Java. Techniki zaawansowane. Wydanie VIII

Autor: Cay S. Horstmann, Gary Cornell
Tłumaczenie: Jaromir Senczyk
ISBN: 978-83-246-1483-7
Tytuł oryginału: Core Java(tm), Volume II–Advanced Features: Eighth Edition
Format: 172x245, stron: 1064

- Jak wykorzystać strumienie?
- Jak stworzyć efektowny interfejs użytkownika?
- Jak zapewnić bezpieczeństwo w tworzonych aplikacjach?

Co spowodowało, że język programowania Java zyskał tak wielką popularność? Przyczyn jest kilka: możliwość przenoszenia kodu między programami, wydajność i to, co programiści lubią najbardziej – mechanizm automatycznego oczyszczania pamięci. Nie bez znaczenia jest również to, że Java jest językiem zorientowanym obiektowo, udostępnia obsługę programowania rozproszonego oraz świetną dokumentację. Ponadto liczne publikacje oraz pomocna społeczność sprawiają, że Java zajmuje poczesne miejsce wśród innych języków programowania.

Kolejne wydanie książki „Java. Techniki zaawansowane. Wydanie VIII" zostało zaktualizowane o wszystkie te elementy, które pojawiły się w wersji szóstej platformy Java Standard Edition. Dzięki tej książce dowiesz się, w jaki sposób wykorzystać strumienie, jak parsować dokumenty XML czy też w jaki sposób tworzyć aplikacje sieciowe. Poznasz interfejs JDBC, sposób wykorzystania transakcji oraz wykonywania zapytań SQL. Autorzy w szczegółowy sposób pokażą Ci, jak tworzyć aplikacje z wykorzystaniem biblioteki Swing. Dodatkowo przedstawią, w jaki sposób zapewnić bezpieczeństwo w tworzonych przez Ciebie aplikacjach. Wszystkie te – oraz wiele innych – zagadnienia zostaną przedstawione w przystępny i sprawdzony sposób!

- Wykorzystanie strumieni
- Dokumenty XML i ich wykorzystanie w języku Java
- Programowanie aplikacji sieciowych
- Wykorzystanie interfejsu JDBC
- Tworzenie aplikacji wielojęzycznych
- Możliwości pakietu Swing
- Wykorzystanie biblioteki AWT
- Bezpieczeństwo w aplikacjach
- Zastosowanie podpisu cyfrowego
- Sposoby wykorzystania obiektów rozproszonych (RMI)

Wykorzystaj zaawansowane możliwości języka Java w swoich projektach!

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

1

Strumienie i pliki

W tym rozdziale:

- strumienie,
- strumienie tekstowe,
- odczyt i zapis danych binarnych,
- strumienie plików ZIP,
- strumienie obiektów i serializacja,
- zarządzanie plikami,
- ulepszona obsługa wejścia i wyjścia,
- wyrażenia regularne.

W tym rozdziale omówimy metody obsługi plików i katalogów, a także metody zapisywania do i wczytywania informacji z plików w formacie tekstowym i binarnym. W rozdziale przedstawiony jest również mechanizm serializacji obiektów, który umożliwia przechowywanie obiektów z taką łatwością, z jaką przechowujesz tekst i dane numeryczne. Następnie omówimy szereg ulepszeń, które do obsługi wejścia i wyjścia wprowadził pakiet `java.nio` udostępniony w wersji Java SE 1.4. Rozdział zakończymy przedstawieniem problematyki wyrażeń regularnych, mimo że nie jest ona bezpośrednio związana ze strumieniami i plikami. Nie potrafiliśmy jednak znaleźć dla niej lepszego miejsca w książce. W naszym wyborze nie byliśmy zresztą osamotnieni, ponieważ zespół Javy dołączył specyfikację interfejsów programowych związanych z przetwarzaniem wyrażeń regularnych do specyfikacji ulepszonej obsługi wejścia i wyjścia w Java SE 1.4.

Strumienie

W języku Java obiekt, z którego możemy odczytać sekwencję bajtów, nazywamy *strumieniem wejścia*. Obiekt, do którego możemy zapisać sekwencję bajtów, nazywamy *strumieniem wyjścia*. Źródłem bądź celem tych sekwencji bajtów mogą być, i często właśnie są, pliki,

ale także i połączenia sieciowe, a nawet bloki pamięci. Klasy abstrakcyjne `InputStream` i `Out ↪putStream` stanowią bazę hierarchii klas opisujących wejście i wyjście programów Java.

Ponieważ strumienie binarne nie są zbyt wygodne do manipulacji danymi przechowywanymi w standardzie Unicode (przypomnijmy tutaj, że Unicode opisuje każdy znak za pomocą dwóch bajtów), stworzono osobną hierarchię klas operujących na znakach Unicode i dziedziczących po klasach abstrakcyjnych `Reader` i `Writer`. Klasy te są przystosowane do wykonywania operacji odczytu i zapisu, opartych na dwubajtowych znakach Unicode, nie przydają się natomiast do znaków jednobajtowych.

Odczyt i zapis bajtów

Klasa `InputStream` posiada metodę abstrakcyjną:

```
abstract int read()
```

Metoda ta wczytuje jeden bajt i zwraca jego wartość lub –1, jeżeli natrafi na koniec źródła danych. Projektanci konkretnych klas strumieni wejścia przeładowują tę metodę, dostarczając w ten sposób użytecznej funkcjonalności. Dla przykładu, w klasie `FileInputStream` metoda `read` czyta jeden bajt z pliku. `System.in` to predefiniowany obiekt klasy pochodnej od `Input ↪Stream`, pozwalający pobierać informacje z klawiatury.

Klasa `InputStream` posiada również nieabstrakcyjną metodę pozwalającą pobrać lub zignorować tablicę bajtów. Metody te wywołują abstrakcyjna metodę `read`, tak więc podklasy muszą przeładować tylko tę jedną metodę.

Analogicznie, klasa `OutputStream` definiuje metodę abstrakcyjną

```
abstract void write(int b)
```

która wysyła jeden bajt do aktualnego wyjścia.

Metody `read` i `write` potrafią *zablokować* wątek, dopóki dany bajt nie zostanie wczytany lub zapisany. Oznacza to, że jeżeli strumień nie może natychmiastowo wczytać lub zapisać danego bajta (zazwyczaj z powodu powolnego połączenia sieciowego), Java zawiesza wątek dokonujący wywołania. Dzięki temu inne wątki mogą wykorzystać czas procesora, w którym wywołana metoda czeka na udostępnienie strumienia.

Metoda `available` pozwala sprawdzić liczbę bajtów, które w danym momencie odczytać. Oznacza to, że poniższy kod prawdopodobnie nigdy nie zostanie zablokowany:

```
int bytesAvaible = System.in.available();
if (bytesAvaible > 0)
{
 byte[] dane = new byte[bytesAvaible];
 System.in.read(data);
}
```

Gdy skończymy odczytywać albo zapisywać dane do strumienia, zamykamy go, wywołując metodę `close`. Metoda ta uwalnia zasoby systemu operacyjnego, do tej pory udostępnione wątkowi. Jeżeli aplikacja otworzy zbyt wiele strumieni, nie zamykając ich, zasoby systemu

mogą zostać naruszone. Co więcej, zamknięcie strumienia wyjścia powoduje *opróżnienie* bufora używanego przez ten strumień — wszystkie znaki, przechowywane tymczasowo w buforze, aby mogły zostać zapisane w jednym większym pakiecie, zostaną natychmiast wysłane. Jeżeli nie zamkniemy strumienia, ostatni pakiet bajtów może nigdy nie dotrzeć do odbiorcy. Bufor możemy również opróżnić własnoręcznie, przy użyciu metody `flush`.

Mimo iż klasy strumieni udostępniają konkretne metody wykorzystujące funkcje `read` i `write`, programiści Javy rzadko z nich korzystają, ponieważ nieczęsto się zdarza, żeby programy musiały czytać i zapisywać sekwencje bajtów. Dane, którymi jesteśmy zwykle bardziej zainteresowani, to liczby, łańcuchy znaków i obiekty.

Java udostępnia wiele klas strumieni pochodzących od podstawowych klas `InputStream` i `OutputStream`, które pozwalają operować na danych w sposób bardziej dogodny aniżeli w przypadku pracy na poziomie pojedynczych bajtów.

API `java.io.InputStream` **1.0**

- `abstract int read()`

  pobiera jeden bajt i zwraca jego wartość. Metoda `read` zwraca –1, gdy natrafi na koniec strumienia.

- `int read(byte[] b)`

  wczytuje dane do tablicy i zwraca liczbę wczytanych bajtów, a jeżeli natrafi na koniec strumienia, zwraca –1. Metoda `read` czyta co najwyżej `b.length` bajtów.

- `int read(byte[] b, int off, int len)`

  wczytuje dane do tablicy bajtów. Zwraca liczbę wczytanych bajtów, a jeżeli natrafi na koniec strumienia, zwraca –1.

  Parametry:

`b`	tablica, w której zapisywane są dane.
`off`	indeks tablicy `b`, pod którym powinien zostać umieszczony pierwszy wczytany bajt.
`len`	maksymalna liczba wczytywanych bajtów.

- `long skip(long n)`

  ignoruje `n` bajtów w strumieniu wejścia. Zwraca faktyczną liczbę zignorowanych bajtów (która może być mniejsza niż `n`, jeżeli natrafimy na koniec strumienia).

- `int available()`

  zwraca liczbę bajtów dostępnych bez konieczności zablokowania wątku (pamiętajmy, że zablokowanie oznacza, że wykonanie aktualnego wątku zostaje wstrzymane).

- `void close()`

  zamyka strumień wejścia.

- `void mark(int readlimit)`

  ustawia znacznik na aktualnej pozycji strumienia wejścia (nie wszystkie strumienie obsługują tę możliwość). Jeżeli ze strumienia zostało pobranych więcej niż `readlimit` bajtów, strumień ma prawo usunąć znacznik.

- `void reset()`

  wraca do ostatniego znacznika. Późniejsze wywołania `read` będą powtórnie czytać pobrane już bajty. Jeżeli znacznik nie istnieje, strumień nie zostanie zresetowany.

- `boolean markSupported()`

  zwraca `true`, jeżeli strumień obsługuje znaczniki.

