

Język programowania JAVA

© 2011-12 Radosław Klimek



Vincent Van GOGH: *Mężczyzna pijący filiżankę kawy*

Trochę z historii języka



- Początek lat 90-tych – prace nad językiem do oprogramowania sprzętu elektronicznego i urządzeń AGD (nowoczesnych, interaktywnych) – prace w laboratoriach Sun Microsystems pod kierunkiem Jamesa Goslinga,





- Początek lat 90-tych – prace nad językiem do oprogramowania sprzętu elektronicznego i urządzeń AGD (nowoczesnych, interaktywnych) – prace w laboratoriach Sun Microsystems pod kierunkiem **Jamesa Goslinga**,
- ok. 1993 – napisane całkowicie w nowym języku przeglądarki WebRunner coraz bardziej popularnego Internetu, z czasem nowa nazwa przeglądarki HotJava (pierwsza przeglądarka obsługująca aplety),



- Początek lat 90-tych – prace nad językiem do oprogramowania sprzętu elektronicznego i urządzeń AGD (nowoczesnych, interaktywnych) – prace w laboratoriach Sun Microsystems pod kierunkiem **Jamesa Goslinga**,
- ok. 1993 – napisane całkowicie w nowym języku przeglądarki WebRunner coraz bardziej popularnego Internetu, z czasem nowa nazwa przeglądarki **HotJava** (pierwsza przeglądarka obsługująca aplety),
- rok 1995 – oficjalne ogłoszenie nowoczesnego języka dostosowanego dla potrzeb Internetu,



- Początek lat 90-tych – prace nad językiem do oprogramowania sprzętu elektronicznego i urządzeń AGD (nowoczesnych, interaktywnych) – prace w laboratoriach Sun Microsystems pod kierunkiem **Jamesa Goslinga**,
- ok. 1993 – napisane całkowicie w nowym języku przeglądarki WebRunner coraz bardziej popularnego Internetu, z czasem nowa nazwa przeglądarki **HotJava** (pierwsza przeglądarka obsługująca aplety),
- rok 1995 – oficjalne ogłoszenie nowoczesnego języka dostosowanego dla potrzeb Internetu,
- początkowo nowy język została nazwany Oak, z czasem wybrano inną nazwę Java.



- Początek lat 90-tych – prace nad językiem do oprogramowania sprzętu elektronicznego i urządzeń AGD (nowoczesnych, interaktywnych) – prace w laboratoriach Sun Microsystems pod kierunkiem **Jamesa Goslinga**,
- ok. 1993 – napisane całkowicie w nowym języku przeglądarki WebRunner coraz bardziej popularnego Internetu, z czasem nowa nazwa przeglądarki **HotJava** (pierwsza przeglądarka obsługująca aplety),
- rok 1995 – oficjalne ogłoszenie nowoczesnego języka dostosowanego dla potrzeb Internetu,
- początkowo nowy język została nazwany Oak, z czasem wybrano inną nazwę **Java**.

Znaczenie języka Java



Znaczenie języka Java

- Pewne koncepcje zapożyczone z języka Smalltalk (silna obiektowość, maszyna wirtualna, zarządzanie pamięcią) oraz z C++ (słowa kluczowe, składnia),



Znaczenie języka Java

- Pewne koncepcje zapożyczone z języka Smalltalk (silna obiektowość, maszyna wirtualna, zarządzanie pamięcią) oraz z C++ (słowa kluczowe, składnia),
- programy kompilowane do kodu bajtowego, a więc postaci pośredniej wykonywanej przez maszynę wirtualną, niezależność od architektury,



Znaczenie języka Java

- Pewne koncepcje zapożyczone z języka Smalltalk (silna obiektowość, maszyna wirtualna, zarządzanie pamięcią) oraz z C++ (słowa kluczowe, składnia),
- programy kompilowane do kodu bajtowego, a więc postaci pośredniej wykonywanej przez maszynę wirtualną, niezależność od architektury,
- bezpieczeństwo konstrukcji, m.in. względnie silne typowanie, brak wskaźników, brak dziedziczenia wielobazowego,



Znaczenie języka Java

- Pewne koncepcje zapożyczone z języka Smalltalk (silna obiektowość, maszyna wirtualna, zarządzanie pamięcią) oraz z C++ (słowa kluczowe, składnia),
- programy kompilowane do kodu bajtowego, a więc postaci pośredniej wykonywanej przez maszynę wirtualną, niezależność od architektury,
- bezpieczeństwo konstrukcji, m.in. względnie silne typowanie, brak wskaźników, brak dziedziczenia wielobazowego,
- wsparcie dla programowania rozproszonego i sieciowego,



Znaczenie języka Java

- Pewne koncepcje zapożyczone z języka Smalltalk (silna obiektowość, maszyna wirtualna, zarządzanie pamięcią) oraz z C++ (słowa kluczowe, składnia),
- programy kompilowane do kodu bajtowego, a więc postaci pośredniej wykonywanej przez maszynę wirtualną, niezależność od architektury,
- bezpieczeństwo konstrukcji, m.in. względnie silne typowanie, brak wskaźników, brak dziedziczenia wielobazowego,
- wsparcie dla programowania rozproszonego i sieciowego,
- niezawodność i bezpieczeństwo (np. wyjątki, logi, asercje).



