PROGRAMOWANIE W JAVIE

Andrzej Marciniak

Uniwersytet Zielonogórski
Instytut Sterowania i Systemów Informatycznych

Klasyczne konstrukcje programowania imperatywnego w Javie:

- Wewnętrzne typy danych
- Literały
- Wyrażenia
- Zmienne
- Instrukcje

Typy danych i zmienne w Javie

- Każdy obiekt przechowuje swój stan za pomocą zmiennych.
- Zmienna jest elementem danych nazwanym i wywoływanym za pomocą identyfikatora
- Ustalenie identyfikatora i powiązanie z nim zmiennej odbywa się w procesiedeklaracji. Wszystkie zmienne w Javie muszą być zadeklarowane, zanim zostaną użyte. Deklaracja zmiennej może znajdować się w dowolnym miejscu definiowanej metody (pojęcie metody jest równoważne identycznemu pojęciu w języku C++).
- W Javie mamy dwa typy zmiennych: prymitywne(wewnętrzne) i referencyjne (referencja - element danych którego wartością jest adres)
- Typy referencyjne: tablice, klasy i interfejsy (analogiczne do typów wskaźnikowych w innych językach, mimo że Java nie pozwala na jawne użycie adresów)

Wewnętrzne (prymitywne) typy danych:

Język Java udostępnia programiście szereg standardowych typów wartości, z których większość można spotkać także w innych językach programowania. Typy te nazywane są w oryginale 'intrinsic types'.

typ	rozmiar (bity)	przedział zmienności
boolean	8	true - false
byte	8	-128 - 127
char	16	Unicode 0-65536
short	16	-32 768 - 32 767
int	32	-2 147 483 648 - 2 147 483 647
long	64	-9,2 $\cdot 10^{18}$ - 9,2$\cdot 10^{18}$
float	32 IEEE 754	3,4 $\cdot 10^{-38}$ - 3,4 $\cdot 10^{38}$
double	64 IEEE 754	1,7 $\cdot 10^{-308}$ - 1,7 $\cdot 10^{308}$

Unicode

UNICODE jest standardem kodowania pozwalającym na zdefiniowanie 65536 znaków, bez potrzeby używania różnych stron kodowych dla alfabetów narodowych co występuje dla standardu ASCII. Znaki ASCII są reprezentowane przez liczby od 0 do 127.

Polskie litery

Litera	Kod Unicode	Litera	Kod Unicode
Ą	0104	Ó	00D3
ą	0105	ó	00F3
Ć	0106	Ś	015A
ć	0107	ś	015B
Ę	0118	Ź	0179
ę	0119	ź	017A
Ł	0141	Ż	017B
ł	0142	ż	017C

Przykład wykorzystania Unicode dla polskich znaków

Aby w programie Javy użyć znaków Unicode stosujemy następującą konwencję: \ukodznaku.

```
public void paint(Graphics g) {
    g.drawString("Dzi\u0119kuj\u0119",10,10);
}
```

Znaki Unicode są wyświetlane, jeśli system, w którym uruchamiane są programy stosujące znaki Unicode implementuje kodowanie znaków Unicode. W przypadku, gdy w zbiorze czcionek nie jest dostępny obraz graficzny reprezentujący dany znak, powinien być wyświetlony automatycznie znak podobny (np. gdy brak litery Ś wyświetlana jest litera S).

Deklaracje zmiennych

Deklaracja zmiennej składa się z nazwy typu zmiennej, za którą znajduje się nazwa zmiennej oraz, opcjonalnie, wyrażenie nadające wartość początkową zmiennej. Cały napis zakończony jest średnikiem. Deklarację tę możemy przedstawić schematycznie w następujący sposób:

$$\text{Nazwa\_Typu Nazwa\_Zmiennej } [= \text{Wartość\_Początkowa}]$$

Nazwa musi zaczynać się literą alfabetu, znakiem podkreślenia '_' lub znakiem dolara '$'. Dalej, identyfikator może zawierać cyfry lub litery alfabetu. W skład alfabetu wchodzą znaki: 'a' do 'z' 'A' do 'Z' oraz zbiór znaków Unicodu (czyli napisy 'Mąka' i 'Tëmço' są poprawnymi identyfikatorami w Javie). Ponadto zastosowano mechanizm nadawania zmiennym wewnętrznym domyślnych wartości początkowych, tj. boolean false, byte 0, char 'x0', short 0, int 0, long 0, float 0.0F, double 0.0D.