API `java.io.OutputStream` **1.0**

- `abstract void write(int n)`

  zapisuje jeden bajt.

- `void write(byte[] b)`

- `void write(byte[] b, int off, int len)`

  zapisują wszystkie bajty tablicy `b lub pewien ich zakres`.

  Parametry: `b` tablica, z której pobierane są dane.
  `off` indeks tablicy `b`, spod którego powinien zostać pobrany pierwszy zapisywany bajt.
  `len` liczba zapisywanych bajtów.

- `void close()`

  opróżnia i zamyka strumień wyjścia.

- `void flush()`

  opróżnia strumień wyjścia, czyli wysyła do odbiorcy wszystkie dane znajdujące się w buforze.

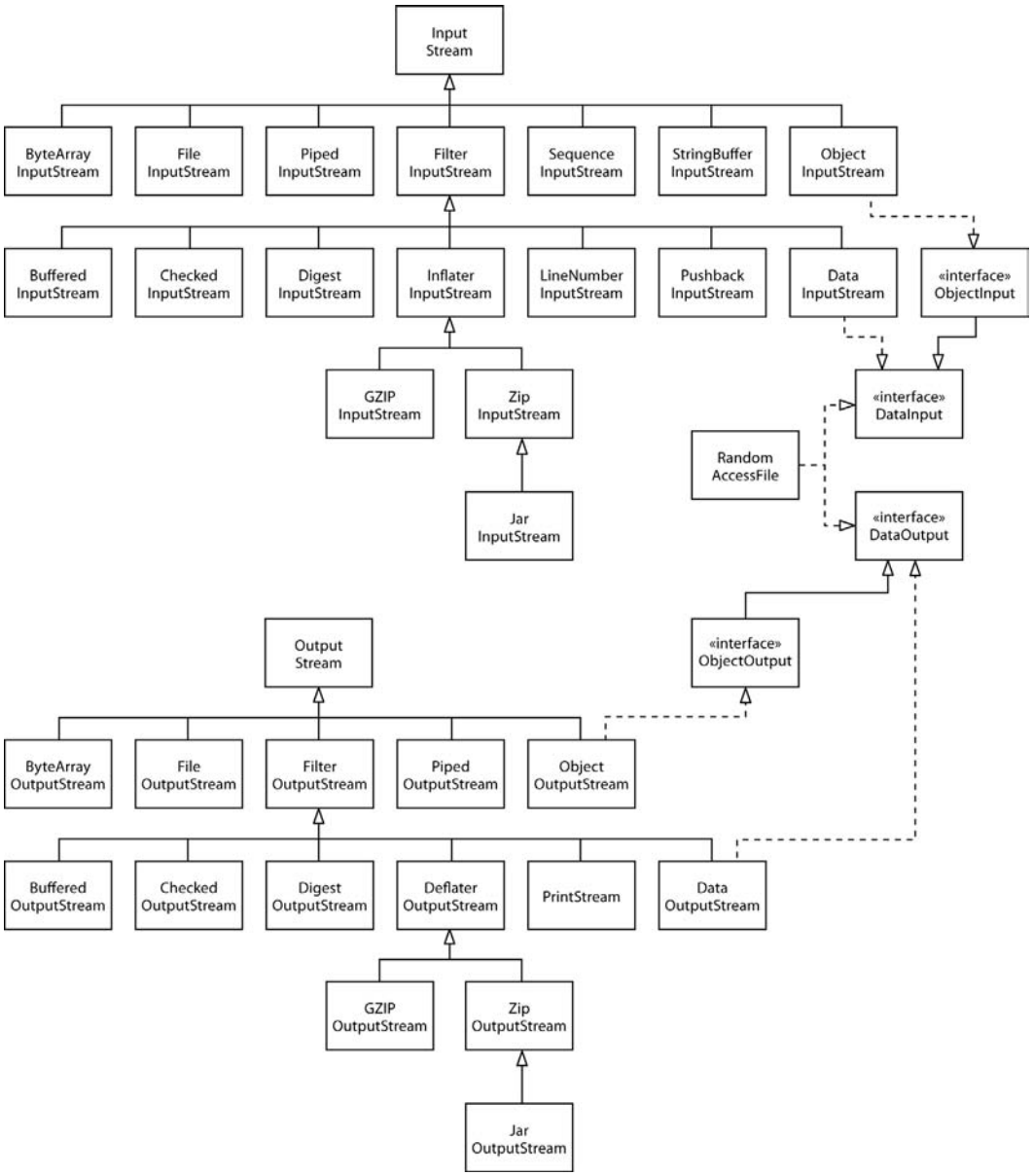
Zoo pełne strumieni

W przeciwieństwie do języka C, który w zupełności zadowala się jednym typem `FILE*`, Java posiada istne zoo ponad 60 (!) różnych typów strumieni (patrz rysunki 1.1 i 1.2).

Podzielmy gatunki należące do zoo klas strumieni zależnie od ich przeznaczenia. Istnieją osobne hierarchie klas przetwarzających bajty i znaki.

Jak już o tym wspomnieliśmy, klasy `InputStream` i `OutputStream` pozwalają pobierać i wysyłać jedynie pojedyncze bajty oraz tablice bajtów. Klasy te stanowią bazę hierarchii pokazanej na rysunku 1.1. Do odczytu i zapisu liczb i łańcuchów znakowych używamy ich podklas. Na przykład, `DataInputStream` i `DataOutputStream` pozwalają wczytywać i zapisywać wszystkie podstawowe typy Javy w postaci binarnej. Istnieje wiele pożytecznych klas strumieni, na przykład ZipInputStream i ZipOutputStream pozwalające odczytywać i zapisywać dane w plikach skompresowanych w formacie ZIP.

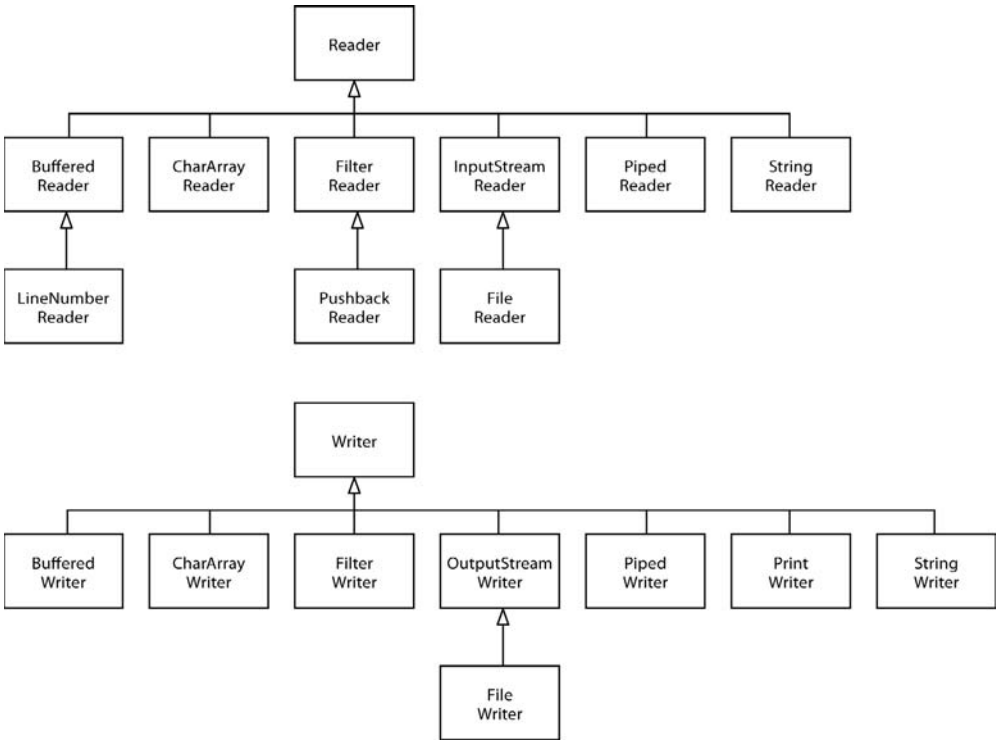
Z drugiej strony, o czym już wspominaliśmy, do obsługi tekstu Unicode używamy klas pochodzących od klas abstrakcyjnych `Reader` i `Writer` (patrz rysunek 1.2) Podstawowe metody klas `Reader` i `Writer` są podobne do tych należących do `InputStream` i `OutputStream`.



Rysunek 1.1. *Hierarchia strumieni wejścia i wyjścia*

```
abstract int read()
abstract void write(int b)
```

Metoda `read` zwraca albo kod znaku Unicode (jako liczbę z przedziału od 0 do 65535), albo –1, jeżeli natrafi na koniec pliku. Metoda write jest wywoływana dla podanego kodu znaku Unicode (więcej informacji na temat kodów Unicode znjadziesz w rozdziale 3. książki *Java 2. Podstawy*).



Rysunek 1.2. *Hierarchia klas Reader i Writer*

Począwszy od Java SE 5.0, wprowadzono cztery dodatkowe interfejsy: `Closeable`, `Flushable`, `Readable` i `Appendable` (patrz rysunek 1.3). Pierwsze dwa z nich są wyjątkowo proste i zawierają odpowiednio metody:

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

i

```
void flush()
```

Klasy `InputStream`, `OutputStream`, `Reader` i `Writer` implementują interfejs `Closeable`. Klasy `OutputStream` i `Writer` implementują interfejs `Flushable`.

