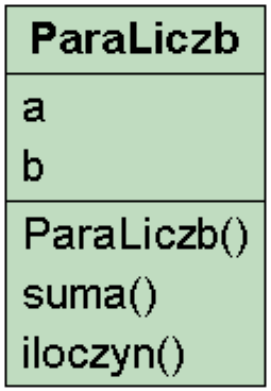
```
public class NazwaKlasy
{
   [spDostępu] typ nazwaPola;
   ...

   [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`.

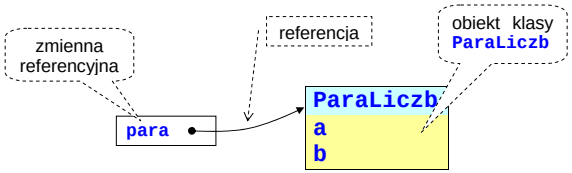
Uwaga:

- 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.:

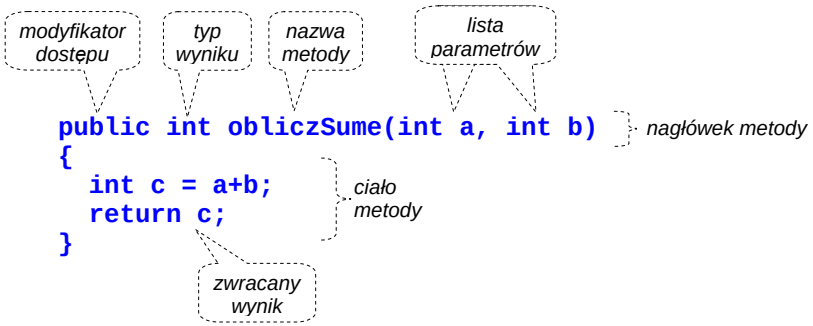
```
class ParaLiczb
{ int a, b;

   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

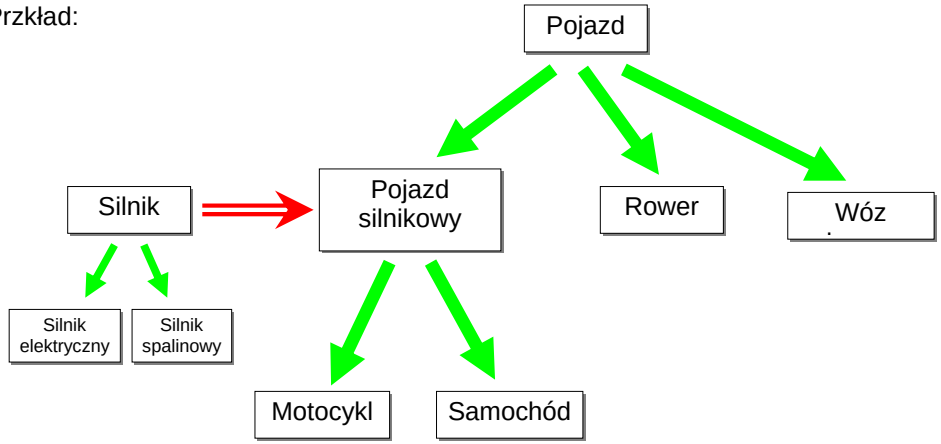
w metodzie statycznej nie wolno odwoływać się do pól i metod niestatycznych

wywołania metody statycznej

wywołania metody niestatycznej dla obiektów t1 i t2

Związki między klasami: „jest” i „zawiera”

Przkład:



Pojazd silnikowy jest szczególnym rodzajem **Pojazdu**

Motocykl jest szczególnym rodzajem **Pojazdu silnikowego**

Pojazd silnikowy zawiera **Silnik**

Ponowne wykorzystanie klas

Przy tworzeniu nowych klas można wykorzystywać już istniejące klasy za pomocą:

- kompozycji,
- dziedziczenia.

Kompozycję stosuje się wtedy, gdy między klasami zachodzi relacja typu „całość ↔ część” tzn. nowa klasa **zawiera** w sobie istniejącą klasę.

Dziedziczenie stosuje się wtedy, gdy między klasami zachodzi relacja „generalizacja ↔ specjalizacja” tzn. nowa klasa **jest szczególnym rodzajem** juz istniejącej klasy.

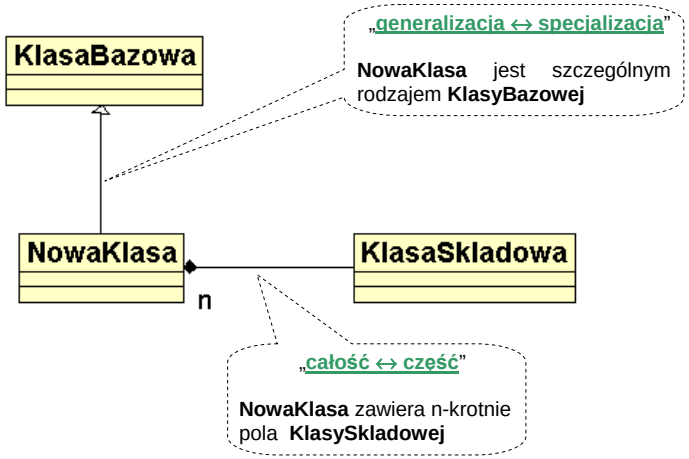
Uwaga:

Zwykle tworzy się nowe klasy wykorzystując jednocześnie kompozycję i dziedziczenie np.:
klasa **Pojazd silnikowy** jest uszczegółowieniem klasy **Pojazd** oraz zawiera w sobie klasę **Silnik**.

Diagramy klas w języku UML

UML (ang. Unified Modeling Language) – zunifikowany język modelowania do tworzenia systemów obiektowo zorientowanych.

Diagram klas pokazuje klasy i zachodzące między nimi relacje.



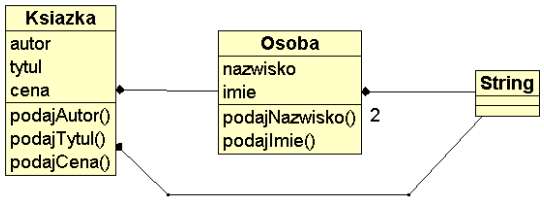
Kompozycja

Kompozycję uzyskujemy poprzez definiowanie w nowej klasie pól, które są obiektami istniejących klas.