Proponowana konwencja identyfikatorów

Twórcy Javy proponują, aby wszystkim metodom i zmiennym publicznym przydzielać identyfikatory rozpoczynające się od małych liter i zawierające duże litery na początku każdego nowego słowa, np. nextItem, currentValue, getTimeOfDay .
Identyfikatory metod prywatnych oraz zmiennych prywatnych i lokalnych powinny zawierać wyłącznie małe litery oraz znaki podkreślenia oddzielające kolejne wyrazy, np. next_val, temp_val .
Zmienne finalne, czyli zmienne o stałych wartościach najlepiej jest wyróżniać dużymi literami np. TOK_BRACE, GREEN .
Powyższa konwencja jest stosowana w standardowych bibliotekach klas Javy.

Słowa kluczowe i zarezerwowane

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

* słowa kluczowe, które nie są aktualnie używane

** słowa kluczowe dodane w standardzie Java 2

Uwaga: true, false, null nie są słowami kluczowymi, ale są również zarezerwowane.

Przykłady deklaracji i wyświetlania zmiennych

```
public class MaxVariablesDemo {
    public static void main(String args[]) {
        // całkowite
        byte largestByte = Byte.MAX_VALUE;
        short largestShort = Short.MAX_VALUE;
        int largestInteger = Integer.MAX_VALUE;
        long largestLong = Long.MAX_VALUE;
        // rzeczywiste
        float largestFloat = Float.MAX_VALUE;
        double largestDouble = Double.MAX_VALUE;
        // wyświetlanie wartości maksymalnych
        System.out.println("The largest byte value is " + largestByte);
        System.out.println("The largest short value is " + largestShort);
        System.out.println("The largest integer value is " + largestInteger);
        System.out.println("The largest long value is " + largestLong);
        System.out.println("The largest float value is " + largestFloat);
        System.out.println("The largest double value is " + largestDouble);
    }
}
```

Zmienne finalne

Zmienne definiowane ze słowem kluczowym final mogą być inicjalizowane tylko raz, po czym ich wartość nie może się zmieniać. Słowo kluczowe final odnosić się może również do klas i metod: klasa finalna nie może mieć podklas, natomiast metoda finalna nie może być przykryta. Przykładowe definicje:

```
final int aFinalVar = 0;
// lub
final int blankfinal;
 . . .
blankfinal = 0;
```

Druga definicja pokazuje deklarację z odroczeniem inicjalizacji zmiennej (tzw. pusta zmienna finalna - ang. *blank final*). Każda następna próba przypisania wartości tej zmiennej spowoduje błąd podczas kompilacji.

Zmienne tablicowe

Tablica jest typem umożliwiającym grupowanie zmiennych tego samego typu i odwoływanie się do nich za pomocą wspólnej nazwy. Tablice można deklarować z podaniem rozmiaru lub bez, np.

```
int month[], week[7];
month = new int[12];
```

Tablice można inicjalizować automatycznie np.

```
int month[]={ 1, 2,3}
```

Tablice wielowymiarowe

W Javie podobnie jak w C/C++ nie ma tablic wielowymiarowych, a jedynie tablice tablic, np.

int macierz[ ] [ ]= new double [3] [3]

Tę samą deklarację można przedstawić również na inny sposób, np. int macierz[ ] [ ]= new double [3] [ ]

macierz[0]=new double [3]

macierz[1]=new double [3]

macierz[2]=new double [3]

