Java. Ćwiczenia praktyczne. Wydanie II

Autor: Marcin Lis
ISBN: 83-246-0327-1
Format: B5, stron: 192

Rozpocznij przygodę z programowaniem w Javie

- Poznaj podstawowe elementy języka Java
- Opanuj zasady programowania obiektowego
- Napisz własne aplety i aplikacje
- Wykorzystaj komponenty do tworzenia interfejsów użytkownika

Java w ciągu kilku ostatnich lat przebyła drogę od niemal nieznanej technologii do jednego z najpopularniejszych języków programowania na świecie. Dziś jej głównym atutem nie są aplety, które w założeniu twórców miały umilać czas osobom odwiedzającym witryny WWW, lecz rozbudowane aplikacje przetwarzające setki danych. Java to uznana platforma programistyczna stosowana zarówno przez największe firmy z branży informatycznej, jak i przez programistów amatorów do realizacji przeróżnych zadań z wykorzystaniem technik obiektowych.

„Java. Ćwiczenia praktyczne. Wydanie II" to zbiór krótkich ćwiczeń, dzięki którym poznasz podstawy programowania w tym języku. Przeczytasz o głównych elementach Javy i technikach obiektowych. Dowiesz się, jak definiować zmienne, przetwarzać dane tekstowe, tworzyć proste aplety i bardziej złożone aplikacje. Nauczysz się korzystać z komponentów, budując interfejsy użytkownika swoich aplikacji i zaimplementujesz operacje wejścia i wyjścia na plikach. Zdobędziesz solidne podstawy do dalszej nauki Javy.

- Instalacja Java Development Kit w Windows i Linuksie
- Deklarowanie zmiennych
- Operatory i ich priorytety
- Instrukcje warunkowe i pętle
- Obiekty i klasy
- Wyjątki i obsługa błędów
- Tworzenie apletów
- Grafika i dźwięk w Javie
- Tworzenie interfejsów użytkownika za pomocą komponentów
- Operacje na plikach

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Zmienne, operatory i instrukcje

Zmienne

Zmienna jest to miejsce, w którym możemy przechowywać jakieś dane, np. liczby czy ciągi znaków. Każda zmienna musi mieć swoją nazwę, która ją jednoznacznie identyfikuje, a także typ, który informuje o tym, jakiego rodzaju dane można w niej przechowywać. Np. zmienna typu `int` przechowuje liczby całkowite, a zmienna typu `float` liczby zmiennoprzecinkowe. Typy w Javie dzielą się na dwa rodzaje: *typy podstawowe* (ang. *primitive types*) oraz *typy odnośnikowe* (ang. *reference types*).

Typy podstawowe

Typy podstawowe dzielą się na:

- typy całkowitoliczbowe (z ang. *integral types*),
- typy zmiennopozycyjne (rzeczywiste, z ang. *floating-point types*),
- typ `boolean`,
- typ `char`.

Typy całkowitoliczbowe

Rodzina typów całkowitoliczbowych składa się z czterech typów:

- `byte`,
- `short`,
- `int`,
- `long`.

W przeciwieństwie do C++ dokładnie określono sposób reprezentacji tych danych. Niezależnie więc od tego, na jakim systemie pracujemy (16-, 32- czy 64-bitowym), dokładnie wiadomo, na ilu bitach zapisana jest zmienna danego typu. Wiadomo też dokładnie, z jakiego zakresu wartości może ona przyjmować, nie ma więc dowolności, która w przypadku języka C mogła prowadzić do sporych trudności przy przenoszeniu programów pomiędzy różnymi platformami. W tabeli 2.1 zaprezentowano zakresy poszczególnych typów danych oraz liczbę bitów niezbędną do zapisania zmiennych danego typu.

Tabela 2.1. *Zakresy typów arytmetycznych w Javie*

Typ	Liczba bitów	Liczba bajtów	Zakres
`byte`	8	1	od –128 do 127
`short`	16	2	od –32 768 do 32 767
`int`	32	4	od –2 147 483 648 do 2 147 483 647
`long`	64	8	od –9 223 372 036 854 775 808 do 9 223 372 036 854 775 807

Typy zmiennoprzecinkowe

Typy zmiennoprzecinkowe występują tylko w dwóch odmianach:

- `float` (pojedynczej precyzji),
- `double` (podwójnej precyzji).

Zakres oraz liczbę bitów i bajtów potrzebnych do zapisu tych zmiennych prezentuje tabela 2.2.

Tabela 2.2. *Zakresy dla typów zmiennoprzecinkowych w Javie*

Typ	Liczba bitów	Liczba bajtów	Zakres
`float`	32	4	od –3,4e38 do 3,4e38
`double`	64	8	od –1,8e308 do 1,8e308

Format danych `float` i `double` jest zgodny ze specyfikacją standardu ANSI/IEEE 754. Zapis 3,4e48 oznacza $3,4 * 10^{38}$.

Typ boolean

Jest to typ logiczny. Może on reprezentować jedynie dwie wartości: `true` (prawda) i `false` (fałsz). Może być wykorzystywany przy sprawdzaniu różnych warunków w instrukcjach `if`, a także w pętlach i innych konstrukcjach programistycznych, które zostaną przedstawione w dalszej części rozdziału.

Typ char

Typ `char` służy do reprezentacji znaków (liter, znaków przestankowych, ogólnie wszelkich znaków alfanumerycznych), przy czym w Javie jest on 16-bitowy i zawiera znaki *Unicode*. Ponieważ znaki reprezentowane są tak naprawdę jako 16-bitowe kody liczbowe, typ ten zalicza się czasem do typów arytmetycznych.

Deklarowanie zmiennych typów podstawowych

Aby móc użyć jakiejś zmiennej w programie, najpierw trzeba ją zadeklarować, tzn. podać jej typ oraz nazwę. Ogólna deklaracja wygląda następująco:

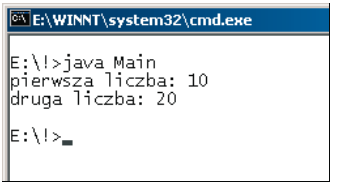
```
typ_zmiennej nazwa_zmiennej;
```

Po takiej deklaracji zmienna jest już gotowa do użycia, tzn. możemy jej przypisywać różne wartości bądź też wykonywać na niej różne operacje, np. dodawanie. Przypisanie wartości zmiennej odbywa się przy użyciu znaku (operatora) =.

ĆWICZENIE

2.1. Deklarowanie zmiennych

Zadeklaruj dwie zmienne całkowite i przypisz im dowolne wartości. Wyniki wyświetl na ekranie (rysunek 2.1).