Niektóre środowiska Javy

JSE (Java Standard Edition) – środowisko programistyczne do tworzenia aplikacji desktopowych oraz apletów na strony WWW;

- JSE (Java Standard Edition) – środowisko programistyczne do tworzenia aplikacji desktopowych oraz apletów na strony WWW;
- JEE (Java Enterprise Edition) – środowisko programistyczne do tworzenia rozbudowanych biznesowych aplikacji, rozszerzenie platformy JSE;

- JSE (Java Standard Edition) – środowisko programistyczne do tworzenia aplikacji desktopowych oraz apletów na strony WWW;
- JEE (Java Enterprise Edition) – środowisko programistyczne do tworzenia rozbudowanych biznesowych aplikacji, rozszerzenie platformy JSE;
- JME (Java Micro Edition) – środowisko programistyczne do tworzenia aplikacji na niewielkie urządzenia typu telefony komórkowe;

- JSE (Java Standard Edition) – środowisko programistyczne do tworzenia aplikacji desktopowych oraz apletów na strony WWW;
- JEE (Java Enterprise Edition) – środowisko programistyczne do tworzenia rozbudowanych biznesowych aplikacji, rozszerzenie platformy JSE;
- JME (Java Micro Edition) – środowisko programistyczne do tworzenia aplikacji na niewielkie urządzenia typu telefony komórkowe;
- Java Card Technology – środowisko programistyczne do tworzenia oprogramowania dla inteligentnych kart (np. kart bankomatowych) o bardzo małej pamięci i niewielkiej mocy obliczeniowej.

Niektóre konwencje notacyjne

Proponowana konwencja nazewnictwa:

nazwy klas – rozpoczynamy wielką literą:

```
class Klasa { ... }
```

nazwy metod – rozpoczynamy z małej litery:

```
void metoda () { ... }
```

nazwy zmiennych – rozpoczynamy od małej litery:

```
int zmienna ;
```

nazwy wielocłonowe – kolejne człony z wielkiej litery:

```
1 class NazwaMojejKlasy { ... }  
2 void toMojaMetoda () { ... }  
3 int jakasZmienna ;
```

Niektóre konwencje notacyjne (cd.)

nazwy stałych – wielkimi literami, ze znakami podkreślenia:

```
final int TO_JEST_STALA;
```

Ponadto, klamry otwierające warto stosować odpowiednio:

```
1 class Klasa{  
2     void metoda{  
3         int zmienna;  
4     }  
5 }
```

Wielkość liter może mieć znaczenie:

```
1 int liczba ;  
2 int Liczba ;
```

aplet – (ang. *applet*) niewielki program napisany w taki sposób, aby mógł zostać osadzony w stronie WWW wprost i wykonany przez przeglądarkę internetową na komputerze, na którym jest ona uruchomiona, może być dostarczany w postaci kodu bajtowego Javy. Może zostać uruchomiony w przeglądarce internetowej wykorzystując wirtualną maszynę Javy albo w samodzielnej aplikacji AppletViewer służącej do testowania apletów Javy.

serwlet – klasa Javy działająca po stronie serwera WWW w modelu żądanie-odpowiedź, rozszerzająca jego możliwości. Uruchamiane są w bezpiecznym środowisku serwera aplikacji (np. GlassFish) albo kontenera webowego (np. Apache Tomcat). Jako część platformy JEE, serwlety mają dostęp do całego API Javy.

- W Javie podobnie jak w innych językach wyróżniamy wiele typów danych mogących przechowywać zarówno liczby stałoprzecinkowe, zmiennoprzecinkowe, znaki, ciągi znaków, oraz typ logiczny.
- Typy proste reprezentują pojedyncze wartości, a nie złożone obiekty.
- Choć Java jest językiem zorientowanym obiektowo, to typy proste nie bazują na modelu obiektowym.
- Każdy obiekt musi mieć określony typ.

Typy proste

W języku Java zdefiniowano osiem typów prostych, które można podzielić na cztery grupy:

W języku Java zdefiniowano osiem typów prostych, które można podzielić na cztery grupy:

- 1 **typy całkowite:** byte, short, int i long – reprezentują liczby całkowite,

W języku Java zdefiniowano osiem typów prostych, które można podzielić na cztery grupy:

- 1 **typy całkowite:** byte, short, int i long – reprezentują liczby całkowite,
- 2 **typy rzeczywiste:** float i double – reprezentują liczby rzeczywiste,

W języku Java zdefiniowano osiem typów prostych, które można podzielić na cztery grupy:

- 1 **typy całkowite:** byte, short, int i long – reprezentują liczby całkowite,
- 2 **typy rzeczywiste:** float i double – reprezentują liczby rzeczywiste,
- 3 **typ znakowy:** char – reprezentuje pojedyncze znaki,

W języku Java zdefiniowano osiem typów prostych, które można podzielić na cztery grupy:

- 1 **typy całkowite**: byte, short, int i long – reprezentują liczby całkowite,
- 2 **typy rzeczywiste**: float i double – reprezentują liczby rzeczywiste,
- 3 **typ znakowy**: char – reprezentuje pojedyncze znaki,
- 4 **typ logiczny**: boolean – reprezentuje wartości logiczne (true, false).

nazwa	szerokość	zakres
byte	8	-128 .. 127
short	16	-32 768 .. 32 767
int	32	-2 147 483 648 .. 2 147 483 647
long	64	-9 223 372 036 854 775 808 .. 9 223 372 036 854 775 807

Java nie posiada typu **unsigned** (bez znaku), czego konsekwencją jest to, że przekraczając zakres danego typu przejdziemy na zakres ujemny.

Typ **byte** najbardziej nadaje się do wczytywania danych z plików lub sieci, gdyż umożliwia rozpoznawanie protokołów sieciowych i formatów plików. Używa się go także do wykonywania operacji bitowych.