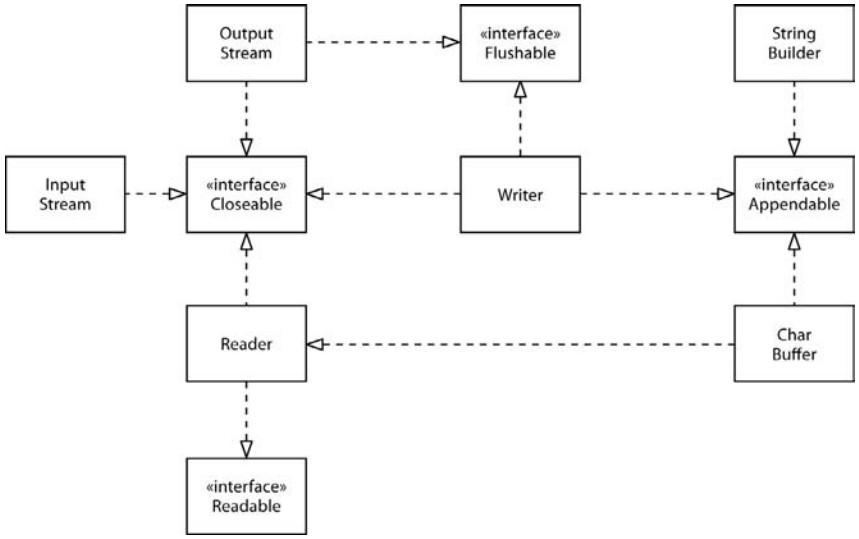
Interfejs `Readable` ma tylko jedną metodę

```
int read(CharBuffer cb)
```

Klasa `CharBuffer` ma metody do sekwencyjnego oraz swobodnego odczytu i zapisu. Reprezentuje ona bufor w pamięci lub mapę pliku w pamięci. (Patrz punkt „Struktura bufora danych” na stronie 89).

Interfejs `Appendable` ma dwie metody umożliwiające dopisywanie pojedynczego znaku bądź sekwencji znaków:

```
Appendable append(char c)
Appendable append(CharSequence s)
```



Rysunek 1.3. *Interfejsy Closeable, Flushable, Readable i Appendable*

Interfejs `CharSequence` opisuje podstawowe właściwości sekwencji wartości typu `char`. Interfejs ten implementują klasy `String`, `CharBuffer`, `StringBuilder` i `StringBuffer`.

Spośród klas strumieni jedynie klasa `Writer` implementuje interfejs `Appendable`.

API `java.io.Closeable` **5.0**

- `void close()`
  zamyka obiekt implemetujący interfejs `Closeable`. Może wyrzucić wyjątek `IOException`.

API `java.io.Flushable` **5.0**

- `void flush()`
  opróżnia bufor danych związany z obiektem implementującym interfejs `Flushable`.

API `java.lang.Readable` **5.0**

- `int read(CharBuffer cb)`
  próbuje wczytać tyle wartości typu `char`, ile może pomieścić `cb`. Zwraca liczbę wczytanych wartości lub -1, jeśli obiekt `Readable` nie ma już wartości do pobrania.

API `java.lang.Appendable` **5.0**

- `Appendable append(char c)`

- `Appendable append(CharSequence cs)`
  dopisuje podany kod znaku lub wszystkie kody podanej sekwencji do obiektu `Appendable`; zwraca `this`.

API `java.lang.CharSequence` **1.4**

- `char charAt(int index)`
  zwraca kod o podanym indeksie.
- `int length()`
  zwraca liczbę kodów w sekwencji.
- `CharSequence subSequence(int startIndex, int endIndex)`
  zwraca sekwencję `CharSequence` złożoną z kodów od `startIndex` do `endIndex - 1`.
- `String toString()`
  zwraca łańcuch znaków składający się z kodów danej sekwencji.

Łączenie filtrów strumieni

Klasy `FileInputStream` i `FileOutputStream` obsługują strumienie wejścia i wyjścia przyporządkowane określonemu plikowi na dysku. W konstruktorze tych klas podajemy nazwę pliku lub pełną ścieżkę dostępu do niego. Na przykład

```
FileInputStream fin = new FileInputStream("employee.dat");
```

spróbuje odszukać w aktualnym katalogu plik o nazwie *employee.dat*.

Ponieważ wszystkie klasy w `java.io` uznają relatywne ścieżki dostępu za rozpoczynające się od aktualnego katalogu roboczego, powinieneś wiedzieć, co to za katalog. Możesz pobrać tę informację poleceniem `System.getProperty("user.dir")`.

Tak jak klasy abstrakcyjne `InputStream` i `OutputStream`, powyższe klasy obsługują odczyt i zapis plików na poziomie pojedynczego bajta. Oznacza to, że z obiektu `fin` możemy czytać wyłącznie pojedyncze bajty oraz tablice bajtów.

```
byte b = (byte)fin.read();
```

W następnym podrozdziale przekonamy się, że korzystając z `DataInputStream`, moglibyśmy wczytywać typy liczbowe:

```
DataInputStream din = . . .;
double p = din.readDouble();
```

Ale tak jak `FileInputStream` nie posiada metod czytających typy liczbowe, tak `DataInput`
`↪Stream` nie posiada metody pozwalającej czytać dane z pliku.

Java korzysta ze sprytnego mechanizmu rozdzielającego te dwa rodzaje funkcjonalności. Niektóre strumienie (takie jak `FileInputStream` i strumień wejścia zwracany przez metodę `openStream` klasy `URL`) mogą udostępniać bajty z plików i innych, bardziej egzotycznych loka-

lizacji. Inne strumienie (takie jak `DataInputStream` i `PrintWriter`) potrafią tworzyć z bajtów reprezentację bardziej użytecznych typów danych. Programista Javy musi połączyć te dwa mechanizmy w jeden. Dla przykładu, aby wczytywać liczby z pliku, powinien utworzyć obiekt typu `FileInputStream`, a następnie przekazać go konstruktorowi `DataInputStream`.

```
FileInputStream fin = new FileInputStream("employee.dat");
DataInputStream din = new DataInputStream(fin);
double s = din.readDouble();
```

Wróćmy do rysunku 1.1, gdzie przedstawione są klasy `FilterInputStream` i `FilterOutputStream`. Ich podklasy możemy wykorzystać do rozbudowy obsługi strumieni zwykłych bajtów.

Różne funkcjonalności możemy dodawać poprzez zagnieżdżanie filtrów. Na przykład — domyślnie strumienie nie są buforowane. Wobec tego każde wywołanie metody read oznacza odwołanie się do usług systemu operacyjnego, który odczytuje kolejny bajt. Dużo efektywniej będzie żądać od systemu operacyjnego całych bloków danych i umieszczać je w buforze. Jeśli chcemy uzyskać buforowany dostęp do pliku, musimy skorzystać z poniższej, monstrualnej sekwencji konstruktorów:

```
DataInputStream din = new DataInputStream
 (new BufferedInputStream
  (new FileInputStream("employee.dat")));
```

Zwróćmy uwagę, że `DataInputStream` znalazł się na *ostatnim* miejscu w łańcuchu konstruktorów, ponieważ chcemy używać metod klasy `DataInputStream` i chcemy, aby korzystały *one* z buforowanej metody `read`.

Czasami będziemy zmuszeni utrzymywać łączność ze strumieniami znajdującymi się pośrodku łańcucha. Dla przykładu, czytając dane, musimy często podejrzeć następny bajt, aby sprawdzić, czy jego wartość zgadza się z naszymi oczekiwaniami. W tym celu Java dostarcza klasę `PushbackInputStream`.

```
PushbackInputStream pbin = new PushbackInputStream
 (new BufferedInputStream
  (new FileInputStream("employee.dat")));
```

Teraz możemy odczytać wartość następnego bajta:

```
int b = pbin.read();
```

i umieścić go z powrotem w strumieniu, jeżeli jego wartość nie odpowiada naszym oczekiwaniom.

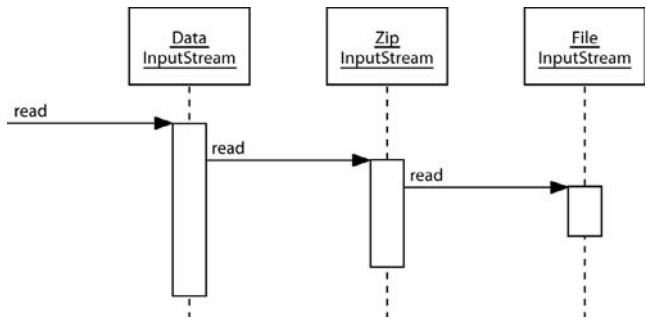
```
if (b != '<') pbin.unread(b);
```

Ale wczytywanie i powtórne wstawianie to *jedyne* metody obsługiwane przez klasę `PushbackInputStream`. Jeżeli chcemy podejrzeć kolejne bajty, a także wczytywać liczby, potrzebujemy referencji zarówno do `PushbackInputStream`, jak i do `DataInputStream`.

```
DataInputStream din = DataInputStream
 (pbin = new PushbackInputStream
  (new BufferedInputStream
   (new FileInputStream("employee.dat"))));
```

Oczywiście, w bibliotekach strumieni innych języków programowania takie udogodnienia jak buforowanie i kontrolowanie kolejnych bajtów są wykonywane automatycznie, więc konieczność tworzenia ich kombinacji w języku Java wydaje się niepotrzebnym zawracaniem głowy. Jednak możliwość łączenia klas filtrów i tworzenia w ten sposób naprawdę użytecznych sekwencji strumieni daje nam niespotykaną elastyczność. Na przykład, korzystając z poniższej sekwencji strumieni, możemy wczytywać liczby ze skompresowanego pliku ZIP (patrz rysunek 1.4).

```
ZipInputStream zin
 = new ZipInputStream(new FileInputStream("employee.zip"));
DataInputStream din = new DataInputStream(zin);
```



Rysunek 1.4. *Sekwencja filtrowanych strumieni*

Aby dowiedzieć się więcej o obsłudze formatu ZIP, zajrzyj do podrozdziału poświęconego strumieniom plików ZIP na stronie 48.

API `java.io.FileInputStream` **1.0**

- `FileInputStream(String name)`

  tworzy nowy obiekt typu `FileInputStream`, używając pliku, którego ścieżka dostępu znajduje się w łańcuchu `nazwa`.

- `FileInputStream(File file)`

  tworzy nowy obiekt typu `FileInputStream`, używając pliku, którego ścieżkę dostępu zawiera parametr name, lub używając informacji zawartych w obiekcie `file` (klasa `File` zostanie omówiona pod koniec tego rozdziału). Ścieżki dostępu są podawane względem katalogu roboczego skonfigurowanego podczas uruchamiania maszyny wirtualnej Java.

API `java.io.FileOutputStream` **1.0**

- `FileOutputStream(String name)`
- `FileOutputStream(String name, boolean append)`
- `FileOutputStream(File file)`

  tworzy nowy obiekt typu

- `FileOutputStream(File file, boolean append)`

  tworzy nowy strumień wyjściowy pliku określonego za pomocą łańcucha file lub obiektu file (klasa `File` zostanie omówiona pod koniec tego rozdziału). Jeżeli parametr `append` ma wartość `true`, dane dołączane są na końcu pliku, a istniejący plik o tej samej nazwie nie zostanie skasowany. W przeciwnym razie istniejący plik o tej samej nazwie zostanie skasowany.