Rysunek 2.1. *Wynik działania programu z ćwiczenia 2.1*

```
public
class Main
{
  public static void main (String args[])
  {
    int pierwszaLiczba;
    int drugaLiczba;
    pierwszaLiczba = 10;
    drugaLiczba = 20;
    System.out.println ("pierwsza liczba: " + pierwszaLiczba);
    System.out.println ("druga liczba: " + drugaLiczba);
  }
}
```

Instrukcja `System.out.println` pozwala wyprowadzić ciąg znaków na ekran. Wartość zmiennej można również przypisać już w trakcie deklaracji, pisząc:

```
typ_zmiennej nazwa_zmiennej = wartość;
```

Można również zadeklarować wiele zmiennych danego typu, oddzielając ich nazwy przecinkami. Część z nich może być też od razu zainicjowana:

```
typ_zmiennej nazwa1, nazwa2, nazwa3;
typ_zmiennej nazwa1 = wartość1, nazwa2, nazwa3 = wartość2;
```

Zmienne w Javie, podobnie jak w C czy C++, ale inaczej niż w Pascalu, można deklarować wedle potrzeb wewnątrz funkcji czy metody.

ĆWICZENIE

2.2. Jednoczesna deklaracja i inicjacja zmiennych

Zadeklaruj i jednocześnie zainicjalizuj dwie zmienne typu całkowitego. Wynik wyświetl na ekranie.

```
public
class Main
{
  public static void main (String args[])
  {
    int pierwszaLiczba = 10;
    int drugaLiczba = 20;
    System.out.println ("pierwsza liczba: " + pierwszaLiczba);
    System.out.println ("druga liczba: " + drugaLiczba);
  }
}
```

ĆWICZENIE

2.3. Deklarowanie zmiennych w jednym wierszu

Zadeklaruj kilka zmiennych typu całkowitego w jednym wierszu. Kilka z nich zainicjuj.

```
public
class Main
{
  public static void main (String args[])
  {
    int pierwszaLiczba = 10, drugaLiczba = 20, i, j, k;
    System.out.println ("pierwsza liczba: " + pierwszaLiczba);
    System.out.println ("druga liczba: " + drugaLiczba);
  }
}
```

Przy nazywaniu zmiennych obowiązują pewne zasady. Otóż nazwa może się składać z wielkich i małych liter oraz cyfr, ale nie może się zaczynać od cyfry. Choć nie jest to zabronione, raczej unika się stosowania polskich znaków diakrytycznych. Nazwa zmiennej powinna także odzwierciedlać funkcję pełnioną w programie. Jeżeli na przykład określa ona liczbę punktów w jakimś zbiorze, to najlepiej nazwać ją `liczbaPunktow` lub nawet `liczbaPunktowWZbiorze`. Mimo że tak długa nazwa może wydawać się dziwna, jednak bardzo poprawia czytelność programu oraz ułatwia jego analizę. Naprawdę warto ten sposób stosować. Przyjmuje się też, co również jest bardzo wygodne, że nazwę zmiennej rozpoczynamy małą literą, a poszczególne człony

tej nazwy (wyrazy, które się na nią składają) rozpoczynamy wielką literą — dokładnie tak jak w powyższych przykładach.

Typy odnośnikowe

Typy odnośnikowe (ang. *reference types*) możemy podzielić na dwa umowne rodzaje:

- typy klasowe (ang. *class types*)[1],
- typy tablicowe (ang. *array types*).

Zacznijmy od typów tablicowych. Tablice są to wektory elementów danego typu i służą do uporządkowanego przechowywania wartości tego typu. Mogą być jedno- bądź wielowymiarowe. Dostęp do danego elementu tablicy jest realizowany poprzez podanie jego indeksu, czyli miejsca w tablicy, w którym się on znajduje. Dla tablicy jednowymiarowej będzie to po prostu kolejny numer elementu, dla tablicy dwuwymiarowej trzeba już podać numer wiersza i kolumny itd. Jeśli chcemy zatem przechować w programie 10 liczb całkowitych, najwygodniej będzie użyć w tym celu 10-elementowej tablicy typu `int`.

Typy klasowe pozwalają na tworzenie klas i deklarowanie zmiennych obiektowych. Zajmiemy się nimi w rozdziale 3.

Deklarowanie zmiennych typów odnośnikowych

Zmienne typów odnośnikowych deklarujemy podobnie jak w przypadku zmiennych typów podstawowych, tzn. pisząc:

```
typ_zmiennej nazwa_zmiennej;
```

lub:

```
typ_zmiennej nazwa_zmiennej_1, nazwa_zmiennej_2, nazwa_zmiennej_3;
```

Stosując taki zapis, inaczej niż w przypadku typów prostych, zadeklarowaliśmy jednak jedynie tzw. *odniesienie* (ang. *reference*) do obiektu, a nie sam byt, jakim jest obiekt! Takiemu odniesieniu domyślnie

[1] Typy klasowe moglibyśmy podzielić z kolei na obiektowe i interfejsowe; są to jednak rozważania, którymi nie będziemy się w niniejszej publikacji zajmować.

przypisana jest wartość pusta (`null`), czyli praktycznie nie możemy wykonywać na nim żadnej operacji. Dopiero po utworzeniu odpowiedniego obiektu w pamięci możemy powiązać go z tak zadeklarowaną zmienną. Jeśli zatem napiszemy np.:

```
int a;
```

będziemy mieli gotową do użycia zmienną typu całkowitego. Możemy jej przypisać np. wartość `10`. Żeby jednak móc skorzystać z tablicy, musimy zadeklarować zmienną odnośnikową typu tablicowego, utworzyć obiekt tablicy i powiązać go ze zmienną. Dopiero wtedy będziemy mogli swobodnie odwoływać się do kolejnych elementów. Pisząc zatem:

```
int tablica[];
```

zadeklarujemy odniesienie do tablicy, która będzie mogła zawierać elementy typu `int`, czyli 32-bitowe liczby całkowite. Samej tablicy jednak jeszcze wcale nie ma. Przekonamy się o tym, wykonując kolejne ćwiczenia.

ĆWICZENIE

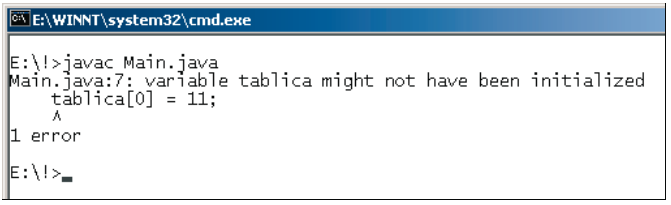
2.4. Deklarowanie tablicy

Zadeklaruj tablicę elementów typu całkowitego. Przypisz zerowemu elementowi tablicy dowolną wartość. Spróbuj skompilować i uruchomić program.

```
public
class Main
{
  public static void main (String args[])
  {
    int tablica[];
    tablica[0] = 11;
    System.out.println ("Zerowy element tablicy to: " + tablica[0]);
  }
}
```

Już przy próbie kompilacji kompilator wypisze na ekranie tekst: `Variable tablica might not have been initialized`, informujący nas, że chcemy odwołać się do zmiennej, która prawdopodobnie nie została zainicjalizowana (rysunek 2.2). Widzimy też wyraźnie, że w razie wystąpienia błędu na etapie kompilacji otrzymujemy kilka ważnych i pomocnych informacji. Przede wszystkim jest to nazwa pliku, w którym wystąpił błąd (jest to ważne, gdyż program może składać się z bardzo wielu klas, a każda z nich jest zazwyczaj definiowana w oddzielnym pliku), numer wiersza w tym pliku oraz konkretne

miejsce wystąpienia błędu. Na samym końcu kompilator podaje też całkowitą liczbę błędów.