Przykładowe instrukcje deklarujące zmienne typu byte:

```
1 byte b ;  
2 byte c = 0x55 ;
```

Typ **short** jest najrzadziej używanym typem Javy, ze względu na odwrotną kolejność zapisywania bajtów.

Przykładowe instrukcje deklarujące zmienne typu short:

```
1 short a ;  
2 short b = 0x55ab ;
```

Typ `int` jest najczęściej używanym typem całkowitym ze względu na możliwość zapisywania bardzo dużych liczb, nadaje się do stosowania w pętlach i indeksowania tablic.

Deklaracja zmiennych typu `int`:

```
1 int a ;  
2 int b = 0x55aa0000 ;
```

Typ **long** umożliwia zapisywanie olbrzymich liczb. Przypadki kiedy typ **int** okazuje się niewystarczający są bardzo rzadkie. (Np. zliczanie milisekund w okresie roku lub dłuższym, albo mnożenie bardzo dużych liczb).

Deklaracja zmiennych typu **long**:

```
1 long m;  
2 long n = 0x55aa000055aa0000 ;
```


Typy zmiennoprzecinkowe

Istnieją dwa typy zmiennoprzecinkowe **float** i **double**:

nazwa	szerokość	zakres
float	8	$1,7 * 10^{-308} .. 1,7 * 10^{308}$
double	16	$3,4 * 10^{-38} .. 3,4 * 10^{38}$

Inaczej typy rzeczywiste, używa się ich na przykład przy obliczaniu pierwiastka kwadratowego albo wykonywania funkcji trygonometrycznych.

- Typ **char** reprezentuje znaki w języku Java. Są to 16 bitowe kody zgodnie ze standardem **Unicode**.
- Ich dopuszczalny zakres wynosi od 0 do 65536.
- Standardowe znaki ASCII to zakres od 0 do 127.
- ISO-Latin-1 to zakres od 0 do 255.

- W Javie istnieje typ prosty **boolean**, który umożliwia stosowanie wartości logicznych (true, false).
- Jest to typ określający wartości zwracane przez wszystkie operacje porównań np. $a < b$.

Deklaracja zmiennej typu **boolean**:

```
1 boolean done = false ;  
2 boolean active = true ;
```

Czasami zdarzają się sytuacje, w których wartość określonego typu trzeba zapisać w zmiennej innego typu.

Czasami zdarzają się sytuacje, w których wartość określonego typu trzeba zapisać w zmiennej innego typu.

- **Poszerzanie typów** – np. rzutowanie typu **byte** do typu **int**.
Typ źródłowy i typ docelowy są zgodne albo typ docelowy jest większy od typu źródłowego to kompilator dokonuje automatycznej konwersji typów.

Czasami zdarzają się sytuacje, w których wartość określonego typu trzeba zapisać w zmiennej innego typu.

- **Poszerzanie typów** – np. rzutowanie typu **byte** do typu **int**.
Typ źródłowy i typ docelowy są zgodne albo typ docelowy jest większy od typu źródłowego to kompilator dokonuje automatycznej konwersji typów.
- **Zawężanie typów** – np. rzutowanie typu **int** do typu **byte**.
Wartość typu większego zapisywana jest do zmiennej typu mniejszego. Dokonujemy tego jawnie w tekście programu następującą instrukcją: (typ docelowy) wartość.

```
1      int a = 100;  
2      byte b = (byte) a;
```

- **Konwersja typów całkowitych** – wartość typu większego jest zredukowana modulo maksymalny zakres typu docelowego, np.: (byte) i, dla $i = 257$ to 1
- **Zawężanie typów rzeczywistych na całkowite** – wartość typu rzeczywistego traci część dziesiętną, a otrzymana wartość całkowita jest przekształcana jak liczby całkowite.

Tablica jest typem umożliwiającym grupowanie zmiennych tego samego typu i odwoływanie się do nich za pomocą wspólnej nazwy. Elementy tablic mogą być zmiennymi dowolnego typu, zarówno prostego jak i złożonego. Tworzenie tablicy przebiega w dwóch etapach:

Tablica jest typem umożliwiającym grupowanie zmiennych tego samego typu i odwoływanie się do nich za pomocą wspólnej nazwy. Elementy tablic mogą być zmiennymi dowolnego typu, zarówno prostego jak i złożonego. Tworzenie tablicy przebiega w dwóch etapach:

❶ **deklaracja zmiennej typu tablicowego:**

```
typ ZmiennaTablicowa[ ];
```

deklaracja ta tworzy zmienną tablicową z przypisaną wartością **null** reprezentującą tablicę bez elementów.

Tablica jest typem umożliwiającym grupowanie zmiennych tego samego typu i odwoływanie się do nich za pomocą wspólnej nazwy. Elementy tablic mogą być zmiennymi dowolnego typu, zarówno prostego jak i złożonego. Tworzenie tablicy przebiega w dwóch etapach:

- 1 **deklaracja zmiennej typu tablicowego:**
typ `ZmiennaTablicowa[]`;
deklaracja ta tworzy zmienną tablicową z przypisaną wartością **null** reprezentującą tablicę bez elementów.
- 2 **alokacja pamięci niezbędnej do przechowywania elementów tablicy** (alokacja dynamiczna):
`ZmiennaTablicowa = new typ [rozmiar]`;
typ jest typem elementów tablicy
rozmiar jest liczbą elementów tablicy

