

Podstawy programowania w języku JAVA wykład 2



dr inż. Jacek Czerniak

Zakład Informatyki
jczerniak@ukw.edu.pl



- **Komentarze**
- **słowa kluczowe**
- **Identyfikatory**
- **Typy danych**
- **Operatory**
- **Instrukcje**
- **Klasy i obiekty**
- **Kompozycja i dziedziczenie klas**
- **Klasy abstrakcyjne**
- **Interfejsy**

Komentarz wierszowy

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

Komentarz blokowy

```
/* Program wypisujący tekst powitania  
    Bydgoszcz, 01 listopada 2009 r.  
*/
```

Komentarz dokumentacyjny

```
/*  
    * Klasa parsująca plik PHP. Wersja PHP  
    * zależy od naciśniętego przycisku.  
    * @version 1.0.5  
    */
```

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
<u>boolean</u>	double	import	protected	throw
break	else	<u>instanceof</u>	public	throws
byte	extends	<u>int</u>	return	transient
case	false	interface	short	true
catch	final	long	static	try
char	finally	native	<u>strictfp</u>	void
class	float	new	super	volatile
const	for	null	switch	while
continue	<u>goto</u>			

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 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.

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**

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

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

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

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

- Identyfikatory `Zielony`, `Zolty`, `Czerwony` nazywamy 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.
```

- **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. 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

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

Zestawienie operatorów cz.2

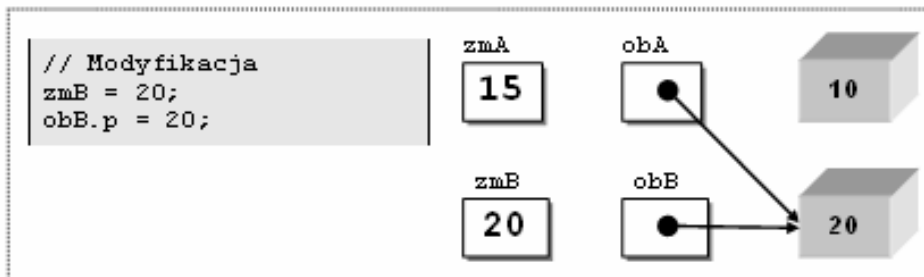
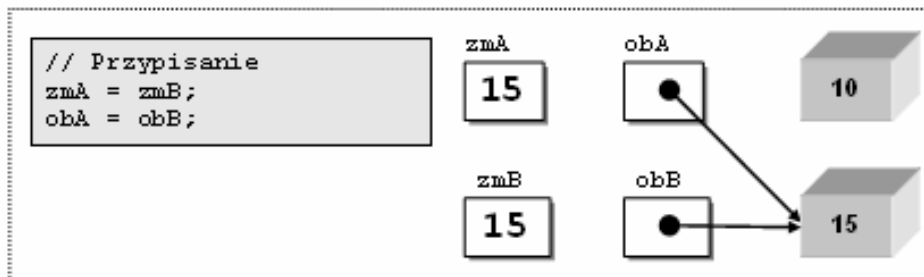
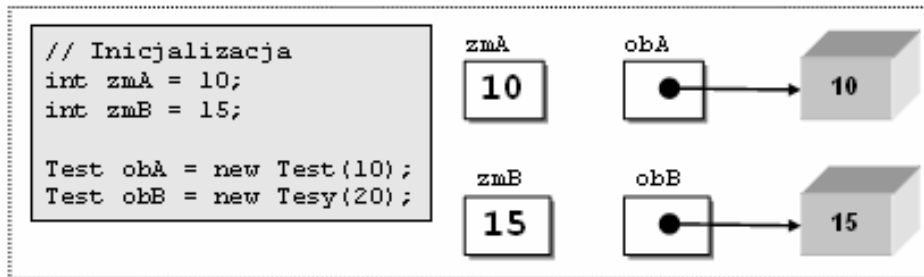
wiązanie i priorytet		operator	sposób użycia	działanie
lewe	4	+	<u>wyrażenie+wyrażenie</u>	dodawanie, łączenie łańcuchów,
		-	<u>wyrażenie-wyrażenie</u>	odejmowanie
lewe	5	<<	<u>wyrażenie<<wyrażenie</u>	przesunięcie bitowe w lewo
		>>	<u>wyrażenie>>wyrażenie</u>	przesunięcie bitowe w prawo
		>>>	<u>wyrażenie>>>wyrażenie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

Zestawienie operatorów cz.3

wiązanie i priorytet	operator	sposób użycia	działanie
13	? :	<u>wyraż ? wyraż : wyraż</u>	operator warunku
prawe	=	<u>zmienna=wyrażenie</u>	proste przypisanie
	=	<u>zmienna=wyrażenie</u>	pomnóż i przypisz
	/=	<u>zmienna/=wyrażenie</u>	podziel i przypisz
	%=	<u>zmienna%=wyrażenie</u>	oblicz modulo i przypisz
	+=	<u>zmienna+=wyrażenie</u>	dodaj i przypisz
	-=	<u>zmienna-=wyrażenie</u>	odejmij i przypisz
	<<=	<u>zmienna<<=wyrażenie</u>	przesuń w lewo i przypisz
	>>=	<u>zmienna>>=wyrażenie</u>	przesuń w prawo i przypisz
	>>>=	<u>zmienna>>>=wyrażenie</u>	przesuń w prawo bez znaku i przypisz
	&=	<u>zmienna&=wyrażenie</u>	koniunkcja bitowa i przypisz
	^=	<u>zmienna^=wyrażenie</u>	różnica bitowa i przypisz
	=	<u>zmienna =wyrażenie</u>	alternatywa bitowa i przypisz

- 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