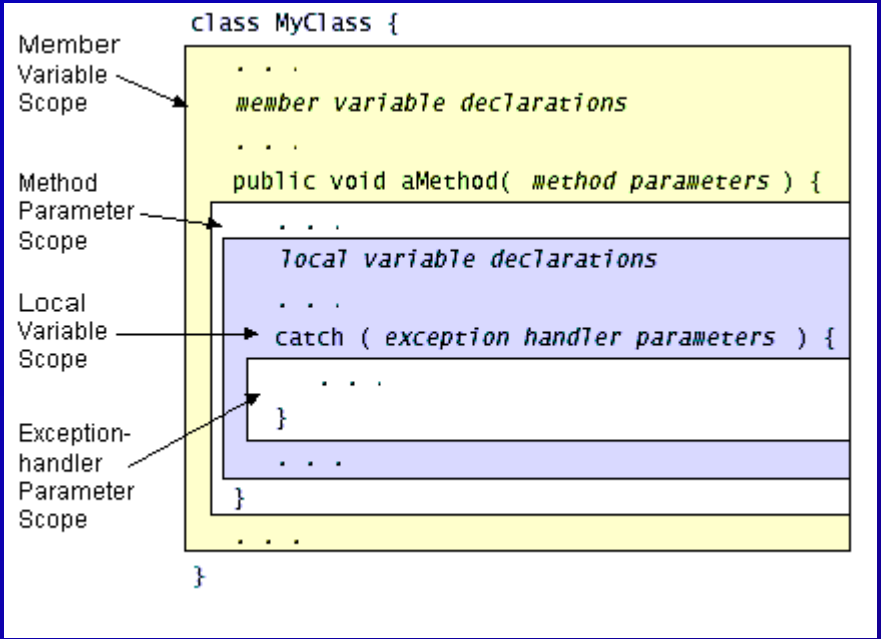
Zasięg zmiennych

Położenie deklaracji zmiennej w programie determinuje jej zasięg, umieszczając go w jednej z czterech kategorii:

- zmienna składowa (member variable),
- zmienna lokalna (local variable),
- parametr metody (method parameter),
- parametr procedury obsługi wyjątku (exception-handler parameter).

Instrukcje złożone, czyli sekwencje instrukcji prostych, rozpoczynają się i kończą nawiasami klamrowymi. Każda zmienna jest widoczna od miejsca jej deklaracji do końca zawierającej ją instrukcji złożonej. Instrukcje złożone mogą być zagnieżdżone, czyli w jednej instrukcji złożonej może występować inna, i każda może zawierać własne zmienne lokalne; niedozwolone jest jednak stosowanie w odniesieniu do nich takich samych identyfikatorów.

Zasięg zmiennych - ilustracja



Zasięg zmiennych - przykład błędu

Podana niżej definicja klasy jest nieprawidłowa, gdyż występują w niej deklaracje dwóch zmiennych o takiej samej nazwie. W języku C lub C++ obie zmienne miałyby różne zakresy i nie byłoby błędu.

```
class Zasięg {
    public static void main(String args[]) {
       int zmienna=1;
            {   //utworzenie nowego zakresu
                int zmienna=2;  //błąd czasu kompilacji
            }
    }
}
```

Literały

- literały **całkowite** (dziesiętne, ósemkowe i szesnastkowe), odpowiadające typom int i long (domyślnie int), np. 169, 0251 (ósemkowy), 0xA9, 0XA9, 0xa9, 2.34l, 2.59L (Literały o wartościach większych niż 2 147 483 647dziesiętnie automatycznie przyjmują postać liczby typu long)
- literały **rzeczywiste** (stałoprzecinkowe i zmiennoprzecinkowe), odpowiadające typom double i float (domyślnie double), np. 4.565, 50.0d, 30.5E-11, 4.5f, 0.2F, 30.5E-11f
- literały **znakowe i łańcuchowe**, związane z typem char i String, np. 'K', '\x20'(spacja), "Napis", '\141' (litera 'a' ósemkowo)
- literały **logiczne** reprezentujące dwie wartości logiczne typu boolean, tj. true, false , które odmiennie, niż w języku C, nie dają się konwertować niejawnie do wartości typu int.

Specjalne literały znakowe

Literały łańcuchowe są sekwencjami znaków ujętymi w cudzysłów. Literały tego typu w Javie nie są, jak to ma miejsce w CC++, tablicami znaków zakończonymi znakiem pustym.

