Programowanie obiektowe

Podstawy języka Java

Paweł Rogaliński
Instytut Informatyki, Automatyki i Robotyki
Politechniki Wrocławskiej

pawel.rogalinski@pwr.wroc.pl

Podstawowe elementy języka

- Komentarze
- słowa kluczowe
- Identyfikatory
- Typy danych
- Literały
- Operatory
- Instrukcje warunkowe
- Instrukcje iteracyjne
- Instrukcje skoku

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),

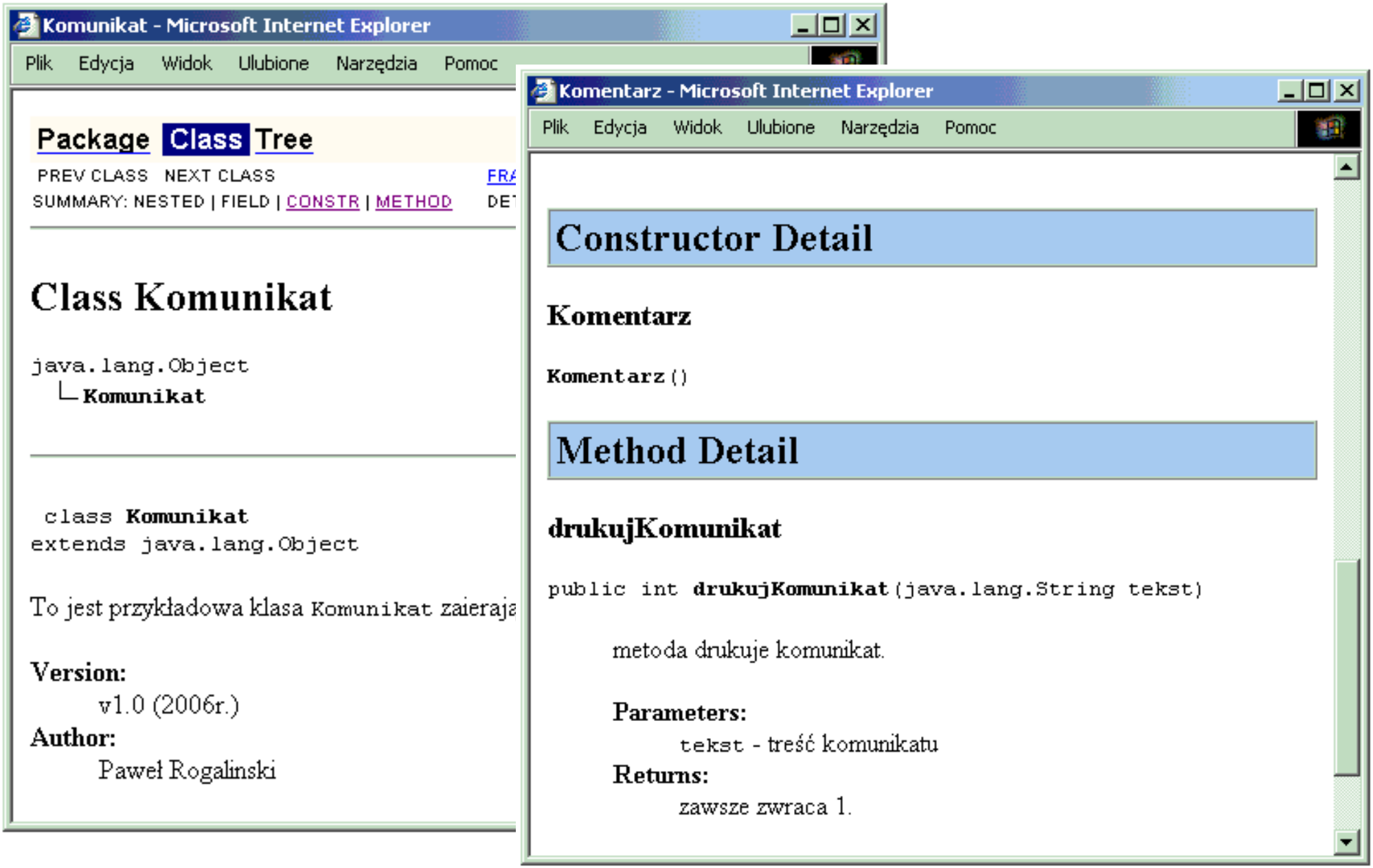
`@param` – opis parametru wywołania metody,

`@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 podstawowe**:
  - 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 podstawowe

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

Rozmiar i zakres wartości typów podstawowych

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

Typy podstawowe

- Wszystkie typy całkowite reprezentują liczby ze znakiem tzn. pierwszy bit liczby jest traktowany jako znak.
- Implementacja typów zmiennoprzecinkowych w Javie jest zgodna ze standardem IEEE-754.
- Znaki są w Javie reprezentowane jako 16-bitowe kody *Unicode UTF-16* (ponad 65 tys. znaków; m.in. alfabet grecki i znaki innych alfabetów europejskich, rosyjski, arabski, hebrajski, japoński). Powszechnie obecnie stosowany kod *ASCII* (7 bitowy) jest podzbiorem *Unikodu*.
- Dane typu logicznego mogą przyjmować tylko jedną z dwóch wartości: **`true`** (prawda) lub **`false`** (fałsz). Nie można stosować wartości 0 i 1, tak jak to jest w języku C/C++.