API `java.io.BufferedInputStream` **1.0**

- `BufferedInputStream(InputStream in)`

  tworzy nowy obiekt typu `BufferedInputStream`, o domyślnym rozmiarze bufora. Strumień buforowany wczytuje znaki ze strumienia danych, nie wymuszając za każdym razem dostępu do urządzenia. Gdy bufor zostanie opróżniony, system prześle do niego nowy blok danych.

API `java.io.BufferedOutputStream` **1.0**

- `BufferedOutputStream(OutputStream out)`

  tworzy nowy obiekt typu `Buffered-OutputStream`, o domyślnym rozmiarze bufora. Strumień umieszcza w buforze znaki, które powinny zostać zapisane, nie wymuszając za każdym razem dostępu do urządzenia. Gdy bufor zapełni się lub gdy strumień zostanie opróżniony, dane są przesyłane odbiorcy.

API `java.io.PushbackInputStream` **1.0**

- `PushbackInputStream(InputStream in)`

  tworzy strumień sprawdzający wartość następnego w kolejce bajta.

- `PushbackInputStream(InputStream we, int size)`

  tworzą strumień umożliwiający podgląd kolejnego bajta wraz z buforem o podanym rozmiarze.

- `void unread(int b)`

  wstawia bajt z powrotem do strumienia, dzięki czemu przy następnym wywołaniu `read` zostanie on ponownie odczytany.

  Parametry: `b` zwracany bajt

Strumienie tekstowe

Zapisując dane, możemy wybierać pomiędzy formatem binarnym i tekstowym. Dla przykładu: jeżeli liczba całkowita 1234 zostanie zapisana w postaci binarnej, w pliku pojawi się sekwencja bajtów `00 00 04 D2` (w notacji szesnastkowej). W formacie tekstowym liczba ta zostanie zapisana jako łańcuch `"1234"`. Mimo iż zapis danych w postaci binarnej jest szybki i efektywny, to uzyskany wynik jest kompletnie nieczytelny dla ludzi. W poniższym podrozdziale skoncentrujemy się na *tekstowym* wejściu-wyjściu.

Zapisując łańcuchy znakowe, musimy uwzględnić sposób *kodowania znaków*. W przypadku kodowania UTF-16 łańcuch `"1234"` zostanie zakodowany jako `00 31 00 32 00 33 00 34` (w notacji szesnastkowej). Jednakże obecnie większość środowisk, w których uruchamiamy programy w języku Java, używa swojego własnego formatu tekstu. W kodzie ISO 8859-1, najczęściej stosowanym w USA i Europie Zachodniej, nasz przykładowy łańcuch zostanie zapisany jako `31 32 33 34`, bez bajtów o wartości zero.

Klasa `OutputStreamWriter` zamienia strumień znaków Unicode na strumień bajtów, stosując odpowiednie kodowanie znaków. Natomiast klasa `InputStreamReader` zamienia strumień wejścia, zawierający bajty (reprezentujące znaki za pomocą określonego kodowania), na obiekt udostępniający znaki Unicode.

Poniżej przedstawiamy sposób utworzenia obiektu wejścia, wczytującego znaki z konsoli i automatycznie konwertującego je na Unicode.

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

Obiekt wejścia korzysta z domyślnego kodowania lokalnego systemu, na przykład ISO 8859-1 Możemy wybrać inny sposób kodowania, podając jego nazwę w konstruktorze `InputStream ↪Reader`, na przykład:

```
InputStreamReader in = new InputStreamReader(new FileInputStream("kremlin.dat"),
↪"ISO8859_5");
```

Więcej informacji na temat kodowania znaków znajdziesz w punkcie „Zbiory znaków” na stronie 35.

Ponieważ obiekty tekstowego wejścia i wyjścia są tak często dołączane do plików, Java dostarcza w tym celu dwie wygodne klasy: `FileReader` i `FileWriter`. Na przykład instrukcja

```
FileWriter out = new FileWriter("output.txt");
```

jest równoznaczna z

```
FileWriter out = new FileWriter(new FileOutputStream("output.txt");
```

Zapisywanie tekstu

W celu zapisania tekstu korzystamy z klasy `PrintWriter`. Dysponuje ona metodami umożliwiającymi zapis łańcuchów i liczb w formacie tekstowym. Dla wygody programistów ma ona konstruktor umożliwiający połączenie obiektu klasy PrintWriter z FileWriter. Zatem instrukcja

```
PrintWriter out = new PrintWriter("employee.txt");
```

stanowi odpowiednik instrukcji

```
PrintWriter out = new PrintWriter(new FileWriter("employee.txt"));
```

Do zapisywania danych za pomocą obiektu klasy `PrintWriter` używamy tych samych metod `print` i `println`, których używaliśmy dotąd z obiektem `System.out`. Możemy wykorzystywać je do zapisu liczb (`int`, `short`, `long`, `float`, `double`), znaków, wartości logicznych, łańcuchów znakowych i obiektów.

Spójrzmy na poniższy kod:

```
String name = "Harry Hacker";
double salary = 75000;
out.print(name);
out.print(' ');
out.println(salary);
```

Rezultatem jego wykonania będzie wysłanie napisu

```
Harry Hacker 75000.0
```

do strumienia `out`. Następnie znaki zostaną skonwertowane na bajty i zapisane w pliku *employee.txt*.

Metoda `println` automatycznie dodaje znak końca wiersza, odpowiedni dla danego systemu operacyjnego (`"\r\n"` w systemie Windows, `"\n"` w Unix). Znak końca wiersza możemy pobrać, stosując wywołanie `System.getProperty("line.separator")`.

Jeżeli obiekt zapisu znajduje się w *trybie automatycznego opróżniania*, w chwili wywołania metody `println` wszystkie znaki w buforze zostaną wysłane do odbiorcy (obiekty PrintWriter zawsze są buforowane). Domyślnie automatyczne opróżnianie jest *wyłączone*. Automatyczne opróżnianie możemy włączać i wyłączać przy użyciu konstruktora `PrintWriter(Writer out, boolean autoFlush)`:

```
PrintWriter out = new PrintWriter(new FileWriter("employee.txt", true);
// automatyczne opróżnianie
```

Metody `print` nie wyrzucają wyjątków. Aby sprawdzić, czy ze strumieniem jest wszystko w porządku, wywołujemy metodę `checkError`.

Weterani Javy prawdopodobnie zastanawiają się, co się stało z klasą `PrintStream` i obiektem `System.out`. W języku Java 1.0 klasa `PrintStream` obcinała znaki Unicode do znaków ASCII, po prostu opuszczając górny bajt. Takie rozwiązanie nie pozwalało na przenoszenie kodu na inne platformy i w języku Java 1.1 zostało zastąpione przez koncepcję obiektów odczytu i zapisu. Ze względu na konieczność zachowania zgodności `System.in`, `System.out` i `System.err` wciąż są strumieniami, nie obiektami odczytu i zapisu. Ale obecna klasa `PrintStream` konwertuje znaki Unicode na schemat kodowania lokalnego systemu w ten sam sposób, co klasa `PrintWriter`. Gdy używamy metod `print` i `println`, obiekty `PrintStream` działają tak samo jak obiekty `PrintWriter`, ale w przeciwieństwie do PrintWriter pozwalają wysyłać bajty za pomocą metod `write(int)` i `write(byte[])`.

`java.io.PrintWriter` **1.1**

- `PrintWriter(Writer out)`
- `PrintWriter(Writer out, boolean autoFlush)`

  tworzy nowy obiekt klasy `PrintWriter`.

  Parametry: `out` obiekt zapisu tekstu.

  `autoFlush` `true` oznacza, że metody `println` będą opróżniać bufor (domyślnie: false).

- `PrintWriter(OutputStream out)`
- `PrintWriter(OutputStream out, boolean autoFlush)`

  tworzy nowy obiekt klasy `PrintWriter` na podstawie istniejącego obiektu typu `OutputStream`, poprzez utworzenie pośredniczącego obiektu klasy OutputStreamWriter.
- `PrintWriter(String filename)`
- `PrintWriter(File file)`

  tworzy nowy obiekt klasy `PrintWriter` zapisujący dane do pliku poprzez utworzenie pośredniczącego obiektu klasy `FileWriter`.
- `void print(Object obj)`

  drukuje łańcuch zwracany przez metodę `toString` danego obiektu.

  Parametry: `obj` drukowany obiekt.
- `void print(String p)`

  drukuje łańcuch Unicode.
- `void println(String p)`

  drukuje łańcuch zakończony znakiem końca wiersza. Jeżeli automatyczne opróżnianie jest włączone, opróżnia bufor strumienia.
- `void print(char[] p)`

  drukuje tablicę znaków Unicode.
- `void print(char c)`

  drukuje znak Unicode.
- `void print(int i)`
- `void print(long l)`
- `void print(float f)`
- `void print(double d)`
- `void print(boolean b)`

  drukuje podaną wartość w formacie tekstowym.
- `void printf(String format, Object... args)`

  drukuje podane wartości według łańcucha formatującego. Specyfikację łańcucha formatującego znajdziesz w rozdziale 3. książki *Java 2. Podstawy*.
- `boolean checkError()`

  zwraca `true`, jeżeli wystąpił błąd formatowania lub zapisu. Jeżeli w strumieniu danych wystąpi błąd, strumień zostanie uznany za niepewny (ang. *tainted*) i wszystkie następne wywołania metody `checkError` będą zwracać `true`.

Wczytywanie tekstu

Wiemy już, że:

- aby zapisać dane w formacie binarnym, używamy klasy `DataOutputStream`;
- aby zapisać dane w formacie tekstowym, używamy klasy `PrintWriter`.