```
C:\Testjava>Java Test
Inicjalizacja:
zmA = 10    zmB = 15
obA = 10    obB = 15

Przypisanie:
zmA = 15    zmB = 15
obA = 15    obB = 15

Modyfikacja:
zmA = 15    zmB = 20
obA = 20    obB = 20

C:\Testjava>
```

```
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) ;
    }
}
```

Instrukcja pusta – nie powoduje wykonania żadnych działań np. ;

Instrukcje wyrażeniowe:

- przypisanie np. `a = b;`
- preinkrementacja np. `++a;`
- predekrementacja np. `--b;`
- postinkrementacja 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.

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.

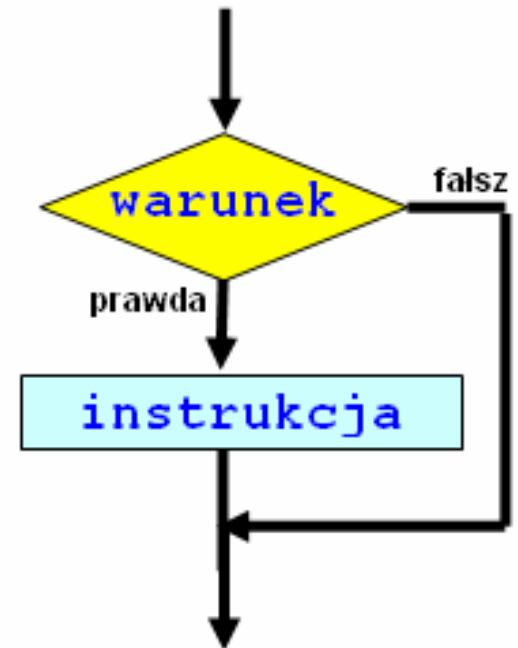
Instrukcja synchronized – wymuszanie synchronizacji przy współbieżnym wykonywaniu różnych wątków programu

Instrukcja warunkowa IF



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

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

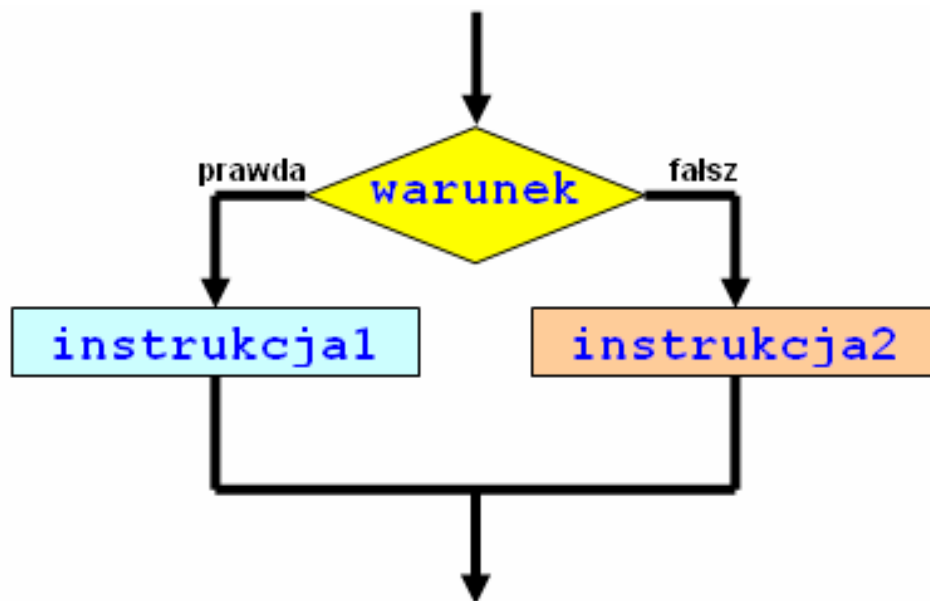


Instrukcja warunkowa if-else

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 *if*