Rysunek 2.2. *Błąd kompilacji. Nie zainicjowaliśmy zmiennej tablica*

Skoro jednak wystąpił błąd, należy go natychmiast naprawić.

ĆWICZENIE

2.5. Deklaracja i utworzenie tablicy

Zadeklaruj i utwórz tablicę elementów typu całkowitego. Przypisz zerowemu elementowi tablicy dowolną wartość. Spróbuj wyświetlić zawartość tego elementu na ekranie.

```
public
class Main
{
  public static void main (String args[])
  {
    int tablica[] = new int[10];
    tablica[0] = 11;
    System.out.println ("Zerowy element tablicy to: " + tablica[0]);
  }
}
```

Wyrażenie `new tablica[10]` oznacza utworzenie nowej, jednowymiarowej tablicy liczb typu `int` o rozmiarze 10 elementów. Ta nowa tablica została przypisana zmiennej odnośnikowej o nazwie `tablica`. Po takim przypisaniu możemy odwoływać się do kolejnych elementów tej tablicy, pisząc:

```
tablica[index]
```

Warto przy tym zauważyć, że elementy tablicy numerowane są od zera, a nie od 1. Oznacza to, że pierwszy element tablicy 10-elementowej ma indeks 0, a ostatni 9 (a nie 10!).

Co się jednak stanie, jeśli — nieprzyzwyczajeni do takiego sposobu indeksowania — odwołamy się do indeksu o numerze `10`?

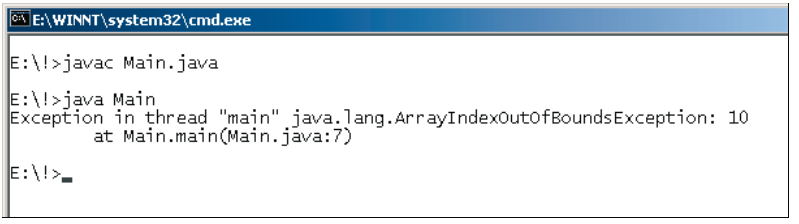
ĆWICZENIE

2.6. Odwołanie do nieistniejącego indeksu

Zadeklaruj i zainicjalizuj tablicę dziesięcioelementową. Spróbuj przypisać elementowi o indeksie 10 dowolną liczbę całkowitą.

```
public
class Main
{
  public static void main (String args[])
  {
    int tablica[] = new int[10];
    tablica[10] = 11;
    System.out.println ("Dziesiąty element tablicy to: "
    ➥+ tablica[10]);
  }
}
```

Efekt działania kodu jest widoczny na rysunku 2.3. Wbrew pozorom nie stało się jednak nic strasznego. Wystąpił błąd, został on jednak obsłużony przez maszynę wirtualną Javy. Konkretnie został wygenerowany tzw. wyjątek i program standardowo zakończył działanie. Taki wyjątek możemy jednak przechwycić i tym samym zapobiec niekontrolowanemu zakończeniu aplikacji. Jest to jednak odrębny, aczkolwiek bardzo ważny temat; zajmiemy się nim więc nieco później. Godne uwagi jest to, że próba odwołania się do nieistniejącego elementu została wykryta i to odwołanie tak naprawdę nie wystąpiło! Program nie naruszył więc niezarezerwowanego dla niego obszaru pamięci.



Rysunek 2.3. *Próba odwołania się do nieistniejącego elementu tablicy*

Operatory

Poznaliśmy już zmienne, musimy jednak wiedzieć, jakie operacje możemy na nich wykonywać. Operacje wykonujemy za pomocą różnych operatorów, np. odejmowania, dodawania, przypisania itd. Operatory te możemy podzielić na następujące grupy[2]:

- arytmetyczne,
- bitowe,
- logiczne,
- przypisania,
- porównania.

Operatory arytmetyczne

Wśród tych operatorów znajdziemy standardowo działające:

- `+` — dodawanie,
- `-` — odejmowanie,
- `*` — mnożenie,
- `/` — dzielenie.

ĆWICZENIE

2.7. Operacje arytmetyczne na zmiennych

Zadeklaruj dwie zmienne typu całkowitego. Wykonaj na nich kilka operacji arytmetycznych. Wyniki wyświetl na ekranie.

```
public
class Main
{
  public static void main(String args[])
  {
    int a, b, c;
    a = 10;
    b = 25;
    c = b - a;
    System.out.println("a = " + a);
```

[2] Można wydzielić również inne grupy, co wykracza jednak poza ramy tematyczne niniejszej publikacji.

```
    System.out.println("b = " + b);
    System.out.println("b - a = " + c);
    c = a * b;
    System.out.println("a * b = " + c);
  }
}
```

Do operatorów arytmetycznych należy również znak %, przy czym nie oznacza on obliczania procentów, ale dzielenie modulo (resztę z dzielenia). Np. wynik działania 12 % 5 wynosi 2, piątka mieści się bowiem w dwunastu 2 razy, pozostawiając resztę 2 (5 * 2 = 10, 10 + 2 = 12).

ĆWICZENIE

2.8. Dzielenie modulo

Zadeklaruj kilka zmiennych. Wykonaj na nich operacje dzielenia modulo. Wyniki wyświetl na ekranie.

```
public
class Main
{
  public static void main(String args[])
  {
    int a, b, c;
    a = 10;
    b = 25;
    c = b % a;
    System.out.println("b % a = " + c);
    System.out.println("a % 3 = " + a % 3);
    c = a * b;
    System.out.println("(a * b) % 120 = " + c % 120);
  }
}
```

Kolejne operatory typu arytmetycznego to operator inkrementacji i dekrementacji. Operator inkrementacji (czyli zwiększenia), którego symbolem jest ++, powoduje przyrost wartości zmiennej o jeden. Może występować w formie przyrostkowej bądź przedrostkowej. Oznacza to, że jeśli mamy zmienną, która nazywa się np. x, forma przedrostkowa będzie wyglądać: ++x, natomiast przyrostkowa: x++.

Oba te wyrażenia zwiększą wartość zmiennej x o jeden, jednak nie są one równoważne. Otóż operacja x++ zwiększa wartość zmiennej po jej wykorzystaniu, natomiast ++x przed jej wykorzystaniem. Czasem takie rozróżnienie jest bardzo pomocne przy pisaniu programu.