Przykład:

Klasa **`Osoba`** zawiera:
- pola **`nazwisko`** i **`imie`**, które należą do klasy **`String`**.

Klasa **`Książka`** zawiera:
- pole **`autor`** należące do klasy **`Osoba`**,
- pole **`tytul`** należące do klasy **`String`**,
- pole **`cena`** typu **`double`**.



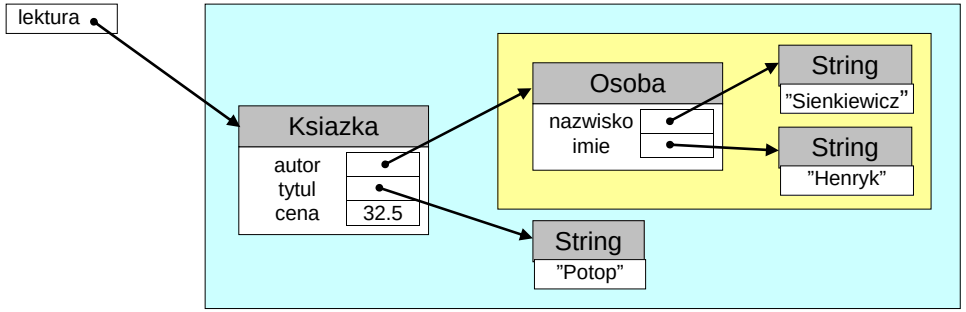
Kompozycja - definicja klasy `Osoba` oraz `Ksiazka`

```
class Osoba
{ private String nazwisko, imie;

  public Osoba(String nazwisko, String imie)
  { this.nazwisko = nazwisko;     this.imie = imie;  }

  public String podajNazwisko()
  { return nazwisko; }

  public String podajImie()
  { return imie; }
}

class Ksiazka
{ private Osoba autor;
  private String tytul;
  double cena;

  public Ksiazka(Osoba autor, String tytul, double cena)
  { this.autor = autor;     this.tytul = tytul;     this.cena = cena;  }

  public Osoba podajAutor()
  { return autor; }

  public String podajTytul()
  { return tytul; }

  public double podajCena()
  { return cena; }
}
```


Kompozycja cd.

Przykładowa instrukcja tworząca nowy obiekt klasy **`Ksiazka`**:

```
Ksiazka lektura = new Ksiazka( new Osoba(”Sienkiewicz”, ”Henryk”),
                               ”Potop”,
                               32.5 );
```

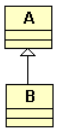


Dziedziczenie

Dziedziczenie polega na przejęciu właściwości i funkcjonalności obiektów innej klasy i ewentualnej modyfikacji tych właściwości i funkcjonalności w taki sposób, by były one bardziej wyspecjalizowane.

Do wyrażania relacji dziedziczenia jednej klasy przez drugą służy słowo kluczowe **`extends`**

```
class B extends A
{
  ...
}
```



Klasa **B** **dziedziczy** (rozszerza) klasę **A**, tzn.

- klasa **A** jest **klasą bazową**, (superklasą) klasy **B**
- klasa **B** jest **klasą pochodną** klasy **A**

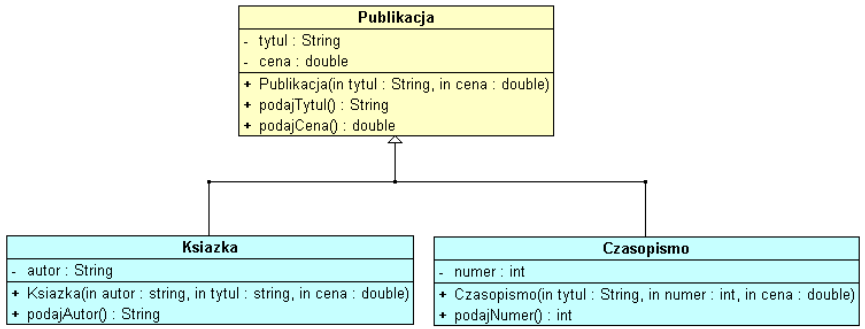
Dziedziczenie cd.

Przykład:

Klasa **`Publikacja`** zawiera:
- pole **`tytul`** z klasy **`String`** i pole **`cena`** typu **`double`**.

Klasa **`Książka`** dziedziczy po klasie **`Publikacja`** i dodatkowo zawiera:
- pole **`autor`** należące do klasy **`String`**.

Klasa **`Czasopismo`** dziedziczy po klasie **`Publikacja`** i dodatkowo zawiera:
- pole **`numer`** typu **`int`**.



Dziedziczenie cd.

Definicja klasy bazowej **Publikacja**

```
class Publikacja
{
    private String tytul;
    private double cena;

    Publikacja(String tytul, double cena)
    { this.tytul = tytul;
      this.cena = cena;
    }


    public String podajTytul()
    { return tytul;
    }

    public double podajCene()
    { return cena;
    }
}
```


Dziedziczenie cd.

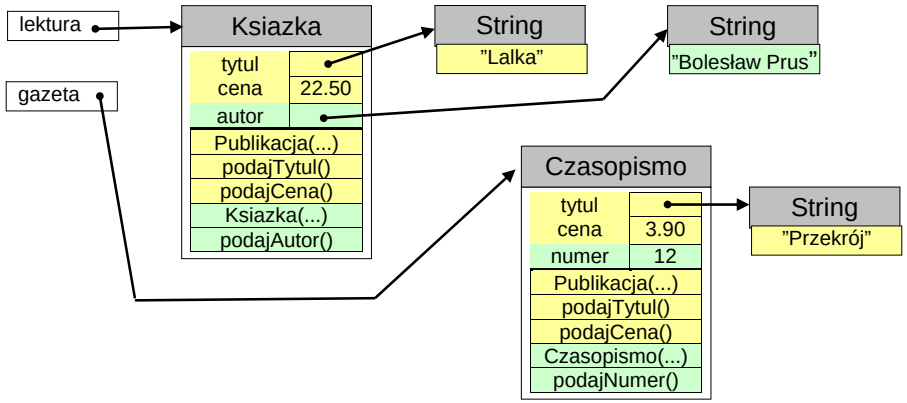
Definicja klas pochodnych **Ksiazka** i **Czasopismo**, które dziedziczą po klasie **Publikacja**

```
class Ksiazka extends Publikacja
{
    private String autor;

    Ksiazka(String autor, String tytul, double cena)
    { super(tytul, cena);    // Wywołanie konstruktora klasy bazowej Publikacja
      this.autor = autor;
    }

    public String podajAutor()
    { return autor;
    }
}

class Czasopismo extends Publikacja
{
    private int numer;

    Czasopismo(String tytul, int numer, double cena)
    { super(tytul, cena);    // Wywołanie konstruktora klasy bazowej Publikacja
      this.numer = numer;
    }

    public int podajNumer()
    { return numer;
    }
}
```


Dziedziczenie cd.

