ORACLE

Java™

Java
Kompendium programisty

Wydanie X

Herbert Schildt

Helion

Oracle Press™

Tytuł oryginału: Java™: The Complete Reference, Tenth Edition

Tłumaczenie: Piotr Rajca oraz Mikołaj Szczepaniak (fragmenty pochodzące z „Java. Kompendium programisty. Wydanie VIII”

ISBN: 978-83-283-4613-0

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/javk10
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/javk10.zip

Printed in Poland.

Spis treści

CZĘŚĆ II
Biblioteka języka Java

CZĘŚĆ III
Wprowadzenie do programowania GUI przy użyciu pakietu Swing

CZĘŚĆ IV
Wprowadzenie do programowania GUI przy użyciu platformy JavaFX

CZĘŚĆ V
Stosowanie Javy w praktyce

DODATKI

ROZDZIAŁ

13

Wejście-wyjście, instrukcja try z zasobami i inne tematy

Niniejszy rozdział zawiera wprowadzenie do jednego z najważniejszych pakietów: `io`. Pakiet `io` zawiera podstawowy podsystem wejścia-wyjścia stosowany w Javie, włącznie z obsługą plików. Obsługa wejścia-wyjścia nie stanowi części samego rdzenia języka Java, ale jest zawarta w podstawowych bibliotekach. Z tego powodu dokładne omówienie tego tematu znajduje się w części II, która zajmuje się kilkoma pakietami tworzącymi API Javy. Ten rozdział to tylko wprowadzenie do zagadnień tego ważnego podsystemu, które ma pokazać, jakie jest jego miejsce w bardziej ogólnym kontekście programowania w Javie i jej środowiska wykonawczego. Zawiera on również omówienie instrukcji `try` zarządzającej zasobami i następujących słów kluczowych Javy: `transient`, `volatile`, `instanceof`, `native`, `strictfp` i `assert`. Na końcu rozdziału przeanalizujemy działanie mechanizmu importu statycznego i alternatywne zastosowanie słowa kluczowego `this`.

Podstawowa obsługa wejścia i wyjścia

W poprzednich 12 rozdziałach w zasadzie nie pojawiały się żadne informacje na temat obsługi wejścia-wyjścia. Przykładowe programy korzystały tylko i wyłącznie z metod `print()` oraz `println()`. Powód jest bardzo prosty: niewiele rzeczywistych programów Javy działa w trybie tekstowym. Współczesne aplikacje albo oferują graficzne interfejsy użytkownika tworzone na bazie jednego z pakietów: Swing, AWT lub JavaFX, albo mają postać aplikacji internetowych. Choć programy tekstowe doskonale nadają się do nauki programowania, bardzo rzadko występują w rzeczywistym świecie. Korzystanie z konsoli w Javie jest ograniczone i nieco dziwaczne — nawet w prostych programach. Po prostu konsola nigdy nie była bardzo ważnym elementem programowania w języku Java.

Oczywiście poprzedni akapit nie neguje, iż Java zawiera bardzo elegancką i elastyczną obsługę wejścia-wyjścia związanego z plikami i siecią. System wejścia-wyjścia Javy jest zwarty i spójny. Wystarczy poznać jego podstawy, aby niezwykle szybko opanować pozostałe zagadnienia. W tym rozdziale dokonamy ogólnego przeglądu podsystemu wejścia-wyjścia; szczegółowy opis poszczególnych mechanizmów można znaleźć w rozdziałach 21. i 22.

Strumienie

Programy Javy wykonują operacje wejścia-wyjścia za pośrednictwem strumieni. **Strumień** to pewien abstrakcyjny byt, który produkuje lub konsumuje informacje. Strumień łączy się z fizycznym urządzeniem dzięki systemowi wejścia-wyjścia Javy. Strumienie zachowują się zawsze tak samo, niezależnie od tego, jakiego urządzenia fizycznego dotyczą. Oznacza to, iż te same klasy i metody wejścia-wyjścia służą do obsługi różnych urządzeń: strumień wejściowy służy do odczytu danych z pliku, klawiatury lub gniazdka sieciowego, natomiast strumień wyjściowy wysyła dane do pliku, na konsolę lub do innego komputera. Strumienie to wręcz doskonała abstrakcja, gdyż poszczególne części kodu nie muszą znać różnicy między klawiaturą i siecią. Implementacja strumieni w Javie opiera się na hierarchii klas zawartych w pakiecie `java.io`.

Oprócz operacji wejścia-wyjścia na strumieniach zaimplementowanych w pakiecie `java.io` język Java obsługuje też operacje wejścia-wyjścia na buforach i kanałach, które zaimplementowano w pakiecie `java.nio` i jego podpakietach. Te operacje zostaną omówione w rozdziale 22.

Strumienie znakowe i bajtowe

Java definiuje dwa rodzaje strumieni: znakowe i bajtowe. **Strumień bajtów** stanowi wygodny sposób obsługi wejścia i wyjścia bajtowego. Używa się go na przykład do zapisu i odczytu danych binarnych. **Strumień znaków** zapewnia wygodny sposób przekazywania znaków. Strumień korzysta ze standardu Unicode, więc obsługuje wiele różnych języków. W pewnych zastosowaniach strumienie znaków są bardziej wydajne od strumieni bajtów.

Oryginalne wydanie Javy (1.0) nie zawierało strumieni znaków, więc cała komunikacja odbywała się bajtowo. Strumienie znaków pojawiły się w Javie 1.1, a pewne klasy i metody związane z bajtami zostały zabronione. Właśnie z tego powodu warto uaktualnić kod, który jeszcze nie korzysta ze strumieni znaków.

Trzeba pamiętać o tym, iż na najniższym poziomie cała komunikacja wejścia-wyjścia odbywa się bajtowo. Strumienie znaków po prostu ułatwiają przekazywanie znaków.

Kolejne podpunkty omawiają dostępne strumienie bajtów i znaków.

Klasy strumieni bajtów

Strumienie bajtów zaimplementowano w ramach dwóch hierarchii klas. Na samym szczycie tych hierarchii znajdują się dwie klasy abstrakcyjne: `InputStream` i `OutputStream`. Dla każdej z tych klas abstrakcyjnych istnieje wiele konkretnych klas pochodnych obsługujących różnego rodzaju urządzenia, na przykład pliki, połączenia sieciowe, a nawet bufory pamięci. Klasy strumieni bajtów należące do pakietu `java.io` opisano w tabeli 13.1. Niektóre z klas zostaną omówione w niniejszym rozdziale; pozostałe są opisane w części II. Pamiętaj, że korzystanie z klas strumieni wymaga zaimportowania pakietu `java.io`.

Klasy abstrakcyjne `InputStream` i `OutputStream` definiują kilka kluczowych metod implementowanych przez pozostałe klasy strumieni. Najważniejszymi metodami są `read()` i `write()`, które odpowiednio odczytują i zapisują bajty danych. Obie metody są zadeklarowane jako abstrakcyjne w klasach `InputStream` i `OutputStream`. Klasy pochodne przesłaniają je konkretnymi implementacjami.

Klasy strumieni znaków

Strumienie znaków korzystają z dwóch hierarchii klas. Na samym szczycie tych hierarchii znajdują się dwie klasy abstrakcyjne: `Reader` i `Writer`. Obie klasy obsługują strumienie znaków Unicode. Istnieje kilka konkretnych klas pochodnych wspomnianych klas abstrakcyjnych. Tabela 13.2 zawiera listę klas strumieni znaków.

Klasy abstrakcyjne `Reader` i `Writer` definiują kilka kluczowych metod implementowanych przez pozostałe klasy strumieni. Najważniejszymi metodami są `read()` i `write()`, które odpowiednio odczytują i zapisują znaki danych. Klasy pochodne przesłaniają je konkretnymi implementacjami.

Tabela 13.1. *Klasy strumieni bajtowych*

Klasa strumienia	Znaczenie
`BufferedInputStream`	Buforowany strumień wejściowy
`BufferedOutputStream`	Buforowany strumień wyjściowy
`ByteArrayInputStream`	Strumień wejściowy czytający z tablicy bajtów
`ByteArrayOutputStream`	Strumień wyjściowy zapisujący do tablicy bajtów
`DataInputStream`	Strumień wejściowy zawierający metody do odczytywania podstawowych typów danych Javy
`DataOutputStream`	Strumień wyjściowy zawierający metody do zapisywania podstawowych typów danych Javy
`FileInputStream`	Strumień wejściowy odczytujący z pliku
`FileOutputStream`	Strumień wyjściowy zapisujący do pliku
`FilterInputStream`	Implementuje `InputStream`
`FilterOutputStream`	Implementuje `OutputStream`
`InputStream`	Klasa abstrakcyjna opisująca strumień wejściowy
`ObjectInputStream`	Strumień wejściowy dla obiektów
`ObjectOutputStream`	Strumień wyjściowy dla obiektów
`OutputStream`	Klasa abstrakcyjna opisująca strumień wyjściowy
`PipedInputStream`	Potok wejściowy
`PipedOutputStream`	Potok wyjściowy
`PrintStream`	Strumień wyjściowy zawierający metody `print()` i `println()`
`PushbackInputStream`	Strumień wejściowy pozwalający zwracać do strumienia odczytane z niego bajty
`SequenceInputStream`	Strumień wejściowy łączący kilka innych strumieni wejściowych odczytywanych jeden po drugim

Tabela 13.2. *Klasy strumieni znaków*

Klasa strumienia	Znaczenie
`BufferedReader`	Buforowany strumień wejściowy znaków
`BufferedWriter`	Buforowany strumień wyjściowy znaków
`CharArrayReader`	Strumień wejściowy czytający z tablicy znaków
`CharArrayWriter`	Strumień wyjściowy zapisujący do tablicy znaków
`FileReader`	Strumień wejściowy czytający z pliku
`FileWriter`	Strumień wyjściowy zapisujący do pliku
`FilterReader`	Filtrowany strumień wejściowy
`FilterWriter`	Filtrowany strumień wyjściowy
`InputStreamReader`	Strumień wejściowy zamieniający bajty na znaki
`LineNumberReader`	Strumień wejściowy zliczający wiersze
`OutputStreamWriter`	Strumień wyjściowy zamieniający znaki na bajty
`PipedReader`	Potok wejściowy
`PipedWriter`	Potok wyjściowy
`PrintWriter`	Strumień wyjściowy zawierający metody `print()` i `println()`
`PushbackReader`	Strumień wejściowy umożliwiający zwrócenie znaku z powrotem do strumienia
`Reader`	Klasa abstrakcyjna opisująca strumień wejściowy znaków
`StringReader`	Strumień wejściowy czytający z łańcucha
`StringWriter`	Strumień wyjściowy zapisujący w łańcuchu
`Writer`	Klasa abstrakcyjna opisująca strumień wyjściowy znaków

Predefiniowane strumienie

Wszystkie programy Javy automatycznie importują pakiet `java.lang`. Pakiet ten definiuje klasę o nazwie `System` dotyczącą kilku aspektów środowiska, w którym działa aplikacja. Na przykład przy użyciu jej metod można pobrać aktualny czas lub ustawienia zmiennych systemowych. Klasa System zawiera trzy predefiniowane zmienne strumieni: `in`, `out` i `err`. Wymienione pola zdeklarowano jako zmienne publiczne, statyczne i finalne. Oznacza to, iż można z nich korzystać w pozostałych częściach programu bez konieczności tworzenia egzemplarza klasy `System`.