ĆWICZENIE

2.9. Operator inkrementacji

Przeanalizuj poniższy kod. Nie uruchamiaj programu, ale zastanów się, jaki będzie wyświetlony ciąg liczb. Następnie, po uruchomieniu kodu, sprawdź swoje przypuszczenia.

```
public
class Main
{
  public static void main (String args[])
  {
    /*1*/ int x = 1, y;
    /*2*/ System.out.println (++x);
    /*3*/ System.out.println (x++);
    /*4*/ System.out.println (x);
    /*5*/ y = x++;
    /*6*/ System.out.println (y);
    /*7*/ y = ++x;
    /*8*/ System.out.println (++y);
  }
}
```

Dla ułatwienia poszczególne wiersze w programie zostały oznaczone kolejnymi liczbami. Wynikiem działania tego programu będzie ciąg liczb: 2, 2, 3, 3, 6. Dlaczego? Na początku zmienna `x` przyjmuje wartość 1. W 2. wierszu występuje operator `++x`, zatem najpierw jest ona zwiększana o jeden (x = 2), a dopiero potem wyświetlana na ekranie. W wierszu o numerze 3 jest odwrotnie. Najpierw wartość zmiennej `x` jest wyświetlana (x = 2), a dopiero potem zwiększana o 1 (x = 3). W wierszu 4. po prostu wyświetlamy wartość `x` (x = 3). W wierszu 5. najpierw zmiennej `y` jest przypisywana dotychczasowa wartość `x` (x = 3, y = 3), a następnie wartość `x` jest zwiększana o jeden (x = 4). W wierszu 6. wyświetlamy wartość `y` (y = 3). W wierszu 7. najpierw zwiększamy wartość `x` o jeden (x = 5), a następnie przypisujemy tę wartość zmiennej `y`. W wierszu ostatnim, ósmym, zwiększamy `y` o jeden (y = 6) i wyświetlamy na ekranie.

Operator dekrementacji (`--`) działa analogicznie, z tym że zamiast zwiększać wartości zmiennych — zmniejsza je, oczywiście zawsze o jeden.

ĆWICZENIE

2.10. Operator dekrementacji

Zmień kod z ćwiczenia 2.9 tak, aby operator `++` został zastąpiony operatorem `--`. Następnie przeanalizuj jego działanie i sprawdź, czy

otrzymany wynik jest taki sam, jak otrzymany na ekranie po uruchomieniu kodu.

```
public
class Main
{
  public static void main (String args[])
  {
    /*1*/ int x = 1, y;
    /*2*/ System.out.println (--x);
    /*3*/ System.out.println (x--);
    /*4*/ System.out.println (x);
    /*5*/ y = x--;
    /*6*/ System.out.println (y);
    /*7*/ y = --x;
    /*8*/ System.out.println (--y);
  }
}
```

Działania operatorów arytmetycznych na liczbach całkowitych nie trzeba chyba wyjaśniać, z dwoma może wyjątkami. Otóż co się stanie, jeżeli wynik dzielenia dwóch liczb całkowitych nie będzie liczbą całkowitą? Odpowiedź na szczęście jest prosta, wynik zostanie zaokrąglony w dół. Zatem wynikiem działania 7/2 w arytmetyce liczb całkowitych będzie 3 („prawdziwym" wynikiem jest oczywiście 3,5, która to wartość zostaje zaokrąglona w dół do najbliższej liczby całkowitej, czyli trzech).

ĆWICZENIE

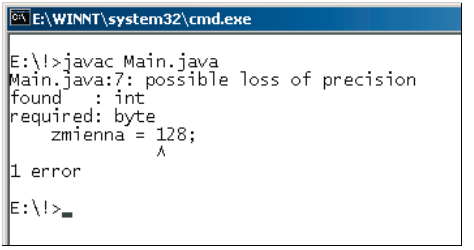
2.11. Dzielenie liczb całkowitych

Wykonaj dzielenie zmiennych typu całkowitego. Sprawdź rezultaty w sytuacji, gdy rzeczywisty wynik jest ułamkiem.

```
public
class Main
{
  public static void main(String args[])
  {
    int a, b, c;
    a = 8;
    b = 3;
    c = 2;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("a / b  = " + a / b);
    System.out.println("a / c  = " + a / c);
    System.out.println("b / c  = " + b / c);
  }
}
```

Drugim problemem jest to, co się stanie, jeżeli przekroczymy zakres jakiejś zmiennej. Pamiętamy np., że zmienna typu `byte` jest zapisywana na 8 bitach i może przyjmować wartości od –128 do 127 (patrz tabela 2.1). Spróbujmy zatem przypisać zmiennej tego typu wartość `128`. Szybko przekonamy się, że kompilator do tego nie dopuści (rysunek 2.4).

Rysunek 2.4. *Próba przekroczenia dopuszczalnej wartości zmiennej*



ĆWICZENIE

2.12. Przekroczenie zakresu w trakcie kompilacji

Zadeklaruj zmienną typu `byte`. Przypisz jej wartość `128`. Spróbuj dokonać kompilacji otrzymanego kodu.

```
public
class Main
{
  public static void main (String args[])
  {
    byte zmienna;
    zmienna = 128;
    System.out.println(zmienna);
  }
}
```

Niestety, kompilator nie zawsze będzie w stanie wykryć tego typu błąd. Może się bowiem zdarzyć, że zakres przekroczymy w trakcie wykonywania programu. Co wtedy?

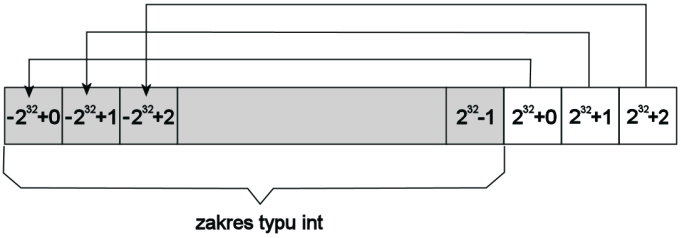
ĆWICZENIE

2.13. Przekroczenie zakresu w trakcie działania kodu

Zadeklaruj zmienne typu `long`. Wykonaj operacje arytmetyczne przekraczające dopuszczalną wartość takiej zmiennej. Wynik wyświetl na ekranie.

```
public
class Main
{
  public static void main (String args[])
  {
    long a, b = (long) Math.pow(2, 63) + 1;
    a = b + b;
    System.out.println ("a = " + a);
  }
}
```

Operacja `(long) Math.pow(2, 63)` oznacza podniesienie liczby 2 do potęgi 63., a następnie skonwertowanie wyniku (który jest liczbą typu `double`) do typu `long`. Zmiennej `a` jest przypisywany wynik działania `b + b` i okazuje się, że jest to 0. Dlaczego? Otóż jeżeli jakaś wartość przekracza dopuszczalny zakres swojego typu, jest „zawijana" do początku tego zakresu. Obrazowo ilustruje to rysunek 2.5.