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;
   }
```

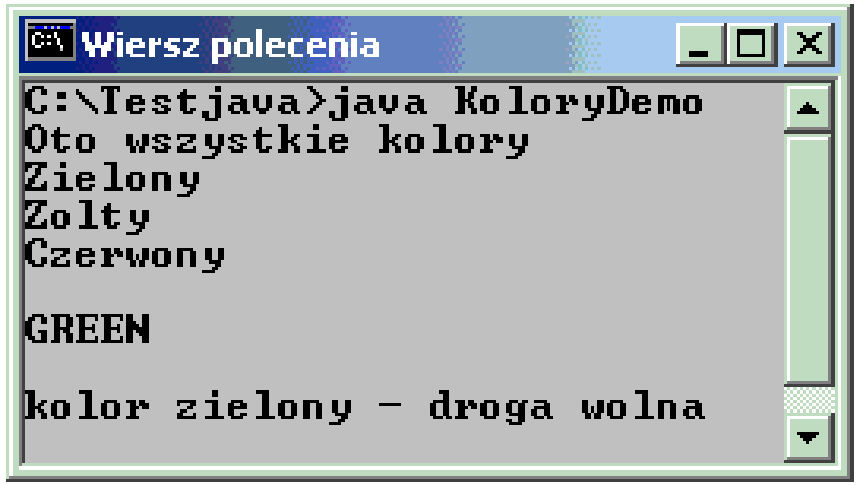
- Wszystkie wyliczenia automatycznie zawierają dwie predefiniowane metody:

```
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;
}   } }
```



Literały

Literał to napis reprezentujący w sposób bezpośredni wartość danej. W Javie wyróżniamy literały **liczbowe**, **znakowe**, **łańcuchowe** i **logiczne**.

Literał liczbowy to bezpośredni zapis konkretnej liczby.

- Liczby całkowite mogą być zapisywane w systemie:
  - dziesiętnym, w naturalny sposób np. `3` lub `121`
  - ósemkowym, poprzez poprzedzenie liczby zerem np. `03`
  - szesnastkowym, poprzez poprzedzenie liczby znakami `0x` lub `0X` np. `0xFF` (cyfry szesnastkowe powyżej 9 mogą być zapisywane wielkimi lub małymi literami).
- Każda liczba całkowita zapisana literalnie (np. `100`) jest traktowana jako liczba typu `int`.
- Liczby całkowite typu `long` są zapisywane z przyrostkiem `L` lub `l` np. `300L`.

Literały cd.

Literały liczbowe cd.

- Przy zapisie liczb rzeczywistych jako separator miejsc dziesiętnych jest stosowana kropka (a nie przecinek) np. `3.1415`
- Liczby rzeczywiste mogą być zapisywane w notacji naukowej z wykorzystaniem litery e lub E np. `9.3e-9` (9.3 pomnożone przez 10 do potęgi -9)
- Każda liczba rzeczywista zapisana literalnie (z kropką dziesiętną lub w notacji naukowej) jest traktowana jako liczba typu `double`.
- Liczby rzeczywiste typu `float` są zapisywane z przyrostkiem `F` lub `f` np. `3.7f`.

Literały cd.

Literał znakowy określa jeden znak zapisany bezpośrednio w programie.

- Literały znakowe typu **`char`** zapisujemy jako:
  - pojedyncze znaki w apostrofach np. **`'A'`**, **`'+'`**,
  - znaki specjalne ujęte w apostrofy np. **`'\n'`**, **`'\t'`**,
  - jako kod szesnastkowy *Unicode* ujęty w apostrofy np. **`'\u006E'`** (litera *n*), **`'\u001B'`** (znak *Esc*),
- Bezpośrednich kodów *Unikodu* nie wolno stosować dla znaków specjalnych *LF* (**`'\u000a'`**) i *CR* (**`'\000d'`**). Zamiast tego należy stosować znaki **`'\n'`** i **`'\r'`**.

Literały cd.

Znaki specjalne i ich zapis stosowany w literałach znakowych i łańcuchowych

znaki specjalne	zapis
przejście do nowego wiersza (*line feed – LF*)	`\n`
tabulacja (*TAB*)	`\t`
backspace (*BS*)	`\b`
powrót karetki (*carrage return – CR*)	`\r`
nowa strona (*form feed – FF*)	`\f`
apostrof	`\'`
cudzysłów	`\"`
lewy ukośnik (*backslash*)	`\\`
dowolny znak o kodzie *NNNN* w *unikodzie* (*N* to cyfra szesnastkowa)	`\uNNNN`

Literały cd.

Literał łańcuchowy to bezpośrednio zapisane ciągi znaków (napisy), które są traktowane jako teksty.

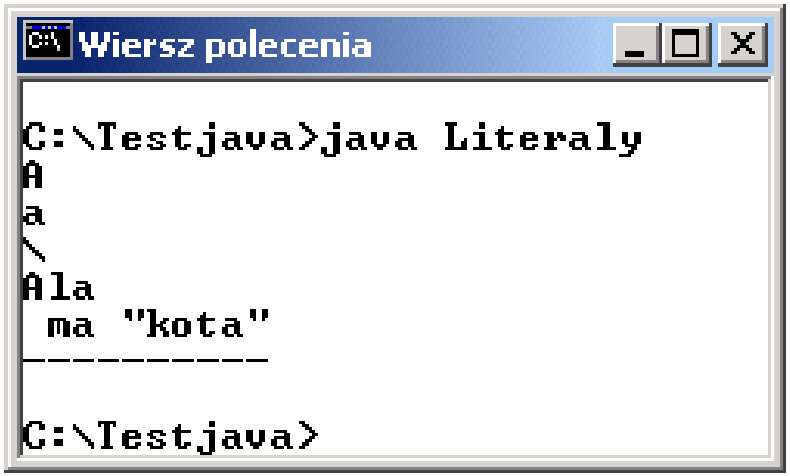
- Łańcuchy znakowe zapisujemy jako ciągi znaków ujęte w cudzysłowy
  np. `"Ala ma kota"`.
- W łańcuchach można wstawiać znaki specjalne oraz znaki *unikodu*
  np. `"dwa\nwiersze i litera \u0061"`.
- W Javie literały łańcuchowe są zawsze traktowane jako obiekty klasy `String`.

Literały logiczne są zapisywane wyłącznie za pomocą słów `false` i `true`.

Literały cd.

Przykład drukowania literałów znakowych i łańcuchowych

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

        System.out.println('A');
        System.out.println('\u0061');
        System.out.println('\\');
        System.out.println("Ala \n ma \"kota\"");
        System.out.println("----------");
    }
}
```