Składowa `System.out` to standardowy strumień wyjściowy. Domyślnie jest to konsola. Składowa `System.in` to standardowy strumień wejściowy, którym domyślnie jest klawiatura. Składowa `System.err` to standardowy strumień błędów, który domyślnie dotyczy konsoli. Wszystkie te strumienie można przekierować do dowolnego innego, zgodnego urządzenia wejścia-wyjścia.

Składowa `System.in` jest obiektem typu `InputStream`. Składowe `System.out` i `System.err` to obiekty typu `OutputStream`. Są to strumienie bajtowe, choć na ogół służą do odczytywania i zapisywania znaków do konsoli. W razie konieczności strumienie te można otoczyć strumieniami znakowymi.

W wielu przykładach z poprzednich rozdziałów stosowaliśmy składową `System.out`. Składowej `System.err` używa się dokładnie tak samo. Używanie składowej `System.in` jest nieco bardziej złożone.

Odczyt danych z konsoli

W początkowym okresie stosowania Javy jedynym sposobem odczytania danych z konsoli było zastosowanie strumienia bajtów. Także dzisiaj odczyt informacji z konsoli za pomocą strumienia bajtów jest dopuszczalny. W komercyjnych aplikacjach zalecaną metodą odczytywania danych wejściowych z konsoli jest stosowanie strumieni znaków. Takie rozwiązanie ułatwia umiędzynarodawianie i konserwację oprogramowania.

W Javie odczyt danych z konsoli odbywa się za pomocą `System.in`. Aby odczytać strumień znaków z konsoli, należy opakować obiekt `System.in` obiektem klasy `BufferedReader`. Klasa `BufferedReader` obsługuje buforowany odczyt ze strumienia wejściowego. Najczęściej stosowana wersja konstruktora jest następująca.

```
BufferedReader(Reader czytnikWej)
```

Parametr *czytnikWej* to strumień, który zostanie połączony z właśnie tworzonym egzemplarzem klasy `BufferedReader`. Klasa `Reader` to klasa abstrakcyjna. Jedną z jej konkretnych klas pochodnych jest `InputStreamReader`, która konwertuje bajty na znaki. Aby utworzyć obiekt `InputStreamReader` powiązany z `System.in`, korzysta się z następującego konstruktora.

```
InputStreamReader(InputStream strumieńWej)
```

Ponieważ pole `System.in` wskazuje na obiekt typu `InputStream`, można go przekazać jako parametr *strumieńWej*. Po połączeniu wszystkiego razem poniższy wiersz kodu tworzy obiekt `BufferedReader` związany z klawiaturą.

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

Po wykonaniu tej instrukcji zmienna `br` zawiera strumień znaków połączony z konsolą za pośrednictwem składowej `System.in`.

Odczyt znaków

Do odczytania znaku z obiektu `BufferedReader` służy metoda `read()`. Stosowana przez nas wersja `read()` ma następującą postać.

```
int read() throws IOException
```

Każde wywołanie metody powoduje odczytanie jednego znaku ze strumienia wejściowego i zwrócenie jego wartości całkowitoliczbowej. Zwrócenie wartości –1 oznacza osiągnięcie końca strumienia wejściowego. Co więcej, metoda może zgłosić wyjątek `IOException`.

Poniższy program odczytuje znaki z klawiatury za pomocą metody `read()` aż do momentu przekazania znaku „q”. Warto zwrócić uwagę na to, że ewentualne wyjątki operacji wejścia-wyjścia, które mogą zostać zgłoszone w trakcie działania tego programu, będą zgłaszane dalej przez metodę `main()`. To dość typowe rozwiązanie podczas odczytywania danych wejściowych z konsoli w prostych programach (w tym w przykładach prezentowanych w tej książce), jednak w bardziej zaawansowanych aplikacjach należałoby wprost zaimplementować obsługę wszystkich potencjalnych wyjątków.

```
// R13\BRRead.java

// Użycie BufferedReader do odczytu znaków z konsoli
import java.io.*;

class BRRead {
  public static void main(String args[]) throws IOException
  {
    char c;
    BufferedReader br = new
            BufferedReader(new InputStreamReader(System.in));
    System.out.println("Wpisz znaki, wpisanie 'q' powoduje zakończenie programu.");

    // Odczyt znaków
    do {
      c = (char) br.read();
      System.out.println(c);
    } while(c != 'q');
  }
}
```

Oto przykładowy wynik działania programu.

```
Wpisz znaki, wpisanie 'q' powoduje zakończenie programu.
123abcq
1
2
3
a
b
c
q
```

Wynik może wyglądać inaczej, niż niektórzy się spodziewali, ponieważ `System.in` jest domyślne buforowane wierszami. Oznacza to, że wejście trafia do programu dopiero po naciśnięciu klawisza *Enter*. Jak łatwo się domyślić, nie czyni to metody `read()` zbyt użyteczną w interaktywnym odczycie danych z klawiatury.

Odczyt łańcuchów

Do odczytania łańcucha z klawiatury służy metoda `readLine()` należąca do klasy `BufferedReader`. Ogólna postać metody jest następująca.

```
String readLine() throws IOException
```

Metoda zwraca obiekt klasy `String`.

Poniższy program ilustruje wykorzystanie metody `readLine()` klasy `BufferedReader`. Odczytuje i wyświetla przekazane wiersze tekstu aż do momentu wpisania słowa „stop”.

```
// R13\BRReadLines.java

// Odczyt łańcucha z konsoli za pomocą klasy BufferedReader
import java.io.*;

class BRReadLines {
  public static void main(String args[]) throws IOException
  {
```

```
    // Utworzenie BufferedReader na podstawie System.in
    BufferedReader br = new BufferedReader(new
                            InputStreamReader(System.in));
    String str;

    System.out.println("Wpisz wiersze tekstu.");
    System.out.println("Wpisz 'stop', aby wyjść.");
    do {
      str = br.readLine();
      System.out.println(str);
    } while(!str.equals("stop"));
  }
}
```

Kolejny przykład to bardzo prosty edytor tekstu. Tworzy tablicę obiektów `String`, a następnie odczytuje wiersze tekstu i zapamiętuje je w tablicy. Zakończenie wpisywania kończy się w momencie wpisania słowa „stop” lub zapamiętania 100 wierszy. Do odczytu danych z konsoli służy klasa `BufferedReader`.

```
// R13\TinyEdit.java

// Prosty edytor
import java.io.*;

class TinyEdit {
  public static void main(String args[])
    throws IOException
  {
    // Utworzenie BufferedReader na podstawie System.in
    BufferedReader br = new BufferedReader(new
                            InputStreamReader(System.in));
    String str[] = new String[100];
    System.out.println("Wpisz wiersze tekstu.");
    System.out.println("Wpisz 'stop', aby wyjść.");
    for(int i=0; i<100; i++) {
      str[i] = br.readLine();
      if(str[i].equals("stop")) break;
    }

    System.out.println("\nOto wpisane dane:");
    // Wyświetlenie tekstu
    for(int i=0; i<100; i++) {
      if(str[i].equals("stop")) break;
      System.out.println(str[i]);
    }
  }
}
```

Oto przykład wykonania programu.

```
Wpisz wiersze tekstu.
Wpisz 'stop', aby wyjść.
To jest pierwszy wiersz.
To jest drugi wiersz.
Java ułatwia korzystanie z łańcuchów.
Wystarczy tworzyć obiekty String.
stop
Oto wpisane dane:
To jest pierwszy wiersz.
To jest drugi wiersz.
Java ułatwia korzystanie z łańcuchów.
Wystarczy tworzyć obiekty String.
```

Wyświetlanie informacji na konsoli

Wyświetlanie informacji na konsoli najłatwiej wykonać, wywołując metodę `print()` lub `println()`. Metody te były już stosowane w wielu przykładach z niniejszej książki. Są one zdefiniowane w klasie `PrintStream` (jest to typ zmiennej referencyjnej `System.out`). Choć `System.out` to w zasadzie strumień bajtów, można go stosować do wyświetlania komunikatów. Alternatywa w postaci rozwiązania znakowego jest opisana w kolejnym podrozdziale.

Ponieważ `PrintStream` jest klasą pochodną klasy `OutputStream`, implementuje również niskopoziomową metodę `write()`. Służy ona do przekazania do konsoli pojedynczego bajta. Najprostsza wersja tej metody ma następującą postać.

```
void write(int wartośćBajta)
```

Metoda zapisuje do strumienia bajt podany jako argument *wartośćBajta*. Choć argument jest typu `int`, na wyjście przekazywany jest tak naprawdę tylko najniższy bajt wartości (osiem najmniej znaczących bitów). Poniższy prosty przykład używa metody `write()` do wyświetlenia na ekranie znaku „A” oraz przejścia do kolejnego wiersza.

```
// R13\WriteDemo.java

// Przykład użycia System.out.write()
class WriteDemo {
  public static void main(String args[]) {
    int b;

    b = 'A';
    System.out.write(b);
    System.out.write('\n');
  }
}
```

Bardzo rzadko korzysta się z metody `write()` do wyświetlania informacji na konsoli (choć w pewnych sytuacjach jest to przydatne). Metody `print()` i `println()` są w tej kwestii znacznie łatwiejsze w użyciu.