Rysunek 2.5. *Przekroczenie dopuszczalnego zakresu dla typu int*

Operatory bitowe

Operacje te, jak sama nazwa wskazuje, dokonywane są na bitach. Przypomnijmy zatem podstawowe wiadomości o systemach liczbowych. W systemie dziesiętnym, z którego korzystamy na co dzień, wykorzystywanych jest dziesięć cyfr — od 0 do 9. W systemie dwójkowym będą zatem wykorzystywane jedynie dwie cyfry — 0 i 1. Kolejne liczby budowane są z tych dwóch cyfr, dokładnie tak samo jak w systemie dziesiętnym; przedstawia to tabela 2.3. Widać wyraźnie, że np. 4 dziesiętnie to 100 dwójkowo, a 10 dziesiętnie to 1010 dwójkowo.

Tabela 2.3. *Reprezentacja liczb w systemie dwójkowym i dziesiętnym*

System dwójkowy	System dziesiętny
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Na tak zdefiniowanych liczbach możemy dokonywać znanych ze szkoły operacji bitowych *AND* (iloczyn bitowy), *OR* (suma bitowa) oraz *XOR* (bitowa alternatywa wykluczająca). Symbolem operatora *AND* jest znak `&` (ampersand), operatora *OR* znak `|` (pionowa kreska), natomiast operatora *XOR* znak `^` (strzałka w górę). Oprócz tego można również wykonywać operacje przesunięć bitów. Zestawienie występujących w Javie operatorów bitowych zostało przedstawione w tabeli 2.4.

Operatory logiczne

Argumentami operacji takiego typu muszą być wyrażenia posiadające wartość logiczną, czyli `true` lub `false` (prawda i fałsz). Przykładowo, wyrażenie `10 < 20` jest niewątpliwie prawdziwe (10 jest mniejsze od 20),

Tabela 2.4. *Operatory bitowe w Javie*

Operator	Symbol
AND	`&`
OR	`|`
NOT	`~`
XOR	`^`
Przesunięcie bitowe w prawo	`>>`
Przesunięcie bitowe w lewo	`<<`
Przesunięcie bitowe w prawo z wypełnieniem zerami	`>>>`

zatem jego wartość logiczna jest równa `true`. W grupie tej wyróżniamy trzy operatory:

- logiczne *AND* (`&&`),
- logiczne *OR* (`||`),
- logiczna negacja (`!`).

Warto zauważyć, że w części przypadków stosowania operacji logicznych, aby otrzymać wynik, wystarczy obliczyć tylko pierwszy argument. Wynika to, oczywiście, z właściwości operatorów. Jeśli bowiem wynikiem obliczenia pierwszego argumentu jest wartość `true`, a wykonujemy operację *OR*, to niezależnie od stanu drugiego argumentu wartością całego wyrażenia będzie `true`. Podobnie przy stosowaniu operatora *AND* — jeżeli wartością pierwszego argumentu będzie `false`, to i wartością całego wyrażenia będzie `false`.

Operatory przypisania

Operacje przypisania są dwuargumentowe i powodują przypisanie wartości argumentu znajdującego się z prawej strony do argumentu znajdującego się z lewej strony. Najprostszym operatorem tego typu jest oczywiście klasyczny znak równości. Zapis `liczba = 5` oznacza, że zmiennej `liczba` chcemy przypisać wartość `5`. Oprócz tego mamy jeszcze do dyspozycji operatory łączące klasyczne przypisanie z innym operatorem arytmetycznym bądź bitowym. Zostały one zebrane w tabeli 2.5.

Tabela 2.5. *Operatory przypisania i ich znaczenie w Javie*

Argument 1	Operator	Argument 2	Znaczenie
x	=	y	x = y
x	+=	y	x = x + y
x	-=	y	x = x – y
x	*=	y	x = x * y
x	/=	y	x = x / y
x	%=	y	x = x % y
x	<<=	y	x = x << y
x	>>=	y	x = x >> y
x	>>>=	y	x = x >>> y
x	&=	y	x = x & y
x	|=	y	x = x | y
x	^=	y	x = x ^ y

Operatory porównania (relacyjne)

Operatory porównania, czyli relacyjne, służą oczywiście do porównywania argumentów. Wynikiem takiego porównania jest wartość logiczna `true` (jeśli jest ono prawdziwe) lub `false` (jeśli jest fałszywe). Zatem wynikiem operacji `argument1 == argument2` będzie `true`, jeżeli argumenty są sobie równe, lub `false`, jeżeli argumenty są różne. Czyli `4 == 5` ma wartość `false`, a `2 == 2` ma wartość `true`. Do dyspozycji mamy operatory porównania zawarte w tabeli 2.6.

Operator warunkowy

Operator warunkowy ma następującą składnię:

```
warunek ? wartość1 : wartość2;
```

Wyrażenie takie przybiera *wartość1*, jeżeli warunek jest prawdziwy, lub *wartość2* w przeciwnym przypadku.

Tabela 2.6. *Operatory porównania w Javie*

Operator	Opis
==	jeśli argumenty są sobie równe, wynikiem jest `true`
!=	jeśli argumenty są różne, wynikiem jest `true`
>	jeśli argument prawostronny jest mniejszy od lewostronnego, wynikiem jest `true`
<	jeśli argument prawostronny jest większy od lewostronnego, wynikiem jest `true`
>=	jeśli argument prawostronny jest mniejszy lub równy lewostronnemu, wynikiem jest `true`
<=	jeśli argument prawostronny jest większy lub równy lewostronnemu, wynikiem jest `true`

ĆWICZENIE

2.14. Wykorzystanie operatora warunkowego

Wykorzystaj operator warunkowy do zmodyfikowania wartości dowolnej zmiennej typu całkowitego (`int`).

```
public
class Main
{
  public static void main (String args[])
  {
    int x = 1, y;
    y = (x == 1 ? 10 : 20);
    System.out.println ("y = " + y);
  }
}
```

W powyższym ćwiczeniu najważniejszy jest oczywiście wiersz:

```
y = (x == 1? 10 : 20);
```

który oznacza: jeżeli `x` jest równe `1`, przypisz zmiennej `y` wartość `10`, w przeciwnym przypadku przypisz zmiennej `y` wartość `20`. Ponieważ zmienną `x` zainicjalizowaliśmy wartością `1`, na ekranie zostanie wyświetlony ciąg znaków `y = 10`.

Priorytety operatorów

Sama znajomość operatorów to jednak nie wszystko. Niezbędna jest jeszcze wiedza na temat tego, jaki mają one priorytet, czyli jaka jest kolejność ich wykonywania. Wiadomo na przykład, że mnożenie jest „silniejsze" od dodawania, zatem najpierw mnożymy, potem dodajemy. W Javie jest podobnie, siła każdego operatora jest ściśle określona. Przedstawia to tabela 2.7[3]. Im wyższa pozycja w tabeli, tym wyższy priorytet operatora. Operatory znajdujące się na jednym poziomie (w jednym wierszu) mają ten sam priorytet.