Przykładowe instrukcje tworzące nowe obiekty klas **Ksiazka** i **Czasopismo**:

```
Ksiazka lektura = new Ksiazka("Bolesław Prus", "Lalka", 22.50 );

Czasopismo gazeta = new Czasopismo("Przekrój", 12, 3.90 );
```



Inicjowanie obiektów przy dziedziczeniu

Pola klasy bazowej można inicjować za pomocą wywołania z konstruktora klasy pochodnej konstruktora klasy bazowej:

```
super(args);
```

gdzie *args* jest listą argumentów przekazanych do konstruktora klasy bazowej

Jeśli konstrukcja **super(...)** występuje, **MUSI** być pierwszą instrukcją konstruktora klasy pochodnej.

Gdy nie występuje, przed wykonaniem kodu klasy pochodnej zostanie wywołany konstruktor bezparametrowy klasy bazowej.

Przykład:

```
Czasopismo(String tytul, int numer, double cena)
{ super(tytul, cena);
  this.numer = numer;
}
```

Wywołanie konstruktora klasy bazowej

Inicjowanie obiektów przy dziedziczeniu cd.

Przy tworzeniu obiektów klas pochodnych podstawową regułą jest, że **najpierw muszą być zainicjowane pola odziedziczone z klasy bazowej, potem dodatkowe pola deklarowane w klasie pochodnej**.

Sekwencja inicjowania obiektu klasy pochodnej jest następująca:

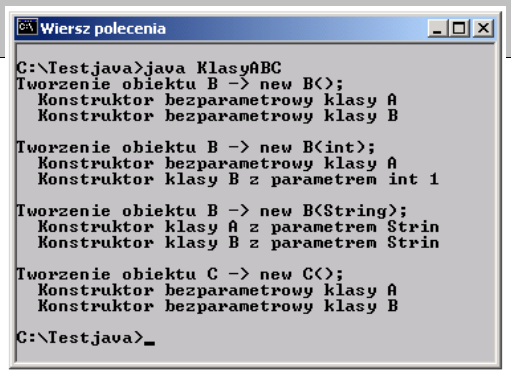
1. Wywoływany jest konstruktor klasy pochodnej,
2. Jeśli pierwszą instrukcją jest `super(args)`, wykonywany jest konstruktor klasy bazowej z argumentami `args`.
3. Jeśli nie ma `super(args)`, wykonywany jest konstruktor bezparametrowy klasy bazowej.
4. Wykonywane są instrukcje wywoływanego konstruktora klasy pochodnej.

Inicjowanie obiektów przy dziedziczeniu cd.

```
class A
{
    A()
    { System.out.println("  Konstruktor bezparametrowy klasy A");
    }

    A(String t)
    { System.out.println("  Konstruktor klasy A z parametrem String " + t);
    }
}

class B extends A
{
    B()
    { System.out.println("  Konstruktor bezparametrowy klasy B");
    }

    B(int i)
    { System.out.println("  Konstruktor klasy B z parametrem int " + i);
    }

    B(String t)
    { super(t);
      System.out.println("  Konstruktor klasy B z parametrem String " + t);
    }
}

class C extends B
{
}
```


Inicjowanie obiektów przy dziedziczeniu cd.

```
class KlasyABC
{
    public static void main(String [] args)
    {
        System.out.println("Tworzenie obiektu B -> new B();");
        new B();

        System.out.println("\nTworzenie obiektu B -> new B(int);");
        new B(1);

        System.out.println("\nTworzenie obiektu B -> new B(String);");
        new B("Ala");

        System.out.println("\nTworzenie obiektu C -> new C();");
        new C();
    }
}
```



Przeciążanie metod

Przeciążanie metod w klasie polega na definiowaniu wielu metod o tej samej nazwie ale odmiennej liście parametrów. Przeciążane mogą być zarówno konstruktory jak i metody statyczne i niestatyczne.

Przykład:

```
class Para
{ int a, b;

   Para()
   { System.out.println(" Konstruktor bezparametrowy");
     a = b = 0;
   }

   Para(int x, int y)
   { System.out.println(" Konstruktor z parametrami");
     a = x;  b = y;
   }

   void pokaz()
   { System.out.println("Metoda bez parametru");
     System.out.println("Para("+ a +", "+ b +")" );
   }

   void pokaz(String tekst)
   { System.out.println("Metoda z parametrem");
     System.out.print(tekst);
     pokaz();
   }
```

Przeciążone konstruktory

Przeciążone metody

Wywołanie metody przeciążonej

Przeciążanie metod cd.

```
   public static void main(String[] args)
   {
    System.out.println("<=== Tworzenie obiektow ===>");
    Para p1 = new Para();
    Para p2 = new Para(3,7);


    System.out.println("\n<=== Metoda pokaz bez parametru ===>");
    p2.pokaz();

    System.out.println("\n<=== Metoda pokaz z parametrem ===>");
    p2.pokaz("Para p2: ");

   }
}
```

```
Wiersz polecenia

C:\Testjava>java Para
<=== Tworzenie obiektow ===>
 Konstruktor bezparametrowy
 Konstruktor z parametrami

<=== Metoda pokaz bez parametrow ===>
Metoda bez parametrow
Para(3, 7)

<=== Metoda pokaz z parametrem ===>
Metoda z parametrem
Para p2: Metoda bez parametrow
Para(3, 7)

C:\Testjava>
```


Przedefiniowanie metody

Przedefiniowanie metody klasy bazowej w klasie pochodnej następuje wtedy, gdy w klasie pochodnej zdefiniujemy metodę z taką samą sygnaturą (nazwa i lista parametrów) i typem wyniku jak sygnatura i typ wyniku nieprywatnej i niestatycznej metody klasy bazowej.

Wówczas metoda klasy bazowej zostaje ukryta.