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, jednoargumentowy minus
		`-`	`-wyrażenie`	
		`!`	`!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**.

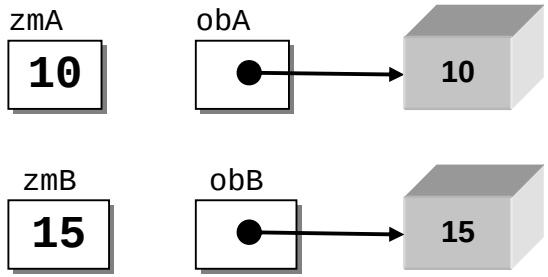
Jeśli zmienne **a** i **b** są typu referencyjnego (zawierają odwołanie do obiektu) to wykonanie instrukcji **a=b** powoduje skopiowanie do zmiennej **a** referencji do obiektu wskazywanego przez zmienną **b**. W efekcie zmienne **a** i **b** wskazują na ten sam obiekt. Późniejsza modyfikacja obiektu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

Operator przypisania - przykład dla typów prostych i referencyjnych

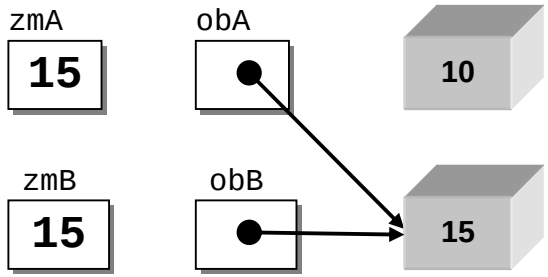
```
class Test
{   int p;

    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

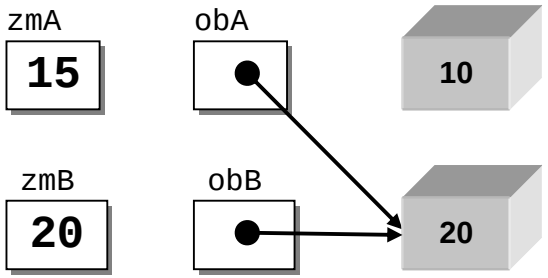
```
// Inicjalizacja
int zmA = 10;
int zmB = 15;

Test obA = new Test(10);
Test obB = new Tesy(20);
```

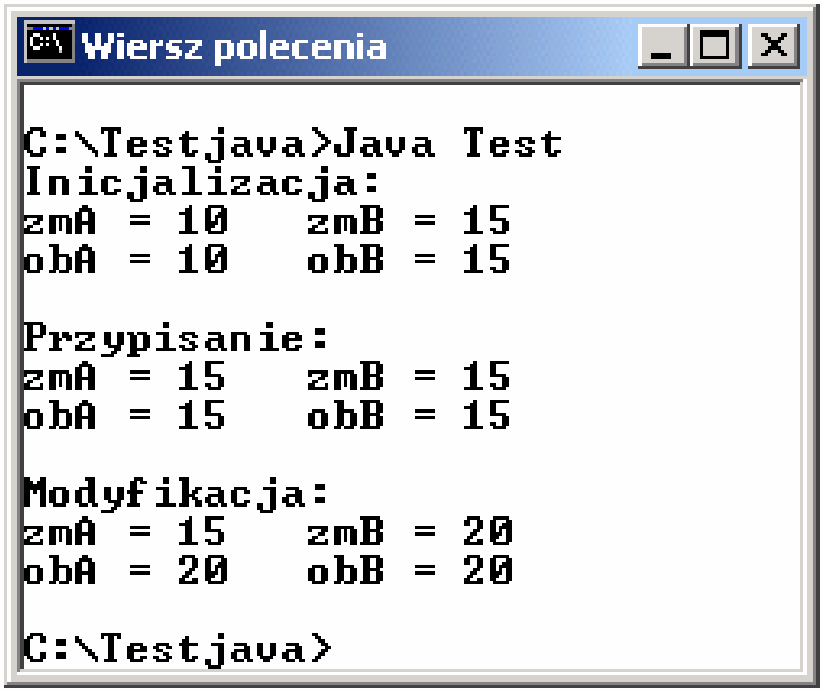