Opis	Literał
Znak nowej linii	\n
Tabulacja pionowa	\t
Backspace	\b
Powrót karetki	\r
Znak nowej strony	\f
Apostrof	\'
Cudzysłów	\"
Backslash (lewy ukośnik)	\\

Rzutowanie typów

- Jeżeli zakres i wartość zmiennej docelowej są na tyle duże, że można w niej zapisać wartość źródłową bez utraty informacji, to kompilator dokonuje automatycznej konwersji typów co nazywa się poszerzaniem (np. $byte \rightarrow int$).
- W przypadku przeciwnym (zawężania), konieczne jest jawne rzutowanie typu (np. $int \rightarrow byte$).
- Gdy jest prawdopodobne że wynik operacji na danym typie (np. mnożenie liczb typu $byte$) przekroczy dopuszczalny zakres typu, kompilator Javy automatycznie przekształca go do typu wyniku. Typ każdego wyrażenia jest zawsze automatycznie przekształcany do najszerszego typu występujących w nim zmiennych i literałów.

Rzutowanie typów - przykład

```
class Rzutowanie {
 public static void main (String args[])
{   byte b=42;
    char c='a';
    short s=1024;
    int i=50000;
    float f=5.67f;
    double d=.1234;
    double rezultat=(f*b)+(i/c)-(d*s);
    }
}
```

1. $f * b \rightarrow float * byte \rightarrow float$
2. $i/c \rightarrow int/char \rightarrow int$
3. $d * s \rightarrow double * short \rightarrow double$
4. $(1) + (2) \rightarrow float + int \rightarrow float$
5. $(4) - (3) \rightarrow float - double \rightarrow double$

Rzutowanie typów - błędy

```
byte b=50;
b=b*2;
 ^ Incompatible type for =.
   Explicit cast needed to convert int to byte.
```

Błąd wynika z faktu, że liczba 2 jest traktowana jako wartość typu int, w związku z czym wyrażenie $b*2$ również. Próba przypisania wymaga zatem jawnego operatora konwersji, mimo że liczba 100 mieści się w dziedzinie typu $byte$.

Operatory

- unarne (posiadające 1 operand) np. "++"oraz pre- lub post-fix'ową notację (operator występuje przed lub za operandem)
- binarne np. "=" , które zawsze wykorzystują notację infix ,
- ternarny: "?:", który również jest zapisywany w notacji infix, czyli op1 ? op2 : op3.

Uwaga: W Javie nie ma możliwości przeciążania operatorów.

Priorytet operatorów

Priorytet	Operatory	Priorytet	Operatory
1	. [] ()	9	^
2	++ -- ! ~ *instanceof*	10	|
3	* / %	11	&&
4	+ -	12	||
5	<< >> >>>	13	? :
6	< > <= >=	14	= op=
7	== !=	15	,
8	&		

Operatory arytmetyczne

Operator	Użycie	Opis
+	op1 + op2	Dodaje op1 i op2
+	+op	Konwertuje op do int, jeżeli jest on typu byte, short, lub char
-	op1 - op2	odejmuje op2 od op1
-	-op	Zmienia znak op
*	op1 * op2	Mnoży op1 przez op2
/	op1 / op2	Dzieli op1 przez op2
%	op1 % op2	Oblicza resztę z dzielenia całkowitego op1 przez op2. W przeciwieństwie do CC++ można go stosować z typami zmiennoprzecinkowymi
++	op++	Inkrementuje op o 1; przypisując wartość op przed zwiększeniem
++	++op	Inkrementuje op o 1; przypisując wartość op po zwiększeniu
−−	op−−	Dekrementuje op o 1; przypisując wartość op przed zmniejszeniem
−−	−−op	Dekrementuje op o 1; przypisując wartość op po zmniejszeniu

Operatory relacyjne

Operator	Użycie	Opis
$>$	op1 $>$ op2	op1 jest większy niż op2
$>=$	op1 $>=$ op2	op1 jest większy lub równy niż op2
$<$	op1$<$ op2	op1 jest mniejszy niż op2
$<=$	op1 $<=$ op2	op1 jest mniejszy lub równy niż op2
$==$	op1 $==$ op2	op1 i op2 są równe
$!=$	op1 $!=$ op2	op1 i op2 nie są równe

Każdy operator relacyjny zwraca wartość logiczną (boolean).