Na tej podstawie można się domyślać, że istnieje również klasa analogiczna do `DataInput➥Stream`, która pozwoli nam czytać dane w formacie tekstowym. Najbliższym odpowiednikiem jest w tym przypadku klasa `Scanner`, którą wykorzystywaliśmy intensywnie w książce *Java 2. Podstawy*. Niestety, przed wprowadzeniem Java SE 5.0 można było użyć w tym celu jedynie klasy `BufferedReader`. Ma ona metodę `readLine` pozwalającą pobrać wiersz tekstu. Aby ją wykorzystać, musimy najpierw połączyć obiekt typu `BufferedReader` ze źródłem wejścia.

```
BufferedReader in = new BufferedReader(new FileReader("employee.txt"));
```

Jeżeli dalsze wczytywanie nie jest możliwe, metoda `readLine` zwraca `null`. Typowa pętla pobierania danych wygląda więc następująco:

```
String line;
while ((line = in.readLine()) != null)
{
 operacje na danych line
}
```

Jednak klasa `BufferedReader` nie udostępnia metod odczytu danych liczbowych. Dlatego do odczytu danych sugerujemy zastosowanie klasy `Scanner`.

Zapis obiektów w formacie tekstowym

W tym podrozdziale przeanalizujemy działanie przykładowego programu, który będzie zapisywać tablicę obiektów typu `Employee` w pliku tekstowym. Dane każdego obiektu zostaną zapisane w osobnym wierszu. Wartości pól składowych zostaną oddzielone od siebie separatorami. Jako separatora używamy pionowej kreski (`|`) (innym popularnym separatorem jest dwukropek (`:`), zabawa polega na tym, że każdy programista używa innego separatora). Naturalnie, taki wybór stawia przed nami pytanie, co będzie, jeśli znak `|` znajdzie się w jednym z zapisywanych przez nas łańcuchów?

Oto przykładowy zbiór danych obiektów:

```
Harry Hacker|35500|1989|10|1
Carl Cracker|75000|1987|12|15
Tony Tester|38000|1990|3|15
```

Zapis tych rekordów jest prosty. Ponieważ korzystamy z pliku tekstowego, używamy klasy `PrintWriter`. Po prostu zapisujemy wszystkie pola składowe, za każdym z nich stawiając `|`, albo też, po ostatnim polu, `\n`. Operacje te wykona poniższa metoda `writeData`, którą dodamy do klasy `Employee`.

```
public void writeData(PrintWriter out) throws IOException
{
 GregorianCalendar calendar = new GregorianCalendar();
 kalendarz.setTime(hireDay);
 out.println(name + "|"
  + salary + "|"
  + calendar.get(Calendar.YEAR) + "|"
  + (calendar.get(Calendar.MONTH) + 1) + "|"
  + calendar.get(Calendar.DAY_OF_MONTH));
}
```

Aby odczytać te dane, wczytujemy po jednym wierszu tekstu i rozdzielamy pola składowe. Do wczytania wierszy użyjemy obiektu klasy `Scanner`, a metoda `String.split` pozwoli nam wyodrębnić poszczególne tokeny.

```
public void readData(Scanner in)
{
   String line = in.nextLine();
   String[] tokens = line.split("\\|");
   name = tokens[0];
   salary = Double.parseDouble(tokens[1]);
   int y = Integer.parseInt(tokens[2]);
   int m = Integer.parseInt(tokens[3]);
   int d = Integer.parseInt(tokens[4]);
   GregorianCalendar calendar = new GregorianCalendar(y, m - 1, d);
   hireDay = calendar.getTime();
}
```

Parametrem metody `split` jest wyrażenie regularne opisujące separator. Wyrażenie regularne omówimy bardziej szczegółowo pod koniec bieżącego rozdziału. Ponieważ pionowa kreska ma specjalne znaczenie w wyrażeniach regularnych, to musimy poprzedzić ją znakiem \. Ten z kolei musimy poprzedzić jeszcze jednym znakiem \ — w efekcie uzyskując wyrażenie postaci `"\\|"`.

Kompletny program został przedstawiony na listingu 1.1. Metoda statyczna

```
void writeData(Employee[] e, PrintWriter out)
```

najpierw zapisuje rozmiar tablicy, a następnie każdy z rekordów. Metoda statyczna

```
Employee[] readData(BufferedReader in)
```

najpierw wczytuje rozmiar tablicy, a następnie każdy z rekordów. Wymaga to zastosowania pewnej sztuczki:

```
int n = in.nextInt();
in.nextLine(); // konsumuje znak nowego wiersza
Employee[] employees = new Employee[n];
for (int i = 0; i < n; i++)
{
   employees[i] = new Employee();
   employees[i].readData(in);
}
```

Wywołanie metody `nextInt` wczytuje rozmiar tablicy, ale nie następujący po nim znak nowego wiersza. Musimy zatem go pobrać (wywołując metodę `nextLine`), aby metoda `readData` mogła uzyskać kolejny wiersz.

Listing 1.1. *TextFileTest.java*

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

/**
 * @version 1.12 2007-06-22
 * @author Cay Horstmann
 */
public class TextFileTest
{
   public static void main(String[] args)
   {
      Employee[] staff = new Employee[3];

      staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
      staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);

      try
      {
         // zapisuje wszystkie rekordy pracowników w pliku employee.dat
         PrintWriter out = new PrintWriter("employee.dat");
         writeData(staff, out);
         out.close();

         // wczytuje wszystkie rekordy do nowej tablicy
         Scanner in = new Scanner(new FileReader("employee.dat"));
         Employee[] newStaff = readData(in);
         in.close();

         // wyświetla wszystkie wczytane rekordy
         for (Employee e : newStaff)
            System.out.println(e);
      }
      catch (IOException exception)
      {
         exception.printStackTrace();
      }
   }

   /**
    * Zapisuje dane wszystkich obiektów klasy Employee
    * umieszczonych w tablicy
    * do obiektu klasy PrintWriter
    * @param employees tablica obiektów klasy Employee
    * @param out obiekt klasy PrintWriter
    */
   private static void writeData(Employee[] employees, PrintWriter out) throws
   ↪IOException
   {
      // zapisuje liczbę obiektów
      out.println(employees.length);

      for (Employee e : employees)
         e.writeData(out);
   }
```

```
   /**
    * Wczytuje tablicę obiektów klasy Employee
    * @param in obiekt klasy Scanner
    * @return tablica obiektów klasy Employee
    */
   private static Employee[] readData(Scanner in)
   {
      // pobiera rozmiar tablicy
      int n = in.nextInt();
      in.nextLine(); // pobiera znak nowego wiersza

      Employee[] employees = new Employee[n];
      for (int i = 0; i < n; i++)
      {
         employees[i] = new Employee();
         employees[i].readData(in);
      }
      return employees;
   }
}

class Employee
{
   public Employee()
   {
   }

   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   public String toString()
```

```
    {
       return getClass().getName() + "[name=" + name + ",salary=" + salary +
       ↪",hireDay=" + hireDay
             + "]";
    }

    /**
     * Zapisuje dane obiektu klasy Employee
     * do obiektu klasy PrintWriter
     * @param out obiekt klasy PrintWriter
     */
    public void writeData(PrintWriter out)
    {
       GregorianCalendar calendar = new GregorianCalendar();
       calendar.setTime(hireDay);
       out.println(name + "|" + salary + "|" + calendar.get(Calendar.YEAR) + "|"
             + (calendar.get(Calendar.MONTH) + 1) + "|" +
             ↪calendar.get(Calendar.DAY_OF_MONTH));
    }

    /**
     * Wczytuje dane obiektu klasy Employee
     * @param in obiekt klasy Scanner
     */
    public void readData(Scanner in)
    {
       String line = in.nextLine();
       String[] tokens = line.split("\\|");
       name = tokens[0];
       salary = Double.parseDouble(tokens[1]);
       int y = Integer.parseInt(tokens[2]);
       int m = Integer.parseInt(tokens[3]);
       int d = Integer.parseInt(tokens[4]);
       GregorianCalendar calendar = new GregorianCalendar(y, m - 1, d);
       hireDay = calendar.getTime();
    }

    private String name;
    private double salary;
    private Date hireDay;
}
```

Zbiory znaków

We wcześniejszych edycjach platformy Java problem znaków narodowych obsługiwany był w mało systematyczny sposób. Sytuacja zmieniła się z wprowadzeniem pakietu java.nio w Java SE 1.4, który unifikuje konwersje zbiorów znaków, udostępniając klasę `Charset` (zwróćmy uwagę na małą literę `s` w nazwie klasy).

Zbiór znaków stanowi odwzorowanie pomiędzy dwubajtowymi kodami Unicode i sekwencjami bajtów stosowanymi w lokalnych kodowaniach znaków. Jednym z najpopularniejszych kodowań znaków jest ISO-8859-1, które koduje za pomocą jednego bajta pierwszych 256 znaków z zestawu Unicode. Coraz większe znaczenie zyskuje również ISO08859-15, w którym

zastąpiono część mniej przydatnych znaków kodu ISO-8859-1 akcentowanymi znakami języka francuskiego i fińskiego, a przede wszystkim zamiast znaku waluty międzynarodowej ¤ umieszczono symbol euro (€) o kodzie `0xA4`. Innymi przykładami kodowań znaków są kodowania o zmiennej liczbie bajtów stosowane dla języków japońskiego i chińskiego.

Klasa Charset używa nazw zbiorów znaków zgodnie ze standardem określonym przez IANA Character Set Registry (*http://www.iana.org/assignments/character-sets*). Nazwy te różnią się nieco od nazw stosowanych w poprzednich wersjach. Na przykład „oficjalną" nazwą ISO-8859-1 jest teraz `"ISO-8859-1"` zamiast `"ISO8859_1"` preferowaną w Java SE 1.3 i wcześniejszych wersjach.

Opis kodowania ISO 8859 znajdziesz na stronie *http://aspell.net/charsets/iso8859.html*.

Obiekt klasy `Charset` uzyskujemy, wywołując metodę statyczną `forName`, której podajemy oficjalną nazwę zbioru znaków lub jeden z jej synonimów:

```
Charset cset = Charset.forName("ISO-8859-1");
```

Duże i małe litery nie są rozróżniane w nazwach zbiorów znaków.

Ze względu na konieczność zachowania zgodności z innymi konwencjami nazw nazwa każdego zbioru znaków może mieć wiele synonimów. Na przykład dla ISO-8859-1 istnieją następujące synonimy:

```
ISO8859-1
ISO_8859_1
ISO8859_1
ISO_8859-1
ISO_8859-1:1987
8859_1
latin1
l1
csISOLatin1
iso-ir-100
cp819
IBM819
IBM-819
819
```

Metoda `aliases` zwraca obiekt klasy `Set` zawierający synonimy. Poniżej przedstawiamy kod umożliwiający przeglądanie synonimów:

```
Set<String> aliases = cset.aliases();
for (String alias : aliases)
   System.out.println(alias);
```

Aby dowiedzieć się, które zbiory znaków są dostępne dla konkretnej implementacji, wywołujemy metodę statyczną `availableCharsets`. Poniższy kod pozwala poznać nazwy wszystkich dostępnych zbiorów znaków:

```
Map<String, Charset> charsets = Charset.availableCharsets();
for (String name : charsets.keySet())
   System.out.println(name);
```

W tabeli 1.1 zostały przedstawione wszystkie kodowania znaków, które musi obsługiwać każda implementacja platformy Java. W tabeli 1.2 wymienione zostały schematy kodowania instalowane domyślnie przez pakiet JDK. Zbiory znaków przedstawione w tabeli 1.3 są instalowane tylko w przypadku systemów operacyjnych używających języków innych niż europejskie.

Tabela 1.1. *Kodowania znaków wymagane na platformie Java*

Standardowa nazwa obiektu Charset	Nazwa tradycyjna	Opis
US-ASCII	ASCII	American Standard Code for Information Exchange
ISO-8859-1	ISO8859_1	ISO 8859-1, alfabet Latin 1
UTF-8	UTF8	8-bitowy Unicode Transformation Format
UTF-16	UTF-16	16-bitowy Unicode Transformation Format, porządek bajtów określony przez opcjonalny znacznik
UTF-16BE	UnicodeBigUnmarked	16-bitowy Unicode Transformation Format, porządek bajtów od najstarszego
UTF-16LE	UnicodeLittleUnmarked	16-bitowy Unicode Transformation Format, porządek bajtów od najmłodszego

Tabela 1.2. *Podstawowe kodowania znaków*

Standardowa nazwa obiektu Charset	Nazwa tradycyjna	Opis
ISO8859-2	ISO8859_2	ISO 8859-2, alfabet Latin 2
ISO8859-4	ISO8859_4	ISO 8859-4, alfabet Latin 4
ISO8859-5	ISO8859_5	ISO 8859-5, alfabet Latin/Cyrillic
ISO8859-7	ISO8859_7	ISO 8859-7, alfabet Latin/Greek
ISO8859-9	ISO8859_9	ISO 8859-9, alfabet Latin 5
ISO8859-13	ISO8859_13	ISO 8859-13, alfabet Latin 7
ISO8859-15	ISO8859_15	ISO 8859-15, alfabet Latin 9
windows-1250	Cp1250	Windows, wschodnioeuropejski
windows-1251	Cp1251	Windows Cyrillic
windows-1252	Cp1252	Windows, Latin 1
windows-1253	Cp1253	Windows, grecki
windows-1254	Cp1254	Windows, turecki
windows-1257	Cp1257	Windows, bałtycki

Tabela 1.3. *Rozszerzone kodowania znaków*

Standardowa nazwa obiektu Charset	Nazwa tradycyjna	Opis
Big5	Big5	Big5, tradycyjny chiński
Big5-HKSCS	Big5_HKSCS	Big5, tradycyjny chiński z rozszerzeniami Hongkong
EUC-JP	EUC_JP	JIS X 0201, 0208, 0212, kodowanie EUC, japoński
EUC-KR	EUC_KR	KS C 5601, kodowanie EUC, koreański
GB18030	GB18030	uproszczony chiński, standard PRC
GBK	GBK	GBK, uproszczony chiński
ISCII91	ISCII91	ISCII91, indu
ISO-2022-JP	ISO2022JP	JIS X 0201, 0208 w postaci ISO 2022, japoński
ISO-2022-KR	ISO2022KR	ISO 2022 KR, koreański
ISO8859-3	ISO8859_3	ISO 8859-3, Latin 3
ISO8859-6	ISO8859_6	ISO 8859-6, alfabet łaciński/arabski
ISO8859-8	ISO8859_8	ISO 8859-8, alfabet łaciński/hebrajski
Shift_JIS	SJIS	Shift_JIS, japoński
TIS-620	TIS620	TIS620, tajski
windows-1255	Cp1255	Windows, hebrajski
windows-1256	Cp1256	Windows, arabski
windows-1258	Cp1258	Windows, wietnamski
windows-3lj	MS392	Windows, japoński
x-EUC-CN	EUC_CN	GB2313, kodowanie EUC, uproszczony chiński
x-EUC-JP-LINUX	EUC_JP_LINUX	JIS X 0201, 0208, kodowanie EUC, japoński
x-EUC-TW	EUC_TW	CNS11643 (Plane 1-3), kodowanie EUC, tradycyjny chiński
x-MS950-HKSCS	MS950_HKSCS	Windows, tradycyjny chiński z rozszerzeniami Hongkong
x-mswin-936	MS936	Windows, uproszczony chiński
x-windows-949	MS949	Windows, koreański
x-windows-950	MS950	Windows, tradycyjny chiński

Lokalne schematy kodowania nie mogą oczywiście reprezentować wszystkich znaków Unicode. Jeśli znak nie jest reprezentowany, to zostaje przekształcony na znak ?.

Dysponując zbiorem znaków, możemy użyć go do konwersji łańcuchów Unicode i sekwencji bajtów. Oto przykład kodowania łańcucha Unicode:

```
String str = . . .;
ByteBuffer buffer = cset.encode(str);
byte[] bytes = buffer.array();
```

Natomiast aby dokonać konwersji w kiedunku przeciwnym, potrzebny będzie bufor. Wykorzystamy metodę statyczną wrap tablicy `ByteBuffer`, aby przekształcić tablicę bajtów w bufor. W wyniku działania metody `decode` otrzymujemy obiekt klasy `CharBuffer`. Wystarczy wywołać jego metodę `toString`, aby uzyskać łańcuch znaków.

```
byte[] bytes = . . .;
ByteBuffer bbuf = ByteBuffer.wrap(bytes, offset, length);
CharBuffer cbuf = cset.decode(bbuf);
String str = cbuf.toString();
```

API `java.nio.charset.Charset` **1.4**

- `static SortedMap availableCharsets()`
  pobiera wszystkie zbiory znaków dostępne dla maszyny wirtualnej. Zwraca mapę, której kluczami są nazwy zbiorów znaków, a wartościami same zbiory.
- `static Charset forName(String name)`
  zwraca zbiór znaków o podanej nazwie.
- `Set aliases()`
  zwraca zbiór synonimów nazwy danego zbioru znaków.
- `ByteBuffer encode(String str)`
  dokonuje konwersji podanego łańcucha na sekwencję bajtów.
- `CharBuffer decode(ByteBuffer buffer)`
  dokonuje konwersji sekwencji bajtów. Nierozpoznane bajty są zamieniane na specjalny znak Unicode (`'\uFFFD'`).

API `java.nio.ByteBuffer` **1.4**

- `byte[] array()`
  zwraca tablicę bajtów, którą zarządza ten bufor.
- `static ByteBuffer wrap(byte[] bytes)`
- `static ByteBuffer wrap(byte[] bytes, int offset, int length)`
  zwraca bufor, który zarządza podaną tablicą bajtów lub jej określonym zakresem.

API `java.nio.CharBuffer`

- `char[] array()`
  zwraca tablicę kodów, którą zarządza ten bufor.
- `char charAt(int index)`
  zwraca kod o podanym indeksie.

- `String toString()`
  zwraca łańcuch, który tworzą kody zarządzane przez ten bufor.

Odczyt i zapis danych binarnych

Aby zapisać liczbę, znak, wartość logiczną lub łańcuch, korzystamy z jednej z poniższych metod interfejsu `DataOutput`:

```
writeChars
writeByte
writeInt
writeShort
writeLong
writeFloat
writeDouble
writeChar
writeBoolean
writeUTF
```

Na przykład, `writeInt` zawsze zapisuje liczbę `integer` jako wartość czterobajtową, niezależnie od liczby jej cyfr, a `writeDouble` zawsze zapisuje liczby `double` jako wartości ośmiobajtowe. Rezultat tych działań nie jest czytelny dla człowieka, ale ponieważ wymagana ilość bajtów jest taka sama dla każdej wartości danego typu, to wczytanie ich z powrotem będzie szybsze niż parsowanie zapisu tekstowego.

Zależnie od platformy użytkownika, liczby całkowite i zmiennoprzecinkowe mogą być przechowywane w pamięci na dwa różne sposoby. Załóżmy, że pracujesz z czterobajtową wartością, taką jak `int`, na przykład 1234, czyli 4D2 w zapisie szesnastkowym (1234 = 4×256+13×16+2). Może ona zostać przechowana w ten sposób, że pierwszym z czterech bajtów pamięci będzie bajt najbardziej znaczący (ang. *most significant byte*, *MSB*): 00 00 04 D2. Albo w taki sposób, że będzie to bajt najmłodszy (ang. *least significant byte*, *LSB*): D2 04 00 00. Pierwszy sposób stosowany jest przez maszyny SPARC, a drugi przez procesory Pentium. Może to powodować problemy z przenoszeniem nawet najprostszych plików danych pomiędzy różnymi platformami. W języku Java zawsze stosowany jest pierwszy sposób, niezależnie od procesora. Dzięki temu pliki danych programów w języku Java są niezależne od platformy.

Metoda `writeUTF` zapisuje łańcuchy, używając zmodyfikowanej wersji 8-bitowego kodu UTF (ang. *Unicode Text Format*). Zamiast po prostu zastosować od razu standardowe kodowanie UTF-8 (przedstawione w tabeli 1.4), znaki łańcucha są najpierw reprezentowane w kodzie UTF-16 (patrz tabela 1.5), a dopiero potem przekodowywane na UTF-8. Wynik takiego kodowania różni się dla znaków o kodach większych od 0xFFFF. Kodowanie takie stosuje się dla zachowania zgodności z maszynami wirtualnymi powstałymi, gdy Unicode zadowalał się tylko 16 bitami.

Ponieważ opisana modyfikacja kodowania UTF-8 stosowana jest wyłącznie na platformie Java, to metody `writeUTF` powinniśmy używać tylko do zapisu łańcuchów przetwarzanych przez programy wykonywane przez maszynę wirtualną Java. W pozostałych przypadkach należy używać metody `writeChars`.

Tabela 1.4. *Kodowanie UTF-8*

Zakres znaków	Kodowanie
0...7F	$0a_6a_5a_4a_3a_2a_1a_0$
80...7FF	$110a_{10}a_9a_8a_7a_6\ 10a_5a_4a_3a_2a_1a_0$
800...FFFF	$1110a_{15}a_{14}a_{13}a_{12}\ 10a_{11}a_{10}a_9a_8a_7a_6\ 10a_5a_4a_3a_2a_1a_0$
10000...10FFFF	$11110a_{20}a_{19}a_{18}\ 10a_{17}a_{16}a_{15}a_{14}a_{13}a_{12}\ 10a_{11}a_{10}a_9a_8a_7a_6\ 10a_5a_4a_3a_2a_1a_0$

Tabela 1.5. *Kodowanie UTF-16*

Zakres znaków	Kodowanie
0...FFFF	$a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}a_9a_8\ a_7a_6a_5a_4a_3a_2a_1a_0$
10000...10FFFF	$110110b_{19}b_{18}b_{17}b_{16}a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}\ 110111a_9a_8\ a_7a_6{}_5a_4a_3a_2a_1a_0$ gdzie $b_{19}b_{18}b_{17}b_{16} = a_{20}a_{19}a_{18}a_{17}a_{16} - 1$

Definicje kodów UTF-8 i UTF-16 znajdziesz w dokumentach, odpowiednio: RFC 2279 (*http://ietf.org/rfc/rfc2279.txt*) i RFC 2781 (*http://ietf.org/rfc/rfc2781.txt*).

Aby odczytać dane, korzystamy z poniższych metod interfejsu DataInput:

```
readInt
readShort
readLong
readFloat
readDouble
readChar
readBoolean
readUTF
```

Klasa `DataInputStream` implementuje interfejs `DataInput`. Aby odczytać dane binarne z pliku, łączymy obiekt klasy `DataInputStream` ze źródłem bajtów, takim jak na przykład obiekt klasy `FileInputStream`:

```
DataInputStream in = new DataInputStream(new FileInputStream("employee.dat"));
```

Podobnie, aby zapisać dane binarne, używamy klasy `DataOutputStream` implementującej interfejs `DataOutput`:

```
DataOutputStream out = new DataOutputStream(new FileOutputStream("employee.dat"));
```

API `java.io.DataInput` **1.0**

- `boolean readBoolean()`
- `byte readByte()`
- `char readChar()`
- `double readDouble()`
- `float readFloat()`

- `int readInt()`
- `long readLong()`
- `short readShort()`

  wczytuje wartość określonego typu.

- `void readFully(byte[] b)`

  wczytuje bajty do tablicy `b`, blokując wątek, dopóki wszystkie bajty nie zostaną wczytane.

  Parametry: `b` bufor, do którego zapisywane są dane.

- `void readFully(byte[] b, int off, int len)`

  wczytuje bajty do tablicy `b`, blokując wątek, dopóki wszystkie bajty nie zostaną wczytane.

  Parametry: `b` bufor, do którego zapisywane są dane.
  `off` indeks pierwszego bajta.
  `len` maksymalna ilość odczytanych bajtów.

- `String readUTF()`

  wczytuje łańcuch znaków zapisanych w zmodyfikowanym formacie UTF-8.

- `int skipBytes(int n)`

  ignoruje `n` bajtów, blokując wątek, dopóki wszystkie bajty nie zostaną zignorowane.

  Parametry: `n` liczba ignorowanych bajtów.

API `java.io.DataOutput` 1.0

- `void writeBoolean(boolean b)`
- `void writeByte(int b)`
- `void writeChar(char c)`
- `void writeDouble(double d)`
- `void writeFloat(float f)`
- `void writeInt(int i)`
- `void writeLong(long l)`
- `void writeShort(short s)`

  zapisują wartość określonego typu.

- `void writeChars(String s)`

  zapisuje wszystkie znaki podanego łańcucha.

- `void writeUTF(String s)`

  zapisuje łańcuch znaków w zmodyfikowanym formacie UTF-8.

Strumienie plików o swobodnym dostępie

Strumień `RandomAccessFile` pozwala pobrać lub zapisać dane w dowolnym miejscu pliku. Do plików dyskowych możemy uzyskać swobodny dostęp, inaczej niż w przypadku strumieni danych pochodzących z sieci. Plik o swobodnym dostępie możemy otworzyć w trybie tylko do odczytu albo zarówno do odczytu, jak i do zapisu. Określamy to, używając jako drugiego argumentu konstruktora łańcucha `"r"` (odczyt) lub `"rw"` (odczyt i zapis).

```
RandomAccesFile in = new RandomAccesFile("employee.dat", "r");
RandomAccesFile inOut = new RandomAccesFile("employee.dat", "rw");
```

Otwarcie istniejącego pliku przy użyciu `RandomAccessFile` nie powoduje jego skasowania.

Plik o swobodnym dostępie posiada *wskaźnik pliku*. Wskaźnik pliku opisuje pozycję następnego bajta, który zostanie wczytany lub zapisany. Metoda `seek` zmienia położenie wskaźnika, określając numer bajta, na który wskazuje. Argumentem metody `seek` jest liczba typu `long` z przedziału od 0 do długości pliku w bajtach.

Metoda `getFilePointer` zwraca aktualne położenie wskaźnika pliku.

Klasa `RandomAccessFile` implementuje zarówno interfejs `DataInput`, jak i `DataOutput`. Aby czytać z pliku o swobodnym dostępie, używamy tych samych metod, np. `readInt/writeInt` lub `readChar/writeChar`, które omówiliśmy w poprzednim podrozdziale.

Prześledzimy teraz działanie programu, który przechowuje rekordy pracowników w pliku o swobodnym dostępie. Każdy z rekordów będzie mieć ten sam rozmiar, co ułatwi nam ich wczytywanie. Załóżmy na przykład, że chcemy ustawić wskaźnik pliku na trzecim rekordzie. Musimy zatem wyznaczyć bajt, na którym należy ustawić ten wskaźnik, a następnie możemy już wczytać rekord.

```
long n = 3;
in.seek((n - 1) * RECORD_SIZE);
Employee e = new Employee();
e.readData(in);
```

Jeśli zmodyfikujemy rekord i będziemy chcieli zapisać go w tym samym miejscu pliku, musimy pamiętać, aby przywrócić wskaźnik pliku na początek tego rekordu:

```
in.seek((n - 1) * RECORD_SIZE);
e.writeData(out);
```

Aby określić całkowitą liczbę bajtów w pliku, używamy metody `length`. Całkowitą liczbę rekordów w pliku ustalamy, dzieląc liczbę bajtów przez rozmiar rekordu.

```
long nbytes = in.length(); // długość w bajtach
int nrecords = (int) (nbytes / RECORD_SIZE);
```

Liczby całkowite i zmiennoprzecinkowe posiadają reprezentację binarną o stałej liczbie bajtów. W przypadku łańcuchów znaków sytuacja jest nieco trudniejsza. Stworzymy zatem dwie metody pomocnicze pozwalające zapisywać i wczytywać łańcuchy o ustalonym rozmiarze.

Metoda `writeFixedString` zapisuje określoną liczbę kodów, zaczynając od początku łańcucha. (Jeśli jest ich za mało, to dopełnia łańcuch wartościami zerowymi).

```
public static void writeFixedString(String s, int size, DataOutput out)
   throws IOException
{
   for (int i = 0; i < size; i++)
   {
      char ch = 0;
      if (i < s.length()) ch = s.charAt(i);
      out.writeChar(ch);
   }
}
```

Metoda `readFixedString` wczytuje `size` kodów znaków ze strumienia wejściowego lub do momentu napotkania wartości zerowej. Wszystkie pozostałe wartości zerowe zostają pominięte. Dla lepszej efektywności metoda używa klasy `StringBuilder` do wczytania łańcucha.

```
public static String readFixedString(int size, DataInput in)
   throws IOException
{
   StringBuilder b = new StringBuilder(size);
   int i = 0;
   boolean more = true;
   while (more && i < size)
   {
      char ch = in.readChar();
      i++;
      if (ch == 0) more = false;
      else b.append(ch);
   }
   in.skipBytes(2 * (size - i));
   return b.toString();
}
```

Metody `writeFixedString` i `readFixedString` umieściliśmy w klasie pomocniczej `DataIO`.

Aby zapisać rekord o stałym rozmiarze, zapisujemy po prostu wszystkie jego pola w formacie binarnym.

```
public void writeData(DataOutput out) throws IOException
{
   DataIO.writeFixedString(name, NAME_SIZE, out);
   out.writeDouble(salary);

   GregorianCalendar calendar = new GregorianCalendar();
   calendar.setTime(hireDay);
   out.writeInt(calendar.get(Calendar.YEAR));
   out.writeInt(calendar.get(Calendar.MONTH) + 1);
   out.writeInt(calendar.get(Calendar.DAY_OF_MONTH));
}
```

Odczyt rekordu jest równie prosty.