```
// Przypisanie
zmA = zmB;
obA = obB;
```




```
// Modyfikacja
zmB = 20;
obB.p = 20;
```





Operator przypisania cd.

Wiązanie prawostronne operatora przypisania powoduje, że instrukcje przypisania są opracowywane od prawej strony. W wyrażeniu: **`a=b=c=10;`** najpierw zostanie wykonane przypisanie **`c=10`**, następnie przypisanie **`b=c`**, a na koniec **`a=b`**. W rezultacie zmienne **`a b`** i **`c`** będą miały wartość **`10`**.

W Javie występują tzw. złożone operatory przypisania w postaci **`op=`** gdzie **`op`** jest to jeden z operatorów: **`*`**, **`/`**, **`%`**, **`+`**, **`-`**, **`<<`**, **`>>`**, **`>>>`**, **`&`**, **`^`**, **`|`**.

Uwaga: Instrukcja **`x op= wyrażenie;`** jest równoważna instrukcji:

`x = x op wyrażenie;`

Jest to bardzo wygodny sposób skracania zapisu,

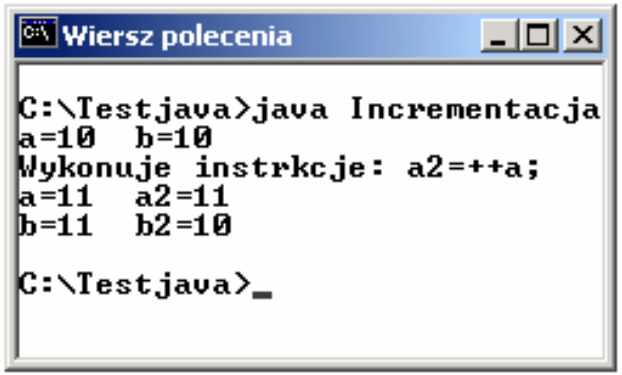
Operatory inkrementacji i dekrementacji

Operator **++** zwiększa, a **--** zmniejsza o jeden wartość argumentu (zmiennej). Oba występują w dwóch postaciach:

- przyrostkowej (operator po argumencie **--** zmiennej),
- przedrostkowej (operator przed argumentem **--** zmienną).

Przyrostkowa forma operatorów modyfikuje wartość argumentu po jej wykorzystaniu w wyrażeniu, przedrostkowa przed wykorzystaniem tej wartości.

```
class Incrementacja
{
    public static void main(String[] args){
        int a, a2, b, b2;
        a=10; b=10;
        System.out.println("a=" +a+ "  b=" +b);
        a2=++a;
        b2=b++;
        System.out.println("Wykonuje instrkcje: a2=++a;   b2=b++;");
        System.out.println("a=" +a+ "  a2=" +a2);
        System.out.println("b=" +b+ "  b2=" +b2);
    }
}
```



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`

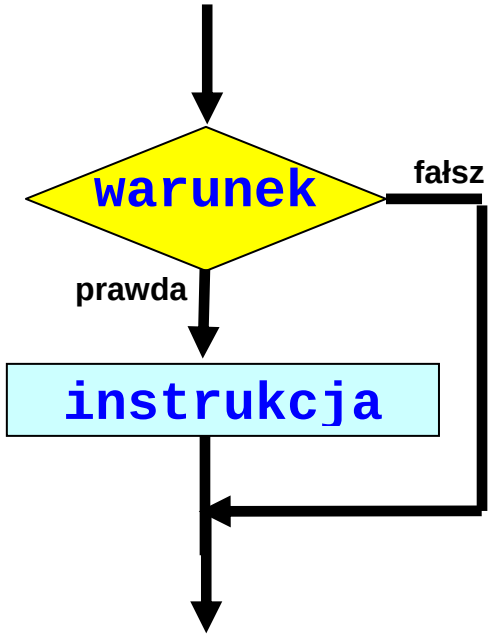
Instrukcja throw – zgłaszanie wyjątku przerywającego normalny tok działania programu.

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;
    }
```