Operatory logiczne

Operator	Użycie	Opis
&&	op1 && op2	Oba operandy są prawdziwe, warunkowo szacuje op2
||	op1 || op2	Jeden z operandów jest prawdziwy, warunkowo szacuje op2
!	!op	op jest fałszywy (false)
&	op1 & op2	Oba operandy są prawdziwe, szacuje oba
|	op1 | op2	Jeden z operandów jest prawdziwy, szacuje oba
^	op1 ^ op2	Prawda jeśli op1 i op2 są różne (XOR)

Operatory logiczne działają na wartościach typu boolean.

Operatory bitowe

Operator	Użycie	Opis
&	op1 & op2	bitowy iloczyn logiczny (AND)
|	op1 | op2	bitowa suma logiczna (OR)
^	op1 ^ op2	bitowa róznica symetryczna (XOR)
~	~op2	bitowa negacja (NOT)
>>	op1 >> op2	przesuwa w prawo bity op1 o wartość op2
<<	op1 << op2	przesuwa w lewo bity op1 o wartość op2
>>>	op1 >>> op2	przesuwa w prawo bity op1 o wartość op2 (bez znaku - zawsze uzupełnia zerami)

Operatory bitowe działają na wartościach typów całkowitych.

Operatory przypisania

Operator	Użycie	Analogiczny do
$+=$	op1 $+=$ op2	op1 $=$ op1 $+$ op2
$-=$	op1 $-=$ op2	op1 $=$ op1 - op2
$*=$	op1 $*=$ op2	op1 $=$ op1 $*$ op2
$/=$	op1 $/=$ op2	op1 $=$ op1 $/$ op2
$\%=$	op1 $\%$ =op2	op1 $=$ op1 $\%$op2
$\&=$	op1 $\&=$ op2	op1 $=$ op1 $\&$ op2
$\mid=$	op1 $\mid=$ op2	op1 $=$ op1 $\mid$ op2
$\hat{}=$	op1 $\hat{}=$ op2	op1 $=$ op1 $\hat{}$ op2
$<<=$	op1 $<<=$ op2	op1 $=$ op1 $<<$ op2
$>>=$	op1 $>>=$ op2	op1 $=$ op1 $>>$ op2
$>>>=$	op1 $>>>=$ op2	op1 $=$ op1 $>>>$ op2

Inne operatory

Operator	Opis
$?:$	Skrócone wyrażenie warunkowe if-else
[]	Używany do tworzenia i indeksowania tablic
.	Używany do odwoływania się do metod i zmiennych
(*parametry*)	przekazywanie listy parametrów
(*typ*)	Konwertuje wartość do typu podanego w ()
new	Tworzy obiekt lub tablicę
instanceof	Określa czy pierwszy operand jest obiektem typu podanego przez drugi operand

Wyrażenia, instrukcje i bloki

Wyrażenia są to ciągi zmiennych, operatorów i wywołań funkcji, służące do oszacowania pojedynczej wartości (np. wyrażenie warunkowe). Zbiór instrukcji języka Java, wzorowany na zbiorze instrukcji języka C, zawiera:

- instrukcje wyrażeniowe (przypisania, inkrementacji)
- instrukcje deklaracyjne,
- instrukcje sterujące,
- bloki - ciągi instrukcji w nawiasach klamrowych.

Instrukcje sterujące - pętle (instr. iteracyjne)

- instrukcję while, pozwalającą programować obliczenia tak długo, jak prawdziwy jest pewien warunek,
- instrukcje do ... while, podobną do poprzedniej, z modyfikacją która pozwala uruchomić blok instrukcji przynajmniej raz,
- instrukcję for, służącą do programowania obliczeń powtarzających się tak długo, jak długo pewna zmienna sterująca przyjmuje wartości z pewnego zbioru.

Instrukcja while- przykład

Efektem działania poniższego programu będzie wypisanie na ekranie konsoli "Kopiuj ten łańcuc"

```
public class WhileDemonstracja {
    public static void main(String[] args) {
        String źródło = "Kopiuj łańcuch dopóki " +
                        "napotkasz literę 'h'.";
        StringBuffer docelowy = new StringBuffer();
        int i = 0;
        char c =źródło.charAt(i);
        while (c != 'h') {
            docelowy.append(c);
            c = źródło.charAt(++i);
        }
        System.out.println(docelowy);
    }
}
```