Tablice (2/3)

```
1 class Array
2 {
3     public static void main (string args[])
4     {
5         int MonthDays[];
6         MonthDays = new int [12];
7         MonthDays[0] = 31;
8         MonthDays[1] = 29;
9         MonthDays[2] = 31;
10        MonthDays[3] = 30;
11        MonthDays[4] = 31;
12        MonthDays[5] = 30;
13        MonthDays[6] = 31;
14        MonthDays[7] = 31;
15        MonthDays[8] = 30;
16        MonthDays[9] = 31;
17        MonthDays[10] = 30;
18        MonthDays[11] = 31;
19        System.out.println ( " Kwiecień ma " +MonthDays[3]+ " dni . " );
20    }
21 }
```

Wynik działania programu to:

```
1 C:\>java Array
2 Kwiecień ma 30 dni.
```

Tablice (3/3)

Deklarację i alokację tablicy można zapisać w jednej linii:

```
1 int MonthDays[] = new int [12];
```

Tablice można też inicjalizować automatycznie, podobnie jak w przypadku typów prostych.

Nie trzeba podawać rozmiaru tablicy, jest on automatycznie dostosowywany do liczby określonych wartości. Tablica zostanie utworzona automatycznie z alokowanym miejscem wystarczającym dla wszystkich elementów tablicy.

```
1 class Array
2 {
3     public static void main (string args [])
4     {
5         int MonthDays[]= {31, 29, 31, 30, 31,30, 31, 31, 30, 31, 30, 31};
6         System.out.println ("Kwiecień ma "+MonthDays[3]+" dni.");
7     }
8 }
```

Efekt działania tego programu będzie identyczny z poprzednim:

```
1 C:\>java Array
2 Kwiecień ma 30 dni.
```

Tablice wielowymiarowe

Zasadniczo w Javie nie ma tablic wielowymiarowych, a jedynie tablice tablic, które poza drobnymi wyjątkami działają tak samo.

Alokując pamięć na tablicę wielowymiarową należy określić co najmniej jeden najbardziej lewy wymiar (liczba wierszy).

Liczbę elementów w każdym wierszu można alokować oddzielnie.

Tablice wielowymiarowe

Zasadniczo w Javie nie ma tablic wielowymiarowych, a jedynie tablice tablic, które poza drobnymi wyjątkami działają tak samo.

Alokując pamięć na tablicę wielowymiarową należy określić co najmniej jeden najbardziej lewy wymiar (liczba wierszy).

Liczbę elementów w każdym wierszu można alokować oddzielnie.

- Deklaracja tablicy wielowymiarowej:

```
typ nazwaTablicy [][];
```

Alokacja tablicy:

```
nazwaTablicy = new typ[rozmiar1][rozmiar2];
```

Tablice wielowymiarowe

Zasadniczo w Javie nie ma tablic wielowymiarowych, a jedynie tablice tablic, które poza drobnymi wyjątkami działają tak samo.

Alokując pamięć na tablicę wielowymiarową należy określić co najmniej jeden najbardziej lewy wymiar (liczba wierszy).

Liczbę elementów w każdym wierszu można alokować oddzielnie.

- Deklaracja tablicy wielowymiarowej:

```
typ nazwaTablicy [][];
```

Alokacja tablicy:

```
nazwaTablicy = new typ[rozmiar1][rozmiar2];
```

- Deklaracja i alokacja tablicy:

```
typ nazwaTablicy [][] = new typ[rozmiar1][rozmiar2];
```

Tablice wielowymiarowe (cd.)

Oto przykładowy program, który tworzy tablicę dwuwymiarową 4x4, a następnie inicjuje ją w taki sposób, że na przekątnej będą jedynki (tablica jednostkowa):

```
1  class Tablica
2  {
3      public static void main (String args[])
4      {
5          {int tab [][];
6          tab = new int [4][4];
7          tab [0][0] = 1;
8          tab [1][1] = 1;
9          tab [2][2] = 1;
10         tab [3][3] = 1;
11     }
12 }
```


Tablice wielowymiarowe – niesymetryczne

Tablice wielowymiarowe nie muszą być symetryczne, tzn nie wszystkie wiersze muszą mieć taką samą ilość elementów.

Tablice wielowymiarowe – niesymetryczne

Tablice wielowymiarowe nie muszą być symetryczne, tzn nie wszystkie wiersze muszą mieć taką samą ilość elementów.

- Oto deklaracja takiej tablicy:

```
1 int [][] tab = new int [3][];  
2 tab[0] = new int [3];  
3 tab[1] = new int [2];  
4 tab[2] = new int [1];
```

Tablice wielowymiarowe – niesymetryczne

Tablice wielowymiarowe nie muszą być symetryczne, tzn nie wszystkie wiersze muszą mieć taką samą ilość elementów.

- Oto deklaracja takiej tablicy:

```
1 int [][] tab = new int [3] [] ;  
2 tab [0] = new int [3] ;  
3 tab [1] = new int [2] ;  
4 tab [2] = new int [1] ;
```

- W ten sposób możemy zinicjować elementy takiej tablicy:

```
1 for (int i=0; i < tab.length; i++)  
2     for (int j=0; j < tab[i].length; j++)  
3         tab[i][j] = 1;
```

Tablice wielowymiarowe – niesymetryczne

Tablice wielowymiarowe nie muszą być symetryczne, tzn nie wszystkie wiersze muszą mieć taką samą ilość elementów.