Tabela 2.7. *Priorytety operatorów w Javie*

Grupa operatorów	Symbole
inkrementacja przyrostkowa	`++, --`
inkrementacja przedrostkowa, negacje	`++, --, ~, !`
mnożenie, dzielenie	`*, /, %`
przesunięcia bitowe	`<<, >>, >>>`
porównania	`<, >, <=, >=`
porównania	`==, !=`
bitowe AND	`&`
bitowe XOR	`^`
bitowe OR	`|`
logiczne AND	`&&`
logiczne OR	`||`
warunkowe	`?`
przypisania	`=, +=, -=, *=, /=, %=, >>=, <<=, >>>=, &=, ^=, |=`

[3] Tabela nie przedstawia wszystkich operatorów występujących w Javie, a jedynie omawiane w książce.

Instrukcje

Instrukcja warunkowa if...else

Bardzo często w programie zachodzi potrzeba sprawdzenia jakiegoś warunku i w zależności od tego, czy jest on prawdziwy, czy fałszywy, wykonanie różnych instrukcji. Do takiego sprawdzania służy właśnie instrukcja warunkowa `if...else`. Ma ona ogólną postać:

```
if (wyrażenie warunkowe){
  //instrukcje do wykonania, jeżeli warunek jest prawdziwy
}
else{
  //instrukcje do wykonania, jeżeli warunek jest fałszywy
}
```

wyrażenie warunkowe, inaczej niż w C i C++, musi dać w wyniku wartość typu `boolean`, tzn. `true` lub `false`.

ĆWICZENIE

2.15. Użycie instrukcji warunkowej if...else

Wykorzystaj instrukcję warunkową `if...else` do stwierdzenia, czy wartość zmiennej typu całkowitego jest mniejsza od zera.

```
public
class Main
{
  public static void main (String args[])
  {
    int a = -10;
    if (a > 0){
      System.out.println ("Zmienna a jest większa od zera");
    }
    else{
      System.out.println ("Zmienna a nie jest większa od zera");
    }
  }
}
```

Spróbujmy teraz czegoś nieco bardziej skomplikowanego. Zajmijmy się klasycznym przykładem liczenia pierwiastków równania kwadratowego. Przypomnijmy, że jeśli mamy równanie o postaci $A * x^2 + B * x + C = 0$, to — aby obliczyć jego rozwiązanie — liczymy tzw. deltę (Δ), która jest równa $B^2 - 4 * A * C$. Jeżeli delta jest większa od zera, mamy dwa pierwiastki: $x_1 = (-B + \sqrt{\Delta}) / (2 * A)$ i $x_2 = (-B - \sqrt{\Delta}) / (2 * A)$.

Jeżeli delta jest równa zero, istnieje tylko jedno rozwiązanie — mianowicie x = –B / (2 * A). W przypadku trzecim, jeżeli delta jest mniejsza od zera, równanie takie nie ma rozwiązań w zbiorze liczb rzeczywistych.

Skoro jest tutaj tyle warunków do sprawdzenia, to jest to doskonały przykład do potrenowania zastosowania instrukcji `if...else`. Aby nie komplikować zagadnienia, nie będziemy się w tej chwili zajmować wczytywaniem parametrów równania z klawiatury, ale podamy je bezpośrednio w kodzie.

ĆWICZENIE

2.16. Pierwiastki równania kwadratowego

Wykorzystaj operacje arytmetyczne oraz instrukcję `if...else` do obliczenia pierwiastków równania kwadratowego o parametrach podanych bezpośrednio w kodzie programu.

```
public
class Main
{
  public static void main (String args[])
  {
    int parametrA = 1, parametrB = -1, parametrC = -6;

    System.out.println ("Parametry równania:\n");
    System.out.println ("A: " + parametrA + " B: " + parametrB + " C: "
    ➥+ parametrC + "\n");

    if (parametrA == 0){
      System.out.println ("To nie jest równanie kwadratowe: A = 0!");
    }
    else{
      double delta = parametrB * parametrB - 4 * parametrA * parametrC;
      if (delta < 0){
        System.out.println ("Delta < 0.");
        System.out.println ("To równanie nie ma rozwiązania w zbiorze
        ➥liczb rzeczywistych");
      }
      else{
        double wynik;
        if (delta == 0){
          wynik = - parametrB / 2 * parametrA;
          System.out.println ("Rozwiązanie: x = " + wynik);
        }
        else{
          wynik = (- parametrB + Math.sqrt(delta)) / 2 * parametrA;
          System.out.print ("Rozwiązanie: x1 = " + wynik);
          wynik = (- parametrB - Math.sqrt(delta)) / 2 * parametrA;
          System.out.println (", x2 = " + wynik);
        }
```

```
        }
      }
    }
  }
```

Jak łatwo zauważyć, instrukcję warunkową można zagnieżdżać, tzn. po jednym `if` może występować kolejne, po nim następne itd. Jednak jeżeli zapiszemy to w sposób podany w poprzednim ćwiczeniu, przy wielu zagnieżdżeniach otrzymamy bardzo nieczytelny kod. Możemy więc posłużyć się konstrukcją `if...else if`. Zamiast tworzyć mniej wygodną konstrukcję, taką jak przedstawiona poniżej:

```
if (warunek1){
  //instrukcje 1
}
else{
  if (warunek2){
    //instrukcje 2
  }
  else{
    if (warunek3){
      //instrukcje 3
    }
    else{
      //instrukcje 4
    }
  }
}
```

całość możemy zapisać dużo prościej i czytelniej w postaci:

```
if (warunek1){
  //instrukcje 1
}
else if (warunek2){
  //instrukcje 2
}
else if (warunek3){
  //instrukcje 3
}
else{
  //instrukcje 4
}
```

ĆWICZENIE

2.17. Zastosowanie instrukcji if...else if

Napisz kod obliczający pierwiastki równania kwadratowego o parametrach zadanych w programie. Wykorzystaj instrukcję `if...else if`.

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

```
  {
    int parametrA = 1, parametrB = -1, parametrC = -6;

    System.out.println ("Parametry równania:\n");
    System.out.println ("A: " + parametrA + " B: " + parametrB + " C: "
    ➥+ parametrC + "\n");

    if (parametrA == 0){
      System.out.println ("To nie jest równanie kwadratowe: A = 0!");
    }
    else{
      double delta = parametrB * parametrB - 4 * parametrA * parametrC;
      double wynik;

      if (delta < 0){
        System.out.println ("Delta < 0.");
        System.out.println ("To równanie nie ma rozwiązania w zbiorze
        ➥liczb rzeczywistych");
      }
      else if (delta == 0){
        wynik = - parametrB / 2 * parametrA;
        System.out.println ("Rozwiązanie: x = " + wynik);
      }
      else{
        wynik = (- parametrB + Math.sqrt(delta)) / 2 * parametrA;
        System.out.print ("Rozwiązanie: x1 = " + wynik);
        wynik = (- parametrB - Math.sqrt(delta)) / 2 * parametrA;
        System.out.println (" x2 = " + wynik);
      }
    }
  }
}
```

Instrukcja wyboru switch

Instrukcja `switch` pozwala w wygodny i przejrzysty sposób sprawdzać ciąg warunków i wykonywać różny kod w zależności od tego, czy są one prawdziwe, czy fałszywe. Można nią zastąpić ciąg instrukcji `if...else if`. Jeżeli mamy w kodzie przykładową konstrukcję w postaci:

```
if (a == 1){
  instrukcje1;
}
else if (a == 50){
  instrukcje2;
}
else if (a == 23){
  instrukcje3;
}
else{
  instrukcje4;
}
```

to możemy zastąpić ją następująco:

```
switch (a){
  case 1:
    instrukcje1;
    break;
  case 50:
    instrukcje2;
    break;
  case 23:
    instrukcje3;
    break;
  default:
    instrukcje4;
}
```

Po kolei jest tu sprawdzane, czy `a` jest równe 1, potem 50 i w końcu 23. Jeżeli równość zostanie w jednym z przypadków stwierdzona, wykonywane są instrukcje po odpowiedniej klauzuli `case`. Jeżeli `a` nie jest równe żadnej z wymienionych liczb, wykonywane są instrukcje po słowie `default`. Instrukcja `break` powoduje wyjście z bloku `switch`.