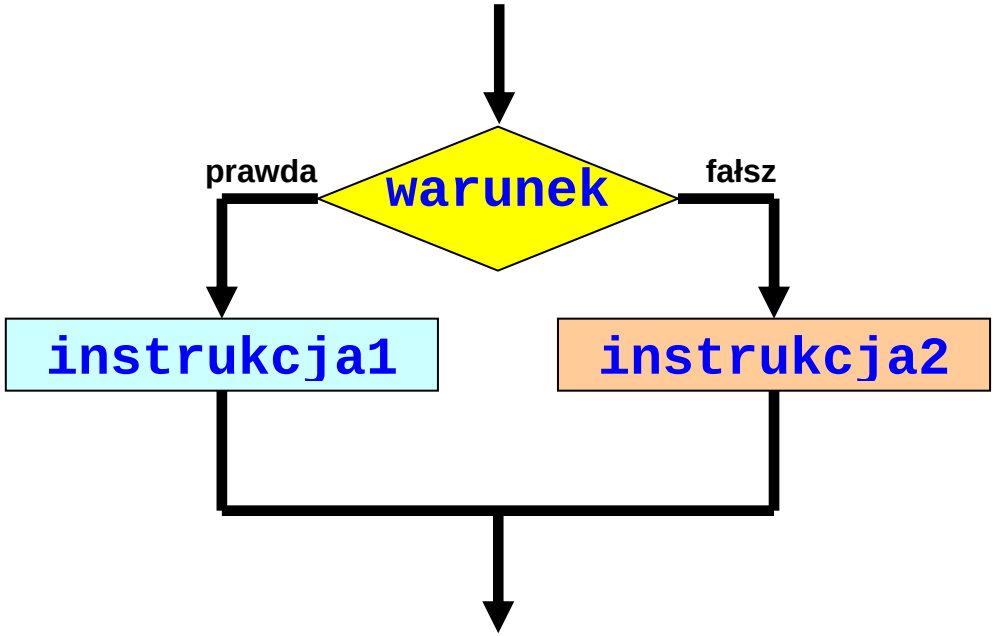


Postać instrukcji warunkowej `if - else`

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

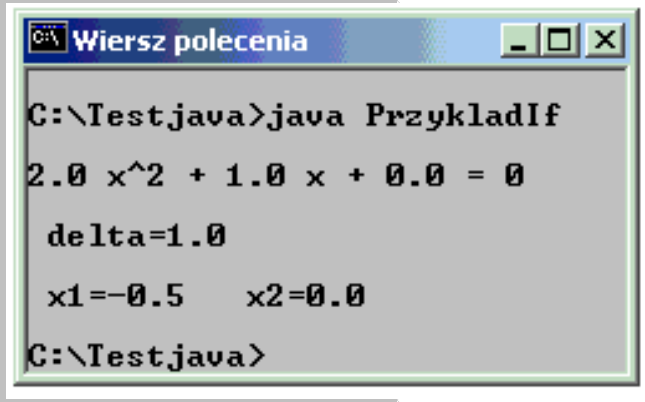
Jeśli *warunek* jest prawdziwy to wykonywana jest *instrukcja1*, w przeciwnym wypadku wykonywana jest *instrukcja2*.

```
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");
}   }
```