```
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);
        }

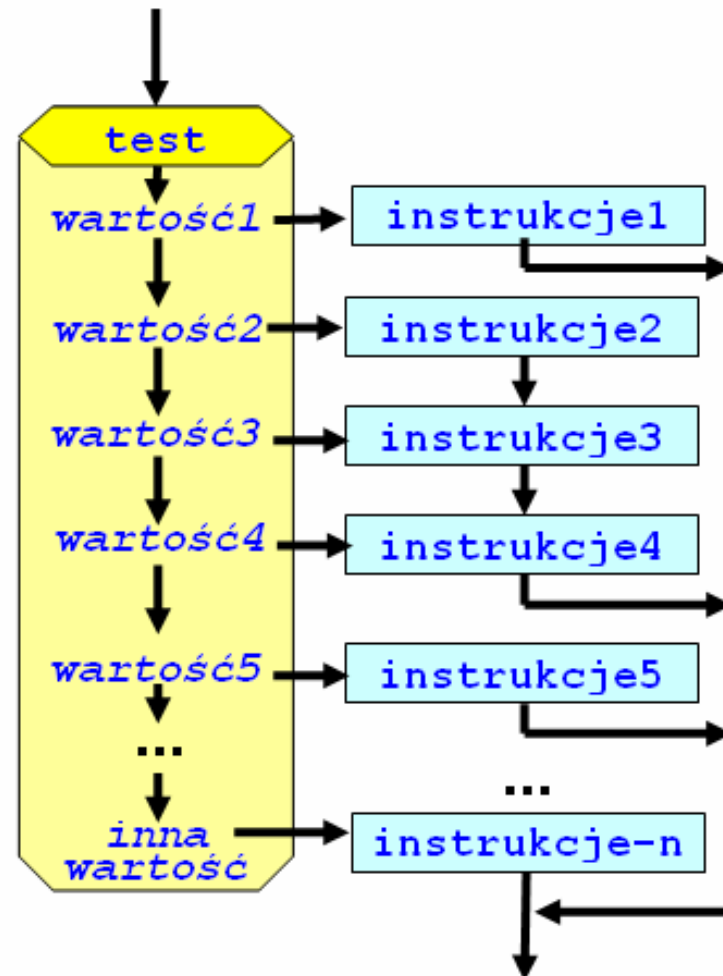
        delta = b*b-4*a*c;
        System.out.println("\n delta=" + delta);

        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");
    }
}
```

```
C:\Testjava>java PrzykladIf
2.0 x^2 + 1.0 x + 0.0 = 0
delta=1.0
x1=-0.5    x2=0.0
C:\Testjava>
```

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.