ĆWICZENIE

2.18. Użycie instrukcji wyboru switch

Używając instrukcji `switch`, napisz program sprawdzający, czy wartość zadeklarowanej zmiennej jest równa 1, czy 10. Wyświetl na ekranie stosowny komunikat.

```
public
class Main
{
  public static void main (String args[])
  {
    int a = 10;
    switch (a){
      case 1 :
        System.out.println("a = 1");
        break;
      case 10:
        System.out.println("a = 10");
        break;
      default:
        System.out.println("a nie jest równe ani 1, ani 10.");
    }
  }
}
```

Uwaga! Jeżeli zapomnimy o słowie `break`, wykonywanie instrukcji `switch` będzie kontynuowane, co może prowadzić do otrzymania niespodziewanych efektów. W szczególności zostanie wtedy wykonany

blok instrukcji występujący po `default`. Może to być, oczywiście, efektem zamierzonym, może też jednak powodować trudne do wykrycia błędy.

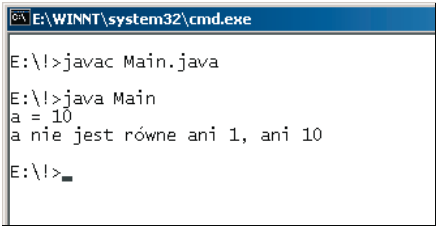
ĆWICZENIE

2.19. Efekt pominięcia instrukcji break

Zmodyfikuj kod z ćwiczenia 2.18, usuwając instrukcję `break`. Zaobserwuj, jak zmieniło się działanie programu.

```
public
class Main
{
  public static void main (String args[])
  {
    int a = 10;
    switch (a){
      case 1:
        System.out.println("a = 1");
      case 10:
        System.out.println("a = 10");
      default:
        System.out.println("a nie jest równe ani 1, ani 10");
    }
  }
}
```

Widać wyraźnie (rysunek 2.5), że teraz według naszego programu zmienna `a` jest jednocześnie równa 10, jak i różna od dziesięciu.



Rysunek 2.5. *Ilustracja błędu z ćwiczenia 2.19*

Pętla for

Pętle w językach programowania pozwalają na wykonywanie powtarzających się czynności. Nie inaczej jest w Javie. Jeśli chcemy np. wypisać na ekranie 10 razy napis Java, to możemy zrobić to, pisząc

10 razy `System.out.println("Java");`. Jeżeli jednak chcielibyśmy mieć już 150 takich napisów, to, pomijając oczywiście sensowność tej czynności, byłby to już problem. Na szczęście z pomocą przychodzą nam właśnie pętle. Pętla typu `for` ma następującą składnię:

```
for (wyrażenie początkowe; wyrażenie warunkowe; wyrażenie
modyfikujące){
  //instrukcje do wykonania
}
```

wyrażenie początkowe jest stosowane do zainicjalizowania zmiennej używanej jako licznik liczby wykonań pętli. *wyrażenie warunkowe* określa warunek, jaki musi być spełniony, aby dokonać kolejnego przejścia w pętli, *wyrażenie modyfikujące* jest zwykle używane do modyfikacji zmiennej będącej licznikiem.

ĆWICZENIE

2.20. Budowa pętli for

Wykorzystując pętlę typu `for`, napisz program wyświetlający na ekranie 10 razy napis `Java`.

```
public
class Main
{
  public static void main (String args[])
  {
    for (int i = 1; i <= 10; i++){
      System.out.println ("Java");
    }
  }
}
```

Zmienna `i` to tzw. zmienna iteracyjna, której na początku przypisujemy wartość `1` (`int i = 1`). Następnie w każdym przebiegu pętli jest ona zwiększana o jeden (`i++`) oraz wykonywana jest instrukcja `System.out.println ("Java");`. Wszystko trwa tak długo, aż `i` osiągnie wartość `10` (`i <= 10`).

Wyrażenie modyfikujące jest zwykle używane do modyfikacji zmiennej iteracyjnej. Takiej modyfikacji możemy jednak dokonać również wewnątrz pętli. Struktura tego typu wygląda następująco:

```
for (wyrażenie początkowe; wyrażenie warunkowe;){
  instrukcje do wykonania
  wyrażenie modyfikujące
}
```

ĆWICZENIE

2.21. Wyrażenie modyfikujące w bloku instrukcji

Zmodyfikuj pętlę typu `for` z ćwiczenia 2.20 tak, aby wyrażenie modyfikujące znalazło się w bloku instrukcji.

```
public
class Main
{
  public static void main (String args[])
  {
    for (int i = 1; i <= 10;){
      System.out.println ("Java");
      i++;
    }
  }
}
```

Zwróćmy uwagę, że mimo iż wyrażenie modyfikujące jest teraz wewnątrz pętli, średnik znajdujący się po `i <= 10` jest niezbędny! Jeśli o nim zapomnimy, kompilator zgłosi błąd.

Kolejną ciekawą możliwością jest połączenie wyrażenia warunkowego i modyfikującego.

ĆWICZENIE

2.22. Łączenie wyrażenia warunkowego i modyfikującego

Napisz taką pętlę typu `for`, aby wyrażenie warunkowe było jednocześnie wyrażeniem modyfikującym.

```
public
class Main
{
  public static void main (String args[])
  {
    for (int i = 1; i++ <= 10;){
      System.out.println ("Java");
    }
  }
}
```

W podobny sposób jak w poprzednich przykładach możemy się pozbyć wyrażenia początkowego, które przeniesiemy przed pętlę. Schemat wygląda następująco:

```
wyrażenie początkowe;
for (; wyrażenie warunkowe;){
  instrukcje do wykonania
  wyrażenie modyfikujące
}
```

ĆWICZENIE

2.23. Wyrażenie początkowe przed pętlą

Zmodyfikuj pętlę typu `for` w taki sposób, aby wyrażenie początkowe znalazło się przed pętlą, a wyrażenie modyfikujące wewnątrz niej.

```
public
class Main
{
  public static void main (String args[])
  {
    int i = 1;
    for (; i <= 10;){
      System.out.println ("Java");
      i++;
    }
  }
}
```

Skoro zaszliśmy już tak daleko w pozbywaniu się wyrażeń sterujących, usuńmy również wyrażenie warunkowe. Jest to jak najbardziej możliwe!