Klasa PrintWriter

Choć zastosowanie `System.out` do przekazywania informacji do konsoli jest poprawne, używa się go jedynie w celach testowych lub przy pisaniu przykładowych programów, takich jak w niniejszej książce. W aplikacjach rzeczywiście stosowanych przez użytkowników zaleca się stosowanie klasy `PrintWriter`, czyli jednej z klas znakowych. Takie rozwiązanie ułatwia umiędzynarodawianie oprogramowania.

Klasa `PrintWriter` definiuje kilka konstruktorów. Skorzystamy z następującej wersji.

```
PrintWriter(OutputStream strumieńWyjściowy, boolean wysłanieDla)
```

Parametr *strumieńWyjściowy* to obiekt typu `OutputStream`, natomiast parametr *wysłanieDla* informuje, czy Java ma przesyłać dane po każdym wywołaniu metody `println()` (lub kilku innych). Dla wartości `true` przesyłanie jest wykonywane automatycznie. Dla wartości `false` nie jest przeprowadzane.

Klasa `PrintWriter` udostępnia metody `print()` i `println()`. Wymienione metody działają dokładnie tak samo jak ich odpowiedniki dostępne dla obiektu `System.out`. Jeżeli argument nie jest typem prostym, metody te wywołują najpierw metodę `toString()` dla przekazanego obiektu i dopiero wtedy wyświetlają wynik.

Aby zapisywać dane do konsoli za pomocą klasy `PrintWriter`, wystarczy wskazać obiekt `System.out` jako strumień wyjściowy i użyć automatycznego wysyłania zawartości strumienia. Poniższy wiersz kodu tworzy obiekt `PrintWriter` połączony z konsolą.

```
PrintWriter pw = new PrintWriter(System.out, true);
```

Następujący program ilustruje wykorzystanie klasy `PrintWriter`.

```
// R13\PrintWriterDemo.java

// Przykład użycia klasy PrintWriter
import java.io.*;

public class PrintWriterDemo {
  public static void main(String args[]) {
    PrintWriter pw = new PrintWriter(System.out, true);

    pw.println("To jest łańcuch.");
    int i = -7;
    pw.println(i);
    double d = 4.5e-7;
    pw.println(d);
  }
}
```

Wynik działania programu jest następujący.

```
To jest łańcuch.
-7
4.5E-7
```

Pamiętaj, że nie ma nic złego w korzystaniu z `System.out` do wyświetlania na konsoli prostych tekstów lub danych testowych. Zaletą klasy `PrintWriter` jest możliwość łatwiejszego umiędzynarodawiania oprogramowania. Ponieważ klasa `PrintWriter` nie zapewnia żadnych dodatkowych zalet ponad standardowy strumień `System.out`, nie korzystam z niej w przedstawionych przykładach.

Odczyt i zapis plików

Java dostarcza wiele klas i metod związanych z odczytem i zapisem plików. Warto na początku podkreślić, że zagadnienia związane z plikowymi operacjami wejścia-wyjścia są dość złożone i zostaną szczegółowo omówione w części II. W tym podrozdziale ograniczymy się tylko do wprowadzenia podstawowych technik odczytywania i zapisywania danych. Mimo że będziemy posługiwali się strumieniami bajtowymi, opisane techniki można bez trudu dostosować także do strumieni znakowych.

Najczęściej stosuje się klasy `FileInputStream` i `FileOutputStream`, które tworzą strumienie bajtów powiązane z plikami. Aby otworzyć plik, wystarczy utworzyć egzemplarz jednej ze wspomnianych klas, jako argument konstruktora podając nazwę pliku. Choć obie klasy posiadają dodatkowe, przeciążone konstruktory, będziemy korzystali tylko z dwóch przedstawionych poniżej postaci.

```
FileInputStream(String nazwaPliku) throws FileNotFoundException
FileOutputStream(String nazwaPliku) throws FileNotFoundException
```

Parametr *nazwaPliku* to nazwa otwieranego pliku. Jeśli tworzy się strumień wejściowy, a podany plik nie istnieje, konstruktor zgłasza wyjątek `FileNotFoundException`. Jeśli tworzy się strumień wyjściowy i podany plik nie może zostać utworzony, konstruktor zgłasza wyjątek `FileNotFoundException`. Gdy plik podany w strumieniu wyjściowym jest otwierany, wcześniejszy plik o tej samej nazwie jest niszczony.

W razie stosowania menedżera zabezpieczeń wiele spośród klas operujących na plikach, w tym klasy `FileInputStream` i `FileOutputStream`, zgłasza wyjątek `SecurityException` w odpowiedzi na każdą próbę naruszenia zabezpieczeń podczas otwierania pliku. Ponieważ jednak aplikacje uruchamiane za pomocą polecenia *java* domyślnie nie stosują menedżera zabezpieczeń, prezentowane w tej książce przykłady operacji wejścia-wyjścia nie muszą przechwytywać ani obsługiwać wyjątku `SecurityException`. Warto jednak pamiętać, że pozostałe typy aplikacji mogą używać menedżera zabezpieczeń i muszą być przygotowane na generowanie wyjątku `SecurityException` przez operacje wejścia-wyjścia — w tych aplikacjach koniecznie należy zaimplementować właściwą obsługę tego wyjątku.

Po zakończeniu działań na pliku należy go zamknąć za pomocą metody `close()`. Jest ona zdefiniowana w klasach `FileInputStream` i `FileOutputStream` w następujący sposób.

```
void close() throws IOException
```

Zamknięcie pliku zwalnia związane z nim zasoby i umożliwia wykorzystanie tych zasobów na potrzeby operacji na innym pliku. Brak wywołania zamykającego plik może skutkować tzw. wyciekami pamięci (ang. *memory leaks*), ponieważ nieużywane zasoby nie są zwalniane.

Począwszy od wersji JDK 7, metoda `close()` należy do interfejsu `AutoCloseable` wchodzącego w skład pakietu `java.lang`. Interfejs `AutoCloseable` dziedziczy po interfejsie `Closeable` z pakietu `java.io`. Oba te interfejsy są implementowane przez klasy strumieni, w tym `FileInputStream` i `FileOutputStream`.

Zanim przystąpimy do analizy kolejnych zagadnień, warto zwrócić uwagę na istnienie dwóch podstawowych technik zamykania plików po zakończeniu pracy. Pierwszym, tradycyjnym rozwiązaniem jest wywołanie wprost metody `close()` dla pliku, który nie jest już potrzebny. Właśnie to rozwiązanie obowiązywało we wszystkich wersjach sprzed wydania JDK 7 i jako takie występuje w całym wcześniejszym kodzie Javy. Druga technika polega na użyciu instrukcji `try`-*with-resources* dodanego w wersji JDK 7 — instrukcja ta automatycznie zamyka plik, który nie jest już potrzebny. W tym modelu nie jest konieczne bezpośrednie wywołanie metody `close()`. Ponieważ wciąż można spotkać kod opracowany przed wydaniem wersji JDK 7, który jeszcze jest stosowany i podlega konserwacji, warto znać i dobrze rozumieć to tradycyjne rozwiązanie. Naszą analizę rozpoczniemy właśnie od tego modelu; do zautomatyzowanego zamykania plików wrócimy w kolejnym podrozdziale.

Do odczytania danych z pliku może służyć wersja metody `read()` zdefiniowana w klasie `File`
`InputStream`. Jej postać używana w tym rozdziale jest następująca.

```
int read() throws IOException
```

Każde jej wywołanie powoduje odczytanie z pliku pojedynczego bajta i zwrócenie go jako wartość `int`. Metoda zwraca wartość –1, gdy zostanie podjęta próba odczytu po przekroczeniu końca pliku. Co więcej, metoda może zwrócić wyjątek `IOException`.

Przedstawiony poniżej program używa metody `read()` do odczytania i wyświetlenia zawartości pliku tekstowego zawierającego tekst ASCII. Nazwę pliku należy przekazać w formie argumentu wiersza poleceń.

```
// R13\ShowFile.java

/* Wyświetla zawartość pliku tekstowego

   Aby użyć programu, określ nazwę
   pliku do wyświetlenia
   Na przykład wyświetlenie pliku o nazwie TEST.TXT
   wymaga wpisania poniższego wiersza

   java ShowFile TEST.TXT
*/

import java.io.*;

class ShowFile {
  public static void main(String args[])
  {
    int i;
    FileInputStream fin;

    // Sprawdza, czy podano nazwę pliku
    if(args.length != 1) {
      System.out.println("Sposób użycia: ShowFile nazwa-pliku");
      return;
    }

    // Próba otwarcia pliku
```

```
    try {
      fin = new FileInputStream(args[0]);
    } catch(FileNotFoundException e) {
      System.out.println("Nie można otworzyć pliku ");
      return;
    }

    // Na tym etapie plik jest otwarty i gotowy do odczytu
    // Poniższy kod odczytuje znaki do osiągnięcia końca pliku
    try {
      do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
      } while(i != -1);
    } catch(IOException e) {
      System.out.println("Błąd odczytu pliku ");
    }

    // Zamyka plik
    try {
      fin.close();
    } catch(IOException e) {
      System.out.println("Błąd zamykania pliku ");
    }
  }
}
```

Warto zwrócić uwagę na blok `try-catch` w powyższym programie. Zadaniem tego bloku jest obsługa ewentualnych błędów operacji wejścia-wyjścia. Każda taka operacja jest monitorowana pod kątem wyjątków; jeśli wystąpi jakiś wyjątek, zostanie prawidłowo obsłużony. O ile w prostych programach czy przykładowym kodzie (jak w dotychczasowych przykładach) zgłaszanie wyjątków operacji wejścia-wyjścia przez metodę `main()` jest uzasadnione, o tyle w rzeczywistych aplikacjach wyjątki należy propagować do kodu wywołującego, tak aby dysponował informacjami o niepowodzeniu żądanych operacji. Większość prezentowanych tutaj przykładów ilustrujących operacje wejścia-wyjścia na plikach wprost obsługuje wszystkie potencjalne wyjątki.