- Oto deklaracja takiej tablicy:

```
1 int [][] tab = new int [3] [] ;  
2 tab [0] = new int [3] ;  
3 tab [1] = new int [2] ;  
4 tab [2] = new int [1] ;
```

- W ten sposób możemy zinicjować elementy takiej tablicy:

```
1 for (int i=0; i < tab.length; i++)  
2     for (int j=0; j < tab[i].length; j++)  
3         tab[i][j] = 1;
```

- Otrzymamy taką tablicę:

```
1 1 1 1  
2 1 1  
3 1
```



Operatory to specjalne znaki instruujące kompilator, aby wykonał **operacje** na **operandach**, będącymi zmiennymi, wyrażeniami lub literałami. W javie istnieją operatory:

- jednoargumentowe
- dwuargumentowe
- trójargumentowy (jeden)

Operatory mogą być:

- przedrostkowe (prefix)
- przyrostkowe(postfix)
- wrostkowe (infix)

W Javie istnieją łącznie 44 operatory.

Dla każdego operatora jest określony typ operandów i rodzaj wykonywanej operacji.

Operatory można podzielić na cztery grupy:

- matematyczne
- bitowe
- logiczne
- relacyjne

Operatory matematyczne

Operandy muszą być jednego z typów numerycznych lub typu **char**.

Niedozwolone jest użycie typu **boolean**.

operator	działanie	operator	działanie
+	dodawanie	+=	przypisanie z dodawaniem
-	odejmowanie, minus jednoargumentowy	-=	przypisanie z odejmowaniem
*	mnożenie	*=	przypisanie z mnożeniem
/	dzielenie	/=	przypisanie z dzieleniem
%	dzielenie modulo	%=	przypisanie z dzieleniem modulo
++	inkrementacja	--	dekrementacja

Operatory matematyczne

Operatory matematyczne

- Operatory dodawania, odejmowania, mnożenia i dzielenia działają tak samo jak w matematyce

Operatory matematyczne

- Operatory dodawania, odejmowania, mnożenia i dzielenia działają tak samo jak w matematyce
- jednoargumentowy operator – oblicza wartość przeciwną

Operatory matematyczne

- Operatory dodawania, odejmowania, mnożenia i dzielenia działają tak samo jak w matematyce
- jednoargumentowy operator – oblicza wartość przeciwną
- dzielenie modulo – można go stosować również z typami zmiennoprzecinkowymi:

```
1 int x = 35;  
2 double y = 35.4;  
3 x = x % 10;  
4 y = y % 10;  
5 // otrzymamy następujące wyniki:  
6 x = 5  
7 y = 5.4
```

Operatory matematyczne (cd.)

- przypisania z operatorami matematycznymi

```
1 a = a + 5 /* można zapisać */ a += 5  
2 a = a % 2 /* można zapisać */ a %= 2
```

Operatory matematyczne (cd.)

- przypisania z operatorami matematycznymi

```
1 a = a + 5 /* można zapisać */ a += 5  
2 a = a % 2 /* można zapisać */ a %= 2
```

- inkrementacja i dekrementacja (zwiększenie i zmniejszenie o jeden)

```
1 i = i + 1 /* można zapisać */ ++i albo i++  
2 i = i - 1 /* można zapisać */ --i albo i--
```

Operatory bitowe

Operator	Działanie	Operator	Działanie
~	Bitowa negacja logiczna (NOT)		
&	Bitowy iloczyn logiczny (AND)	&=	Przypisanie z operacją AND
	Bitowa suma logiczna (OR)	=	Przypisanie z operacją OR
^	Bitowa różnica symetryczna (XOR)	^=	Przypisanie z operacją XOR
>>	Przesunięcie w prawo	>>=	Przypisanie z przesunięciem w prawo
>>>	Przesunięcie w prawo bez znaku	>>>=	Przypisanie z przesunięciem w prawo bez znaku
<<	Przesunięcie w lewo	<<=	Przypisanie z przesunięciem w lewo

Podstawowe operatory bitowe

Operatory bitowe umożliwiają wykonywanie operacji na pojedynczych bitach.

Cztery podstawowe operacje bitowe to:

- negacja logiczna NOT
- iloczyn logiczny AND
- suma logiczna OR
- różnica symetryczna XOR

Działanie podstawowych operacji bitowych:

A	B	A OR B	A AND B	A XOR B	NOT A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

Operatory przesunięcia bitowego

W Javie są trzy przesunięcia:

Operatory przesunięcia bitowego

W Javie są trzy przesunięcia:

- **przesunięcie w lewo** `<<` przesuwa w lewo bity lewego operandu o ilczbę pozycji określoną przez prawy operand. Przesunięcie w lewo o n bitów powoduje utratę pierwszych n bitów i dopisanie n zer z prawej strony. (Przesunięcie o 1 pozycję w lewo odpowiada mnożeniu przez 2)

Operatory przesunięcia bitowego

W Javie są trzy przesunięcia:

- **przesunięcie w lewo** \ll przesuwają w lewo bity lewego operandu o liczbę pozycji określoną przez prawy operand. Przesunięcie w lewo o n bitów powoduje utratę pierwszych n bitów i dopisanie n zer z prawej strony. (Przesunięcie o 1 pozycję w lewo odpowiada mnożeniu przez 2)
- **przesunięcie w prawo** \gg przesuwają w prawo bity lewego operandu o liczbę pozycji określoną przez prawy operand. Przesunięcie w prawo powoduje utratę n pierwszych bitów z prawej strony, a lewej strony dopisywane są bity znaku. (Przesunięcie o 1 pozycję w prawo odpowiada dzieleniu przez 2)

Operatory przesunięcia bitowego

W Javie są trzy przesunięcia:

- **przesunięcie w lewo** \ll przesuwają w lewo bity lewego operandu o liczbę pozycji określoną przez prawy operand. Przesunięcie w lewo o n bitów powoduje utratę pierwszych n bitów i dopisanie n zer z prawej strony. (Przesunięcie o 1 pozycję w lewo odpowiada mnożeniu przez 2)
- **przesunięcie w prawo** \gg przesuwają w prawo bity lewego operandu o liczbę pozycji określoną przez prawy operand. Przesunięcie w prawo powoduje utratę n pierwszych bitów z prawej strony, a lewej strony dopisywane są bity znaku. (Przesunięcie o 1 pozycję w prawo odpowiada dzieleniu przez 2)
- **przesunięcie w prawo bez znaku** \ggg powoduje, że przy przesuwaniu w prawo z prawej strony dopisywane są zera.