Przykład:

```
class Para
{ int a, b;

   Para(int x, int y)
   { System.out.println(" Konstruktor z parametrami");
     a = x;  b = y;
   }

   void pokaz()
   { System.out.println("Metoda bez parametru z klasy Para");
     System.out.println("Para("+ a +", "+ b +")" );
   }

}
```


Przedefiniowanie metody cd.

```
class Trojka extends Para
{ int c;

  Trojka(int x, int y, int z)
  { super(x, y);
    c = z;
  }

  void pokaz()
  { System.out.println("Metoda bez parametru z klasy Trojka");
    System.out.println("Trojka("+ a +", "+ b +", "+ c +")" );
  }

  void pokaz(String tekst)
  { System.out.println("Metoda z parametrem z klasy Trojka");
    System.out.println(tekst);
    pokaz();
    super.pokaz();
  }

  public static void main(String[] args)
  {
    System.out.println("<=== Tworzenie obiektow ===>");
    Trojka t1 = new Trojka(1, 2, 3);

    System.out.println("\n<=== Metoda pokaz bez parametru ===>");
    t1.pokaz();

    System.out.println("\n<=== Metoda pokaz z parametrem ===>");
    t1.pokaz("Trojka t1: ");
   }
}
```

Wywołanie konstruktora klasy bazowej

Przedefiniowanie metody z klasy bazowej

Przeciążenie metody

Wywołanie metody przedefiniowanej

Wywołanie metody z klasy bazowej

Przedefiniowanie metody cd.

```
Wiersz polecenia

C:\Testjava>java Trojka
<=== Tworzenie obiektow ===>
 Konstruktor z parametrami

<=== Metoda pokaz bez parametru ===>
Metoda bez parametru z klasy Trojka
Trojka(1, 2, 3)

<=== Metoda pokaz z parametrem ===>
Metoda z parametrem z klasy Trojka
Trojka t1:
Metoda bez parametru z klasy Trojka
Trojka(1, 2, 3)
Metoda bez parametru z klasy Para
Para(1, 2)

C:\Testjava>
```

Pokrycie metody statycznej

Pokrycie metody statycznej klasy bazowej w klasie pochodnej następuje wtedy, gdy w klasie pochodnej zdefiniujemy statyczną metodę z taką samą sygnaturą (nazwa i lista parametrów) i typem wyniku jak sygnatura i typ wyniku nieprywatnej i statycznej metody klasy bazowej.

Wówczas metoda klasy bazowej zostaje ukryta. Można się do niej odwołać z innej metody poprzedzając wywołanie metody nazwą klasy bazowej i operatorem „**.**” .

Uwaga:

W podobny sposób można pokrywać pola klasy bazowej. Należy w klasie pochodnej zdefiniować pole o tej samej nazwie (może być innego typu).

Pokrycie metody statycznej cd.

```
class KlasaBazowa
{
  static void drukuj()
  { System.out.println("Metoda statyczna klasy bazowej");
  }
}

class KlasaPochodna extends KlasaBazowa
{
  static void drukuj()
  { System.out.println("Metoda statyczna klasy pochodnej");
  }

  public static void main(String[] args)
  {
    drukuj();

    KlasaBazowa.drukuj();
  }
}
```

Pokrycie metody statycznej z klasy bazowej

Wywołanie metody statycznej z klasy pochodnej

Wywołanie metody statycznej z klasy bazowej

Przedefiniowanie i pokrycie - podsumowanie

- Metody prywatne nie mogą być przedefiniowane ani pokryte, gdyż nie są dostępne w klasie pochodnej.
- Metoda nieprywatna i niestatyczna jest przedefiniowana w klasie pochodnej, jeśli ma taką samą sygnaturę jak metoda klasy bazowej.
- Typ wyniku metody przedefiniowującej w klasie pochodnej musi być taki sam jak typ wyniku metody przedefiniowanej z klasy bazowej.
- Przy przedefiniowaniu można za pomocą specyfikatorów dostępu rozszerzać dostęp, ale nie można go zawężać.
- Metody statyczne nie mogą być przedefiniowane ale mogą być pokryte.
- Przy przedefiniowaniu i pokryciu metod trzeba zachować zgodność typów wyjątków zgłaszanych przez metodę i deklarowanych w klauzuli **`throws`**.

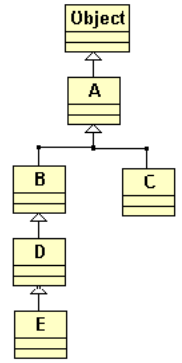
Hierarchia dziedziczenia w Javie

Jeśli przy definicji klasy nie używamy słowa **`extends`** (tzn. nie żądamy jawnie dziedziczenia) to nasza klasa domyślnie dziedziczy klasę **`Object`**.

W Javie każda klasa może bezpośrednio odziedziczyć tylko jedną klasę. Ale pośrednio może mieć dowolnie wiele nadklas.

W Javie wszystkie klasy pochodzą pośrednio od klasy `Object`

```
class A
{
}

class B extends A
{
}

class C extends A
{
}

class D extends B
{
}

class E extends D
{
}
```



Konwersje referencyjne

Można zauważyć, że obiekt klasy pochodnej posiada wszystkie atrybuty i metody klasy bazowej, a więc „zawiera w sobie” obiekt klasy bazowej (nadklasy). Dlatego odniesienie do takiego obiektu można zapamiętać w zmiennej referencyjnej klasy bazowej.

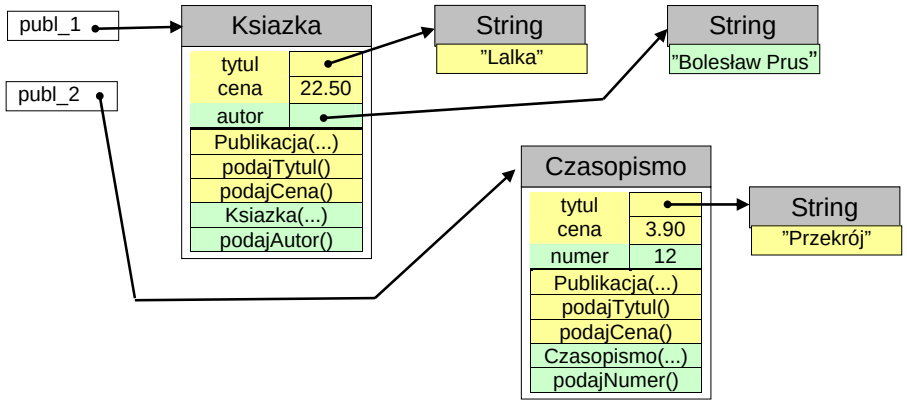
Obiekty klasy **`Ksiazka`** i klasy **`Czasopismo`** mają właściwości obiektów klasy **`Publikacja`** (tzn. posiadają wszystkie atrybuty i metody klasy **`Publikacja`**).

