Java Wydanie III OddEven TABLICE INFORMATYCZNE

Marcin Lis

Helion

BUDOWA APLIKACJI

Program w Javie składa się ze zbioru klas. Każda klasa publiczna musi być zapisana w oddzielnym pliku o takiej samej nazwie, jak nazwa tej klasy (zasada ta nie dotyczy klas wewnętrznych i pakietowych), oraz rozszerzeniu *java*. Wykonywanie programu rozpoczyna się od metody o nazwie `main`, która musi być zadeklarowana jako publiczna i statyczna. Jeżeli program składa się z wielu klas publicznych i więcej niż jedna z nich zawiera taką metodę `main`, to aplikacja ma wiele punktów wejścia (startowych), może być więc uruchamiana na kilka sposobów. Schematyczna konstrukcja programu:

```
public class Main
{
  public static void main(String args[])
  {
    //kod metody main
  }
}
```

KOMPILACJA KODU

W przypadku korzystania ze zintegrowanych środowisk programistycznych sposób kompilacji zależy od wykorzystywanego produktu. W przypadku korzystania z kompilatora pracującego w wierszu poleceń, zawartego w pakiecie Java Development Kit (JDK), kompilacja odbywa się po wydaniu następującego polecenia:

```
javac nazwa_pliku.java
```

Kompilator ten udostępnia m.in. opcje:

Opcja	Znaczenie
`g`	Włącza lub wyłącza generowanie informacji dla debugera
`nowarn`	Wyłącza wyświetlanie ostrzeżeń
`verbose`	Wyświetla dodatkowe informacje o postępie kompilacji
`deprecation`	Wyświetla informacje o użyciu przestarzałych metod API
`classpath`	Specyfikuje położenie plików bibliotecznych
`sourcepath`	Specyfikuje położenie plików źródłowych
`d`	Określa położenie plików wynikowych
`encoding`	Określa standard kodowania znaków w plikach źródłowych
`source`	Określa standard, z którym kompatybilne są pliki źródłowe
`target`	Określa kompatybilność kodów wynikowych z poszczególnymi wersjami maszyn wirtualnych
`help`	Wyświetla skróconą listę opcji kompilatora

Parametr `target` może przyjmować jedną z następujących wartości:

- 1.1 — kod zgodny z maszyną wirtualną w wersji 1.1;
- 1.2 — kod zgodny z wersją 1.2 i wyższymi;
- 1.3 — kod zgodny z wersją 1.3 i wyższymi;
- 1.4 — kod zgodny z wersją 1.4 i wyższymi;
- 1.5 — kod zgodny z wersją 1.5 i wyższymi;
- 1.6 — kod zgodny z wersją 1.6 i wyższymi;
- 1.7 — kod zgodny z wersją 1.7 i wyższymi;
- 5 — kod zgodny z wersją 1.5 i wyższymi;
- 6 — kod zgodny z wersją 1.6 i wyższymi;
- 7 — kod zgodny z wersją 1.7 i wyższymi.

TYPY DANYCH

Java udostępnia pewną liczbę wbudowanych typów danych, czyli takich, które oferuje sam język i z których można korzystać bez potrzeby ich definiowania. Określa się je jako typy proste lub podstawowe (z ang. *primitive types*).

Typ znakowy (char)

Typ `char` służy do reprezentowania wszelkich znaków, m.in. liter. Jest on 16-bitowy i opiera się na standardzie Unicode (czyli standardzie umożliwiającym przedstawienie znaków występujących w większości języków świata). Ponieważ znaki reprezentowane są tak naprawdę jako 16-bitowe kody liczbowe, typ ten można zaliczyć również do typów arytmetycznych.

Typ logiczny (boolean)

Typ `boolean` może reprezentować jedynie dwie wartości: `true` (prawda) i `false` (fałsz).

Typy arytmetyczne

Typy całkowitoliczbowe

Typy arytmetyczne całkowitoliczbowe służą do reprezentowania liczb całkowitych.
W Javie występują cztery ich rodzaje:

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

Rodzaje te różnią się zakresem liczb, które można reprezentować za ich pomocą, co przedstawia poniższa tabela.

Typ	Liczba bitów	Zakres
`byte`	8	od –128 do 127
`short`	16	od –32 768 do 32 767
`int`	32	od -2^{31} do $2^{31}-1$
`long`	64	od -2^{63} do $2^{63}-1$

Osoby programujące w innych językach programowania, takich jak C, C++ czy PHP, powinny zwrócić uwagę, że zakres wartości możliwych do przedstawienia jest z góry ustalony i nie zależy od platformy systemowej, na której uruchamiany jest program w Javie.

Typy zmiennopozycyjne

Typy zmiennopozycyjne występują w dwóch odmianach:

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

różniących się rozmiarem oraz zakresem liczb możliwych do zaprezentowania.

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

KOMENTARZE

W Javie istnieją dwa rodzaje komentarzy, oba zapożyczone z języków takich, jak C i C++:

- blokowy,
- wierszowy.

Komentarz blokowy

Komentarz blokowy rozpoczyna się od znaków `/*`, a kończy się znakami `*/`. Wszystko, co znajduje się pomiędzy tymi znakami, jest traktowane przez kompilator jako komentarz i pomijane w procesie kompilacji. Umiejscowienie komentarza blokowego jest w zasadzie dowolne — może on się znaleźć nawet w środku instrukcji (pod warunkiem, że nie zostanie przedzielone żadne słowo). Komentarzy blokowych nie wolno zagnieżdżać.

```
class Main
{
  public static void main(String args[])
  {
    /*
    to jest komentarz blokowy
    */
    System.out.println("To jest napis");
  }
}
```

Komentarz wierszowy

Komentarz wierszowy zaczyna się od znaków `//` i obowiązuje do końca danej linii programu. Wszystko to, co występuje po tych dwóch znakach aż do końca bieżącej linii, jest ignorowane przez kompilator.

```
class Main
{
  public static void main(String args[])
  {
    //to jest komentarz wierszowy
    System.out.println ("To jest napis.");
  }
}
```

Komentarz wierszowy może znaleźć się w środku komentarza blokowego:

```
/*
//ta konstrukcja jest poprawna
*/
```

ZMIENNE

Deklaracja pojedynczej zmiennej

Deklaracja zmiennej polega na podaniu jej typu oraz nazwy i kończy się znakiem średnika; schematycznie:

```
nazwa_zmiennej typ_zmiennej;
```

Przykładowo:

```
class Main
{
  public static void main(String args[])
  {
    int liczba;
  }
}
```

Deklaracja może być równoczesna z przypisaniem wartości (inicjalizacją zmiennej):

```
nazwa_zmiennej typ_zmiennej = wartość;
```

Przykładowo:

```
int liczba = 100;
```

Nazwy zmiennych

Nazwa zmiennej może się składać z liter (zarówno małych, jak i dużych), cyfr oraz znaku podkreślenia, nie może jednak zaczynać się od cyfry. Dopuszczalny jest także znak dolara, ale przyjmuje się, że jest on zarezerwowany dla narzędzi przetwarzających kod i raczej nie należy go stosować w nazwach zmiennych. Można wykorzystać znaki spoza ścisłego alfabetu łacińskiego (wszelkiego rodzaju znaki narodowe).

Deklaracja wielu zmiennych

W jednym wierszu można deklarować wiele zmiennych, jeśli są one tego samego typu; schematycznie:

```
typ_zmiennej nazwa1, nazwa2, ..., nazwaN;
```

Przykładowo:

```
int pierwsza_liczba, druga_liczba, trzecia_liczba;
```

Deklaracje wielu zmiennych mogą być powiązane z ich równoczesną inicjalizacją:

```
int pierwsza_liczba = 100, druga_liczba, trzecia_liczba = 200;
```

Zmienne referencyjne

Zmienne typów referencyjnych (z ang. *reference types*), inaczej odnośnikowych, deklaruje się tak, jak przedstawiono wyżej, czyli:

```
typ_zmiennej nazwa_zmiennej;
```

z tą różnicą, że konstrukcja ta powoduje powstanie jedynie odniesienia, któremu domyślnie zostanie przypisana wartość pusta `null`. Zmiennej referencyjnej po deklaracji należy przypisać odniesienie do obiektu utworzonego oddzielną instrukcją (patrz sekcja „Klasy i obiekty").

OPERATORY

Operatory arytmetyczne

Operator	Wykonywane działanie
`*`	mnożenie
`/`	dzielenie
`+`	dodawanie
`-`	odejmowanie
`%`	dzielenie modulo (reszta z dzielenia)
`++`	inkrementacja (zwiększanie)
`--`	dekrementacja (zmniejszanie)

Operatory logiczne

Operator	Symbol
Iloczyn (AND)	`&&`
Suma (OR)	`||`
Negacja (NOT)	`!`

Operatory logiczne działają według zasad opisanych poniżej.

Iloczyn logiczny

Wynikiem operacji AND (iloczynu logicznego) jest wartość `true`, wtedy i tylko wtedy, kiedy oba argumenty mają wartość `true`. W każdym innym przypadku wynikiem jest wartość `false`.

Argument 1	Argument 2	Wynik
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

Suma logiczna

Wynikiem operacji OR (sumy logicznej) jest wartość `false`, wtedy i tylko wtedy, kiedy oba argumenty mają wartość `false`. W każdym innym przypadku wynikiem jest `true`.

Argument 1	Argument 2	Wynik
`true`	`true`	`true`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

Negacja logiczna

Operacja NOT (negacja logiczna) zamienia wartość argumentu na przeciwną. Jeśli argument miał wartość `true`, będzie miał wartość `false`, jeśli argument miał wartość `false`, będzie miał wartość `true`.