Przypisania z operatorami bitowymi

Podobnie jak w przypadku operatorów matematycznych, ze wszystkimi operatorami bitowymi są związane odpowiadające im operatory przypisań.

```
a = a >> 5 można zapisać a >>= 5
```

Operatory relacyjne

Operatory relacyjne umożliwiają porównywanie dwóch wartości.

Operator	Działanie	Operator	Działanie
==	równy	!=	różny
>	większy	<	mniejszy
>=	większy lub równy	<=	mniejszy lub równy

Każdy operator relacyjny zwraca jedną z wartości typu **boolean**.
Za pomocą operatorów == i != można porównywać wartości dowolnych typów.

Pozostałe operatory pozwalają porównywać tylko wartości numeryczne (liczby całkowite i zmiennoprzecinkowe oraz znaki).

Operatory logiczne

Operatory logiczne mają znaczenie analogiczne do odpowiednich operatorów bitowych, ale działają one na wartościach logicznych.

Operator	Działanie	Operator	Działanie
!	Negacja logiczna (NOT)	?:	Trójargumentowy operator if-then-else
&	Iloczyn logiczny (AND)	&=	Przypisanie z operacją AND
	Suma logiczna (OR)	=	Przypisanie z operacją OR
^	Logiczna różnica symetryczna (XOR)	^=	Przypisanie z operacją XOR
&&	Szybka operacja AND	==	równy
	Szybka operacja OR	!=	nierówny

Operator trójargumentowy

Podobnie jak języki C/C++, Java zawiera operator trójargumentowy `?:`. Jego ogólna postać:

```
wyrażenie_logiczne ? wyrażenie1 : wyrażenie 2
```

gdzie `wyrażenie_logiczne` może być dowolnym wyrażeniem logicznym, a `wyrażenie1` i `wyrażenie2` – wyrażeniami dowolnych, takich samych typów różnych od `void`.

Jeśli `wyrażenie_logiczne` ma wartość `true`, operator `?:` zwraca wartość pierwszego wyrażenia w przeciwnym wypadku wartość drugiego wyrażenia.

Operator ten najczęściej stosuje się w instrukcjach przypisania:

```
1 int a = 42;
2 int b = 2;
3 int c = 99;
4 int d = 0;
5 int e = (b == 0) ? 0 : ( a / b );
6 int f = {d == 0} ? ( c / d );
```


Instrukcje sterujące umożliwiają zmianę sekwencyjnego wykonywania programu w zależności od aktualnych wartości określonych zmiennych. Obejmują one:

- rozgałęzienia,
- pętle,
- obsługę wyjątków,
- wywołania funkcji.

Instrukcje sterujące Javy działają niemal identycznie ze swoimi odpowiednikami z języków C/C++.

Rozgałęzienia – instrukcja if-else

Instrukcja **if-else** umożliwia obliczanie wartości wyrażenia logicznego i w zależności od jego wartości wykonanie jednej z dwóch instrukcji. Ogólna postać:

```
1 if (wyrażenie_logiczne)
2   instrukcja1
3   [ else
4     instrukcja2 ]
```

Cześć **else** jest opcjonalna. Instrukcja1 i Instrukcja2 może być instrukcją prostą lub instrukcją złożoną.

```
1 int x=5;
2 if ( x==5 )
3 {System.out.println("x jest równe 5");
4 }
```

```
1 int x=12;
2 if ( x==10 )
3 {
4 System.out.println("x jest równe 10");
5 }
6 else
7 {
8 System.out.println("x nie jest równe 10");
9 }
```

Instrukcja break

Język Java nie zawiera instrukcji goto. W Javie odpowiednikiem goto jest instrukcja **break**, która nakazuje kompilatorowi przejście do pierwszej instrukcji znajdującej się za instrukcją złożoną o podanej nazwie. Instrukcje definiuje się za pomocą **etykiet**. (Dla pętli i instrukcji, **switch** można używać bez etykiet)

```
1 class Break
2 {
3     public static void main(String args[])
4     {
5         boolean t = true;
6         a: {
7             b: {
8                 c: {
9                     System.out.println("Przed wykonaniem instrukcji break");
10                    if (t) break b;
11                    System.out.println("Ta instrukcja nie jest wykonywana");
12                }
13                System.out.println("Po wykonaniu instrukcji break");
14            }
15        }
16    }
```

Wynik działania powyższych instrukcji:

```
1 C:\> java Break
2 Przed wykonaniem instrukcji Break
3 Po wykonaniu instrukcji Break
```

Instrukcja switch

Instrukcja switch

- Instrukcja **switch** umożliwia przejście do jednej z podanych sekwencji instrukcji w zależności od wartości określonego wyrażenia.