Referencje do obiektów klas **`Ksiazka`** i **`Czasopismo`** można więc przypisywać do zmiennych referencyjnych klasy **`Publikacja`**.

Konwersje referencyjne cd.

Przykładowe instrukcje tworzące nowe obiekty klas **`Ksiazka`** i **`Czasopismo`**:

```
Publikacja publ_1 = new Ksiazka(”Bolesław Prus”,”Lalka”, 22.50 );

Publikacja publ_2 = new Czasopismo(”Przekrój”, 12, 3.90 );
```



Referencyjna konwersja rozszerzająca

Referencyjna konwersja rozszerzająca to przekształcenie referencji do obiektu klasy pochodnej na referencję do typu wyższego czyli nadklasy (klasy bazowej).

Referencyjna konwersja rozszerzająca jest dokonywana automatycznie przy:

- przypisywaniu zmiennej referencyjnej odniesienia do obiektu klasy pochodnej,
- przekazywaniu argumentów metodzie, gdy parametr metody jest typu „referencja do obiektu nadklasy” przekazywanego argumentu
- zwracaniu wyniku metody, gdy wynik podstawiamy na zmienną będącą „referencją do obiektu nadklasy” zwracanego wyniku.

Referencyjna konwersja rozszerzająca

```
class Wydawnictwo
{
    static Publikacja utworzPublikacje(int rodzaj)
    {
        switch(rodzaj)
        { case 1: // utworz ksiazke
                  return new Ksiazka("Sarteczko", "Java", 45.00);
          case 2: // utworz czasopismo
                  return new Czasopismo("PC Format", 11, 9.50);
        }
        return null;
    }

    static double roznicaCeny(Publikacja p1, Publikacja p2 )
    {    return p1.podajCene() - p2.podajCene();
    }

    public static void main(String [] args)
    {
        Publikacja p1 = utworzPublikacje(1);
        Publikacja p2 = utworzPublikacje(2);
        roznicaCeny(p1, p2);

        Ksiazka k = new Ksiazka("Sarteczko", "Java", 45.00);
        Czasopismo c = new Czasopismo("PC Format", 11, 9.50);
        roznicaCeny(k, c);

        p1 = k;
        p2 = c;
        roznicaCeny(p1, p2);
    }
}
```

Konwersja referencyjna przy zwracaniu wyniku

Konwersja referencyjna przy przekazywaniu argumentow

Konwersja referencyjna przy przypisywaniu

Referencyjna konwersja zawężająca

Referencyjna konwersja zawężająca to przekształcenie referencji klasy bazowej na referencję do typu niższego czyli podklasy (klasy pochodnej). Taka konwersja jest dozwolona tylko wtedy, gdy referencja klasy bazowej wskazuje na obiekt, który w rzeczywistości należy do klasy pochodnej.

Referencyjna konwersja zawężająca (konwersja „w dół") :

- zawsze wymaga jawnego użycia operatora konwersji,
- jest bezpieczna. Java w trakcie wykonywania programu sprawdza czy obiekt, na który wskazuje referencja, jest faktycznie obiektem należącym do klasy pochodnej. Gdy tak nie jest to zostanie zgłoszony wyjątek **`ClassCastException`**.

Referencyjna konwersja zawężająca

```
class Wydawnictwo
{
    static Publikacja utworzPublikacje(int rodzaj)
    {   switch(rodzaj)
        { case 1: // utworz ksiazke
                  return new Ksiazka("Barteczko", "Java", 45.00);
          case 2: // utworz czasopismo
                  return new Czasopismo("PC Format", 11, 9.50);
        }
        return null;
    }

    public static void main(String [] args)
    {
        Publikacja publ_1, publ_2;
        String autor_1, autor_2;
        Ksiazka ksiaz;

        publ_1 = utworzPublikacje(1); // utworzenie obiektu klasy Ksiazka
        publ_2 = utworzPublikacje(2); // utworzenie obiektu klasy Publikacja

        ksiaz = (Ksiazka)publ_1;
        autor_1 = ksiaz.podajAutor();

        ksiaz = (Ksiazka)publ_2;
        autor_2 = ksiaz.podajAutor();

        autor_1 = ((Ksiazka)publ_1).podajAutor();

        autor_2 = ((Ksiazka)publ_2).podajAutor();
    }
}
```

Zawężająca konwersja referencyjna

Tu zostanie zgłoszony wyjątek ClassCastException

Operator `instanceof`

Operator **`instanceof`** jest wykorzystywany do stwierdzenia , do jakiej klasy należy obiekt. Wyrażenie:

`nazwaZmiennej` **`instanceof`** ***`nazwaKlasy`***

ma wartość **`true`**, jeśli zmienna ***`nazwaZmiennej`*** wskazuje na obiekt należący do klasy ***`nazwaKlasy`***, albo dowolnej jej podklasy.

Operator `instanceof`

```
public static void main(String [] args)
    {
        Publikacja publ_1, publ_2;
        String autor_1, autor_2;
        Ksiazka ksiaz;

        publ_1 = utworzPublikacje(1); // utworzenie obiektu klasy Ksiazka
        publ_2 = utworzPublikacje(2); // utworzenie obiektu klasy Publikacja

        if (publ_1 instanceof Ksiazka)
           {  ksiaz = (Ksiazka)publ_1;
              autor_1 = ksiaz.podajAutor();
           }

        if (publ_2 instanceof Ksiazka)
           {  ksiaz = (Ksiazka)publ_2;
              autor_2 = ksiaz.podajAutor();
           }

        if (publ_1 instanceof Ksiazka)
           {  autor_1 = ((Ksiazka)publ_1).podajAutor();
           }

        if (publ_2 instanceof Ksiazka)
           {  autor_2 = ((Ksiazka)publ_2).podajAutor();
           }
    }
```

Użycie operatora **`instanceof`** zabezpiecza przed próbą wykonania niedozwolonej konwersji zawężającej

Metody wirtualne

Jeśli w podklasie (klasie pochodnej) zostanie przedefiniowana jakaś metoda, zdefiniowana pierwotnie w nadklasie (klasie bazowej) to przy wywołaniu tej metody zostanie uruchomiona metoda tej klasy, do której faktycznie należy obiekt, a nie tej klasy która jest typem zmiennej referencyjnej zawierającej odniesienie do obiektu.