Instrukcja do-while- przykład

Efektem działania poniższego programu będzie również wypisanie na ekranie konsoli "Kopiuj ten łańcuc"

```
public class doWhileDemonstracja {
    public static void main(String[] args) {
        String źródło = "Kopiuj łańcuch dopóki " +
                        "napotkasz literę 'h'.";
        StringBuffer docelowy = new StringBuffer();
        int i = 0;
        char c =źródło.charAt(i);
        do{
            docelowy.append(c);
            c = źródło.charAt(++i);
        }
        while (c != 'h');
        System.out.println(docelowy);
    }
}
```

Instrukcja for - przykład

Składnia instrukcji for

for (*inicjacja; warunkowanie; inkrementacja*) { *ciało pętli* }

```
public class ForDemo {
    public static void main(String[] args) {
        int[] arrayOfInts = { 32, 87, 3, 589, 12, 1076,
                              2000, 8, 622, 127 };

        for (int i = 0; i < arrayOfInts.length; i++) {
            System.out.print(arrayOfInts[i] + " ");
        }
        System.out.println();
    }
}
```

Instrukcje warunkowe - if

Składnia prostej instrukcji if

if ($wyrażenie\ boolean$) { $instrukcje$ }

którą można poszerzyć o to co ma się wydarzyć w przeciwnym przypadku:

if ($wyrażenie\ boolean$) { $instrukcje$} else { $instrukcje$ }

```
public class IfElseDemo {
    public static void main(String[] args) {
        int testscore = 76;  char grade;
        if (testscore >= 90) {
            grade = 'A';
        } else if (testscore >= 80) {
            grade = 'B';
        } else {
            grade = 'C';
        }
        System.out.println("Grade = " + grade);
    }
}
```

Instrukcje warunkowe - switch

Składnia instrukcji switch:

switch ($wyrażenie$) {
case $wyrażenie$: $instrukcje$ break;
...
default : $instrukcje$ break;}

```
public class SwitchDemo {
    public static void main(String[] args) {
        int ocena = 3;
        switch (ocena) {
            case 2:  System.out.println("D"); break;
            case 3:  System.out.println("C"); break;
            case 4:  System.out.println("B"); break;
            case 5:  System.out.println("A"); break;
            default: System.out.println("Zla wartosc"); break;
        }
    }
}
```

Instrukcja obsługi wyjątków

Składnia instrukcji obsługi wyjątków:

try *instrukcja_lub_blok_instrukcji*
catch (*typ_wyjątku_1*) *instrukcja_obsługi_wyjątku_1*
catch (*typ_wyjątku_2*) *instrukcja_obsługi_wyjątku_2*
...
catch (*typ_wyjątku_N*) *instrukcja_obsługi_wyjątku_N*
finally *domyślna_instrukcja_obsługi_wyjątku*

Więcej informacji na temat obsługi wyjątków znajdzie się na następnych wykładach.

Instrukcje sterujące rozgałęzieniami

Instrukcje break i continue pozwalają na wcześniejsze opuszczenie pętli.

- Jeśli break występuje bez etykiety (ang. label) to sterowanie programem przekazywane jest poza instrukcję pętli.
- Jeśli instrukcja break posiada etykietę to sterowanie przekazywane jest poza blok instrukcji oznaczonych etykietą i możliwe jest natychmiastowe opuszczenie wielu zagnieżdżonych instrukcji pętli.
- Instrukcja continue nie powoduje opuszczenia pętli ale natychmiastowe przejście do następnego kroku iteracji.
- Jeśli użyjemy instrukcji continue z etykietą sterowanie przekazywane jest do nawiasu otwierającego blok instrukcji oznaczonych etykietą.

Instrukcja return umożliwia natychmiastowe zakończenie wykonywania bieżącej metody.

Instrukcje sterujące rozgałęzieniami - przykład

```
int i=0; int k=10;
NaszBlok:
    {  while(i++<10)
        {
            if (--k==0)
                break NaszBlok;
            System.out.println(i/k);
            // wyświetlenie na ekranie wyniku dzielenia
        }
        //Tu zostałoby przekazane sterowanie gdyby instrukcja
        //break nie miała etykiety NaszBlok.
    } //koniec bloku instrukcji NaszBlok
    //Tu zostaje przekazane sterowanie programem po wykonaniu
    //instrukcji break NaszBlok
```