Mimo że w powyższym przykładzie strumień został zamknięty dopiero po odczytaniu zawartości pliku, w wielu sytuacjach stosuje się nieco inne rozwiązanie. Alternatywnym sposobem realizacji tego zadania jest wywołanie metody `close()` w ramach bloku `finally`. W tym modelu wszystkie metody uzyskujące dostęp do pliku powinny znajdować się w bloku `try`, a blok `finally` powinien służyć do zamknięcia tego pliku. Oznacza to, że niezależnie od tego, jak zakończy się wykonywanie bloku `try`, plik zostanie ostatecznie zamknięty. Poniżej przedstawiono nową wersję bloku `try` z poprzedniego przykładu, w której zaimplementowano opisany mechanizm:

```
try {
  do {
    i = fin.read();
    if(i != -1) System.out.print((char) i);
  } while(i != -1);
} catch(IOException e) {
  System.out.println("Błąd odczytu pliku");
} finally {
  // Zamyka plik zawsze po zakończeniu bloku try
  try {
    fin.close();
  } catch(IOException e) {
    System.out.println("Błąd zamykania pliku");
  }
}
```

Jedną z zalet tego rozwiązania jest gwarancja zamknięcia pliku w bloku `finally`, nawet jeśli wykonywanie kodu uzyskującego dostęp do pliku zostanie przerwane wskutek wyjątku niezwiązanego z operacjami wejścia-wyjścia.

W pewnych sytuacjach prostszym rozwiązaniem jest umieszczenie wyrażeń otwierających plik i uzyskujących dostęp do tego pliku w jednym bloku `try` (zamiast w dwóch osobnych blokach), po którym następuje blok `finally` odpowiedzialny za zamknięcie niepotrzebnego pliku. Odpowiednio zmodyfikowaną wersję programu `ShowFile` pokazano poniżej:

```
// R13\ShowFile2.java

/* Wyświetla zawartość pliku tekstowego

   Podczas uruchamiania programu podaj nazwę
   pliku do wyświetlenia
   Na przykład wyświetlenie pliku o nazwie TEST.TXT
   wymaga wpisania poniższego wiersza

   java ShowFile2 TEST.TXT

   Ta wersja programu umieszcza kod otwierający plik i
   uzyskujący dostęp do jego zawartości w jednym bloku
   try, plik jest zamykany w bloku finally
*/

import java.io.*;

class ShowFile2 {
  public static void main(String args[])
  {
    int i;
    FileInputStream fin = null;

    // Sprawdza, czy podano nazwę pliku
    if(args.length != 1) {
      System.out.println("Sposób użycia: ShowFile2 nazwa-pliku");
      return;
    }

    // Poniższy kod otwiera plik, odczytuje jego zawartość do momentu
    // osiągnięcia końca pliku, po czym zamyka plik w bloku finally
    try {
      fin = new FileInputStream(args[0]);

      do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
      } while(i != -1);

    } catch(FileNotFoundException e) {
      System.out.println("Nie znaleziono pliku");
    } catch(IOException e) {
      System.out.println("Wystąpił błąd wejścia-wyjścia");
    } finally {
      // Zamyka plik niezależnie od sytuacji
      try {
        if(fin != null) fin.close();
      } catch(IOException e) {
        System.out.println("Błąd zamykania pliku");
      }
    }
  }
}
```

Warto zwrócić uwagę na sposób inicjalizacji zmiennej `fin`, której początkowo przypisano wartość `null`. Nieco później, już w bloku `finally` plik jest zamykany, pod warunkiem że zmienna `fin` jest różna od `null`. Takie rozwiązanie jest uzasadnione, ponieważ zmienna `fin` jest różna od `null` tylko wtedy, gdy plik został prawidłowo otwarty. Oznacza to, że metoda `close()` nie zostanie wywołana, jeśli w czasie otwierania pliku wystąpił jakiś wyjątek.

Sekwencję try-catch z powyższego przykładu można zapisać w bardziej zwięzłej formie. Ponieważ FileNotFoundException jest klasą pochodną klasy IOException, osobne przechwytywanie wyjątków obu typów nie jest potrzebne. Poniżej pokazano zmienioną sekwencję try-catch, z której wyeliminowano przechwytywanie samego wyjątku FileNotFoundException. W razie wystąpienia jakiegokolwiek wyjątku typu IOException (i jego podtypów) zostanie wyświetlony standardowy komunikat z opisem błędu.

```
try {
  fin = new FileInputStream(args[0]);

  do {
    i = fin.read();
    if(i != -1) System.out.print((char) i);
  } while(i != -1);
} catch(IOException e) {
  System.out.println("Błąd wejścia-wyjścia: " + e);
} finally {
  // Zamyka plik niezależnie od sytuacji
  try {
    if(fin != null) fin.close();
  } catch(IOException e) {
    System.out.println("Błąd zamykania pliku");
  }
}
```

W tym rozwiązaniu wszystkie błędy, w tym błąd podczas otwierania pliku, są obsługiwane przez jedno wyrażenie catch. Z uwagi na prostotę proponowanej konstrukcji właśnie ona będzie stosowana w wielu przykładach stosowania podsystemu wejścia-wyjścia. Warto jednak pamiętać, że powyższe rozwiązanie nie zdaje egzaminu w sytuacji, gdy chcemy inaczej reagować na błędy podczas otwierania plików, na przykład wskutek nieprawidłowego wpisania nazwy pliku. W takim przypadku przed przekazaniem sterowania do bloku try uzyskującego dostęp do tego pliku można na przykład poprosić użytkownika o wpisanie prawidłowej nazwy.

Do zapisu danych do pliku służy metoda write() zdefiniowana w klasie FileOutputStream. Jej najprostsza postać jest następująca.

```
void write(int wartośćBajta) throws IOException
```

Metoda zapisuje *wartośćBajta* do pliku. Choć przekazuje się wartość typu int, do pliku zapisywanych jest tylko 8 najmniej znaczących bitów liczby. Gdy w trakcie zapisu wystąpi błąd, zostaje zgłoszony wyjątek IOException. Kolejny przykład używa metody write() do skopiowania pliku tekstowego.

```
// R13\CopyFile.java

/* Kopiowanie plików tekstowych

   Aby użyć programu, należy wpisać nazwę
   pliku źródłowego i docelowego
   Na przykład aby skopiować plik FIRST.TXT
   do pliku SECOND.TXT, trzeba wpisać poniższe
   polecenie w wierszu poleceń

   java CopyFile FIRST.TXT SECOND.TXT
*/

import java.io.*;

class CopyFile {
  public static void main(String args[]) throws IOException
  {
    int i;
    FileInputStream fin = null;
    FileOutputStream fout = null;

    // Najpierw sprawdza, czy podano nazwy obu plików
```

```
    if(args.length != 2) {
      System.out.println("Sposób użycia: CopyFile źródło cel");
      return;
    }

    // Kopiuje plik
    try {
      // Próba otwarcia plików
      fin = new FileInputStream(args[0]);
      fout = new FileOutputStream(args[1]);

      do {
        i = fin.read();
        if(i != -1) fout.write(i);
      } while(i != -1);

    } catch(IOException e) {
      System.out.println("Błąd wejścia-wyjścia: " + e);
    } finally {
      try {
        if(fin != null) fin.close();
      } catch(IOException e2) {
        System.out.println("Błąd zamykania pliku źródłowego");
      }
      try {
        if(fout != null) fout.close();
      } catch(IOException e2) {
        System.out.println("Błąd zamykania pliku docelowego");
      }
    }
  }
}
```

W powyższym kodzie warto zwrócić szczególną uwagę na dwa odrębne bloki `try` używane podczas zamykania plików. W ten sposób gwarantujemy, że oba pliki zostaną zamknięte, nawet jeśli wywołanie `fin.close()` zgłosi jakiś wyjątek.

Warto zwrócić uwagę także na sposób obsługi potencjalnych błędów operacji wejścia-wyjścia w tym i poprzednim programie. Różni się on od rozwiązań stosowanych w innych językach, w których informacje o błędach są zwracane przy użyciu kodów. Rozwiązanie zastosowane w Javie nie tylko upraszcza obsługę plików, lecz także pozwala na łatwe rozróżnienie osiągnięcia końca pliku od wystąpienia jakichś błędów.

Automatyczne zamykanie pliku

Przykładowe programy prezentowane w poprzednim podrozdziale zawierały bezpośrednie wywołania metody `close()` zamykającej pliki, które nie były już potrzebne. Jak już wspomniano, ten sposób zamykania plików stosowano w wersjach Javy sprzed wydania JDK 7. Mimo że opisany model wciąż jest prawidłowy i przydatny, w wersji JDK 7 dodano rozwiązanie umożliwiające nieco inne zarządzanie zasobami, w tym strumieniami plikowymi. W nowym modelu proces zamykania zasobów jest automatyczny. Mechanizm nazywany czasem **automatycznym zarządzaniem zasobami** (ang. *automatic resource management — ARM*) zaimplementowano na bazie rozszerzonej wersji wyrażenia `try`. Największą zaletą automatycznego zarządzania zasobami jest brak ryzyka przypadkowego pozostawienia otwartego pliku (lub innego zasobu), mimo że nie jest już potrzebny. Jak już wspomniano, brak konsekwentnego zamykania plików może skutkować zjawiskiem wycieku pamięci i prowadzić do innych poważnych problemów.

Automatyczne zarządzanie zasobami zaimplementowano na bazie rozszerzonej formy wyrażenia `try`. Ogólną postać tego wyrażenia pokazano poniżej:

```
try (specyfikacja-zasobu) {
  // Użycie zasobu
}
```

Element *specyfikacja-zasobu* zazwyczaj reprezentuje wyrażenie deklarujące i inicjalizujące jakiś zasób, na przykład strumień plikowy. Wyrażenie składa się z deklaracji i inicjalizacji zmiennej przy użyciu referencji do obiektu, który ma być przedmiotem zarządzania. Po zakończeniu wykonywania bloku `try` tak zadeklarowany i zainicjalizowany zasób zostanie automatycznie zwolniony. W przypadku pliku automatyczne zwolnienie jest równoznaczne z zamknięciem pliku. (Oznacza to, że bezpośrednie wywołanie metody `close()` jest zbędne). Nowa forma wyrażenia `try` może oczywiście obejmować klauzule `catch` i `finally`. Ta forma wyrażenia `try` bywa nazywana `try`-*with-resources*.