ĆWICZENIE

2.24. Pętla bez wyrażeń

Zmodyfikuj pętlę typu `for` w taki sposób, aby wyrażenie początkowe znalazło się przed pętlą, natomiast wyrażenie modyfikujące i warunkowe wewnątrz pętli.

```
public
class Main
{
  public static void main (String args[])
  {
    int i = 1;
    for ( ; ; ){
      System.out.println ("Java");
      if (i++ >= 10) break;
    }
  }
}
```

Przy stosowaniu tego typu konstrukcji pamiętajmy, że oba średniki w nawiasach okrągłych występujących po `for` są niezbędne do prawidłowego funkcjonowania kodu. Warto też zwrócić uwagę na zmianę kierunku nierówności. We wcześniejszych przykładach sprawdzaliśmy bowiem, czy `i` jest mniejsze bądź równe 10, a teraz, czy jest większe bądź równe. Dzieje się tak, dlatego że poprzednio sprawdzaliśmy,

czy pętla ma być dalej wykonywana, natomiast obecnie, czy ma zostać zakończona. Przy okazji wykorzystaliśmy też kolejną instrukcję, mianowicie `break`. Służy ona do natychmiastowego przerwania wykonywania pętli.

Kolejna przydatna instrukcja, `continue`, powoduje rozpoczęcie kolejnej iteracji, tzn. w miejscu jej wystąpienia wykonywanie bieżącej iteracji jest przerywane i rozpoczyna się kolejny przebieg.

ĆWICZENIE

2.25. Zastosowanie instrukcji continue

Napisz program wyświetlający na ekranie liczby od 1 do 20, które nie są podzielne przez 2. Skorzystaj z pętli `for` i instrukcji `continue`.

```
public
class Main
{
  public static void main (String args[])
  {
    for (int i = 1; i <= 20; i++){
      if (i % 2 == 0)
        continue;
      System.out.println (i);
    }
  }
}
```

Przypomnijmy, że `%` to operator dzielenia modulo, tzn. dostarcza on resztę z dzielenia. Nic jednak nie stoi na przeszkodzie, aby działającą w taki sam sposób aplikację napisać bez użycia instrukcji `continue`.

ĆWICZENIE

2.26. Liczby niepodzielne przez dwa

Zmodyfikuj kod z ćwiczenia 2.25 tak, aby nie było konieczności użycia instrukcji `continue`.

```
public
class Main
{
  public static void main (String args[])
  {
    for (int i = 1; i <= 20; i++){
      if (i % 2 != 0) System.out.println (i);
    }
  }
}
```

Pętla while

O ile pętla typu `for` służy raczej do wykonywania znanej z góry liczby operacji, to w przypadku pętli `while` liczba ta nie jest zwykle znana. Nie jest to, oczywiście, obligatoryjny podział. Tak naprawdę obie można napisać w taki sposób, aby były swoimi funkcjonalnymi odpowiednikami. Ogólna konstrukcja pętli typu `while` jest następująca:

```
while (wyrażenie warunkowe){
  instrukcje
}
```

Instrukcje są wykonywane tak długo, dopóki wyrażenie warunkowe jest prawdziwe. Oznacza to, że gdzieś w pętli musi wystąpić modyfikacja warunku bądź też instrukcja `break`. Inaczej będzie się ona wykonywała w nieskończoność!

ĆWICZENIE

2.27. Budowa pętli while

Używając pętli typu `while`, napisz program wyświetlający na ekranie 10 razy napis `Java`.

```
public
class Main
{
  public static void main (String args[])
  {
    int i = 1;
    while (i <= 10){
      System.out.println ("Java");
      i++;
    }
  }
}
```

ĆWICZENIE

2.28. Połączone wyrażenia

Zmodyfikuj kod z ćwiczenia 2.27 tak, aby wyrażenie warunkowe zmieniało jednocześnie wartość zmiennej `i`.

```
public
class Main
{
  public static void main (String args[])
  {
    int i = 1;
```

```
      while (i++ <= 10){
        System.out.println ("Java");
      }
    }
}
```

ĆWICZENIE

2.29. Liczby nieparzyste i pętla while

Korzystając z pętli `while`, napisz program wyświetlający na ekranie liczby od 1 do 20 niepodzielne przez 2.

```
public
class Main
{
  public static void main (String args[])
  {
    int i = 1;
    while (i <= 20){
      if (i % 2 != 0)
        System.out.println (i);
      i++;
    }
  }
}
```

Pętla do...while

Istnieje jeszcze jedna odmiana pętli. Jest to `do...while`. Jej konstrukcja jest następująca:

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

ĆWICZENIE

2.30. Budowa pętli do...while

Korzystając z pętli `do...while`, napisz program wyświetlający na ekranie 10 razy dowolny napis.

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

```
    int i = 1;
    do{
      System.out.println ("Java");
    }
    while (i++ <= 9);
  }
}
```

Wydawać by się mogło, że to przecież to samo, co zwykła pętla `while`. Jest jednak pewna różnica. Otóż w przypadku pętli `do...while` instrukcje wykonane są co najmniej jeden raz, nawet jeśli warunek jest na pewno fałszywy.

ĆWICZENIE

2.31. Pętla do...while z fałszywym warunkiem

Zmodyfikuj kod z ćwiczenia 2.30 w taki sposób, aby wyrażenie warunkowe na pewno było fałszywe. Zaobserwuj wyniki działania programu.

```
public
class Main
{
  public static void main (String args[])
  {
    int i = 10;
    do{
      System.out.println ("Java");
    }
    while (i++ <= 9);
  }
}
```

Pętla foreach

Począwszy od wersji 1.5 (5.0), Java udostępnia nowy rodzaj pętli. Jest ona nazywana pętlą `foreach` lub rozszerzoną pętlą `for` (z ang. *enhanced for*) i pozwala na automatyczną iterację po kolekcji obiektów lub też po tablicy. Jej działanie pokażemy właśnie w tym drugim przypadku. Jeśli bowiem mamy tablicę `tab` zawierającą wartości pewnego typu, to do przejrzenia wszystkich jej elementów możemy użyć konstrukcji w postaci:

```
for(typ val: tablica){
  //instrukcje
}
```

W takim przypadku w kolejnych przebiegach pętli `for` pod `val` będzie podstawiana wartość kolejnej komórki.

ĆWICZENIE

2.32. Wykorzystanie rozszerzonej pętli for

Zadeklaruj tablicę liczb typu `int` i wypełnij ją przykładowymi danymi. Następnie użyj rozszerzonej pętli `for` do wyświetlenia zawartości tablicy na ekranie.

```
public
class Main
{
  public static void main (String args[])
  {
    int tab[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    for(int val: tab){
      System.out.println(val);
    }
  }
}
```