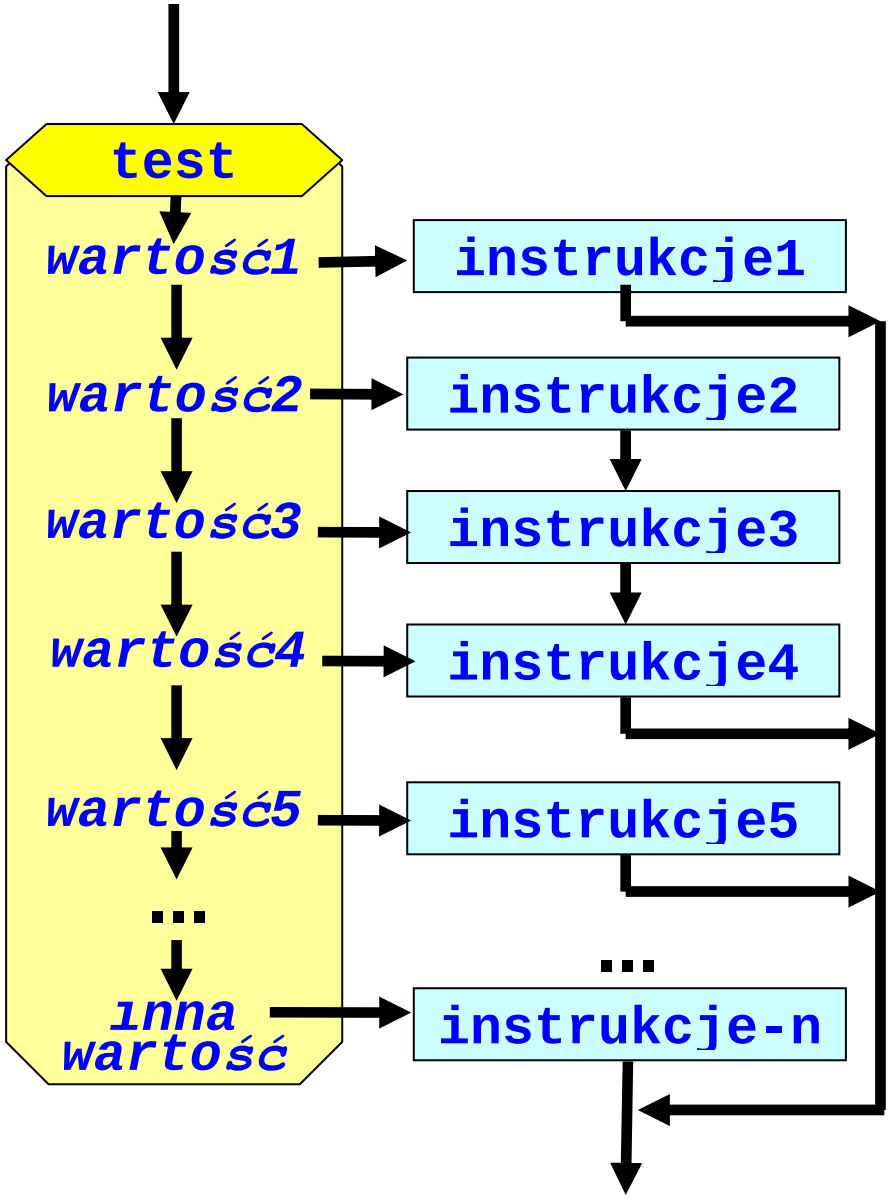


Postać instrukcji wyboru `switch`

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;
    }
```

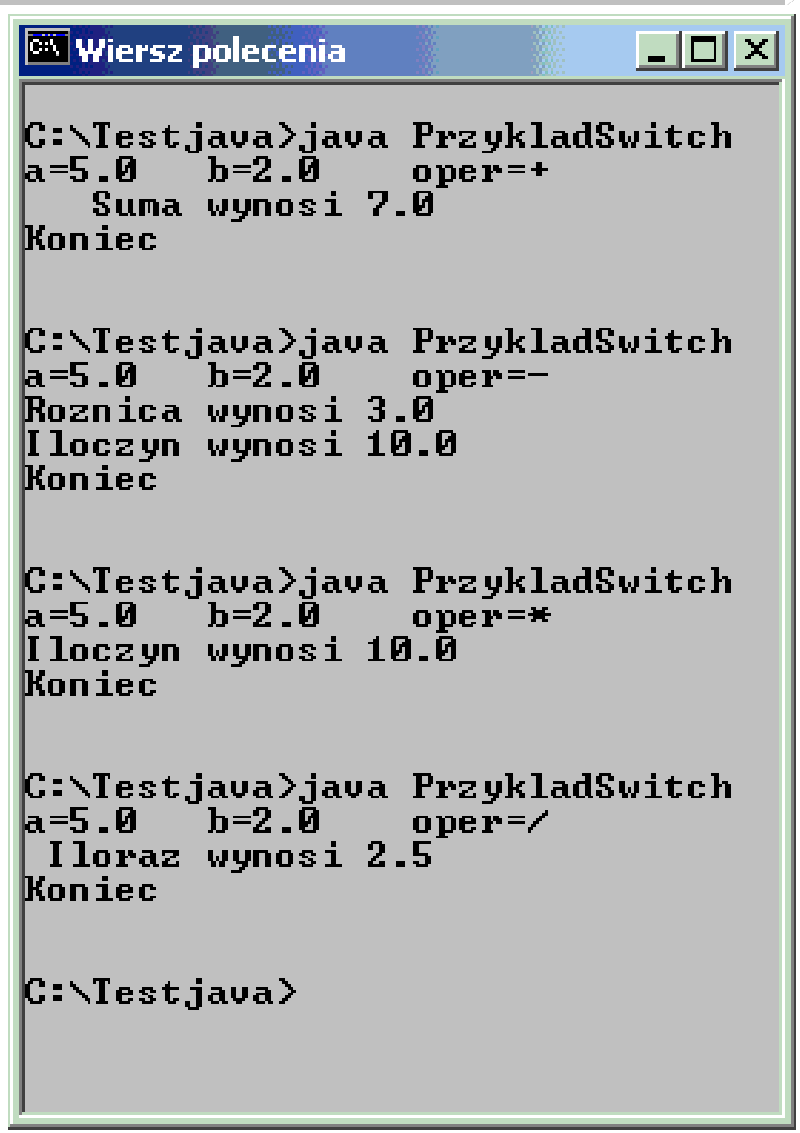


Przykład – instrukcja `switch`

```
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;   <<<<<<<< UWAGA: to jest komentarz
            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");
    }
}
```

Przykład – instrukcja `switch`

```
import  javax.swing.JOptionPane;

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

         a=Double.parseDouble(JOptionPane.showInputDialo
         b=Double.parseDouble(JOptionPane.showInputDialo
         oper=JOptionPane.showInputDialog(null, "Podaj d
         System.out.println("a=" + a + "   b=" + b + "

         switch(oper){
            case '+': System.out.println("   Suma wynosi
                      break;
            case '-': System.out.println("Roznica wynosi
                      // break;   <<<<<<<< UWAGA: to jes
            case '*': System.out.println("Iloczyn wynosi
                      break;
            case '/': System.out.println(" Iloraz wynosi
                      break;
            default: System.out.println("Nieznana operac
         }
         System.out.println("Koniec \n");
    }
}
```

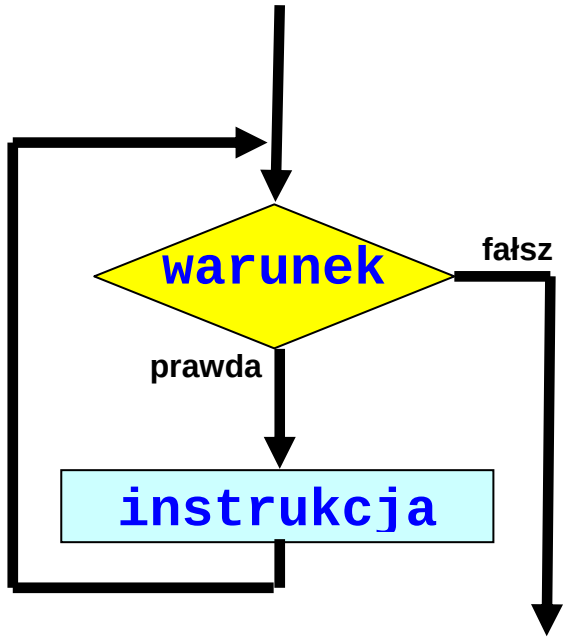


Postać pętli `while`

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;
    }
```