Począwszy od JDK 9 specyfikacja zasobu używana w instrukcji `try`-*with-resources* może także mieć postać zmiennej zadeklarowanej i zainicjalizowanej we wcześniejszej części programu. Jednak musi to być zmienna **praktycznie finalna**, czyli po określeniu wartości początkowej wartość tej zmiennej nie uległa zmianie.

Wyrażenie `try`-*with-resources* można stosować tylko dla zasobów implementujących interfejs `AutoCloseable` zdefiniowany w pakiecie `java.lang`. Wspomniany interfejs definiuje metodę `close()`. Po interfejsie `AutoCloseable` dziedziczy interfejs `Closeable` wchodzący w skład pakietu `java.io`. Oba interfejsy są implementowane przez klasy strumieni. Oznacza to, że wyrażenia `try`-*with-resources* można z powodzeniem stosować w operacjach na strumieniach, w tym na strumieniach plikowych.

W roli pierwszego przykładu automatycznego zamykania plików wykorzystamy przebudowaną wersję programu `ShowFile`:

```
// R13\ShowFile3.java

/* Ta wersja programu ShowFile używa wyrażenia try-with-resources
   do automatycznego zamknięcia pliku, który nie jest już potrzebny
*/

import java.io.*;

class ShowFile3 {
  public static void main(String args[])
  {
    int i;

    // Sprawdza, czy podano nazwę pliku
    if(args.length != 1) {
      System.out.println("Sposób użycia: ShowFile3 nazwa-pliku");
      return;
    }

    // Poniższy kod używa wyrażenia try-with-resources do otwarcia pliku, po czym
    // automatycznie zamyka ten plik w momencie zakończenia bloku try
    try(FileInputStream fin = new FileInputStream(args[0])) {

      do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
      } while(i != -1);

    } catch(FileNotFoundException e) {
      System.out.println("Nie znaleziono pliku.");
    } catch(IOException e) {
      System.out.println("Wystąpił błąd wejścia-wyjścia ");
    }

  }
}
```

W powyższym programie należy zwrócić uwagę na sposób otwierania pliku w ramach wyrażenia `try`:

```
try(FileInputStream fin = new FileInputStream(args[0])) {
```

Szczególnie interesująca jest część specyfikacji zasobu, w której zadeklarowano obiekt klasy `FileInputStream` nazwany `fin`. Przypisano mu referencję do pliku otwartego przez konstruktor klasy `FileInputStream`. Oznacza to, że w tej wersji programu zmienna `fin` ma zasięg lokalny względem bloku `try`, ponieważ została utworzona w momencie wejścia sterowania do tego bloku. Po opuszczeniu bloku `try` łańcuch skojarzony ze zmienną `fin` zostanie automatycznie zamknięty za pomocą niejawnego wywołania metody `close()`. Skoro nie musimy wprost wywoływać metody `close()`, siłą rzeczy nie możemy zapomnieć o tym wywołaniu. To jedna z największych zalet instrukcji `try`-*with-resources*.

Warto mieć na uwadze, że zasób zadeklarowany w wyrażeniu `try` jest niejawnie traktowany jako zmienna finalna. Oznacza to, że po tej deklaracji nie można przypisać użytej zmiennej żadnej innej wartości. Co więcej, zasięg tego zasobu jest ograniczony do wyrażenia `try`-*with-resources*.

W ramach jednego wyrażenia `try` można zarządzać wieloma zasobami. Wystarczy oddzielić specyfikacje poszczególnych zasobów znakiem średnika. Przykładem zastosowania takiej konstrukcji jest poniższy program. Ten przebudowany program `CopyFile` zawiera pojedyncze wyrażenie `try`-*with-resources* zarządzające zarówno strumieniem `fin`, jak i strumieniem `fout`.

```
// R13\CopyFile2.java

/* Wersja programu CopyFile korzystająca z instrukcji try-with-resources
   Program demonstruje sposób zarządzania dwoma zasobami (w tym przypadku
   plikami) przez pojedyncze wyrażenie try
*/

import java.io.*;

class CopyFile2 {
  public static void main(String args[]) throws IOException
  {
    int i;

    // Najpierw sprawdza, czy podano nazwy obu plików
    if(args.length != 2) {
      System.out.println("Sposób użycia: CopyFile2 źródło cel");
      return;
    }

    // Otwiera dwa pliki i zarządza nimi w ramach wyrażenia try
    try (FileInputStream fin = new FileInputStream(args[0]);
         FileOutputStream fout = new FileOutputStream(args[1]))
    {

      do {
        i = fin.read();
        if(i != -1) fout.write(i);
      } while(i != -1);

    } catch(IOException e) {
      System.out.println("Błąd wejścia-wyjścia: " + e);
    }
  }
}
```

W powyższym programie szczególnie interesujący jest sposób otwierania plików wejściowego i wyjściowego w ramach bloku `try`:

```
try (FileInputStream fin = new FileInputStream(args[0]);
     FileOutputStream fout = new FileOutputStream(args[1]))
{
  // ...
```

Po zakończeniu tego bloku oba pliki (reprezentowane przez `fin` i `fout`) zostaną automatycznie zamknięte. Wystarczy porównać tę wersję programu z poprzednią, aby przekonać się, że nowa instrukcja pozwala znacznie skrócić kod operujący na zasobach. Możliwość skrócenia i poprawy czytelności kodu to jedna z niewątpliwych zalet automatycznego zarządzania zasobami.

Warto wspomnieć o jeszcze jednym ważnym aspekcie stosowania instrukcji try-*with-resources*. Ogólnie podczas wykonywania bloku try nie można wykluczyć sytuacji, że wyjątek zgłoszony w ramach tego bloku będzie powodował inny wyjątek występujący podczas zamykania odpowiedniego zasobu w klauzuli finally. W przypadku tradycyjnego wyrażenia try oryginalny wyjątek jest gubiony i zastępowany przez wyjątek zgłoszony jako drugi. Okazuje się, że w przypadku zastosowania instrukcji try-*with-resources* to drugi wyjątek jest ukrywany. Wyjątek nie jest jednak gubiony — trafia na listę wyjątków skojarzonych z pierwszym wyjątkiem. Listę tych wyjątków można uzyskać za pośrednictwem metody getSuppressed() zdefiniowanej w klasie Throwable.

Z uwagi na zalety wyrażenia try-*with-resources* właśnie ta forma będzie stosowana w wielu, choć nie we wszystkich, przykładowych programach w tym wydaniu książki. W niektórych przykładach będziemy stosowali tradycyjny sposób zamykania zasobów. Decyzja o pozostawieniu części przykładów niezmienionych wynika po części z tego, że wciąż istnieje bardzo dużo kodu stosującego tradycyjny model zarządzania zasobami. W tej sytuacji każdy programista Javy powinien dobrze rozumieć dotychczasowe rozwiązania i biegle nimi operować (choćby na potrzeby konserwacji istniejącego kodu). Co więcej, może się zdarzyć, że niektórzy programiści będą jeszcze przez pewien czas pracować w środowiskach sprzed wydania JDK 7. W takich przypadkach rozszerzona forma instrukcji try jest po prostu niedostępna. I wreszcie nie można wykluczyć sytuacji, w której jawne, bezpośrednie zamknięcie zasobu okaże się lepszym rozwiązaniem niż poleganie na automatycznych mechanizmach. Właśnie dlatego część przykładów w tej książce nadal będzie stosowała tradycyjny model polegający na ręcznym wywołaniu metody close(). Ta część przykładów nie tylko ilustruje tradycyjną technikę zarządzania zasobami, ale też może być kompilowana i uruchamiana przez czytelników we wszystkich, także starszych środowiskach.

Część przykładów w tej książce stosuje tradycyjny model zamykania plików, aby przyzwyczaić czytelników do tej wciąż powszechnej techniki. W nowym kodzie należy jednak stosować zautomatyzowany model w formie instrukcji try-*with-resources*.

Modyfikatory transient i volatile

Java definiuje dwa interesujące modyfikatory typu: transient i volatile. Stosuje się je na ogół w dosyć specyficznych sytuacjach.

Gdy zmienną składową zadeklaruje się jako transient, jej wartość nie zostanie utrwalona przy zapisie obiektu na dysku. Oto przykład.

```
class T {
  transient int a; // Nie jest trwała
  int b; // Jest trwała
}
```

Jeśli obiekt T zostanie zapisany w trwałym magazynie danych, zawartość zmiennej a nie zostanie zachowana, natomiast wartość zmiennej b zostanie zapamiętana.

Modyfikator volatile informuje kompilator, iż oznaczona nim zmienna może zostać nieoczekiwanie zmieniona przez inną część programu. Taka sytuacja często zachodzi w przypadku programowania wielowątkowego. W środowisku wielowątkowym często dwa lub więcej wątków współdzieli tę samą zmienną składową. Ze względu na wydajność każdy wątek może przechowywać własną, prywatną kopię współdzielonej zmiennej. Rzeczywista (**macierzysta**) kopia zmiennej jest uaktualniania tylko co jakiś czas, na przykład w momencie rozpoczynania metody typu synchronized. Choć takie rozwiązanie działa, nie zawsze jest poprawne. Czasem macierzysta kopia powinna cały czas przechowywać aktualną wartość. Aby to zapewnić, wystarczy użyć modyfikatora volatile, który informuje kompilator o tym, iż musi on zawsze stosować wersję macierzystą zmiennej (lub przynajmniej stale aktualizować jej kopie i na odwrót). Co więcej, dostęp do współdzielonej zmiennej musi być wykonywany w ściśle określonej kolejności podanej przez program.