Oznacza to, że wiązanie odwołań do metod z kodem programu następuje nie w czasie kompilacji programu, lecz fazie wykonania programu tuż przed każdorazowym wykonaniem instrukcji wywołującej przedefiniowaną metodę.

Metody wirtualne to takie metody, dla których wiązanie odwołań z kodem programu następuje w fazie wykonania programu

Metody wirtualne cd.

Metody wirtualne to takie metody, dla których wiązanie odwołań z kodem programu następuje w fazie wykonania programu

W Javie wszystkie metody są wirtualne za wyjątkiem:

- **metod statycznych** (bo nie dotyczą obiektów, a klasy)
- **metod deklarowanych ze specyfikatorem `final`**, który oznacza, że metoda jest ostateczne i nie może być przedefiniowana,
- **metod prywatnych** (bo metody prywatne nie mogą zostać przedefiniowane).

Odwołania do metod wirtualnych są **polimorficzne**, gdyż efekt każdorazowego odwołania może przybierać różne kształty, w zależności od tego jaki jest faktyczny typ obiektu, na rzecz którego wywołano metodę wirtualną.

Metody wirtualne - przykład.

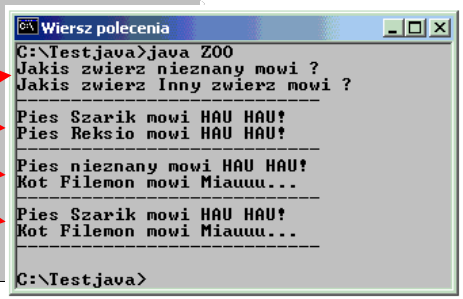
```
class Zwierz
{   String nazwa = "nieznany";

    Zwierz(){ }
    Zwierz(String n){ nazwa = n; }

    String podajGatunek() { return "Jakis zwierz"; }
    String podajNazwe()   { return nazwa; }
    String podajGlos()    { return "?"; }

    void mowa()
    {   System.out.println(podajGatunek() + " " +
                           podajNazwe() + " mowi " +
                           podajGlos() );
    }
}

class Pies extends Zwierz
{
    Pies(){ }
    Pies(String n){ super(n); }
    String podajGatunek() { return "Pies"; }
    String podajGlos()    { return "HAU HAU!"; }
}

class Kot extends Zwierz
{
    Kot() { }
    Kot(String n){ super(n); }
    String podajGatunek() { return "Kot"; }
    String podajGlos()    { return "Miauuu..."; }
}
```

Metody **podajGatunek(), podajNazwe(), podajGlos()** są wirtualne.

Działanie metod wirtualnych wywołanych w metodzie **mowa()** będzie zależeć od klasy obiektu, na rzecz którego zostanie wywołana metoda **mowa().**

Metody wirtualne – przykład cd.

```
class ZOO
{
    static void dialogZwierzat(Zwierz z1, Zwierz z2)
    { z1.mowa();
      z2.mowa();
      System.out.println("---------------------------");
    }

    public static void main(String []args)
    {
          Zwierz z1 = new Zwierz(),
                 z2 = new Zwierz("Inny zwierz");
          Pies pies = new Pies(),
             szarik = new Pies("Szarik"),
             reksio = new Pies("Reksio");
        Kot filemon = new Kot("Filemon");

        dialogZwierzat(z1,     z2);

        dialogZwierzat(szarik, reksio);

        dialogZwierzat(pies,   filemon);

        dialogZwierzat(szarik, filemon);
    }

}
```



Metody i klasy abstrakcyjne

Metoda abstrakcyjna to metoda, która nie ma implementacji (ciała) i jest zadeklarowana ze specyfikatorem **abstract**. Taka metoda może być deklarowana tylko w klasie abstrakcyjnej !

```
abstract int obliczCos();
```

nie ma ciała – tylko średnik

Klasą abstrakcyjną to klasa, opatrzona specyfikatorem **abstract**. Taka klasa może (ale nie musi) zawierać metody abstrakcyjne.

```
abstract class JakasKlasa
{  abstract int obliczCos();
   void wypiszCos(){ System.out.printn(”cos”); }
}
```

metoda abstrakcyjna

Nie można tworzyć obiektów klasy abstrakcyjnej !!!

Metody i klasy abstrakcyjne cd.

Klasa abstrakcyjna może być dziedziczona przez nowe klasy. Klasa pochodna **MUSI** przedefiniować (a właściwie zdefiniować) wszystkie metody abstrakcyjne, które odziedziczyła z abstrakcyjnej klasy bazowej. W przeciwnym wypadku klasa pochodna nadal pozostanie klasą abstrakcyjną i nie będzie można tworzyć jej obiektów.

Metody i klasy abstrakcyjne - przykład.

```
abstract class Zwierz
{   String nazwa = "nieznany";

    Zwierz(){ }
    Zwierz(String n){ nazwa = n; }

    String podajNazwe()   { return nazwa; }

    abstract String podajGatunek();
    abstract String podajGlos();

    void mowa()
    {   System.out.println(podajGatunek() + " " +
                           podajNazwe() + " mowi " +
                           podajGlos() );
    }
}

class Pies extends Zwierz
{
    Pies(){ }
    Pies(String n){ super(n); }
    String podajGatunek() { return "Pies"; }
    String podajGlos()    { return "HAU HAU!"; }
}

class Kot extends Zwierz
{
    Kot() { }
    Kot(String n){ super(n); }
    String podajGatunek() { return "Kot"; }
    String podajGlos()    { return "Miauuu..."; }
}
```

Metody **podajGatunek(), podajGlos()** są abstrakcyjne.

W metodzie **mowa()** są wywoływane metody abstrakcyjne, które nie zostały jeszcze zdefiniowane.