Argument	Wynik
`true`	`false`
`false`	`true`

Operatory bitowe

Operator	Symbol
Iloczyn (AND)	`&`
Suma (OR)	`|`
Negacja (NOT)	`~`
Różnica symetryczna (XOR)	`^`
Przesunięcie bitowe w prawo	`>>`
Przesunięcie bitowe w lewo	`<<`
Przesunięcie bitowe w prawo z wypełnieniem zerami	`>>>`

Operatory przypisania

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`

Ciąg dalszy na stronie 2 ➔

Operatory porównywania

Operator	Opis
==	Wynikiem jest true, jeśli argumenty są sobie równe
!=	Wynikiem jest true, jeśli argumenty są różne
>	Wynikiem jest true, jeśli argument lewostronny jest większy od prawostronnego
<	Wynikiem jest true, jeśli argument lewostronny jest mniejszy od prawostronnego
>=	Wynikiem jest true, jeśli argument lewostronny jest większy od prawostronnego lub równy mu
<=	Wynikiem jest true, jeśli argument lewostronny jest mniejszy od prawostronnego lub równy mu

Priorytet operatorów

(Od najsilniejszych do najsłabszych; operatory w jednym wierszu mają ten sam priorytet).

Grupa operatorów	Symbole
Inkrementacja przyrostkowa	++,--
Inkrementacja przedrostkowa, negacja	++,--,~,!
Mnożenie, dzielenie	*,/,%
Dodawanie, odejmowanie	+,-
Przesunięcia bitowe	<<,>>,>>>
Porównania (mniejsze, większe)	<,>,<=,>=
Porównania (równe, różne)	==,!=
Bitowe AND	&
Bitowe XOR	^
Bitowe OR	|
Logiczne AND	&&
Logiczne OR	||
Warunkowy	?
Przypisania	=,+=,-=,*=,/=,%=,&=,^=,|=,<<=,>>=,>>>=

INSTRUKCJE JĘZYKA

Instrukcja warunkowa if...else

Podstawowa instrukcja warunkowa ma postać:

```
if (warunek){
  instrukcje1
}
else{
  instrukcje2
}
```

Oznacza to, że jeżeli *warunek* jest prawdziwy, to zostaną wykonane *instrukcje1*, w przeciwnym wypadku zostaną wykonane *instrukcje2*. Blok else jest opcjonalny.

Instrukcja warunkowa if...else if

Instrukcja złożona if...else if ma postać:

```
if (warunek1){
  instrukcje1
}
else if (warunek2){
  instrukcje2
}
//dalsze bloki else if
else{
  instrukcjeN
}
```

Oznacza to, że jeżeli *warunek* jest prawdziwy, to zostaną wykonane *instrukcje1*, w przeciwnym wypadku, jeżeli prawdziwy jest *warunek2*, zostaną wykonane *instrukcje2*. Liczba bloków else if nie jest ograniczona. Jeżeli żaden warunek nie będzie spełniony, zostaną wykonane *instrukcjeN* z bloku else. Blok else jest opcjonalny. **Przykład:**

```
class Main
{
  public static void main(String args[])
  {
    int x = 0;
    if (x > 0){
      System.out.println("x większe
↪od zera");
    }
    else if(x < 0){
      System.out.println("x mniejsze
↪od zera");
    }
    else{
      System.out.println("x równe zero");
    }
  }
}
```

Instrukcja switch

Instrukcja warunkowa switch może zastąpić serię instrukcji if...else if:

```
switch(wyrażenie){
  case wartość1 :
    instrukcje1;
    break;
  case wartość2 :
    instrukcje2;
    break;
  case wartość3 :
    instrukcje3;
    break;
  default :
    instrukcje4;
}
```

Znaczenie jest takie samo, jak w innych językach programowania. Jeżeli wartością wyrażenia jest *wartość1*, wykonywane są *instrukcje1*, jeżeli wartością wyrażenia jest *wartość2*, wykonywane są *instrukcje2* itd. Jeżeli nie uda się dopasować wartości wyrażenia do wartości występujących po klauzulach case, wykonywane są instrukcje występujące po słowie default. Instrukcja break przerywa wykonywanie bloku switch.

Operator warunkowy

Operator warunkowy ma postać:

```
warunek ? wartość1 : wartość2
```

Zapis ten oznacza, że jeżeli *warunek* jest prawdziwy, wartością wyrażenia staje się *wartość1*, w przeciwnym wypadku wartością wyrażenia staje się *wartość2*.

```
class Main
{
  public static void main (String args[])
  {
    int liczba = 10;
    int liczba2 = liczba < 0 ? -1 : 1;
    System.out.println(liczba2);
  }
}
```

Pętla for

Pętla for ma ogólną postać:

```
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* używane jest zwykle do modyfikacji zmiennej będącej licznikiem.

Przykład:

```
for(int i = 0; i < 10; i++){
  System.out.println("Przebieg " + i);
}
```

Rozszerzona pętla for

Począwszy od Javy w wersji 5.0 (1.5), dostępna jest rozszerzona pętla for (z ang. *enhanced for*), nazywana też pętlą typu foreach. Umożliwia ona automatyczną iterację po obiekcie udostępniającym iterator (np. po kolekcji lub tablicy). Konstrukcja jest następująca:

```
for(typ nazwa: obj){
  //instrukcje
}
```

W kolejnych przebiegach pętli pod *nazwa* będzie podstawiana wartość kolejnego elementu (np. kolejnej komórki tablicy). Pętla będzie działała tak długo, aż zostaną przejrzane wszystkie elementy obiektu *obj* typu *typ*. **Przykład:**

```
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);
    }
  }
}
```

Pętla while

Ogólna postać pętli while jest następująca:

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

Oznacza to, że dopóki warunek jest prawdziwy, będą wykonywane instrukcje. **Przykład:**

```
int i = 0;
while(i < 10){
  System.out.println("Przebieg " + i);
  i++;
}
```

Pętla do...while

Pętla do...while jest odmianą pętli while i ma postać:

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

Oznacza to, że *instrukcje* będą wykonywane, dopóki warunek jest prawdziwy. **Przykład:**

```
int i = 0;
do{
  System.out.println("Przebieg " + i);
}
while(i++ < 9);
```

Instrukcja break

Pętle pojedyncze

Instrukcja break powoduje przerwanie wykonywania bieżącej iteracji pętli i opuszczenie bloku pętli. **Przykład:**

```
int i = 0;
while(true){
  System.out.println("Przebieg " + i);
  if(i++ >= 9) break;
}
```

Pętle zagnieżdżone

W przypadku pętli zagnieżdżonych instrukcja break powoduje przerwanie bieżącej iteracji jedynie tej pętli, w której została umieszczona:

```
for(int i = 0; i < 3; i++){
  for(int j = 0; j < 3; j++){
    if(i == 1) break; //w tym miejscu
↪ zostanie przerwane wykonywanie pętli
↪ wewnętrznej
    System.out.println(i + " " + j);
  }
}
```

Instrukcja continue

Instrukcja continue powoduje przerwanie bieżącej iteracji pętli i przejście do kolejnej iteracji.

Przykład (wyświetlenie parzystych liczb z zakresu 1 – 20):

```
for(int i = 1; i <= 20; i++){
  if(i % 2 != 0) continue;
  System.out.println(i);
}
```

TABLICE

Deklaracje tablic

Tablice w Javie są obiektami. Aby móc skorzystać z tablicy, należy najpierw zadeklarować zmienną tablicową, a następnie utworzyć samą tablicę (obiekt tablicy). Schematycznie deklaracja taka jest następująca:

```
typ_tablicy nazwa_tablicy[];
```

lub (co ma identyczne znaczenie):

```
typ_tablicy[] nazwa_tablicy;
```

Tablicę tworzy się za pomocą operatora new o postaci:

```
new typ_tablicy[liczba_elementów];
```

Tablicę można również jednocześnie zadeklarować i utworzyć, korzystając z konstrukcji:

```
typ_tablicy nazwa_tablicy[] = new typ_
↪tablicy[liczba_elementów];
```

lub:

```
typ_tablicy[] nazwa_tablicy = new typ_
↪tablicy[liczba_elementów];
```

Przykład:

```
class Main
{
  public static void main (String args[])
  {
    int tab[] = new int[1];
    tab[0] = 10;
    System.out.println("Pierwszy
↪ element tablicy ma wartość: " +
↪ tab[0]);
  }
}
```

Inicjalizacja tablicy

W przypadku niewielkich tablic można dokonać inicjalizacji tablicy, ujmując wartości, które mają się znaleźć w jej komórkach, w nawias klamrowy. Nie trzeba wtedy korzystać z operatora new. Schematycznie deklaracja taka jest następująca:

```
typ_tablicy nazwa_tablicy[] = {wartość1,
↪wartość2, ... , wartośćN};
```

Na przykład deklarację sześcioelementowej tablicy liczb całkowitych typu int i wypełnienie jej kolejnych komórek wartościami od 1 do 6 można wykonać za pomocą instrukcji:

```
int tablica[] = {1, 2, 3, 4, 5, 6};
```

Właściwość length

Każda tablica ma właściwość length, która określa liczbę jej komórek.

Przykład:

```
int tablica[] = new int[10];
for(int i = 0; i < tablica.length;
↪ i++){
  tablica[i] = i;
}
```

Tablice wielowymiarowe

Deklaracja regularnych tablic wielowymiarowych odbywa się w sposób analogiczny do deklaracji tablic jednowymiarowych, dodawane są jedynie kolejne nawiasy kwadratowe. Schematyczna deklaracja tablicy dwuwymiarowej ma postać:

```
typ_tablicy nazwa_tablicy[][];
```

Tablica tego typu może zostać zainicjowana przy użyciu składni z nawiasami klamrowymi:

```
typ_tablicy nazwa_tablicy[][] = {
  {wartość1, wartość2, ... , wartośćN},
  {wartość1, wartość2, ... , wartośćN}
}
```

Przykład (dwuwymiarowa tablica liczb całkowitych wypełniona danymi):

```
int tab[][] = {
  {1, 2, 3, 4},
  {5, 6, 7, 8}
};
```

Do jej utworzenia można użyć także standardowego operatora new w postaci:

```
new typ_tablicy[liczba_elementów]
↪[liczba_elementów];
```

Przykład:

```
int tab[][] = new int[2][4];
int count = 1;
for(int i = 0; i < 2; i++){
  for(int j = 0; j < 4; j++){
    tab[i][j] = count++;
  }
}
```

KLASY I OBIEKTY

Budowa klasy

Definicja klasy ma schematyczną postać:

```
class nazwa_klasy
{
  //treść klasy
  //definicje pól i metod
}
```

Przykład:

```
class Punkt
{
  int x;
  int y;
}
```

Tworzenie obiektów

Zmienna typu obiektowego (odnośnikowego, referencyjnego) tworzona jest za pomocą konstrukcji:

```
nazwa_klasy nazwa_zmiennej;
```

Do tak zadeklarowanej zmiennej można przypisać obiekt utworzony za pomocą operatora new:

```
new nazwa_klasy();
```

Jednoczesna deklaracja zmiennej, utworzenie obiektu i przypisanie go zmiennej odbywa się za pomocą schematycznej konstrukcji:

```
nazwa_klasy nazwa_zmiennej = new
↪nazwa_klasy();
```

Przykład:

```
Punkt mojPunkt = new Punkt();
```

Pola klas

Definicje pól

Pola definiowane są w ciele klasy, podobnie jak zwykłe zmienne. Najpierw należy podać typ pola, a po nim — nazwę pola:

```
class nazwa_klasy
{
  typ_pola1 nazwa_pola1;
  typ_pola2 nazwa_pola2;
  //...
  typ_polaN nazwa_polaN;
}
```

Odwołania do pól obiektu

Po utworzeniu obiektu do jego pól można odwoływać się za pomocą operatora kropki (.); schematycznie:

```
nazwa_obiektu.nazwa_pola;
```

Przykład:

```
Punkt punkt1 = new Punkt();
punkt1.x = 100;
punkt1.y = 200;
```

Wartości domyślne pól

Każde niezainicjowane pole klasy otrzymuje wartość domyślną, zależną od jego typu. Wartości te zaprezentowane są w poniższej tabeli.

Typ	Wartość domyślna
byte	0
short	0
int	0
long	0
float	0.0
double	0.0
char	\0
boolean	false
obiektowy	null

Metody klas

Definicje metod

Metody definiowane są w ciele klasy pomiędzy znakami nawiasu klamrowego. Każda metoda może przyjmować argumenty oraz zwracać wynik. Schematyczna deklaracja metody ma postać:

```
typ_wyniku nazwa_metody(argumenty_metody)
{
  //instrukcje metody
}
```

Po umieszczeniu w ciele klasy deklaracja taka będzie wyglądała następująco:

```
class nazwa_klasy
{
  typ_wyniku nazwa_metody(argumenty_
↪metody)
  {
    //instrukcje metody
  }
}
```

Jeśli metoda nie zwraca żadnego wyniku, jako typ wyniku należy zastosować słowo void, jeśli natomiast nie przyjmuje żadnych argumentów, pomiędzy znakami nawiasu okrągłego nie należy nic wpisywać.

Przykład:

```
class Punkt
{
  int x;
  int y;
  void wyswietlWspolrzedne()
  {
    System.out.println("współrzędna x
↪= " + x);
    System.out.println("współrzędna y
↪= " + y);
  }
}
```

Odwołania do metod obiektu

Po utworzeniu obiektu do jego metod można odwoływać się (podobnie jak w przypadku pól) za pomocą operatora kropki (.); schematycznie:

```
nazwa_obiektu.nazwa_metody();
```

Przykład:

```
Punkt punkt1 = new Punkt();
punkt1.wyswietlWspolrzedne();
```

Argumenty metod

Metoda może mieć dowolną liczbę argumentów umieszczonych w nawiasie okrągłym za jej nazwą. Poszczególne argumenty należy oddzielić od siebie znakami przecinka; schematycznie:

```
typ_wyniku nazwa_metody(typ_argumentu_1
↪nazwa_ argumentu_1, typ_argumentu_2
↪nazwa_argumentu_2, ... , typ_argumentu
↪ _N nazwa_argumentu_N)
```

Przykład:

```
void ustawXY(int wspX, int wspY)
{
  x = wspX;
  y = wspY;
}
```

Argumentami mogą być typy zarówno proste, jak i obiektowe.

Przeciążanie metod

W każdej klasie może istnieć dowolna liczba metod o takiej samej nazwie, jeśli tylko różnią się przyjmowanymi argumentami. Mogą, ale nie muszą się one różnić również typem zwra-

canego wyniku. Technika ta jest nazywana przeciążaniem (z ang. *overloading*) metod. **Przykład:**

```
public class Punkt
{
  int x;
  int y;
  void ustawXY(int wspX, int wspY)
  {
    x = wspX;
    y = wspY;
  }
  void ustawXY(Punkt punkt)
  {
    x = punkt.x;
    y = punkt.y;
  }
}
```

Rodzaje klas

Klasy w Javie można ogólnie podzielić na:

- pakietowe,
- publiczne,
- wewnętrzne,
- interfejsowe (patrz sekcja „Interfejsy").

Jeśli przed nazwą klasy nie znajduje się modyfikator `public`, to jest to klasa pakietowa, czyli klasa dostępna jedynie dla innych klas z tego samego pakietu. Jeśli natomiast przed nazwą klasy znajduje się modyfikator `public`, to jest to klasa publiczna, czyli klasa dostępna dla wszystkich innych klas.
Deklaracja klasy pakietowej ma postać:

```
class nazwa_klasy
{
  //pola i metody klasy
}
```

Deklaracja klasy publicznej ma postać:

```
public class nazwa_klasy
{
  //pola i metody klasy
}
```

Klasy wewnętrzne to klasy zdefiniowane wewnątrz innych klas. Postać schematyczna:

```
[specyfikator dostępu] class klasa_
↪zewnętrzna
{
  [specyfikator dostępu] class klasa_
↪wewnętrzna
  {
    //pola i metody klasy wewnętrznej
  }
  //pola i metody klasy zewnętrznej
}
```

Przykład klasy wewnętrznej (`Inside`):

```
public class Outside
{
  class Inside
  {
  }
}
```

Ponieważ klasy wewnętrzne są definiowane wewnątrz innych klas, w pewnym sensie można je traktować jako składowe tych klas, a zatem można też w stosunku do nich stosować modyfikatory dostępu. W związku z tym klasy wewnętrzne mogą być:

- pakietowe,
- publiczne,
- prywatne,
- chronione

i należy je traktować tak jak składowe wymienionych typów.

Konstruktory

Definicja konstruktora

Konstruktor to specjalna metoda, która jest wywoływana zawsze po utworzeniu obiektu w pamięci. Metoda będąca konstruktorem nigdy nie zwraca wyniku i musi mieć nazwę zgodną z nawą klasy; schematycznie:

```
public class nazwa_klasy
{
  nazwa_klasy()
  {
    //kod konstruktora
  }
}
```

Przykład:

```
public class Punkt
{
  int x;
  int y;
  Punkt()
  {
    x = 1;
    y = 1;
  }
}
```

Argumenty konstruktorów

Konstruktor może być bezargumentowy albo może przyjmować argumenty używane (bezpośrednio lub pośrednio) np. do zainicjowania pól obiektu. Argumenty przekazuje się dokładnie w taki sam sposób, jak w przypadku zwykłych metod; schematycznie:

```
class nazwa_klasy
{
  nazwa_klasy(typ1 argument1, typ2
↪argument2,..., typN argumentN)
  {
    //kod konstruktora
  }
}
```

Przykład:

```
public class Punkt
{
  int x;
  int y;
  Punkt(int wspX, int wspY)
  {
    x = wspX;
    y = wspY;
  }
}
```

Wywołanie konstruktora

Konstruktor bezargumentowy (czyli konstruktor domyślny) jest wywoływany automatycznie w trakcie tworzenia obiektu danej klasy, czyli po wykonaniu instrukcji:

```
new nazwa_klasy();
```

Jeżeli konstruktor wymaga podania argumentów, należy umieścić je w nawiasie okrągłym, tak jak w przypadku zwykłej metody:

```
new nazwa_klasy(argumenty_konstruktora);
```

Przykład:

```
Punkt punkt1 = new Punkt(100, 200);
```

Przeciążanie konstruktorów

Konstruktory, tak jak zwykłe metody, mogą być przeciążane, tzn. każda klasa może mieć kilka konstruktorów, jeśli tylko różnią się one przyjmowanymi argumentami. **Przykład:**

```
class Punkt
{
  int x;
  int y;
  Punkt()
  {
    x = 1;
    y = 1;
  }
  Punkt(int wspX, int wspY)
  {
    x = wspX;
    y = wspY;
  }
  Punkt(Punkt punkt)
  {
    x = punkt.x;
    y = punkt.y;
  }
}
```

Konstruktor domyślny

Jeżeli w klasie nie będzie zdefiniowany żaden konstruktor, zostanie do niej automatycznie dodany bezargumentowy konstruktor domyślny. Klasa taka będzie się zatem zachowywała tak, jakby miała schematyczną postać:

```
public class nazwa_klasy
{
  nazwa_klasy()
  {
  }
  //pola i metody klasy
}
```

Jeżeli w klasie zostanie jawnie zdefiniowany konstruktor bezargumentowy, automatycznie stanie się on konstruktorem domyślnym (jest to ważne przy dziedziczeniu), niezależnie od tego, czy istnieją inne konstruktory.

Słowo kluczowe this

Słowo kluczowe `this` to odwołanie do obiektu bieżącego. Można je traktować jako referencję do aktualnego obiektu. Odwołanie do pól i metod przez wskazanie `this` odbywa się za pomocą operatora kropki (.):

```
this.nazwa_pola = wartość;
this.nazwa_metody(argumenty);
```

Umożliwia to m.in. stosowanie w metodach i konstruktorach argumentów o nazwach identycznych z nazwami pól klasy.
Przykład:

```
public class Punkt
{
  int x;
  int y;
  Punkt(int x, int y)
  {
    this.x = x;
    this.y = y;
  }
}
```

Składowe statyczne

Składowe statyczne są to pola i metody klasy, które mogą istnieć, nawet jeśli nie istnieje obiekt tej klasy. Każda taka metoda lub takie pole są wspólne dla wszystkich obiektów tej klasy. Składowe te oznaczane są słowem `static`.

Metody statyczne

Metody statyczne oznacza się słowem `static`, które zwyczajowo powinno znaleźć się zaraz za modyfikatorem dostępu; schematycznie:

```
modyfikator_dostępu static typ_zwracany
↪nazwa_metody(argumenty)
{
  //treść metody
}
```

Przykład:

```
public class A
{
  public static void f()
  {
    System.out.println("Metoda f klasy
↪A");
  }
}
```

Tak napisaną metodę można wywołać klasycznie, tzn. po utworzeniu obiektu klasy `A`, np. w postaci:

```
A a = new A();
a.f();
```

Ponieważ jednak metody statyczne istnieją nawet wtedy, kiedy nie ma żadnego obiektu danej klasy, możliwe jest wywołanie w postaci:

```
A.f();
```

Ogólniej metodę statyczną można wywołać w postaci:

```
nazwa_klasy.nazwa_metody(argumenty_
↪metody);
```

bez konieczności tworzenia obiektu danej klasy.

Pola statyczne

Pola statyczne są deklarowane przez umieszczenie słowa `static` przed typem pola; schematycznie:

```
static typ_pola nazwa_pola;
```

lub

```
modyfikator_dostępu static typ_pola
↪nazwa_pola;
```

Przykład:

```
public class A
{
  public static int liczba;
}
```

Do pól statycznych można odwoływać się tak, jak do innych pól klasy, poprzedzając je nazwą obiektu, czyli stosując konstrukcję:

```
nazwa_obiektu.nazwa_pola
```

bądź też poprzedzając je nazwą klasy:

```
nazwa_klasy.nazwa_pola
```

Przykład:

```
A.liczba = 100;
```

Dziedziczenie

Klasy potomne

W Javie dziedziczenie wyraża się za pomocą słowa `extends`. Schematyczna konstrukcja jest następująca:

```
class klasa_potomna extends klasa_bazowa
{
  //treść klasy potomnej
}
```

Zapis taki oznacza, że klasa potomna dziedziczy z klasy bazowej.
Przykład:

```
class Punkt
{
  int x;
  int y;
}

class Punkt3D extends Punkt
{
  int z;
}
```

Konstruktory w klasach potomnych

Podczas tworzenia obiektu klasy potomnej zawsze wywoływany jest domyślny, bezargumentowy konstruktor klasy bazowej. Jeżeli w klasie bazowej nie istnieje konstruktor domyślny, wymagane jest jawne wywołanie jednego z pozostałych konstruktorów. Wywołanie to wymaga zastosowania składni ze słowem kluczowym `super`. Słowo to oznacza w tym przypadku wywołanie konstruktora klasy bazowej. Schematyczna konstrukcja jest następująca:

```
public
class klasa_potomna extends klasa_bazowa
{
  klasa_potomna()
  {
    super(argumenty);
    /*
    …dalszy kod konstruktora…
    */
  }
}
```

Jeśli metodzie `super` przekaże się argumenty, zostanie wywołany konstruktor klasy bazowej, który tym argumentom odpowiada. Ważne jest, aby metoda `super` była pierwszą instrukcją konstruktora klasy potomnej.
Przykład:

```
class Punkt
{
  int x;
  int y;
  Punkt(int x, int y)
  {
    this.x = x;
    this.y = y;
  }
}

class Punkt3D extends Punkt
{
  int z;
  Punkt3D(int x, int y, int z)
  {
    super(x, y);
    this.z = z;
  }
}
```

Modyfikatory dostępu

Przed każdym polem i każdą metodą może wystąpić modyfikator (inaczej specyfikator) dostępu, określający prawa dostępu do składowych klasy. Wyróżnia się cztery rodzaje dostępu:

- publiczny,
- prywatny,
- chroniony,
- pakietowy.

Domyślnie, jeżeli przed składową klasy nie występuje żadne określenie, dostęp jest pakietowy, co oznacza, że dostęp do tej składowej mają wyłącznie klasy pakietu, w którym się ona znajduje. Dostęp publiczny jest określany słowem `public`, dostęp prywatny — słowem `private`, a chroniony — słowem `protected`.

Dostęp publiczny

Jeżeli dana składowa klasy jest publiczna, oznacza to, że mają do niej dostęp wszystkie inne klasy, czyli dostęp nie jest w żaden sposób ograniczony. Modyfikator dostępu `public` należy zatem umieścić przed nazwą typu, co schematycznie wygląda tak:

```
public nazwa_typu nazwa_zmiennej;
```

Podobnie jest z metodami — modyfikator dostępu powinien być pierwszym elementem deklaracji:

```
public typ_zwracany nazwa_metody
↪(argumenty)
```

Przykład:

```
class Punkt
{
  public int x;
  public int y;
  public int pobierzX()
  {
    return x;
  }
  public int pobierzY()
  {
    return y;
  }
}
```

Dostęp prywatny

Składowe oznaczone słowem `private` to takie, które dostępne są jedynie z wnętrza danej klasy, tzn. wszystkie metody danej klasy mogą je dowolnie odczytywać, zapisywać (pola) i wywoływać (metody), natomiast żadna inna klasa nie może ich ani odczytać, ani zapisać. Modyfikator dostępu `private` należy umieścić przed nazwą typu; schematycznie:

```
private nazwa_typu nazwa_zmiennej;
```

Podobnie jest z metodami — specyfikator dostępu powinien być pierwszym elementem deklaracji:

```
private typ_zwracany nazwa_metody
↪(argumenty)
```

Przykład:

```
class Punkt
{
  private int x;
  private int y;
  public int pobierzX()
  {
    return x;
  }
  public int pobierzY()
  {
    return y;
  }
}
```

Dostęp chroniony

Składowe klasy, oznaczone słowem `protected`, to składowe chronione. Są one dostępne jedynie dla metod danej klasy, klas potomnych oraz klas z tego samego pakietu. Specyfikator dostępu `protected` należy umieścić przed nazwą typu, co schematycznie wygląda tak:

```
protected nazwa_typu nazwa_zmiennej;
```

Podobnie jest z metodami — specyfikator dostępu powinien być pierwszym elementem deklaracji:

```
protected typ_zwracany nazwa_metody
↪(argumenty)
```

Przykład:

```
class Punkt
{
  protected int x;
  protected int y;
  protected int pobierzX()
  {
    return x;
  }
  protected int pobierzY()
  {
    return y;
  }
}
```

Dostęp pakietowy

Dostęp pakietowy jest dostępem domyślnym, stosowanym, kiedy przed składową klasy nie występuje żaden modyfikator dostępu. Konstrukcja taka oznacza, że dostęp do składowej mają wszystkie klasy pakietu, w którym się ona znajduje.
Przykład:

```
class Punkt
{
  int x;
  int y;
  int pobierzX()
  {
    return x;
  }
  int pobierzY()
  {
    return y;
  }
}
```

Klasy i składowe finalne

Klasy finalne

Klasa finalna to taka, z której nie wolno wyprowadzać innych klas, innymi słowy: taka, z której nie mogą dziedziczyć inne klasy. Pozwala to tworzyć klasy, których postać będzie z góry ustalona. Jeśli klasa ma stać się klasą finalną, należy przed jej nazwą umieścić słowo kluczowe `final`, zgodnie ze schematem:

```
modyfikator_dostępu final class
↪nazwa_klasy
{
  //pola i metody klasy
}
```

Przykład:

```
public final class Example
{
  public int liczba;
  public void wyswietl()
  {
    System.out.println(liczba);
  }
}
```

Pola finalne

Pole klasy oznaczone słowem `final` staje się polem finalnym, czyli takim, którego wartość jest stała i nie można jej zmieniać. Słowo kluczowe `final` umieszcza się zwyczajowo przed nazwą typu danego pola:

```
final typ_pola nazwa_pola;
```

lub ogólniej:

```
modyfikator_dostępu [static] final typ_
↪pola nazwa_pola;
```

Poprawne są wszystkie poniższe deklaracje:

```
final int liczba;
public final double liczba;
public static final char znak;
```

Po takiej deklaracji pierwsze przypisanie wartości ustala niezmienną wartość pola.
Deklaracja może też być połączona z inicjalizacją, np.:

```
public final double liczba = 2.14;
```

Jeśli pole finalne jest typu referencyjnego, np.:

```
final Punkt punkt1 = new Punkt();
```

oznacza to, że nie można zmieniać pierwotnie przypisanej referencji. Nic jednak nie stoi na przeszkodzie, aby modyfikować pola obiektu wskazywanego przez tę referencję, np.:

```
punkt1.x = 100;
```

Metody finalne

Metoda oznaczona słowem `final` staje się metodą finalną, co oznacza, że nie będzie możliwe jej przesłonięcie w klasie potomnej. Słowo `final` umieszczane jest przed typem wartości zwracanej przez metodę:

```
final typ_zwracany nazwa_metody
↪(argumenty)
```

lub ogólniej:

```
modyfikator_dostępu [static] final typ_
↪zwracany nazwa_metody(argumenty)
```

Prawidłowe są następujące przykładowe deklaracje:

```
final void metoda(){/*kod metody*/};
public final int metoda(){/*kod
↪metody*/};
public static final void metoda()
↪{/*kod metody*/};
```

```
public static final int metoda(int
↪argument){/*kod metody*/};
```

Argumenty finalne

Argument finalny to taki, którego nie wolno zmieniać w ciele metody. Aby uczynić argument finalnym, należy umieścić słowo `final` przed jego typem. Schematycznie:

```
modyfikator_dostępu [static][final]
typ_zwracany nazwa_metody(final typ_
↪argumentu nazwa_argumentu)
```

Na przykład deklaracja publicznej metody o nazwie `metoda1`, która nie zwraca żadnej wartości, lecz przyjmuje jeden finalny argument typu `int` o nazwie `argument1`, będzie miała postać:

```
public void metoda1(final int argument1)
{
  /*treść metody*/
}
```

Klasy i metody abstrakcyjne

Klasy i metody abstrakcyjne deklaruje się za pomocą słowa kluczowego `abstract`. Jeżeli w klasie znajduje się co najmniej jedna metoda abstrakcyjna, klasa taka musi być również zadeklarowana jako abstrakcyjna (nie wyklucza to istnienia klas abstrakcyjnych, w których żadna z metod nie jest abstrakcyjna):

```
[public] abstract class nazwa_klasy {
  [specyfikator_dostępu] abstract typ_
↪zwracany nazwa_metody(argumenty);
}
```

Metoda abstrakcyjna ma jedynie definicję, nie może zawierać żadnego kodu. Przykładowa publiczna i abstrakcyjna klasa `Shape` zawierająca abstrakcyjną metodę `draw` będzie miała postać:

```
public abstract class Shape {
  public abstract void draw();
}
```

Po takiej deklaracji nie można będzie tworzyć obiektów klasy `Shape`.

Zadeklarowanie metody jako abstrakcyjnej wymusza jej redeklarację w klasie potomnej.

INTERFEJSY

Tworzenie interfejsu

Interfejs to klasa czysto abstrakcyjna, czyli taka, w której wszystkie metody traktuje się jako abstrakcyjne. Interfejs deklaruje się za pomocą słowa kluczowego `interface`. Interfejs może być publiczny, jeśli jest zdefiniowany w pliku o takiej samej nazwie jak nazwa interfejsu, lub pakietowy (dostępny jedynie dla klas wchodzących w skład danego pakietu). Schematyczna konstrukcja interfejsu jest następująca:

```
[public] interface nazwa_interfejsu {
  typ_zwracany nazwa_metody1(argumenty);
  typ_zwracany nazwa_metody2(argumenty);
  /*...dalsze metody interfejsu...*/
  typ_zwracany nazwa_metodyN(argumenty);
}
```

Przykładowy interfejs o nazwie `Drawable` zawierający deklarację jednej tylko metody o nazwie `draw` będzie miał postać:

```
public interface Drawable {
  public void draw();
}
```

Implementowanie interfejsu

To, że klasa ma implementować dany interfejs, zaznacza się, wykorzystując słowo kluczowe `implements`; schematycznie:

```
[specyfikator_dostępu][abstract]
class nazwa_klasy implements nazwa_
↪interfejsu {
  /*
  …pola i metody klasy…
  */
}
```

Jeśli więc przykładowa klasa `Shape` ma implementować przedstawiony wyżej interfejs `Drawable`, powinna mieć postać:

```
public class Shape implements Drawable {
  public void draw() {
    /* wnętrze metody draw */
  }
}
```

Pola interfejsu

Pola interfejsu są jednocześnie publiczne, statyczne i finalne — trzeba im przypisać wartości już w momencie ich deklaracji. Deklaracja pola interfejsu nie różni się od deklaracji pola klasy. Zgodnie z konwencją przyjmuje się, że nazwy takich pól pisze się wielkimi literami, a poszczególne człony nazwy oddziela się znakiem podkreślenia, np.:

```
public interface NowyInterfejs {
  int POLE_TYPU_INT = 100;
  double POLE_TYPU_DOUBLE = 1.0;
  Object POLE_TYPU_OBJECT = new Object();
}
```

TYPY UOGÓLNIONE

Uogólnianie w klasach

Jeśli w klasie ma być użyty typ uogólniony (z ang. *generic type*), należy to zaznaczyć w jej definicji za pomocą nawiasu kątowego umieszczonego za nazwą klasy. W nawiasie trzeba umieścić identyfikatory typów, które zostaną zastosowane, oddzielając je od siebie znakami przecinka; schematycznie:

```
[public] class nazwa_klasy<id1, id2,
↪…, idN>
{
  //treść klasy
}
```

Każdy z identyfikatorów może być następnie użyty we wnętrzu klasy jako określenie konkretnego typu danych. Zwyczajowo stosuje się identyfikatory jednoliterowe (rozpoczynając od litery T), jednak nie jest to ograniczenie formalne (identyfikator typu może być wielonakowy). Przykład klasy przechowującej jedną wartość dowolnego typu:

```
public class Opakowanie1<T>{
  public T val;
}
```

Przykład klasy przechowującej dwie wartości dwóch dowolnych typów:

```
public class Opakowanie2<T, V>{
  public T val1;
  public V val2;
}
```

Przy deklaracji zmiennych klasy korzystającej z typów uogólnionych należy podać w nawiasie kątowym określenia konkretnych typów; schematycznie:

```
nazwa_klasy<id1, id2, …, idN> nazwa_
↪zmiennej;
```

Przykład:

```
Opakowanie1<Integer> op1;
Opakowanie2<Integer, String> op2;
```

Analogicznie należy postąpić przy tworzeniu obiektów; schematycznie:

```
new nazwa_klasy<id1, id2, …, idN>();
```

Przykład:

```
Opakowanie1<Integer> op1 = new
↪Opakowanie1<Integer>();
Opakowanie2<Integer, String> op2 = new
↪Opakowanie2<Integer, String>();
```

Po takich definicjach możliwe będą m.in. następujące przypisania:

```
op1.val = 10;
op1.val = new Integer(100);
op2.val1 = 20;
op2.val2 = "abcxyz";
```

Uogólnianie metod

Uogólnianie metod jest niezależne od uogólnień klas, więc w klasie uogólnionej mogą się znajdować nieuogólnione metody, a uogólnione metody mogą się znajdować w zwykłych klasach. Jeśli metoda ma operować na argumencie typu ogólnego, to specyfikację tego typu należy umieścić przed typem zwracanym przez metodę; ogólnie:

```
[modyfikator_dostępu] <id1, id2,…,idN>
↪typ_zwracany nazwa_metody(argumenty)
{
  //treść metody
}
```

Przykład:

```
public class Main
{
  public static<U> void show(U val)
  {
    System.out.println(val.toString());
  }
  public static void main (String args[])
  {
    show(new Object());
    show(new Integer(100));
  }
}
```

PAKIETY

Tworzenie pakietów

Klasy w Javie grupowane są w jednostki nazywane pakietami. Pakiet to inaczej biblioteka, zestaw powiązanych tematycznie klas. Do tworzenia pakietów służy słowo kluczowe `package`, po którym następuje nazwa pakietu, zakończona znakiem średnika; schematycznie:

```
package nazwa_pakietu;
```

Instrukcja ta musi znajdować się na początku pliku, przed nią nie może być żadnych innych instrukcji. Przed `package` mogą występować jedynie komentarze:

```
//pakiet i klasa pakietowa
package nazwa_pakietu;

class nazwa_klasy
{
  /*
    treść klasy
  */
}
```

Aby skorzystać z klasy zawartej w pakiecie w innej klasie, należy użyć dyrektywy `import` w postaci:

```
import nazwa_pakietu.nazwa_klasy;
```

Dyrektywa `import` musi znajdować się na początku pliku. Aby zaimportować wszystkie klasy z danego pakietu, dyrektywa `import` powinna mieć postać:

```
import nazwa_pakietu.*;
```

Przykład:

```
import java.io.*;
```

Nazwy pakietów

Nazwy pakietów powinny być pisane w całości małymi literami, a jeśli pakiet ma być udostępniony publicznie, należy poprzedzić go odwróconą nazwą domeny twórcy pakietu. Nie jest to obligatoryjne, ale pozwala na utworzenie, z dużym prawdopodobieństwem, nazwy unikatowej w skali globu. Jeżeli np. domeną autora jest *marcinlis.com* i ma powstać pakiet o nazwie *grafika*, jego pełna nazwa będzie brzmieć: *com.marcinlis.grafika*. Wszystkie klasy tego pakietu będą musiały zostać umieszczone w strukturze katalogów odpowiadających tej nazwie.

WYJĄTKI

Instrukcja try...catch

Do przechwytywania wyjątków służy blok instrukcji `try...catch` o schematycznej, podstawowej postaci:

```
try{
  //instrukcje mogące spowodować wyjątek
}
catch(TypWyjątku identyfikatorWyjątku){
  //obsługa wyjątku
}
```

W nawiasie klamrowym, występującym po słowie `try`, umieszcza się instrukcję (instrukcje), która może spowodować wystąpienie błędu. W bloku występującym po `catch` należy umieścić kod, który ma zostać wykonany, kiedy wystąpi wyjątek.

Przykład:

```
public static void main (String args[])
{
  int tab[] = new int[10];
  try{
    //przekroczenie indeksu tablicy
    tab[10] = 100;
  }
  catch(ArrayIndexOutOfBoundsException e){
    //przechwycenie wyjątku
    System.out.println("Nieprawidłowy
↪indeks tablicy!");
  }
}
```

Hierarchia wyjątków

Każdy wyjątek jest obiektem pewnej klasy. Klasy podlegają z kolei regułom dziedziczenia, zgodnie z którymi powstaje hierarchia klas. Wszystkie typowe wyjątki, które można standardowo przechwytywać w aplikacjach za pomocą bloku `try...catch`, dziedziczą (bezpośrednio lub pośrednio) z klasy `Exception`, dziedziczącej z klas `Throwable` oraz `Object`. Wynika z tego ważna właściwość: jeżeli dana instrukcja może wygenerować wyjątek typu X, to można zawsze przechwycić wyjątek ogólniejszy, czyli wyjątek, którego typem będzie jedna z klas nadrzędnych w stosunku do X.

Przechwytywanie wielu wyjątków

W jednym bloku `try...catch` można przechwytywać wiele wyjątków. Konstrukcja taka zawiera wtedy jeden blok `try` i wiele bloków `catch`:

```
try{
  //instrukcje mogące spowodować
↪wyjątek
}
catch(KlasaWyjątku1 identyfikator
↪Wyjątku1){
  //obsługa wyjątku 1
}
catch(KlasaWyjątku2 identyfikator
↪Wyjątku2){
  //obsługa wyjątku 2
}
/*
…dalsze bloki catch…
*/
catch(KlasaWyjątkuN identyfikator
↪WyjątkuN){
  //obsługa wyjątku n
}
```

Po wygenerowaniu wyjątku maszyna wirtualna sprawdza, czy jego typem jest `KlasaWyjątku1` — jeśli tak, to wykonywane są instrukcje obsługi tego wyjątku i blok `try...catch` jest opuszczany. Jeżeli jednak typem wyjątku nie jest `KlasaWyjątku1`, wtedy sprawdza się, czy jest on typu `KlasaWyjątku2` itd.

Przy przechwytywaniu wielu wyjątków w jednym bloku należy pamiętać o ich hierarchii. Ogólna zasada jest taka: nie ma znaczenia kolejność, jeżeli wszystkie wyjątki są na jednym poziomie hierarchii. Jeśli jednak przechwytywane są wyjątki z różnych poziomów, najpierw muszą to być wyjątki bardziej szczegółowe, czyli stojące niżej w hierarchii, a dopiero po nich wyjątki bardziej ogólne, czyli stojące wyżej w hierarchii.

Przykład:

```
public static void main (String args[])
{
  try{
    int liczba = 10 / 0;
  }
  catch(ArithmeticException e){
    System.out.println(e);
  }
  catch(RuntimeException e){
    System.out.println(e);
  }
  catch(Exception e){
    System.out.println(e);
  }
}
```

Zagnieżdżanie bloków try...catch

Bloki `try...catch` można zagnieżdżać. W jednym bloku przechwytującym wyjątek X może istnieć drugi blok, który będzie przechwytywał wyjątek Y. Schematycznie taka konstrukcja ma postać:

```
try{
  //instrukcje mogące spowodować
↪wyjątek 1
  try{
    //instrukcje mogące spowodować
↪wyjątek 2
  }
  catch (TypWyjątku2 identyfikator
↪Wyjątku2){
    //obsługa wyjątku 2
  }
}
catch (TypWyjątku1 identyfikator
↪Wyjątku1){
  //obsługa wyjątku 1
}
```

Przykład:

```
public static void main (String args[])
{
  Punkt punkt = null;
  int liczba;
  try{
    try{
      liczba = 10 / 0;
    }
    catch(ArithmeticException e){
      System.out.println("Nieprawidłowa
↪operacja arytmetyczna.");
      System.out.println("Przypisuję
↪zmiennej liczba wartość 10.");
      liczba = 10;
    }
    punkt.x = liczba;
  }
  catch(Exception e){
    System.out.println("Błąd ogólny.");
    System.out.println(e);
  }
}
```

Zgłaszanie wyjątków

Zgłoszenie własnego wyjątku polega na utworzeniu nowego obiektu jednej z klas wyjątków. Za pomocą instrukcji `new` należy utworzyć nowy obiekt klasy, która dziedziczy (pośrednio lub bezpośrednio) z klasy `Throwable`. W najbardziej ogólnym przypadku będzie to klasa `Exception`. Tak utworzony obiekt musi stać się parametrem instrukcji `throw`, np.:

```
throw new Exception();
```

Jeśli taki wyjątek zostanie obsłużony przez znajdującą się w danym bloku (danej metodzie) instrukcję `try...catch`, nie trzeba robić nic więcej. Jeśli jednak nie zostanie obsłużony, w specyfikacji metody należy zaznaczyć, że może ona taki wyjątek zgłaszać. Wymaga to zastosowania instrukcji `throws` w ogólnej postaci:

```
specyfikator_dostępu [static] [final]
↪typ_zwracany nazwa_metody(argumenty)
throws KlasaWyjątku1, KlasaWyjątku2,
↪..., KlasaWyjątkuN
{
  //treść metody
}
```

Przykład:

```
public static void main (String args[])
throws Exception
{
  throw new Exception();
}
```

Jeżeli zgłaszany wyjątek ma otrzymać własny komunikat, należy przekazać go jako argument konstruktora klasy `Exception`:

```
throw new Exception("komunikat");
```

lub:

```
Exception exception = new
↪Exception("komunikat");
throw exception;
```

Ponowne zgłaszanie wyjątków

Raz przechwycony wyjątek można zgłosić ponownie („wyrzucić"), wykorzystując instrukcję `throw`:

```
try{
  //instrukcje mogące spowodować
↪wyjątek
}
catch(typWyjątku identyfikatorWyjątku){
  //instrukcje obsługujące sytuację
↪wyjątkową
  throw indentyfikatorWyjątku;
}
```

Przykład:

```
public static void main (String args[])
{
  try{
    int liczba = 10 / 0;
  }
  catch(ArithmeticException e){
    System.out.println("Tu wyjątek
↪został przechwycony.");
    //ponowne zgłoszenie wyjątku
    throw e;
  }
}
```

Tworzenie wyjątków

W Javie można tworzyć własne klasy wyjątków. Należy napisać klasę pochodną, dziedziczącą pośrednio lub bezpośrednio z klasy `Throwable`. W praktyce wyjątki najczęściej są wyprowadzane z klasy `Exception` i klas od niej pochodnych:

```
public class nazwa_klasy extends Exception
{
  //treść klasy
}
```

Przykład (nowy wyjątek `GeneralException`):

```
public class GeneralException extends
↪Exception
{
}
```

Zgłoszenie wyjątku `GeneralException`:

```
public static void main (String args[])
throws GeneralException
{
  throw new GeneralException();
}
```

Sekcja finally

Do bloku `try` możemy dołączyć sekcję `finally`, która będzie wykonywana zawsze, niezależnie od tego, co będzie działo się w bloku `try`; schematycznie:

```
try{
  //instrukcje mogące spowodować wyjątek
}
catch(){
  //instrukcje sekcji catch
}
finally{
  //instrukcje sekcji finally
}
```

Sekcję `finally` można stosować w przypadku dowolnych instrukcji, nie ma też konieczności przechwytywania wyjątku. Stosowana jest wtedy konstrukcja `try...finally` w postaci:

```
try{
  //instrukcje
}
finally{
  //instrukcje
}
```

Kod z bloku `finally` zostanie wykonany zawsze, niezależnie od tego, jakie instrukcje znajdują się w bloku `try`.

WSPÓŁPRACA Z SYSTEMEM

Standardowy strumień wejściowy

Standardowy strumień wejściowy jest reprezentowany przez obiekt `System.in`, czyli obiekt `in` zawarty w klasie `System` (statyczne i finalne pole klasy). Jest to obiekt typu `InputStream` (klasy reprezentującej strumienie wejściowe). Metody udostępniane przez tę klasę zostały zebrane w poniższej tabeli.

Typ zwracany	Metoda	Opis
`int`	`available()`	Zwraca liczbę bajtów, które mogą być odczytane ze strumienia
`void`	`close()`	Zamyka strumień i zwalnia związane z nim zasoby
`void`	`mark(int readlimit)`	Zaznacza bieżącą pozycję w strumieniu
`boolean`	`markSupported()`	Sprawdza, czy strumień może obsługiwać metody `mark` i `reset`
`abstract int`	`read()`	Odczytuje kolejny bajt ze strumienia
`int`	`read(byte[] b)`	Odczytuje ze strumienia liczbę bajtów nie większą niż rozmiar tablicy `b`. Zwraca faktycznie odczytaną liczbę bajtów.
`int`	`read(byte[] b, int off, int len)`	Odczytuje ze strumienia liczbę bajtów nie większą niż wskazywana przez `len`, i zapisuje je w tablicy `b`, począwszy od komórki wskazywanej przez `off`. Zwraca faktycznie przeczytaną liczbę bajtów.
`void`	`reset()`	Wraca do pozycji strumienia wskazywanej przez wywołanie metody `mark`
`long`	`skip(long n)`	Pomija w strumieniu liczbę bajtów wskazywanych przez `n`. Zwraca faktycznie pominiętą liczbę bajtów.

Wczytywanie tekstu za pomocą klasy buforowej

Należy skorzystać z metody `readLine` klasy `BufferedReader`. Aby utworzyć obiekt tej klasy powiązany ze standardowym strumieniem wejściowym `System.in`, trzeba dodatkowo utworzyć obiekt pośredniczący klasy `InputStreamReader`, stosując konstrukcję:

```
BufferedReader brIn = new BufferedReader(
  new InputStreamReader(System.in)
  );
```

Przykład:

```
public static void main(String args[])
  {
    BufferedReader brIn = new BufferedReader(
      new InputStreamReader(System.in)
      );
    System.out.println("Wprowadź wiersz tekstu zakończony znakiem Enter:");
    try{
      String line = brIn.readLine();
      System.out.print("Wprowadzona linia to: " + line);
    }
    catch(IOException e){
      System.out.println("Błąd podczas odczytu strumienia.");
    }
  }
```

Wprowadzanie liczb

Do wprowadzania wartości liczbowych można zastosować klasę `StreamTokenizer`, która dzieli strumień wejściowy na jednostki leksykalne, czyli tokeny. Ma ona pole o nazwie `nval`, które zawiera wartość aktualnego tokena w postaci liczby typu `double` (o ile ten token jest liczbą). Typ tokena można rozpoznać, odczytując stan pola `ttype`, które może przyjmować następujące wartości:

- `StreamTokenizer.TT_EOF` — osiągnięty został koniec strumienia;
- `StreamTokenizer.TT_EOL` — osiągnięty został koniec linii;
- `StreamTokenizer.TT_NUMBER` — token jest liczbą;
- `StreamTokenizer.TT_WORD` — token jest słowem.

Przykład:

```
import java.io.*;

public class Main
{
  public static void main(String args[])
  {
    StreamTokenizer strTok = new StreamTokenizer(
      new BufferedReader(
        new InputStreamReader(System.in)
      )
    );

    System.out.print("Wprowadź liczbę: ");

    try{
      strTok.nextToken();
    }
    catch(IOException e){
      System.out.print("Błąd podczas odczytu danych ze strumienia.");
      return;
    }

    if(strTok.ttype != StreamTokenizer.TT_NUMBER){
      System.out.print("To nie jest prawidłowa liczba.");
      return;
    }

    double liczba = strTok.nval;
    System.out.print("Wprowadzona liczba to " + liczba);
  }
}
```

Klasa Scanner

Począwszy od wersji Javy 5.0 (1.5), do przetwarzania danych wejściowych można używać klasy `Scanner`. Zawiera ona konstruktory, które mogą przyjmować obiekty klas: `File`, `InputReader` i `String`, a także obiekty implementujące interfejsy `Readable` lub `ReadableByteChannel`. Jest to więc zestaw pozwalający na obsługę bardzo wielu formatów wejściowych. Metod klasy `Scanner` jest bardzo wiele, a najbardziej przydatne są te z rodziny `next` i `hasNext`, konstruowane na bardzo prostych zasadach, które schematycznie można przedstawić jako:

`hasNext`*`NazwaTypuProstego`*

oraz:

`next`*`NazwaTypuProstego`*

Istnieją więc metody: `hasNext`, `hasNextInt`, `hasNextDouble`, `hasNextByte` itd. Wszystkie one zwracają wartość `true`, jeśli w powiązanym strumieniu danych kolejną jednostką leksykalną jest wartość danego typu prostego, natomiast „pusta" metoda `hasNext` zwraca wartość `true`, jeżeli w strumieniu istnieje jakakolwiek kolejna jednostka leksykalna. Dodatkowo istnieje metoda `hasNextLine`, która określa, czy w strumieniu znajduje się wiersz tekstu.

Metody z rodziny `next`, a więc: `nextInt`, `nextDouble`, `nextByte` itd., zwracają kolejną jednostkę leksykalną w postaci wartości danego typu prostego. Metoda `next` zwraca z kolei token w postaci ciągu znaków, a `nextLine` — cały wiersz tekstu.

Przykład:

```
import java.util.*;

public class Main {
  public static void main(String
↪args[]) {
    Scanner scanner = new
↪Scanner(System.in);

    System.out.print("Wprowadź wartość
↪całkowitą: ");

    while(!scanner.hasNextInt()){
      System.out.print("To nie jest
↪wartość całkowita: ");
    System.out.println(scanner.
↪next());
      System.out.print("Wprowadź
↪wartość całkowitą: ");
    }
    int value = scanner.nextInt();
    int result = value * 2;
    System.out.println(value + " * 2
↪= " + result);
  }
}
```

Standardowy strumień wyjściowy

Dane można wyprowadzać (np. na ekran konsoli) za pomocą instrukcji `System.out.println` lub `System.out.print`, czyli przez wywołanie metody `println` lub `print` obiektu `System.out`. Jest to obiekt klasy `PrintStream`. Metody udostępniane przez tę klasę zostały zebrane w tabeli u góry po prawej.

Deklaracja	Opis
`PrintStream append(char c)`	Dodaje znak do strumienia
`PrintStream append(CharSequence csq)`	Dodaje do strumienia sekwencję znaków
`PrintStream append(CharSequence csq, int start, int end)`	Dodaje do strumienia część sekwencji znaków wyznaczaną przez indeksy `start` i `end`
`boolean checkError()`	Opróżnia bufor oraz sprawdza, czy nie wystąpił błąd
`protected void clearError()`	Zeruje status błędu
`void close()`	Zamyka strumień
`void flush()`	Powoduje opróżnienie bufora
`PrintStream format(Locale l, String format, Object… args)`	Zapisuje w strumieniu dane określone przez argumenty `args` w formacie zdefiniowanym przez `format`, zgodnie z ustawieniami narodowymi wskazanymi przez `l`
`PrintStream format(String format, Object… args)`	Zapisuje w strumieniu dane określone przez argumenty `args` w formacie określonym przez `format`
`void print(boolean b)`	Wyświetla wartość typu `boolean`
`void print(char c)`	Wyświetla znak
`void print(char[] s)`	Wyświetla tablicę znaków
`void print(double d)`	Wyświetla wartość typu `double`
`void print(float f)`	Wyświetla wartość typu `float`
`void print(int i)`	Wyświetla wartość typu `int`
`void print(long l)`	Wyświetla wartość typu `long`
`void print(Object obj)`	Wyświetla ciąg znaków uzyskany przez wywołanie metody `toString` obiektu `obj`
`void print(String s)`	Wyświetla ciąg znaków `s`
`PrintStream printf(Locale l, String format, Object… args)`	Zapisuje w strumieniu dane określone przez argumenty `args` w formacie zdefiniowanym przez `format`, zgodnie z ustawieniami narodowymi wskazanymi przez `l`
`PrintStream printf(String format, Object… args)`	Zapisuje w strumieniu dane określone przez argumenty `args` w formacie określonym przez `format`
`void println()`	Wyświetla znak końca linii (powoduje przejście do nowej linii)
`void println(boolean b)`	Wyświetla wartość typu `boolean` oraz znak końca linii
`void println(char c)`	Wyświetla znak zapisany w `c` oraz znak końca linii
`void println(char[] tab)`	Wyświetla tablicę znaków `tab` oraz znak końca linii
`void println(double d)`	Wyświetla wartość typu `double` oraz znak końca linii
`void println(float f)`	Wyświetla wartość typu `float` oraz znak końca linii
`void println(int i)`	Wyświetla wartość typu `int` oraz znak końca linii
`void println(long l)`	Wyświetla wartość typu `long` oraz znak końca linii
`void println(Object obj)`	Wyświetla ciąg znaków uzyskany przez wywołanie metody `toString` obiektu `obj` oraz znak końca linii
`void println(String s)`	Wyświetla ciąg znaków `s` oraz znak końca linii
`protected void setError()`	Ustawia strumień w stan błędu
`void write(byte[] buf, int off, int len)`	Zapisuje do strumienia bajty z tablicy `buf`, w liczbie wskazywanej przez `len`, poczynając od komórki określonej przez `off`
`void write(int b)`	Zapisuje bajt `b` do strumienia

System plików

Klasa File

Klasa `File` pozwala na wykonywanie podstawowych operacji na plikach i katalogach, takich jak ich tworzenie i usuwanie, operacje na nazwach czy pobieranie parametrów (np. czasu utworzenia bądź modyfikacji). Nie jest to jednak klasa, która umożliwiałaby modyfikację zawartości pliku. Wybrane metody udostępniane przez klasę `File` zostały zebrane w poniższej tabeli.

Typ	Nazwa metody	Opis
`boolean`	`canExecute()`	Sprawdza, czy aplikacja może uruchomić dany plik
`boolean`	`canRead()`	Sprawdza, czy aplikacja może odczytywać dany plik
`boolean`	`canWrite()`	Sprawdza, czy aplikacja ma prawa zapisu do danego pliku
`int`	`compareTo(File pathname)`	Porównuje ścieżki dostępu do plików
`static File`	`createTempFile(String prefix, String suffix)`	Tworzy pusty plik tymczasowy. Nazwa tego pliku powstaje przy wykorzystaniu prefiksu i sufiksu przekazanych w parametrach.
`static File`	`createTempFile(String prefix, String suffix, boolean deleteOnExit, FileAttribute<?>... attrs)`	Tworzy pusty plik tymczasowy o atrybutach wskazanych przez `attrs`. Argument `deleteOnExit` wskazuje, czy plik ma być automatycznie usunięty po zakończeniu pracy maszyny wirtualnej. Metoda dostępna od JDK 1.7.
`static File`	`createTempFile(String prefix, String suffix, File directory)`	Tworzy pusty plik tymczasowy w katalogu wskazywanym przez argument `directory`
`boolean`	`delete()`	Usuwa plik lub katalog
`void`	`deleteOnExit()`	Zaznacza, że plik ma zostać usunięty, kiedy maszyna wirtualna będzie kończyć pracę
`boolean`	`exists()`	Sprawdza istnienie pliku lub katalogu
`File`	`getAbsoluteFile()`	Zwraca obiekt zawierający bezwzględną nazwę pliku lub katalogu (wraz z pełną ścieżką dostępu)
`String`	`getAbsolutePath()`	Zwraca bezwzględną ścieżkę dostępu do pliku lub katalogu
`File`	`getCanonicalFile()`	Zwraca obiekt zawierający kanoniczną postać nazwy pliku lub katalogu (wraz z pełną ścieżką dostępu)
`String`	`getCanonicalPath()`	Zwraca kanoniczną postać ścieżki dostępu do pliku lub katalogu
`long`	`getFreeSpace()`	Zwraca ilość wolnego miejsca na partycji wskazywanej przez bieżący obiekt typu `File`
`String`	`getName()`	Zwraca nazwę pliku (bez ścieżki dostępu)
`String`	`getParent()`	Zwraca nazwę katalogu nadrzędnego
`File`	`getParentFile()`	Zwraca obiekt wskazujący na katalog nadrzędny
`String`	`getPath()`	Zwraca nazwę bieżącego katalogu lub pliku w postaci obiektu typu `String`
`long`	`getTotalSpace()`	Zwraca całkowity rozmiar partycji wskazywanej przez bieżący obiekt
`long`	`getUsableSpace()`	Zwraca ilość miejsca dostępnego dla maszyny wirtualnej na partycji wskazywanej przez bieżący obiekt
`int`	`hashCode()`	Oblicza wartość funkcji skrótu dla danej ścieżki dostępu
`boolean`	`isAbsolute()`	Sprawdza, czy dana ścieżka dostępu jest ścieżką bezwzględną
`boolean`	`isDirectory()`	Sprawdza, czy ścieżka dostępu wskazuje na katalog
`boolean`	`isFile()`	Sprawdza, czy ścieżka dostępu wskazuje na plik
`boolean`	`isHidden()`	Sprawdza, czy ścieżka dostępu wskazuje na ukryty katalog lub plik
`long`	`lastModified()`	Zwraca czas ostatniej modyfikacji pliku lub katalogu
`long`	`length()`	Zwraca wielkość pliku w bajtach
`String[]`	`list()`	Zwraca zawartość katalogu w postaci tablicy obiektów typu `String`
`String[]`	`list(FilenameFilter filter)`	Zwraca listę plików i podkatalogów spełniających kryteria wskazane przez `filter`
`File[]`	`listFiles()`	Zwraca zawartość katalogu w postaci obiektów typu `File`
`File[]`	`listFiles(FileFilter filter)`	Zwraca zawartość katalogu spełniającą kryteria wskazane przez `filter` w postaci obiektów typu `File`

File[]	listFiles(FilenameFilter ↪filter)	Zwraca zawartość katalogu spełniającą kryteria wskazane przez filter w postaci obiektów typu File
static ↪File[]	listRoots()	Wyświetla wszystkie „korzenie" systemu plików
boolean	mkdir()	Tworzy nowy katalog
boolean	mkdirs()	Tworzy nowy katalog z uwzględnieniem nieistniejących katalogów nadrzędnych
boolean	renameTo(File dest)	Zmienia nazwę na wskazywaną przez argument dest
boolean	setExecutable(boolean ↪executable)	Ustawia prawo wykonywalności dla danego pliku lub katalogu
boolean	setLastModified(long time)	Ustawia datę ostatniej modyfikacji pliku lub katalogu
boolean	setReadable(boolean ↪readable)	Ustawia prawo do odczytu dla danego pliku lub katalogu
boolean	setReadOnly()	Ustawia atrybut ReadOnly pliku lub katalogu
boolean	setWritable(boolean ↪writable)	Ustawia prawo do zapisu dla danego pliku lub katalogu
String	toString()	Zwraca ścieżkę dostępu w postaci obiektu klasy String
URI	toURI()	Przekształca ścieżkę dostępu na obiekt URI

Klasa RandomAccessFile

Klasa RandomAccessFile pozwala na wykonywanie wszelkich operacji na plikach o dostępnie swobodnym, a także na odczytywanie i zapisywanie danych z pliku i do niego oraz przemieszczanie się po pliku. Jest dostępna we wszystkich JDK, począwszy od wersji 1.0. Wybrane metody udostępniane przez RandomAccessFile zostały zebrane w poniższej tabeli.

Typ	Metoda	Opis
void	close()	Zamyka strumień oraz zwalnia wszystkie związane z nim zasoby
FileChannel	getChannel()	Zwraca powiązany z plikiem unikatowy obiekt typu FileChannel
FileDescriptor	getFD()	Zwraca deskryptor pliku powiązanego ze strumieniem
long	getFilePointer()	Zwraca aktualną pozycję w pliku
long	length()	Zwraca długość pliku
int	read()	Odczytuje jeden bajt danych z pliku
int	read(byte[] b)	Odczytuje z pliku liczbę bajtów, nie większą niż rozmiar tablicy b, i umieszcza je w tej tablicy. Zwraca faktycznie odczytaną liczbę bajtów.
int	read(byte[] b, int off, ↪int len)	Odczytuje z pliku liczbę bajtów, nie większą niż wskazywana przez len, i zapisuje je w tablicy b, począwszy od komórki wskazywanej przez off. Zwraca faktycznie przeczytaną liczbę bajtów.
boolean	readBoolean()	Odczytuje wartość typu boolean
byte	readByte()	Odczytuje wartość typu byte
char	readChar()	Odczytuje wartość typu char
double	readDouble()	Odczytuje wartość typu double
float	readFloat()	Odczytuje wartość typu float
int	readFully(byte[] b)	Odczytuje liczbę bajtów równą wielkości tablicy b. Zwraca liczbę faktycznie odczytanych bajtów.
int	readFully(byte[] b, ↪int off, int len)	Odczytuje liczbę bajtów, wskazywaną przez len, i zapisuje je w tablicy b, począwszy od komórki wskazywanej przez off. Zwraca faktycznie przeczytaną liczbę bajtów.
int	readInt()	Odczytuje wartość typu int
String	readLine()	Odczytuje wiersz tekstu
long	readLong()	Odczytuje wartość typu long
short	readShort()	Odczytuje wartość typu short
int	readUnsignedByte()	Odczytuje 8-bitową wartość bez znaku
int	readUnsignedShort()	Odczytuje 16-bitową wartość bez znaku
String	readUTF()	Odczytuje tekst w kodowaniu UTF-8
void	seek(long pos)	Zmienia wskaźnik pozycji w pliku na pos
void	setLength(long new- ↪Length)	Ustawia rozmiar pliku na newLength
int	skipBytes(int n)	Pomija n bajtów
void	write(byte[] b)	Zapisuje tablicę bajtów b do pliku
void	write(byte[] b, int ↪off, int len)	Zapisuje do pliku len bajtów z tablicy b, począwszy od komórki wskazywanej przez off
void	write(int b)	Zapisuje bajt b do pliku
void	writeBoolean(boolean v)	Zapisuje do pliku wartość boolean w postaci jednego bajta
void	writeByte(int v)	Zapisuje bajt v do pliku
void	writeBytes(String s)	Zapisuje do pliku ciąg znaków, wskazywany przez s, w postaci ciągu bajtów
void	writeChar(int v)	Zapisuje do pliku wartość typu char w postaci dwóch bajtów
void	writeChars(String s)	Zapisuje do pliku ciąg wskazywany przez s w postaci ciągu znaków
void	writeDouble(double v)	Konwertuje wartość v na typ long, korzystając z metody doubleToLongBits klasy Double, i tak powstałą wartość zapisuje do pliku
void	writeFloat(float v)	Konwertuje wartość v na typ int, korzystając z metody floatToIntBits klasy Float, i tak powstałą wartość zapisuje do pliku
void	writeInt(int v)	Zapisuje do pliku wartość typu int w postaci czterech bajtów
void	writeLong(long v)	Zapisuje do pliku wartość typu long w postaci ośmiu bajtów
void	writeShort(int v)	Zapisuje do pliku wartość typu short w postaci dwóch bajtów
void	writeUTF(String str)	Zapisuje do pliku ciąg znaków, wskazywany przez s, w kodowaniu UTF-8

Odczyt pliku

Przykład:

```
import java.io.*;

public class Main
{
  public static void main (String args[])
  {
    if(args.length < 1){
      System.out.println("Wywołanie programu: Main nazwa_pliku");
      return;
    }

    File file = new File(args[0]);

    if(!file.exists()){
      System.out.println("Nie ma takiego pliku.");
      return;
    }
    RandomAccessFile raf = null;
    try{
      raf = new RandomAccessFile(file, "r");
    }
    catch(FileNotFoundException e){
      System.out.println("Nie ma takiego pliku.");
      return;
    }
    String line = "";
    try{
      while((line = raf.readLine()) != null){
        System.out.println(line);
      }
      raf.close();
    }
    catch(IOException e){
      System.out.println("Błąd wejścia/wyjścia.");
    }
  }
}
```

Zapis do pliku

Przykład:

```
import java.io.*;

public class Main
{
  public static void main (String args[])
  {
    if(args.length < 1){
      System.out.println("Wywołanie programu: Main nazwa_pliku");
      return;
    }

    BufferedReader brIn = new BufferedReader(
      new InputStreamReader(System.in)
      );

    RandomAccessFile raf = null;

    try{
      raf = new RandomAccessFile(args[0], "rw");
    }
    catch(FileNotFoundException e){
      System.out.println("Błędna nazwa pliku lub brak dostępu.");
      return;
    }

    String line = "";

    try{
      while(true){
        line = brIn.readLine();
        if("quit".equals(line) || line == null)
          break;
        raf.writeBytes(line + "\n");
      }
      raf.close();
    }
    catch(IOException e){
      System.out.print("\nBłąd wejścia/wyjścia.");
      return;
    }
  }
}
```

Strumieniowe operacje na plikach

Odczyt danych

Podstawowe klasy odczytujące dane z plików to:

- FileReader,
- FileInputStream.

Pierwsza z nich powinna być stosowana podczas korzystania ze strumienia znakowego, czyli dla plików tekstowych, druga — podczas korzystania ze strumienia binarnego, czyli dla plików binarnych. Obie klasy mają po trzy przeciążone konstruktory, których argumentami mogą być:

- ciąg znaków zawierający nazwę pliku,
- obiekt klasy FileDescriptor,
- obiekt klasy File.

Metody odczytujące dane klasy FileReader:

Typ	Metoda	Opis
int	read()	Odczytuje pojedynczy znak
int	read(char[] ↪cbuf, int ↪offset, ↪int length)	Odczytuje liczbę znaków, nie większą niż wskazywana przez length, i zapisuje je w tablicy cbuf, począwszy od komórki wskazywanej przez offset. Zwraca faktycznie przeczytaną liczbę znaków.

Metody odczytujące dane klasy FileInputStream:

Typ	Metoda	Opis
int	read()	Odczytuje jeden bajt danych
int	read(byte[] b)	Odczytuje liczbę bajtów, nie większą od długości tablicy b, i zapisuje je w tablicy b. Zwraca faktycznie odczytaną liczbę bajtów.
int	read(byte[] ↪b, int off, ↪int len)	Odczytuje liczbę bajtów, nie większą niż wskazywana przez len, i zapisuje je w tablicy b, począwszy od komórki wskazywanej przez off. Zwraca faktycznie przeczytaną liczbę bajtów.

Metod tych można używać bezpośrednio lub też wykorzystać obiekty klas FileReader i FileInputStream jako argumenty dla konstruktorów klas dających większą funkcjonalność, np. BufferedInputStream lub BufferedReader.

Zapis danych

Podstawowe klasy zapisujące dane z plików to:

- FileWriter,
- FileOutputStream.

Pierwsza z nich powinna być wykorzystywana do zapisu plików tekstowych, druga — do zapisu strumieni binarnych. Obie klasy mają po pięć przeciążonych konstruktorów. Trzy z nich są jednoargumentowe i mogą przyjmować następujące argumenty:

- ciąg znaków zawierający nazwę pliku,
- obiekt klasy FileDescriptor,
- obiekt klasy File.

Pozostałe konstruktory są dwuargumentowe. Pierwszy przyjmuje ciąg znaków oraz wartość boolean, drugi — obiekt klasy File i wartość boolean. W obu przypadkach drugi argument ustawiony na true oznacza, że dane mają być dopisywane na końcu pliku, a ustawiony na false — że dane mają być zapisywane od początku pliku (nadpisując jego wcześniejszą zawartość).

Wszystkie konstruktory generują wyjątek FileNotFoundException, jeśli:

- podana nazwa wskazuje na katalog, a nie na plik;
- podany plik nie istnieje i nie można go również utworzyć;
- nie można z jakiegoś powodu otworzyć istniejącego pliku.

Metody zapisujące dane klasy FileWriter:

Typ zwracany	Metoda	Opis
void	write(char[] ↪cbuf, int ↪off, int ↪len)	Zapisuje do strumienia len znaków z tablicy cbuf, począwszy od komórki wskazywanej przez off
void	write(int c)	Zapisuje do strumienia znak c
void	write(String ↪str, int ↪off, int ↪len)	Zapisuje do strumienia len znaków z ciągu str, począwszy od znaku o indeksie wskazywanym przez off

Metody zapisujące dane klasy FileOutputStream:

Typ zwracany	Metoda	Opis
void	write(byte[] b)	Zapisuje tablicę bajtów b do strumienia
void	write(byte[] b, ↪int off, int ↪len)	Zapisuje do strumienia len bajtów z tablicy b, począwszy od komórki wskazywanej przez off
void	write(int b)	Zapisuje bajt b do strumienia

Wydawnictwo Helion
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

ISBN: 978-83-246-4707-1