Operator instanceof

Czasem zachodzi potrzeba uzyskania typu obiektu w trakcie działania programu. Możemy na przykład dysponować wątkiem generującym obiekty różnych typów. Aby przetwarzać je poprawnie, warto byłoby znać typ każdego z otrzymanych obiektów. Inną ważną sytuacją wymagającą sprawdzenia typu obiektu jest rzutowanie. W Javie rzutowanie na niepoprawny typ powoduje zgłoszenie wyjątku wykonywania. Wiele błędnych rzutowań udaje się wychwycić na etapie kompilacji, ale rzutowania związane z hierarchią klas mogą być sprawdzone tylko w trakcie działania programu. Na przykład klasa bazowa A ma dwie klasy pochodne B i C. Oznacza to, iż rzutowanie obiektu B na typ A lub rzutowanie obiektu C na typ A jest w pełni poprawne, ale rzutowanie obiektu B na typ C (i na odwrót) jest błędem. Skoro zmienna referencyjna typu A może przechowywać referencję do obiektu B lub C, jak bez rzutowania na typ C sprawdzić rzeczywisty typ wskazywanego obiektu? Nic nie stoi na przeszkodzie, by był to obiekt typu A, B lub C. Jeśli znajduje się tam obiekt typu B, próba rzutowania na typ C zakończy się zgłoszeniem wyjątku. Na szczęście Java posiada operator `instanceof`, dzięki któremu łatwo sprawdzić rzeczywisty typ obiektu.

Ogólna postać operatora `instanceof` jest następująca.

obiekt `instanceof` *typ*

Element *`obiekt`* to egzemplarz klasy, natomiast element *`typ`* to typ klasy. Jeśli obiekt jest wskazanego typu lub może być na niego rzutowany, operator `instanceof` zwraca wartość `true`. W przeciwnym razie zwraca wartość `false`. Oznacza to, iż operator ten zapewnia bezpieczny i wygodny sposób sprawdzania typu obiektu w trakcie działania programu.

Oto program wykorzystujący operator `instanceof`.

// R13\InstanceOf.java

```
// Przykład użycia operatora instanceof
class A {
  int i, j;
}

class B {
  int i, j;
}

class C extends A {
  int k;
}

class D extends A {
  int k;
}

class InstanceOf {
  public static void main(String args[]) {
    A a = new A();
    B b = new B();
    C c = new C();
    D d = new D();

    if(a instanceof A)
      System.out.println("a to egzemplarz A");
    if(b instanceof B)
      System.out.println("b to egzemplarz B");
    if(c instanceof C)
      System.out.println("c to egzemplarz C");
    if(c instanceof A)
      System.out.println("c może być rzutowany na A");
```

```
    if(a instanceof C)
      System.out.println("a może być rzutowany na C");

    System.out.println();

    // Porównanie typów pochodnych
    A ob;

    ob = d; // A zawiera referencję do d
    System.out.println("ob zawiera referencję do d");
    if(ob instanceof D)
      System.out.println("ob to egzemplarz D");

    System.out.println();

    ob = c; // A zawiera referencję do c
    System.out.println("ob zawiera referencję do c");

    if(ob instanceof D)
      System.out.println("ob może być rzutowany na D");
    else
      System.out.println("ob nie może być rzutowany na D");

    if(ob instanceof A)
      System.out.println("ob może być rzutowany na A");

    System.out.println();

    // Wszystkie obiekty można rzutować na Object
    if(a instanceof Object)
      System.out.println("a może być rzutowany na Object");
    if(b instanceof Object)
      System.out.println("b może być rzutowany na Object");
    if(c instanceof Object)
      System.out.println("c może być rzutowany na Object");
    if(d instanceof Object)
      System.out.println("d może być rzutowany na Object");
  }
}
```

Wynik działania programu jest następujący.

```
a to egzemplarz A
b to egzemplarz B
c to egzemplarz C
c może być rzutowany na A

ob zawiera referencję do d
ob to egzemplarz D

ob zawiera referencję do c
ob nie może być rzutowany na D
ob może być rzutowany na A

a może być rzutowany na Object
b może być rzutowany na Object
c może być rzutowany na Object
d może być rzutowany na Object
```

Większość programów nie musi stosować operatora `instanceof`, ponieważ programiści zwykle wiedzą, którym typem posługują się w swoim kodzie. Operator staje się niezwykle przydatny wtedy, gdy pisze się ogólne procedury działające na obiektach złożonej hierarchii klas.

Modyfikator strictfp

Wraz z pojawieniem się kilka lat temu Javy 2 model obliczeń zmiennoprzecinkowych stał się mniej restrykcyjny. Nowy model nie wymaga na przykład obcinania pewnych wartości pośrednich występujących w trakcie obliczeń. Obcinanie tych wartości w pewnych przypadkach zapobiega przepełnieniom lub niedomiarom (niedopełnieniom). Zastosowanie modyfikatora `strictfp` dla klasy lub metody wymusza, aby wszystkie przeprowadzane w niej obliczenia zmiennoprzecinkowe były przeprowadzane tak samo, jak w pierwszych wersjach Javy. Gdy modyfikator zostanie użyty dla klasy, wszystkie metody klasy są nim oznaczane automatycznie.

Poniższy kod informuje Javę, aby używała oryginalnego modelu obliczeń zmiennoprzecinkowych dla wszystkich metod zdefiniowanych w klasie `MyClass`.

```
strictfp class MyClass { //...
```

Zdecydowana większość programistów nigdy nie musi korzystać z modyfikatora `strictfp`, ponieważ jego wpływ ogranicza się do bardzo wąskiej kategorii obliczeń.

Metody napisane w kodzie rdzennym

Bardzo rzadko występuje potrzeba wywołania podprogramu napisanego w innym języku niż Java. Tego rodzaju podprogramy zwykle zawierają kod wykonywalny przystosowany do konkretnego procesora i środowiska — innymi słowy, jest to tzw. kod rdzenny (ang. *native code*). Taki kod z reguły stosuje się wtedy, gdy pewne obliczenia trzeba wykonywać niezwykle szybko lub gdy należy skorzystać z wyspecjalizowanej biblioteki innej firmy, na przykład pakietu statystycznego. Ponieważ programy Javy są kompilowane do kodu bajtowego, a ten jest interpretowany (lub kompilowany „w locie") przez system wykonawczy Javy, niektórym osobom może się wydawać, iż wywołanie funkcji kodu rdzennego nie jest możliwe. Nie jest to prawdą. Java zawiera słowo kluczowe `native`, które służy do deklarowania metod napisanych w kodzie rdzennym. Po takiej deklaracji można wywoływać te metody w taki sam sposób jak pozostałe metody Javy. Mechanizm używany do integrowania kodu rdzennego z programami pisanymi w Javie nosi nazwę *Java Native Interface* (w skrócie *JNI*).

W celu deklaracji metody kodu rdzennego należy poprzedzić nazwę metody modyfikatorem `native` i nie określać żadnego ciała metody. Oto przykład.

```
public native int meth();
```

Po takiej deklaracji metody należy jeszcze napisać metodę rdzenną i wykonać dość złożony ciąg kroków, aby dołączyć taką metodę do kodu Javy. Wszelkie szczegóły można znaleźć w dokumentacji Javy.

Stosowanie asercji

Kolejnym interesującym słowem kluczowym Javy jest `assert`. W trakcie tworzenia i testowania programów używa się go do **asercji**, czyli sprawdzania, czy dany warunek jest spełniony w trakcie działania programu. Na przykład jedna z metod zawsze powinna zwrócić wartość dodatnią. Do sprawdzenia poprawności tej wartości można użyć asercji — jeśli metoda zawsze zwraca liczbę większą od 0, nie zostaną podjęte żadne dodatkowe działania. Jeśli jednak warunek nie zostanie spełniony, Java zgłosi wyjątek `AssertionError`. Asercji używa się najczęściej w trakcie testowania programów w celu sprawdzenia, czy spełnione są pewne założenia. Na ogół nie stosuje się ich w wynikowym kodzie dostarczanym klientowi.

Słowo kluczowe `assert` ma dwie postacie. Oto pierwsza z nich.

```
assert warunek;
```

Element *`warunek`* to wyrażenie, które musi dać w wyniku wartość typu `boolean`. Jeśli warunek zwróci wartość `true`, asercja jest spełniona i program działa dalej bez najmniejszych przeszkód. W przeciwnym razie asercja nie zostaje spełniona i jest zgłaszany wyjątek `AssertionError`.

Druga postać asercji jest następująca.

```
assert warunek: wyrażenie;
```

W tej wersji element *wyrażenie* to wartość przekazywana do konstruktora wyjątku `Assertion ↪Error`. Przekazana wartość jest konwertowana na łańcuch i wyświetlona w razie niespełnienia asercji. W większości przypadków jako *wyrażenie* stosuje się gotowy łańcuch komunikatu, ale dopuszczalne są dowolne wyrażenia różne od `void`, o ile można dokonać ich konwersji na łańcuch.

Poniżej znajduje się przykład użycia słowa kluczowego `assert`. Asercja sprawdza, czy wartość zwrócona przez metodę `getnum()` jest dodatnia.

```
// R13\AssertDemo.java

// Przykład asercji
class AssertDemo {
  static int val = 3;

  // Zwraca wartość całkowitoliczbową
  static int getnum() {
    return val--;
  }

  public static void main(String args[])
  {
    int n;

    for(int i=0; i < 10; i++) {
      n = getnum();

      assert n > 0; // Spowoduje zgłoszenie wyjątku dla n równego 0

      System.out.println("n wynosi " + n);
    }
  }
}
```

Aby włączyć sprawdzanie asercji w trakcie działania programu, trzeba użyć opcji `-ea`. Aby włączyć asercje dla programu `AssertDemo`, należy użyć następującego polecenia.

```
java -ea AssertDemo
```

Po kompilacji i uruchomieniu programu w opisany sposób na konsoli pojawią się następujące komunikaty.

```
n wynosi 3
n wynosi 2
n wynosi 1
Exception in thread "main" java.lang.AssertionError
        at AssertDemo.main(AssertDemo.java:17)
```

Metoda `main()` wielokrotnie wywołuje metodę `getnum()`, która zwraca wartość całkowitoliczbową. Zwrócona wartość jest najpierw przypisywana do zmiennej `n`, a następnie testowana za pomocą następującej asercji.

```
assert n > 0; // Spowoduje zgłoszenie wyjątku dla n równego 0
```

Asercja nie będzie poprawna, gdy zmienna `n` będzie zawierała wartość 0 (w tym przypadku będzie to sytuacja po czwartym wywołaniu metody `getnum()`). Spowoduje to zgłoszenie wyjątku.