Przedefiniowanie (Konkretyzacja) metod abstrakcyjnych odziedziczonych z abstrakcyjnej klasy bazowej

Metody i klasy abstrakcyjne – przykład cd.

```
class ZOO
{
    static void dialogZwierzat(Zwierz z1, Zwierz z2)
    { z1.mowa();
      z2.mowa();
      System.out.println("----------------------------");
    }

    public static void main(String []args)
    {
       // Zwierz z1 = new Zwierz(),
       //        z2 = new Zwierz("Inny zwierz");

       Zwierz z1 = new Pies(),
              z2 = new Kot(”Bonifacy”);

          Pies pies = new Pies(),
             szarik = new Pies("Szarik"),
             reksio = new Pies("Reksio");
        Kot filemon = new Kot("Filemon");

        dialogZwierzat(z1,     z2);

        dialogZwierzat(szarik, reksio);

        dialogZwierzat(pies,   filemon);

        dialogZwierzat(szarik, filemon);
    }

}
```

Nie wolno tworzyć obiektów klasy abstrakcyjnej

Tu następują referencyjne konwersje rozszerzające.

```
Wiersz polecenia
C:\Testjava>java ZOO
Pies nieznany mowi HAU HAU!
Kot Bonifacy mowi Miauuu...
----------------------------
Pies Szarik mowi HAU HAU!
Pies Reksio mowi HAU HAU!
----------------------------
Pies nieznany mowi HAU HAU!
Kot Filemon mowi Miauuu...
----------------------------
Pies Szarik mowi HAU HAU!
Kot Filemon mowi Miauuu...
----------------------------

C:\Testjava>_
```

Interfejsy

Interfejs w Javie to deklarowany za pomocą słowa kluczowego **`interface`** nazwany zbiór deklaracji zawierający:

- publiczne abstrakcyjne metody (bez implementacji),
- publiczne statyczne zmienne finalne (stałe) o ustalonych typach i wartościach.

Implementacja interfejsu w klasie polega na zdefiniowaniu w tej klasie wszystkich metod zadeklarowanych w implementowanym interfejsie.

Interfejsy cd.

Ogólna postać definicji interfejsu w języku Java:

```
public interface NazwaInterfejsu
{
   typ nazwaZmiennej = wartosc;
   ...

   typ nazwaMetody(lista_parametrów);
   ...
}
```

Uwagi:

- modyfikator dostępu **`public`** przed słowem **`interface`** może nie występować - wówczas interfejs jest dostępny tylko w bieżącym pakiecie,
- zmienne są zawsze typu **`static final`** i mają przypisaną wartość stałą,
- metody są zawsze abstrakcyjne (bez implementacji).

Implementacja interfejsu

Ogólna postać definicji klasy implementującej interfejs w języku Java:

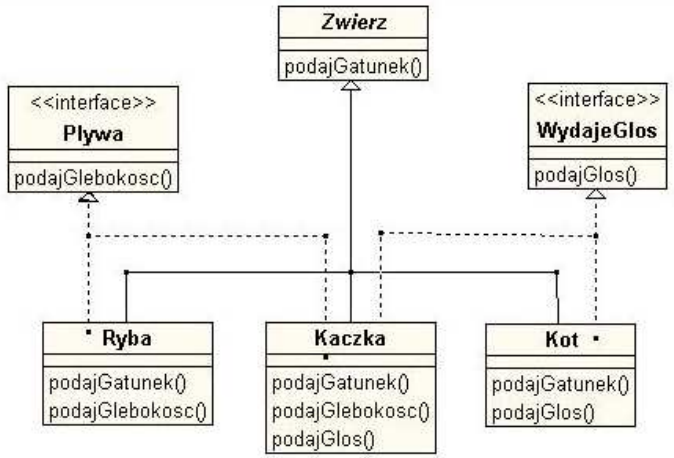
```
public class NazwaKlasy extends KlasaBazowa implements
NazwaInterfejsu_1, ..., NazwaInterjejsu_n
{
   ...
}
```

Uwagi:

- modyfikator dostępu **`public`** przed słowem **`class`** może nie występować - wówczas klasa jest dostępna tylko w bieżącym pakiecie,
- klasa może ale nie musi dziedziczyć inną klasę (słowa **`extends KlasaBazowa`** mogą nie występować)
- klasa może implementować wiele interfejsów,
- klasa musi definiować WSZYSTKIE metody implementowanych interfejsów albo pozostać klasą abstrakcyjną.
- klasa może zawierać własne (nie będące częścią interfejsu) atrybuty i metody.

Implementacja interfejsu – przykład

Przykład zawiera implementację klas reprezentujących różne gatunki zwierząt. Cześć zwierząt potrafi pływać, Niektóre potrafią wydawać odgłosy.



Implementacja interfejsu – przykład

```
/**
 *   Interfejs definiujący metody
 *   dla  obiektów  pływających w wodzie
 */
interface Plywa
{
    float podajGlebokosc();
}

/**
 *   Interfejs definiujący metody
 *   dla  obiektów wydających glosy
 */
interface WydajeGlos
{
    String podajGlos();
}

/**
 *   Klasa abstrakcyjna reprezentująca zwierzęta
 */
abstract class Zwierz
{   String nazwa;

    Zwierz(String nazwa){ this.nazwa = nazwa;}
    public String podajNazwe(){return nazwa; }
    public abstract String podajGatunek();
    public abstract void info();
}
```


Implementacja interfejsu – przykład cd.

Klasa **Ryba** jest klasą pochodną **Zwierz**, która implementuje interfejs **Plywa,** natomiast klasa **Kot** implementuje interfejs **WydajeGlos**

```
class Ryba extends Zwierz implements Plywa
{   float glebokosc;

    Ryba(String nazwa, float glebokosc)
       { super(nazwa);
         this.glebokosc = glebokosc;
       }
    public String podajGatunek()  {  return "Ryba";   }
    public float  podajGlebokosc(){  return glebokosc;}
    public void info()
       { System.out.println(podajNazwe() + " pływa na głębokości " + podajGlebokosc() );
       }
}

class Kot extends Zwierz implements WydajeGlos
{
    Kot(String nazwa){ super(nazwa); }

    public String podajGatunek()  {  return "Kot";    }
    public String podajGlos()     {  return "Miauuu"; }
    public void info()
       { System.out.println(podajNazwe() + " mówi " + podajGlos());
       }
}
```


Implementacja interfejsu – przykład cd.