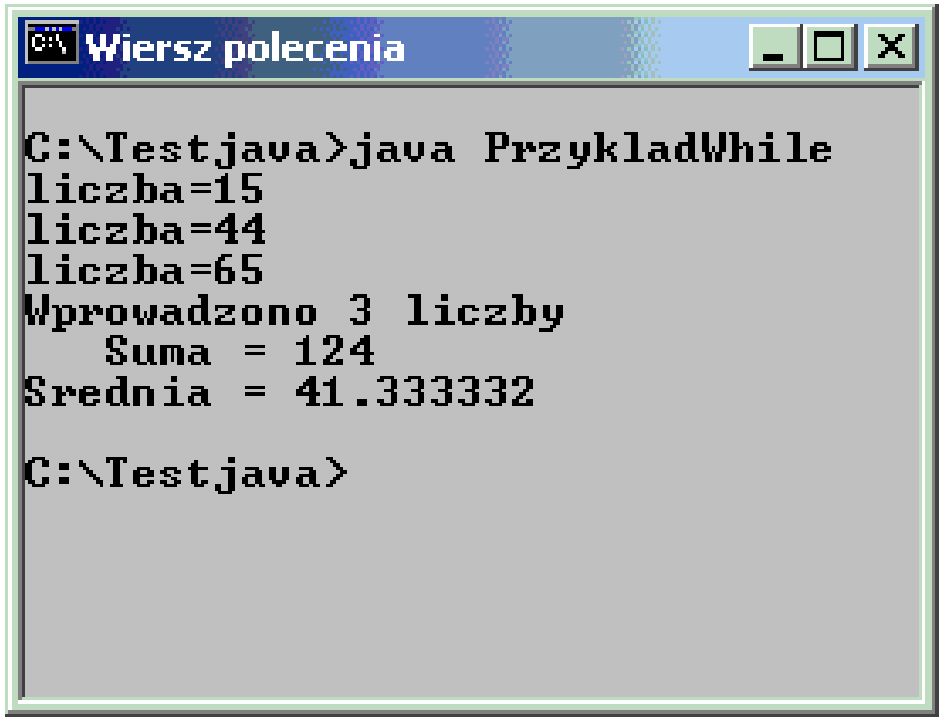


Przykład – pętla `while`

```
import  javax.swing.JOptionPane;

class PrzykladWhile{
    public static void main(String[] args){
        final int LIMIT = 100;
        String dana;
        int liczba, ilosc=0, suma = 0;
        while(suma<LIMIT){
            dana = JOptionPane.showInputDialog("Podaj liczbe");
            if (dana==null) break;
            liczba = Integer.parseInt(dana);
            System.out.println("liczba=" + liczba);
            suma += liczba;
            ilosc++;
        }
        System.out.println("Wprowadzono " + ilosc + " liczb");
        System.out.println("   Suma = " + suma);
        System.out.println("Srednia = " + (float)suma/ilosc);
    }
}
```

Przykład – pętla `while`

```
import  javax.swing.JOptionPane;

class PrzykladWhile{
    public static void main(String[] args){
        final int LIMIT = 100;
        String dana;
        int liczba, ilosc=0, suma = 0;
        while(suma<LIMIT){
            dana = JOptionPane.showInputDialog("Podaj liczbe");
            if (dana==null) break;
            liczba = Integer.parseInt(dana);
            System.out.println("liczba=" + licz
            suma += liczba;
            ilosc++;
        }
        System.out.println("Wprowadzono " + ilo
        System.out.println("   Suma = " + suma)
        System.out.println("Srednia = " + (floa
    }
}
```

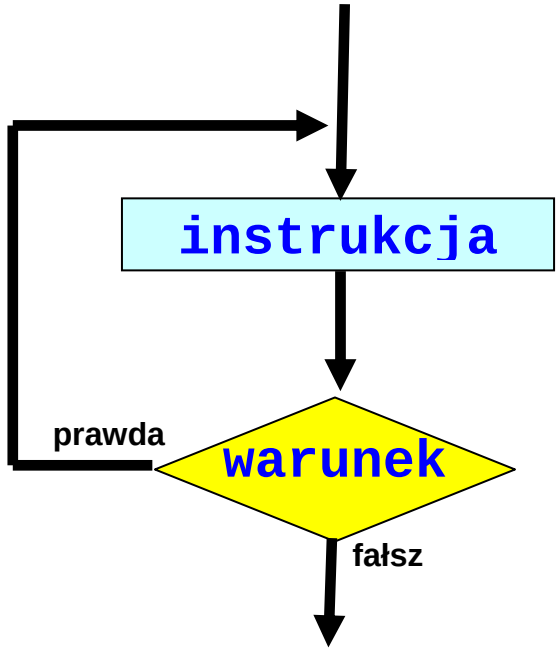


Postać pętli `do ... while`

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.

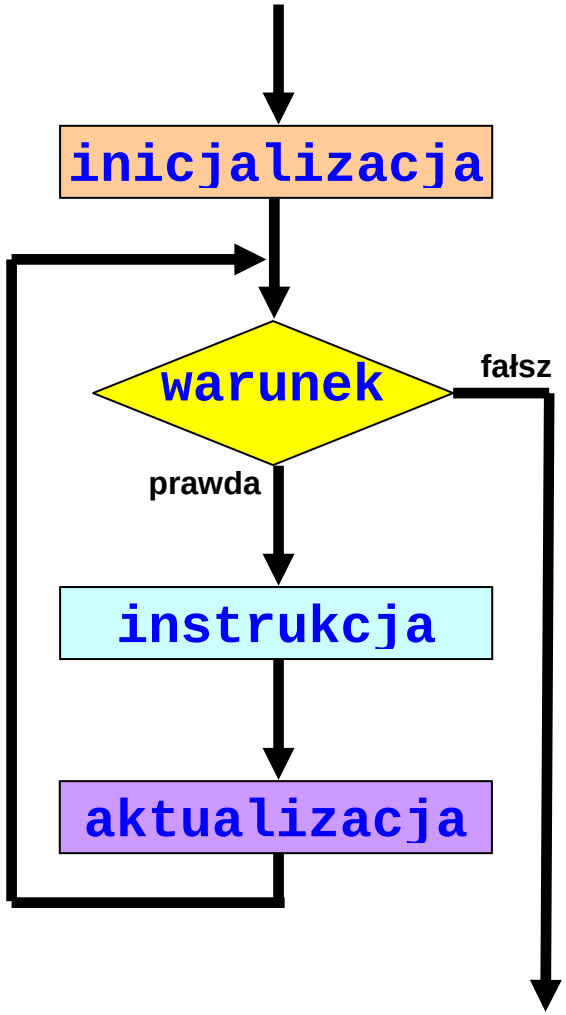
```
do
  {
   instrukcja;
  }
while(warunek);
```