Instrukcja switch

- Instrukcja **switch** umożliwia przejście do jednej z podanych sekwencji instrukcji w zależności od wartości określonego wyrażenia.
- Ogólna postać instrukcji **switch**:

```
1  switch (wyrażenie)
2  {
3      case wartość1 :
4          sekwencja instrukcji
5          [ break ; ]
6      case wartość2 :
7          sekwencja_instrukcji
8          [ break ; ]
9          .....
10     case wartośćN :
11         sekwencja instrukcji
12         [ break ; ]
13     [ default :
14         sekwencja instrukcji ]
15 }
```

Instrukcja switch

- Instrukcja **switch** umożliwia przejście do jednej z podanych sekwencji instrukcji w zależności od wartości określonego wyrażenia.
- Ogólna postać instrukcji **switch**:

```
1  switch (wyrażenie)
2  {
3      case wartość1 :
4          sekwencja_instrukcji
5          [ break ; ]
6      case wartość2 :
7          sekwencja_instrukcji
8          [ break ; ]
9          .....
10     case wartośćN :
11         sekwencja_instrukcji
12         [ break ; ]
13     [ default :
14         sekwencja_instrukcji ]
15 }
```

- Typ wyrażenia jest dowolny. Wartości podane w blokach **case** muszą być niepowtarzającymi się literałami typu zgodnego z typem wyrażenia. Instrukcja **break** przerywa blok **default** oraz sekwencje instrukcji

Instrukcja switch - przykład

```
1 class SwitchSeasons
2 {
3     public static void main(String args[])
4     {
5         int month = 4;
6         string season;
7         switch (month)
8         {
9             case 12:
10            case 1:
11            case 2:
12                season = "zima";
13                break;
14            case 3:
15            case 4:
16            case 5:
17                season = "wiosna";
18                break;
19            case 6:
20            case 7:
21            case 8:
22                season = "lato";
23                break;
24            case 9:
25            case 10:
26            case 11:
27                season = "jesień";
28                break;
29            default:
30                season = "niepoprawny miesiąc";
31        }
32        System.out.println("pora roku: " + season + ".");
33    }
```



Instrukcja return

Instrukcja **return** umożliwia natychmiastowe zakończenie wykonywania bieżącej metody i powrót do procesu, który ją wywołał (co to jest metoda i do czego służy będzie później).

```
1 class ReturnDemo
2 {
3     public static void main(String args[])
4     {
5         boolean t = true;
6         system.out.println("Przed wykonaniem instrukcji return");
7         if (t)
8             return;
9         System.out.println("ta instrukcja nie będzie wykonana");
10    }
11 }
12 }
```

W tym przykładzie instrukcja **return** powoduje zakończenie wykonywania metody **main()** i powrót do interpretera Javy.

Instrukcja warunkowa **if()** nie może być w tym przykładzie pominięta, druga instrukcja **println** byłaby niedostępna – błąd kompilacji.

Jest to jedna z podstawowych konstrukcji wykorzystywana we wszystkich językach programowania. Pozwalają one na cykliczne wykonywanie określonych czynności. Pętle składają się z czterech elementów:

- 1 **inicjalizacja** – sekwencja instrukcji określających warunki początkowe, wykonywana jest jednokrotnie. (opcjonalnie)
- 2 **część główna** – jedna lub więcej instrukcji, które mają być wykonywane cyklicznie. (obowiązkowo, może to być instrukcja pusta)
- 3 **część modyfikująca** – instrukcję wykonywane każdorazowo po zakończeniu wykonywania części głównej, wpływające na wartość warunku. (opcjonalnie)
- 4 **warunek** – wyrażenie logiczne, którego aktualna wartość określa czy należy zakończyć wykonywanie pętli. (obowiązkowo)

W javie istnieją trzy konstrukcje pętli:

- 1 **while**
- 2 **do-while**
- 3 **for**

Pętla while

W programach Javy najczęściej pętli tworzonych jest za pomocą instrukcji **while**. Powoduje ona wykonywanie określonej sekwencji instrukcji (części **główniej**), dopóki wyrażenie logiczne (**warunek**) ma wartość **true**.

Ogólna postać konstrukcji **while**:

```
1 [inicjalizacja]
2 while (wyrażenie_logiczne)
3 {
4     część_główna
5     [część_modyfikująca]
6 }
```

Pętla **while** jest pętlą niepoliczalną, czyli taką, o której nie możemy powiedzieć ile razy się wykonają.

Pętla while – przykład

Poniższy program wyświetla aktualną wartość licznika j.

```
1 class Whiledemo
2 {
3     public static void main(string arg[])
4     {
5         int j = 0;
6         while (j < 0)
7         {
8             System.out.println("j=" + j);
9             j++;
10        }
11    }
12 }
```

Pętla do-while

Jeżeli chcemy aby **część główna** pętli wykonała się co najmniej raz, bez względu na początkową wartość **warunku**. Wtedy należy sprawdzać warunek pętli na końcu a nie na początku. Stosujemy wówczas pętlę **do-while**.

Ogólna postać pętli **do-while**:

```
1 [inicjalizacja]
2 do
3 {
4     część_główna
5     [część_modyfikująca]
6 }
7 while (wyrażenie_logiczne)
```

Pętla **do-while** jest również pętlą niepoliczalną.

Pętla do-while – przykład

Poniższy program działa tak samo jak poprzedni, wyświetla aktualną wartość licznika j.

Ale zamiast pętli **while** użyto pętli **do-while**.

```
1 class DoWhiledemo
2 {
3     public static void main(string arg[])
4     {
5         int j = 0;
6         do
7         {
8             System.out.println("j_u=u" + j);
9         }
10        while (++j > 0)
11    }
12 }
```

Pętla for

Pętla **for** umożliwia tworzenie zwięzłych pętli, zapisywanych nawet w jednym wierszu. Jest to policzalna pętla, gdyż wykonana się określoną liczbę razy.