Klasa **Kaczka** jest klasą pochodną **Zwierz**, która implementuje zarówno interfejs **Plywa** jak i interfejs **WydajeGlos**

```
class Kaczka extends Zwierz implements Plywa, WydajeGlos
{
    Kaczka(String nazwa){ super(nazwa); }
    public String podajGatunek()  {  return "Kaczka"; }
    public float  podajGlebokosc(){  return 0.0F;     }
    public String podajGlos()     {  return "Kwa Kwa";}
    public void info()
    { System.out.println(podajNazwe() + " pływa po powierzchni i mówi ” + podajGlos() );
    }
}
```


Interfejs i zmienne referencyjne

Interfejsy, podobnie jak klasy wyznaczają typy zmiennych referencyjnych.

Przykład:

Deklaracja zmiennej, której typem jest interfejs

```
WydajeGlos wg1, wg2;
```

Wartością takiej zmiennej może być odwołanie do obiektu dowolnej klasy, która implementuje ten interfejs.

```
wg1 = new Kaczka(”Dziwaczka”);
wg2 = new Kot(”Bonifacy”);
```

Przy wykonywaniu powyższych przypisań następuje referencyjna konwersja rozszerzająca referencji wskazującej na nowy obiekt do typu interfejsu (implementowany interfejs jest traktowany podobnie jak klasa bazowa).

Interfejs i zmienne referencyjne cd.

Przez zmienną której typem jest interfejs można wywołać dowolną metodę zadeklarowaną w tym interfejsie:

```
System.out.println( wg1.podajGlos() ); // wypisze "Kwa Kwa"
System.out.println( wg2.podajGlos() ); // wypisze "Miauuu"
```

Wywoła się poprawna wersja metody, odpowiednio dla faktycznego obiektu wskazywanego przez zmienną.

Wywołanie innej metody, która nie została zdefiniowana w interfejsie, jest możliwe wyłącznie po użyciu jawnej konwersji zawężającej:

```
((Zwierz)wg1).info();
      // wypisze "Dziwaczka pływa po powierzchni i mówi Kwa Kwa"

((Zwierz)wg2).info();
      // wypisze "Bonifacy mówi Miauuu"
```


Implementacja interfejsu – przykład cd.

```
class ZOO
{
    static void dialog_1(WydajeGlos z1, WydajeGlos z2)
    { System.out.print ("dialog_1: ");
      System.out.println( z1.podajGlos() + " <===> " + z2.podajGlos());
    }

    static void dialog_2(Zwierz z1, Zwierz z2)
    {
      System.out.print ("dialog_2: ");
      System.out.println( ((WydajeGlos)z1).podajGlos() + " <===> "
                        + ((WydajeGlos)z2).podajGlos() );
    }

    static void dialog_3(Zwierz z1, Zwierz z2)
    { System.out.print ("dialog_3: ");
      if (z1 instanceof WydajeGlos && z2 instanceof WydajeGlos)
         System.out.println( ((WydajeGlos)z1).podajGlos() + " <===> "
                           + ((WydajeGlos)z2).podajGlos() );
      else
         System.out.println( "  Dialog niemozliwy !" );
    }

  // ciąg dalszy na następnej stronie
```

Zawężająca konwersja referencyjna do typu `WydajeGlos`.

Sprawdzanie czy zawężająca konwersja referencyjna do typu `WydajeGlos` jest możliwa.

Implementacja interfejsu – przykład cd.

```
// ciąg dalszy klasy ZOO

    public static void main(String [] args)
    {  Kaczka kaczka = new Kaczka("Dziwaczka");
       Kaczka kaczor = new Kaczka("Donald");
       Ryba rybka = new Ryba("Zlota rybka", 0.32F);
       Kot kot = new Kot("Filemon");

       kaczka.info();
       kaczor.info();
       rybka.info();
       kot.info();

       System.out.println("\n<--- dialog_1 ----------
       dialog_1(kaczka, kaczor);
       dialog_1(kaczor, kot);
     //  dialog_1(kot, rybka);

       System.out.println("\n<--- dialog_2 ----------
       dialog_2(kaczka, kaczor);
       dialog_2(kaczor, kot);
       dialog_2(kot, rybka);

       System.out.println("\n<--- dialog_3 --------------->");
       dialog_3(kaczka, kaczor);
       dialog_3(kaczor, kot);
       dialog_3(kot, rybka);
    }
}  // koniec klasy ZOO
```

Przy przekazywaniu argumentów następuje konwersja referencyjna do typu `WydajeGlos`.

Klasa `Ryba` nie implementuje interfejsu `WydajeGlos`, a więc konwersja referencyjna do typu `WydajeGlos` nie jest możliwa

Implementacja interfejsu – przykład cd.

```
// ciąg dalszy klasy ZOO

    public static void main(String [] args)
    {  Kaczka kaczka = new Kaczka("Dziwaczka");
       Kaczka kaczor = new Kaczka("Donald");
       Ryba rybka = new Ryba("Zlota rybka", 0.32F);
       Kot kot = new Kot("Filemon");

       kaczka.info();
       kaczor.info();
       rybka.info();
       kot.info();

       System.out.println("\n<--- dialog_1 ----
       dialog_1(kaczka, kaczor);
       dialog_1(kaczor, kot);
     //  dialog_1(kot, rybka);

       System.out.println("\n<--- dialog_2 ----
       dialog_2(kaczka, kaczor);
       dialog_2(kaczor, kot);
       dialog_2(kot, rybka);

       System.out.println("\n<--- dialog_3 ----
       dialog_3(kaczka, kaczor);
       dialog_3(kaczor, kot);
       dialog_3(kot, rybka);
    }
}  // koniec klasy ZOO
```

```
Wiersz polecenia
C:\Testjava>java ZOO
Dziwaczka plywa po powierzchni
Donald plywa po powierzchni
Zlota rybka plywa na glebokosci 0.32
Filemon mowi Miauuu

<--- dialog_1 ---------------->
dialog_1: Kwa Kwa <===> Kwa Kwa
dialog_1: Kwa Kwa <===> Miauuu

<--- dialog_2 ---------------->
dialog_2: Kwa Kwa <===> Kwa Kwa
dialog_2: Kwa Kwa <===> Miauuu
dialog_2:
java.lang.ClassCastException: Ryba

<--- dialog_3 ---------------->
dialog_3: Kwa Kwa <===> Kwa Kwa
dialog_3: Kwa Kwa <===> Miauuu
dialog_3:   Dialog niemozliwy !

C:\Testjava>
```