```
switch (test)
{
    case wartość1:
        instrukcje1;
        break;
    case wartość2:
        instrukcje2;
    case wartość3:
        instrukcje3;
    case wartość4:
        instrukcje4;
        break;
    case wartość5:
        instrukcje5;
        break;
    ...
    default:
        instrukcje-n;
}
```



Przykład switch



```
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");
    }
}
```

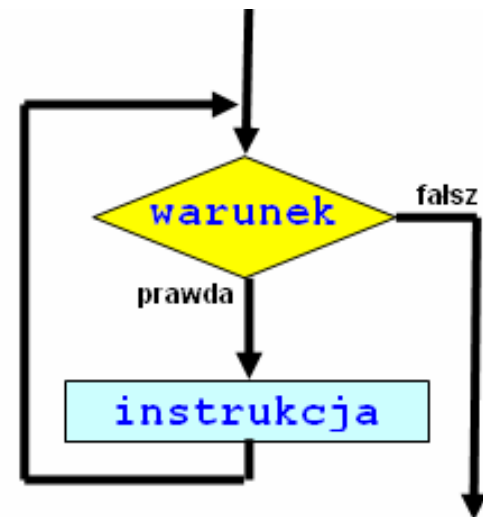
Pętla *while*



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;
}
```

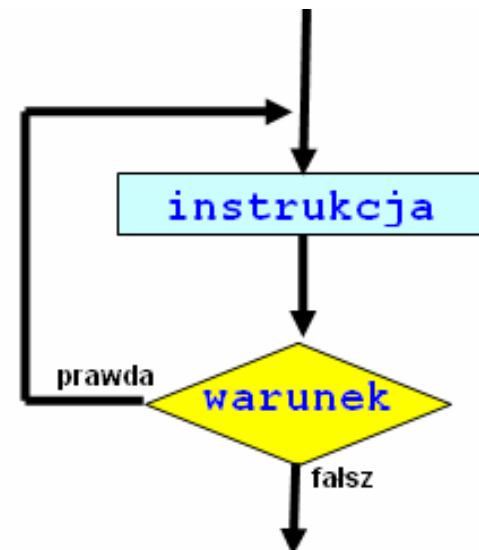


Pętla `do ... while`



- 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.

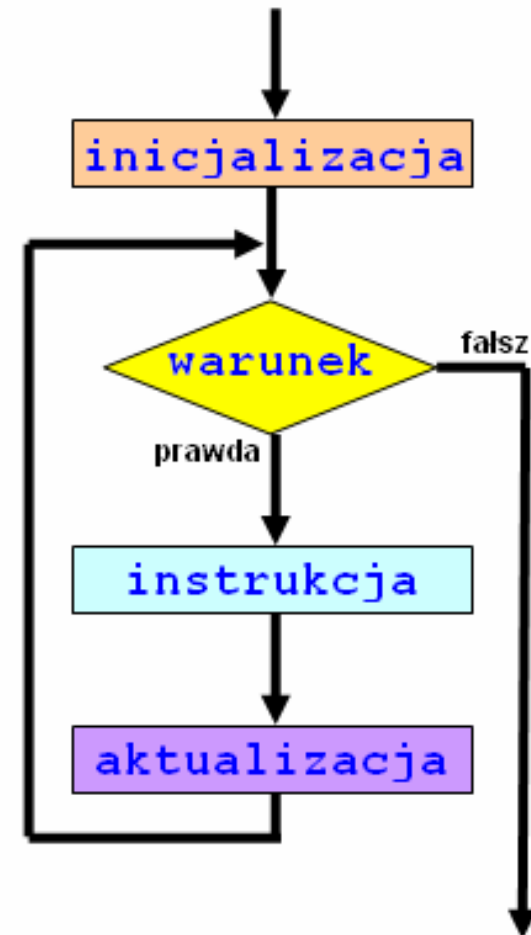
```
do
{
    instrukcja;
}
while(warunek);
```



Pętla *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 następująca po niej aktualizacja. W przeciwnym razie pętla jest przerywana.

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



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 zestawienie:

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

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

- 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 przerywane.
- Instrukcja **break** stosowana jest również do opuszczania instrukcji **switch**.

- Instrukcja **continue** przerywa wykonywanie bieżącego kroku pętli i wznowia 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 wznowiana jest iteracja tej pętli, która jest opatrzona tą etykietą.

Dziękuję za uwagę



dr inż. Jacek Czerniak
jczerniak@ukw.edu.pl