Pętla **for** umożliwia tworzenie zwięzłych pętli, zapisywanych nawet w jednym wierszu. Jest to policzalna pętla, gdyż wykona się określoną liczbę razy.

- Ogólna składnia pętli **for**:

```
1 for (inicjalizacja ; wyrażenie_logiczne ; część _modyfikacyjna)  
2   część_główna
```

Pętla **for** umożliwia tworzenie zwięzłych pętli, zapisywanych nawet w jednym wierszu. Jest to policzalna pętla, gdyż wykona się określoną liczbę razy.

- Ogólna składnia pętli **for**:

```
1 for (inicjalizacja ; wyrażenie_logiczne ; część _modyfikacyjna)
2   część_główna
```

- Każdą pętlę **for** można zastąpić odpowiadającą jej pętlą **while**:

```
1   inicjalizacja
2   while (wyrażenie_logiczne)
3   {
4       część_główna
5       część _modyfikacyjna )
6   }
```

Pętla **for** używa się najczęściej wtedy, gdy pewną instrukcję trzeba wykonać określoną liczbę razy (np. na wszystkich elementach tablicy). Przebiega to w następujący sposób:

- deklaruje się zmienną zwaną licznikiem. Licznik może być dowolnego typu poza **boolean**.
- licznik uzyskuje najpierw wartość minimalną.
- w każdym przebiegu pętli następuje zwiększenie licznika aż do osiągnięcia wartości maksymalnej.
- po osiągnięciu wartości maksymalnej przez licznik następuje przerwanie wykonywania pętli.

Pętla for – przykład

Pętla for – przykład

- Poniższy program wyświetla aktualną wartość licznika.

```
1 class ForDemo
2 {
3     public static void main(String args[])
4     {
5         for (int i = 1; i <= 10; i++)
6             System.out.println( "i_=_ " + i);
7     }
8 }
```

Pętla for – przykład

- Poniższy program wyświetla aktualną wartość licznika.

```
1 class ForDemo
2 {
3     public static void main(String args [])
4     {
5         for (int i = 1; i <= 10; i++)
6             System.out.println( "i_=_ " + i );
7     }
8 }
```

- Część inicjalizacyjna pętli **for** zawiera deklarację zmiennej *i* (licznik). Zakresem zmiennej zadeklarowanej w pętli **for** jest ta pętla. Po zakończeniu jej wykonywania zmienna *i* przestaje być dostępna.

instrukcja continue

instrukcja continue

- Czasami zdarza się, że w danej iteracji trzeba pominąć wszystkie pozostałe instrukcje pętli, ale nadal w niej pozostać. Umożliwia nam to instrukcja **continue**.
 - W pętli **while** i **do-while** powoduje przejście do instrukcji sprawdzającej warunek.
 - W pętli **for** przekazuje sterowanie do jej części modyfikacyjnej do trzeciego wyrażenia w nawiasach.

instrukcja continue

- Czasami zdarza się, że w danej iteracji trzeba pominąć wszystkie pozostałe instrukcje pętli, ale nadal w niej pozostać. Umożliwia nam to instrukcja **continue**.
 - W pętli **while** i **do-while** powoduje przejście do instrukcji sprawdzającej warunek.
 - W pętli **for** przekazuje sterowanie do jej części modyfikacyjnej do trzeciego wyrażenia w nawiasach.

```
1 class ContinueDemo
2 {
3     public static void main(String args[])
4     {
5         for (int i = 1; i <= 10; i++)
6             {
7                 if (i%2 == 0)
8                     continue;
9                 System.out.println("i_u=u" + i);
10            }
11    }
12 }
```

Program ten wypisze tylko nieparzyste wartości licznika.

break a continue w pętłach

Porównajmy użycie instrukcji **break** i **continue** w pętli **for**

```
1 for (int i = 1; i <= 10; i++)
2 {
3     if (i == 5)
4         continue;
5     System.out.println( "i=" + i);
6 }
```

Efekt : Nie zostanie wypisany tylko licznik o wartości 5.

```
1 for (int i = 1; i <= 10; i++)
2 {
3     if (i == 5)
4         break;
5     System.out.println( "i=" + i);
6 }
```

Efekt: wypisane zostaną tylko pierwsze cztery wartości licznika.

break + etykieta

Połączenie instrukcji `break` z etykietą pozwala przerwać działanie wszystkich pętli znajdujących się pomiędzy etykietą a `break`. Dotyczy to również pętli zagnieżdżonych.

```
1  int i , j ;
2  przerwa :
3  for ( j =0; j <4; j ++ )
4  {
5      for ( i =0; i <4; i ++ )
6      {
7          if ( j ==1 ) break przerwa ;
8          System . out . println ( " _ j =" + j + " _ i =" + i ) ;
9      }
10 }
```

W tym przypadku przerwiemy działanie obu pętli.

continue + etykieta

Użycie instrukcji continue z etykietą powoduje przerwanie działania pętli i wykonanie kolejnego przebiegu pętli znajdującej się pod etykietą.

```
1  int i , j ;
2  skok :
3  for ( j =1; j <=3; j ++ )
4  {
5      for ( i =1; i <=3; i ++ )
6      {
7          if ( j ==2 ) continue skok ;
8          System . out . println ( " _j=" + j + " _i=" + i ) ;
9      }
10 }
```