Zgodnie z wcześniejszym opisem można określić komunikat wyświetlany w momencie niespełnienia asercji. Po zastąpieniu wcześniejszego wiersza poniższym:

```
assert n > 0: "n nie jest dodatnie!";
```

i ponownym skompilowaniu programu wynik jego działania byłby następujący.

```
n wynosi 3
n wynosi 2
n wynosi 1
Exception in thread "main" java.lang.AssertionError: n nie jest dodatnie!
        at AssertDemo.main(AssertDemo.java:17)
```

Bardzo ważna uwaga: nie należy polegać na asercjach w kwestii wykonywania istotnych działań wymaganych przez program. Wynika to z faktu, iż kod udostępniany użytkownikom zapewne będzie miał wyłączone asercje. Rozważmy następującą odmianę poprzedniego programu.

```
// R13\AssertDemo2.java

// Bardzo zły sposób stosowania asercji!!!
class AssertDemo2 {
  // Ustalenie wartości początkowej
  static int val = 3;

  // Zwrócenie liczby całkowitej
  static int getnum() {
    return val--;
  }

  public static void main(String args[])
  {
    int n = 0;

    for(int i=0; i < 10; i++) {

      assert (n = getnum()) > 0; // To na pewno nie jest dobry pomysł!

      System.out.println("n wynosi " + n);
    }
  }
}
```

W tej wersji wywołanie `getnum()` zostało przeniesione do asercji. Choć program działa poprawnie dla włączonych asercji, przestanie działać po ich wyłączeniu, ponieważ wtedy metoda `getnum()` nigdy nie zostanie wykonana! Co więcej, teraz zmienna `n` musi zostać zainicjalizowana, gdyż w przeciwnym razie kompilator zgłosiłby błąd (zmienna może nie zostać ustawiona przez asercję).

Asercje mogą być bardzo przydatne, gdyż ułatwiają poszukiwanie błędów w trakcie pracy nad programem. Przed pojawieniem się asercji, aby sprawdzić, czy `n` jest dodatnie, trzeba było użyć następującego fragmentu kodu.

```
if(n < 0) {
  System.out.println("n jest ujemne!");
  return; // Albo zgłoszenie wyjątku
}
```

Stosując asercję, wystarczy napisać tylko jeden wiersz kodu. Co więcej, nie trzeba usuwać instrukcji `assert` z kodu źródłowego wersji programu udostępnianej użytkownikom.

Opcje włączania i wyłączania asercji

W trakcie uruchamiania kodu można wyłączyć asercje za pomocą opcji `-da`. Aby włączyć lub wyłączyć asercje tylko dla wybranego pakietu, wystarczy podać nazwę tego pakietu po opcji `-ea` lub `-da`. Opcja włączenia asercji dla pakietu `MyPack` ma następującą postać:

```
-ea:MyPack...
```

Wyłączenie asercji dla pakietu `MyPack` wymaga użycia następującej opcji.

```
-da:MyPack...
```

Po opcji `-da` lub `-ea` może się również pojawić nazwa klasy. Poniższy kod włącza asercje tylko dla klasy `AssertDemo`.

```
-ea:AssertDemo
```

Import statyczny

Java obsługuje mechanizm tzw. **importu statycznego**, która rozszerza możliwości słowa kluczowego `import`. Zastosowanie przed słowem `import` słowa kluczowego `static` umożliwia zaimportowanie statycznych składowych klasy lub interfejsu. Po takim imporcie można bezpośrednio używać samych nazw składowych statycznych — nie trzeba ich już poprzedzać nazwą klasy. Upraszcza to znacząco składnię związaną ze składowymi statycznymi.

Aby zrozumieć użyteczność importu statycznego, najlepiej przedstawić przykładowy kod, który go *nie* używa. Poniższy program oblicza przeciwprostokątną trójkąta prostokątnego. W tym celu używa dwóch metod statycznych klasy `Math` stanowiącej część pakietu `java.lang`. Pierwsza z metod, `Math.pow()`, zwraca wartość podniesioną do wskazanej potęgi. Druga metoda, o nazwie `Math.sqrt()`, zwraca pierwiastek kwadratowy przekazanej wartości.

// R13\Hypot.java

```
// Obliczenie przeciwprostokątnej trójkąta prostokątnego
class Hypot {
  public static void main(String args[]) {
    double side1, side2;
    double hypot;

    side1 = 3.0;
    side2 = 4.0;

    // Zauważ, że metody sqrt() i pow() trzeba poprzedzać
    // nazwą klasy, w której się znajdują — Math
    hypot = Math.sqrt(Math.pow(side1, 2) +
                      Math.pow(side2, 2));

    System.out.println("Dla przyprostokątnych " +
                       side1 + " i " + side2 +
                       " przeciwprostokątna wynosi " +
                       hypot);
  }
}
```

Ponieważ `pow()` i `sqrt()` to metody statyczne, muszą być poprzedzane nazwą klasy, w której zostały zdefiniowane. Oznacza to znaczne skompilowanie zapisu obliczania przeciwprostokątnej z wzoru Pitagorasa.

```
hypot = Math.sqrt(Math.pow(side1, 2) + Math.pow(side2, 2));
```

Jak łatwo się domyślić, wielokrotne stosowanie przedrostka `Math.` dla metod `pow()` i `sqrt()` (a także wielu innych metod takich jak `sin()`, `cos()` i `tan()`) na pewno nie jest wygodne.

Wystarczy jednak zastosować import statyczny, aby cały problem z podawaniem nazwy klasy zniknął. Oto ulepszona wersja wcześniejszego programu.

// R13\Hypot2.java

```
// Użycie importu statycznego do przeniesienia
// metod sqrt() i pow() do aktualnej przestrzeni nazw
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;


//  Obliczenie przeciwprostokątnej trójkąta prostokątnego
class Hypot2 {
  public static void main(String args[]) {
    double side1, side2;
    double hypot;

    side1 = 3.0;
    side2 = 4.0;

    // Tutaj wywołań sqrt() i pow() nie trzeba
```

```
    // już poprzedzać nazwą klasy
    hypot = sqrt(pow(side1, 2) + pow(side2, 2));

    System.out.println("Dla przyprostokątnych " +
                       side1 + " i " + side2 +
                       " przeciwprostokątna wynosi " +
                       hypot);
  }
}
```

W tej wersji programu nazwy `sqrt` i `pow` są przenoszone do aktualnej przestrzeni nazw za pomocą następujących instrukcji importu statycznego.

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
```

Po wykonaniu tych instrukcji nie trzeba już podawać nazwy klasy przed wywołaniami metod `pow()` i `sqrt()`. Oto krótszy zapis obliczeń przeciwprostokątnej.

```
hypot = sqrt(pow(side1, 2) + pow(side2, 2));
```

Łatwo zauważyć, że taka wersja jest znacznie bardziej czytelna.

Istnieją dwie ogólne postacie instrukcji `import static`. Pierwsza, zastosowana w powyższym przykładzie, ma następującą postać.

```
import static pakiet.nazwa-typu.nazwa-skladowej-statycznej;
```

Element *nazwa-typu* to nazwa klasy lub interfejsu zawierającego składową statyczną. Element *pakiet* to pełna nazwa pakietu, w którym znajduje się klasa lub interfejs. Element *nazwa-skladowej-statycznej* określa nazwę importowanej składowej statycznej.

Druga postać importu statycznego powoduje zaimportowanie wszystkich składowych statycznych wskazanej klasy. Oto jej ogólna wersja.

```
import static pakiet.nazwa-typu.*;
```

Jeżeli chce się używać wielu składowych statycznych wybranej klasy, ta postać znacznie ułatwia zadanie — nie trzeba określać nazw poszczególnych składowych. Aby w poprzednim przykładzie użyć tylko jednej instrukcji importu do zaimportowania metod `pow()` i `sqrt()`, a także **wszystkich innych** metod statycznych klasy `Math`, wystarczy napisać następujący kod.

```
import static java.lang.Math.*;
```

Oczywiście import statyczny nie jest ograniczony tylko i wyłącznie do klasy `Math` lub tylko do metod. Poniższa instrukcja powoduje umieszczenie w aktualnej przestrzeni pola `System.out`.

```
import static java.lang.System.out;
```

W dalszej części kodu zawierającego taką instrukcję nie trzeba już poprzedzać `out` nazwą klasy `System`. Oto przykład.

```
out.println("Po zaimportowaniu System.out, można bezpośrednio używać out.");
```

Czy importowanie `System.out` w ten sposób jest dobrym pomysłem? Choć skraca składnię, inne osoby czytające kod mogą nie domyślić się, iż stosowane w kodzie `out` tak naprawdę dotyczy `System.out`.

Dodatkowa uwaga: import statyczny nie ogranicza się jedynie do klas i interfejsów wbudowanych w Javę. Równie dobrze można w ten sposób importować składowe statyczne własnych klas i interfejsów.

Choć import statyczny jest bardzo wygodny, nie należy go nadużywać. Należy pamiętać, że stosowane w Javie pakiety mają za zadanie ograniczać ryzyko konfliktów nazw. Import statyczny powoduje umieszczenie składowych statycznych w globalnej przestrzeni nazw. Zwiększa się w ten sposób ryzyko konfliktów nazw i przypadkowego ukrycia innych nazw. Jeśli istnieje w programie tylko jedno lub dwa miejsca, gdzie warto byłoby zastosować import statyczny, nie warto tego czynić. Poza tym pewne nazwy, na przykład `System.out`, są na tyle rozpoznawalne, że nie warto ograniczać czytelności kodu importem statycznym. Import statyczny wprowadzono w celu ułatwienia programistom pisania kodu, który wielokrotnie odwołuje się do składowych statycznych, na przykład w trakcie wykonywania wielu operacji matematycznych. Innymi słowy, warto korzystać z tego nowego elementu z rozwagą.

Wywoływanie przeciążonych konstruktorów za pomocą this()

Podczas pracy z przeciążonymi konstruktorami często najwygodniejszym rozwiązaniem jest wywoływanie jednej wersji konstruktora przez inną wersję. W Javie podobne wywołania nie stanowią żadnego problemu — wystarczy użyć nieco innej formy słowa kluczowego `this`. Ogólną postać takiego wywołania pokazano poniżej:

`this(`*`lista-argumentów`*`)`

Wywołanie `this()` spowoduje najpierw wykonanie przeciążonego konstruktora, którego lista parametrów pasuje do parametru *`lista-argumentów`*. Dopiero potem zostaną wykonane ewentualne wyrażenia w oryginalnym konstruktorze (zawierającym to wywołanie). Wywołanie `this()` musi być pierwszym wyrażeniem w ciele konstruktora.

Aby zrozumieć sposób stosowania wywołania `this()`, warto przeanalizować prosty przykład. Przyjrzyjmy się najpierw poniższej klasie, w której wywołanie `this()` nie jest stosowane:

```
// R13\MyClass.java

class MyClass {
  int a;
  int b;

  // Inicjalizuje składowe a oraz b
  MyClass(int i, int j) {
    a = i;
    b = j;
  }

  // Inicjalizuje składowe a oraz b przy użyciu tej samej wartości
  MyClass(int i) {
    a = i;
    b = i;
  }

  // Przypisuje składowym a oraz b domyślną wartość 0
  MyClass( ) {
    a = 0;
    b = 0;
  }
}
```

Klasa zawiera trzy konstruktory, z których każdy odpowiada za inicjalizację składowych `a` oraz `b`. Pierwszy konstruktor otrzymuje na wejściu osobno wartości dla obu składowych. Drugi konstruktor otrzymuje tylko jedną wartość, którą przypisuje obu składowym. Trzeci konstruktor przypisuje tym składowym domyślną wartość 0.

Wywołanie `this()` umożliwia przebudowę klasy `MyClass` do następującej postaci:

```
// R13\MyClass_2.java

class MyClass {
  int a;
  int b;

  // Inicjalizuje składowe a oraz b
  MyClass(int i, int j) {
    a = i;
    b = j;
  }

  // Inicjalizuje składowe a oraz b przy użyciu tej samej wartości
  MyClass(int i) {
    this(i, i); // Wywołuje MyClass(i, i)
```

```
  }

  // Przypisuje składowym a oraz b domyślną wartość 0
  MyClass( ) {
    this(0); // Wywołuje MyClass(0)
  }
}
```

W tej wersji klasy `MyClass` jedynym konstruktorem, który rzeczywiście przypisuje wartości składowym `a` oraz `b`, jest konstruktor w wersji `MyClass(int, int)`. Dwa pozostałe konstruktory ograniczają się do wywołania tego konstruktora za pośrednictwem wyrażenia `this()`. Warto na przykład zastanowić się, jaki będzie efekt wykonania następującego wyrażenia:

```
MyClass mc = new MyClass(8);
```

Wywołanie `MyClass(8)` powoduje wykonanie `this(8, 8)`, czyli w praktyce wywołanie konstruktora `MyClass(8, 8)`, ponieważ lista parametrów tej wersji konstruktora `MyClass` pasuje do listy argumentów przekazanych za pośrednictwem wywołania `this()`. Przeanalizujmy teraz następujące wyrażenie, które odwołuje się do konstruktora domyślnego:

```
MyClass mc2 = new MyClass();
```

Tym razem zostanie użyte wywołanie `this(0)`. Efektem tego wywołania będzie wywołanie konstruktora `MyClass(0)`, którego lista parametrów pasuje do listy użytych argumentów. Jak wiemy, konstruktor `MyClass(0)` wywoła następnie konstruktor `MyClass(0,0)`.

Jednym z argumentów na rzecz wywoływania konstruktorów za pośrednictwem wyrażenia `this()` jest możliwość unikania niepotrzebnego powielania kodu. W wielu przypadkach ograniczenie ilości powielonego kodu pozwala skrócić czas ładowania klasy, ponieważ kod odpowiedniego obiektu jest krótszy. Ten aspekt jest szczególnie ważny w przypadku programów dostarczanych za pośrednictwem internetu, gdzie czas ładowania nierzadko stanowi poważny problem. Stosowanie wywołania `this()` ułatwia też zarządzanie strukturą kodu w sytuacji, gdy poszczególne wersje przeciążonego konstruktora zawierają sporo powtarzającego się kodu.

Warto jednak zachować ostrożność. Wykonywanie konstruktorów zawierających wywołania `this()` przebiega nieco wolniej niż w przypadku konstruktorów zawierających w swoim ciele cały kod inicjalizujący. Wolniejsze działanie wynika z tego, że mechanizm wywołania drugiego konstruktora i zwrócenia wartości wiąże się z pewnymi dodatkowymi kosztami. Jeśli projektowana klasa będzie używana do tworzenia zaledwie kilku obiektów lub jeśli konstruktory tej klasy zawierające wywołania `this()` są dość rzadko stosowane, spadek wydajności w czasie wykonywania najprawdopodobniej będzie nieistotny. Jeśli jednak projektowana klasa ma być używana do tworzenia ogromnej liczby obiektów (liczonych na przykład w tysiącach), negatywny wpływ wspomnianych kosztów na wydajność aplikacji może być dość widoczny. Ponieważ dłuższy proces tworzenia obiektu wpływa na odczucia wszystkich użytkowników, w pewnych przypadkach warto dokładnie rozważyć zalety i wady krótszego czasu ładowania aplikacji kosztem bardziej czasochłonnego tworzenia obiektów (już w czasie działania).

Warto mieć na uwadze jeszcze jeden aspekt. W przypadku bardzo niewielkich konstruktorów (podobnych do tych stosowanych w klasie `MyClass`) różnica w wielkości kodu obiektu wynikająca ze stosowania wywołań `this()` jest stosunkowo niewielka. (W skrajnych przypadkach stosowanie wywołań `this()` w ogóle nie powoduje ograniczenia ilości kodu). Niewielka różnica lub jej brak wynika z tego, że kod bajtowy odpowiedzialny za wywołanie i zwrócenie wartości przez konstrukcję `this()` wymaga dodatkowych instrukcji w kodzie obiektu. W takich przypadkach nawet wyeliminowanie powtórzeń kodu dzięki wywołaniom `this()` nie spowoduje znacznego skrócenia czasu ładowania. Okazuje się jednak, że nawet wówczas trzeba się liczyć z dodatkowymi kosztami podczas konstruowania każdego obiektu. Oznacza to, że wywołanie `this()` należy stosować przede wszystkim w konstruktorach zawierających dużą ilość kodu inicjalizującego (nie w konstruktorach ograniczających się do ustawienia wartości kilku pól).

Istnieją dwa ograniczenia, o których warto pamiętać podczas korzystania z wywołań `this()`. Po pierwsze, w wywołaniu `this()` nie można używać żadnych zmiennych egzemplarza klasy, do której należy dany konstruktor. Po drugie, ten sam konstruktor nie może jednocześnie zawierać wywołań `super()` i `this()`, ponieważ każde z tych wywołań musi być pierwszym wyrażeniem w ciele konstruktora.

Kilka słów o kompaktowych profilach API

W JDK 8 wprowadzono funkcje pozwalające organizować fragmenty biblioteki API w tak zwane **profile kompaktowe** (ang. *compact profile*). Noszą one nazwy `compact1`, `compact2` oraz `compact3`. Każdy z tych profilów zawiera podzbiór biblioteki API. Co więcej, profil `compact2` obejmuje cały profil `compact1`, a profil `compact3` obejmuje cały profil `compact2`. A zatem każdy z nich bazuje na poprzednim. Zaleta profili kompaktowych polega na tym, że aplikacja, która nie potrzebuje pełnej biblioteki API, nie będzie jej pobierać. Stosowanie profili pozwala na zmniejszenie wielkości biblioteki, dzięki czemu można uruchamiać niektóre typy aplikacji Javy na urządzeniach, które z jakichś przyczyn nie są w stanie obsługiwać pełnej wersji Java API. Oprócz tego stosowanie profili kompaktowych pozwala skrócić czas wczytywania programu. Dokumentacja Java JDK 8 API zawiera informacje o tym, do którego z profili kompaktowych należy dany element API (jeśli w ogóle do jakiegoś należy). Warto zauważyć, że nowy mechanizm modułów wprowadzony w JDK 9 zastępuje kompaktowe profile API.

Java: zdobądź solidne podstawy i twórz kod mistrzów!

Niemal od chwili, gdy powstała, Java jest jednym z najważniejszych i najpopularniejszych języków programowania. Dzieje się tak dzięki konsekwentnemu rozwijaniu tego języka i poszukiwaniu coraz to nowszych technologii. Sprawia to, że Java jest pierwszym i najlepszym wyborem dla programistów zainteresowanych tworzeniem aplikacji internetowych. Oprócz tego nadaje się do wielu innych zastosowań. Łatwo można się przekonać, że większość współczesnego świata korzysta z kodu Javy. Dotyczy to nie tylko komputerów czy smartfonów, ale także wielu innych urządzeń.

Ta książka jest dziesiątym wydaniem znakomitego podręcznika, w pełni zaktualizowanym o informacje dotyczące Java SE 9. W przystępny sposób wyjaśniono w nim, jak pisać, kompilować, debugować i uruchamiać kod Javy. Znalazły się tu także informacje o kluczowych elementach biblioteki Java API, takich jak obsługa wejścia-wyjścia, Collections Framework, biblioteka strumieni oraz narzędzia do programowania współbieżnego. W praktyczny sposób przedstawiono bibliotekę Swing, JavaFX, technologię JavaBeans oraz serwletów. Książka zawiera również szczegółowy opis modułów i praktyczne wprowadzenie do JShell, narzędzia do interaktywnego programowania w Javie.

Najważniejsze zagadnienia ujęte w książce:

- **typy danych, zmienne, tablice i operatory oraz instrukcje sterujące**
- **programowanie obiektowe i dziedziczenie**
- **interfejsy i pakiety oraz obsługa wyjątków**
- **programowanie wielowątkowe**
- **wyrażenia lambda, moduły, programowanie sieciowe**
- **obsługa zdarzeń i programowanie współbieżne**

Herbert Schildt — jest uznanym autorytetem w dziedzinie programowania w Javie. Jego książki o programowaniu w Javie, C, C++ i C# przetłumaczono na dziesiątki języków. Doskonale zna wszystkie aspekty technologii komputerowej, jednak najbardziej interesuje się językami programowania. Szczególnie aktywnie działa na rzecz ich standaryzacji.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-4613-0
9 788328 346130

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 79,00 zł

Oracle Press™