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;
    }
```

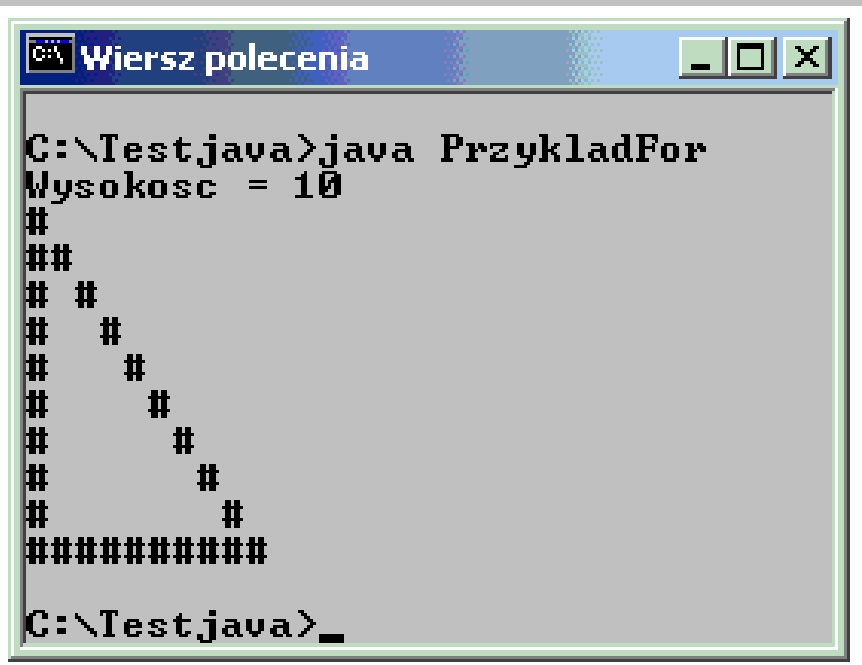


Przykład – pętla `for`

```
import  javax.swing.JOptionPane;

class PrzykladFor{
    public static void main(String[] args){
        int k, w, wysokosc;

        wysokosc = Integer.parseInt(JOptionPane.showInputDialog(null, "Podaj wysokosc"));

        for(w=1; w<=wysokosc; w++)
           {
            for(k=1; k<=wysokosc; k++)
               {
                 if((k==1)||(w==wysokosc)||(w==k))
                    {
                      System.out.print("#");
                    } else
                    {
                      System.out.print(" ");
                    }
               }
            System.out.println("");
           }
    }
}
```

Przykład – pętla `for`

```
import  javax.swing.JOptionPane;

class PrzykladFor{
    public static void main(String[] args){
        int k, w, wysokosc;

        wysokosc = Integer.parseInt(JOptionPane.showInputDialog(null, "Podaj wysokosc"));

        for(w=1; w<=wysokosc; w++)
           {
            for(k=1; k<=wysokosc; k++)
               {
                 if((k==1)||(w==wysokosc)||(w==k))
                    {
                      System.out.print("#");
                    } else
                    {
                      System.out.print(" ");
                    }
               }
            System.out.println("");
           }
    }
}
```



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.

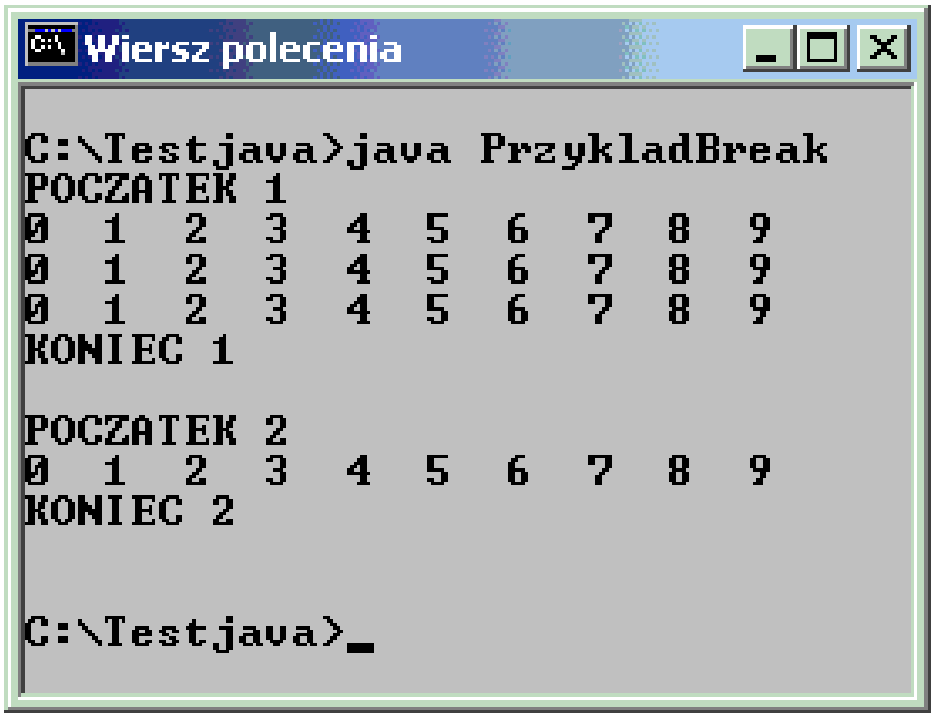
Instrukcja **`break`** stosowana jest również do opuszczania instrukcji **`switch`**.

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();
    }
}
```