```
public void readData(DataInput in) throws IOException
{
   name = DataIO.readFixedString(NAME_SIZE, in);
   salary = in.readDouble();
   int y = in.readInt();
   int m = in.readInt();
```

```
      int d = in.readInt();
      GregorianCalendar calendar = new GregorianCalendar(y, m - 1, d);
      hireDay = calendar.getTime();
   }
```

Wyznaczmy jeszcze rozmiar każdego rekordu. Łańcuchy znakowe przechowujące nazwiska będą miały 40 znaków długości. W rezultacie każdy rekord będzie zajmować 100 bajtów:

- 40 znaków = 80 bajtów dla pola `name`
- 1 `double` = 8 bajtów dla pola `salary`
- 3 `int` = 12 bajtów dla pola `date`

Program przedstawiony na listingu 1.2 zapisuje trzy rekordy w pliku danych, a następnie wczytuje je w odwrotnej kolejności. Efektywne działanie programu wymaga pliku o swobodnym dostępie, ponieważ najpierw zostanie wczytany ostatni rekord.

Listing 1.2. *RandomFileTest.java*

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

/**
 * @version 1.11 2004-05-11
 * @author Cay Horstmann
 */
public class RandomFileTest
{
   public static void main(String[] args)
   {
      Employee[] staff = new Employee[3];

      staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
      staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);

      try
      {
         // zapisuje rekordy wszystkich pracowników w pliku employee.dat
         DataOutputStream out = new DataOutputStream(new
         ↪FileOutputStream("employee.dat"));
         for (Employee e : staff)
            e.writeData(out);
         out.close();

         // wczytuje wszystkie rekordy do nowej tablicy
         RandomAccessFile in = new RandomAccessFile("employee.dat", "r");
         // oblicza rozmiar tablicy
         int n = (int)(in.length() / Employee.RECORD_SIZE);
         Employee[] newStaff = new Employee[n];

         // wczytuje rekordy pracowników w odwrotnej kolejności
         for (int i = n - 1; i >= 0; i--)
         {
            newStaff[i] = new Employee();
            in.seek(i * Employee.RECORD_SIZE);
```

```
            newStaff[i].readData(in);
         }
         in.close();

         // wyświetla wczytane rekordy
         for (Employee e : newStaff)
            System.out.println(e);
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
   }
}

class Employee
{
   public Employee() {}

   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }

   /**
      Podnosi wynagrodzenie pracownika.
      @byPercent podwyżka procentowo
   */
   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   public String toString()
   {
      return getClass().getName()
         + "[name=" + name
         + ",salary=" + salary
```

```
            + ",hireDay=" + hireDay
            + "]";
      }

      /**
         Zapisuje dane pracownika
         @param out obiekt klasy DataOutput
      */
      public void writeData(DataOutput out) throws IOException
      {
         DataIO.writeFixedString(name, NAME_SIZE, out);
         out.writeDouble(salary);

         GregorianCalendar calendar = new GregorianCalendar();
         calendar.setTime(hireDay);
         out.writeInt(calendar.get(Calendar.YEAR));
         out.writeInt(calendar.get(Calendar.MONTH) + 1);
         out.writeInt(calendar.get(Calendar.DAY_OF_MONTH));
      }

      /**
         Wczytuje dane pracownika
         @param in obiekt klasy DataOutput
      */
      public void readData(DataInput in) throws IOException
      {
         name = DataIO.readFixedString(NAME_SIZE, in);
         salary = in.readDouble();
         int y = in.readInt();
         int m = in.readInt();
         int d = in.readInt();
         GregorianCalendar calendar = new GregorianCalendar(y, m - 1, d);
         hireDay = calendar.getTime();
      }

      public static final int NAME_SIZE = 40;
      public static final int RECORD_SIZE = 2 * NAME_SIZE + 8 + 4 + 4 + 4;

      private String name;
      private double salary;
      private Date hireDay;
   }

   class DataIO
   {
      public static String readFixedString(int size, DataInput in)
         throws IOException
      {
         StringBuilder b = new StringBuilder(size);
         int i = 0;
         boolean more = true;
         while (more && i < size)
         {
            char ch = in.readChar();
            i++;
            if (ch == 0) more = false;
            else b.append(ch);
```

```
        }
        in.skipBytes(2 * (size - i));
        return b.toString();
     }

     public static void writeFixedString(String s, int size, DataOutput out)
        throws IOException
     {
        for (int i = 0; i < size; i++)
        {
           char ch = 0;
           if (i < s.length()) ch = s.charAt(i);
           out.writeChar(ch);
        }
     }
  }
```

API `java.io.RandomAccessFile` **1.0**

- `RandomAccessFile(String file, String mode)`
- `RandomAccessFile(File file, String mode)`

  Parametry: `file` — plik, który ma zostać otwarty.

  `tryb` — `"r"` dla samego odczytu, `"rw"` dla odczytu i zapisu, `"rws"` dla odczytu i zapisu danych wraz z synchronicznym zapisem danych i metadanych dla każdej aktualizacji, `"rwd"` dla odczytu i zapisu danych wraz z synchronicznym zapisem tylko samych danych.

- `long getFilePointer()`

  zwraca aktualne położenie wskaźnika pliku.

- `void seek(long pos)`

  zmienia położenie wskaźnika pliku, przesuwając go o `pos` bajtów od początku pliku.

- `long length()`

  zwraca długość pliku w bajtach.

Strumienie plików ZIP

Pliki ZIP to archiwa, w których można przechowywać jeden lub więcej plików w postaci (zazwyczaj) skompresowanej. Każdy plik ZIP posiada nagłówek zawierający informacje, takie jak nazwa pliku i użyta metoda kompresji. W języku Java, aby czytać z pliku ZIP, korzystamy z klasy `ZipInputStream`. Odczyt dotyczy określonej pozycji w archiwum. Metoda `getNext`
`↪Entry` zwraca obiekt typu `ZipEntry` opisujący pozycję archiwum. Metoda `read` klasy `Zip`

↪InputStream zwraca wartość –1, gdy napotka koniec pozycji archiwum, a nie koniec całego pliku ZIP. Aby odczytać kolejną pozycję archiwum, musimy wtedy wywołać metodę close ↪Entry. Oto typowy kod wczytujący zawartość pliku ZIP:

```
ZipInputStream zin = ZipInputStream
 (new FileInputStream(zipname));
ZipEntry entry;
while ((entry = zin.getNextEntry()) != null)
{
 analizuj entry;
 wczytaj zawartość zin;
 zin.closeEntry();
}
zin.close();
```

Wczytując zawartość pozycji pliku ZIP, zwykle zamiast z podstawowej metody read lepiej będzie skorzystać z jakiegoś bardziej kompetentnego filtru strumienia. Dla przykładu, aby wydobyć z archiwum ZIP plik tekstowy, możemy skorzystać z poniższej pętli:

```
Scanner in = new Scanner(zin);
while (in.hasNextLine())
   operacje na in.nextLine();
```

Strumień wejścia ZIP wyrzuca wyjątek ZipException, jeżeli w czasie czytania pliku ZIP nastąpił błąd. Zazwyczaj dzieje się tak, gdy archiwum zostało uszkodzone.

Aby zapisać dane do pliku ZIP, używamy strumień ZipOutputStream. Dla każdej pozycji, którą chcemy umieścić w archiwum ZIP, tworzymy obiekt ZipEntry. Nazwę pliku przekazujemy konstruktorowi ZipEntry; konstruktor sam określa inne parametry, takie jak data pliku i metoda dekompresji. Jeśli chcemy, możemy zmienić ich wartości. Aby rozpocząć zapis nowego pliku w archiwum, wywołujemy metodę putNextEntry klasy ZipOutputStream. Następnie wysyłamy dane do strumienia ZIP. Po zakończeniu zapisu pliku wywołujemy metodę closeEntry. Wymienione operacje powtarzamy dla wszystkich plików, które chcemy skompresować w archiwum. Oto schemat kodu:

```
FileOutputStream fout = new FileOutputStream("test.zip");
ZipOutputStream zout = new ZipOutputStream(fout);
dla wszystkich plików
{
 ZipEntry ze = new ZipEntry(nazwapliku);
 zout.putNextEntry(kze);
 wyślij dane do zout;
 zout.closeEntry();
}
zout.close();
```

Pliki JAR (omówione w rozdziale 10. książki *Java 2. Podstawy* są po prostu plikami ZIP, zawierającymi specjalny rodzaj pliku, tzw. manifest. Do wczytania i zapisania manifestu używamy klas JarInputStream i JarOutputStream.

Strumienie ZIP są dobrym przykładem potęgi abstrakcji strumieni. Odczytując dane przechowywane w skompresowanej postaci, nie musimy zajmować się ich dekompresją. Źródło

bajtów formatu ZIP nie musi być plikiem — dane ZIP mogą być ściągane przez połączenie sieciowe. Na przykład za każdym razem, gdy mechanizm ładowania klas jakiegoś apletu wczytuje plik JAR, tak naprawdę wczytuje i dekompresuje dane pochodzące z sieci.

Artykuł dostępny pod adresem *http://www.javaworld.com/javaworld/jw-10-2000/*
↪jw-1027-toolbox.html przedstawia sposób modyfikacji archiwum ZIP.

Program przedstawiony na listingu 1.3 pozwala użytkownikowi otworzyć archiwum ZIP. Następnie wyświetla pliki przechowywane w tym archiwum w postaci listy, w dolnej części okna. Jeżeli użytkownik kliknie dwukrotnie któryś z plików, jego zawartość zostanie wyświetlona w obszarze tekstowym, tak jak jest to pokazane na rysunku 1.5.

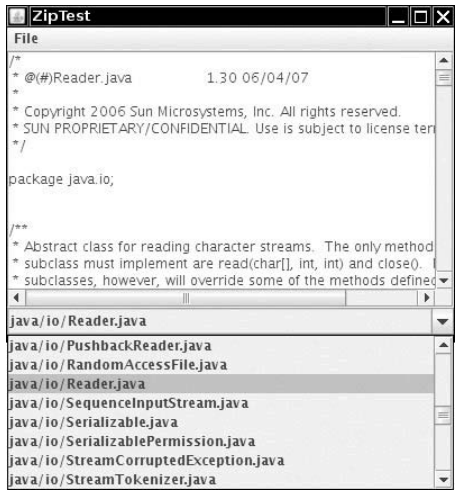
Listing 1.3. *ZipTest.java*

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import java.util.List;
import java.util.zip.*;
import javax.swing.*;

/**
 * @version 1.32 2007-06-22
 * @author Cay Horstmann
 */
public class ZipTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(new Runnable()
         {
            public void run()
            {
               ZipTestFrame frame = new ZipTestFrame();
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
               frame.setVisible(true);
            }
         });
   }
}

/**
  * Ramka zawierająca obszar tekstowy wyświetlający zawartość pliku,
  * listę pozwalającą na wybór plików w archiwum
  * oraz menu pozwalające wczytać nowe archiwum.
  */
class ZipTestFrame extends JFrame
{
   public ZipTestFrame()
   {
      setTitle("ZipTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // dodaje menu i opcje Open oraz Exit
```



Rysunek 1.5.
Program ZipTest w działaniu

```
JMenuBar menuBar = new JMenuBar();
JMenu menu = new JMenu("File");

JMenuItem openItem = new JMenuItem("Open");
menu.add(openItem);
openItem.addActionListener(new ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         JFileChooser chooser = new JFileChooser();
         chooser.setCurrentDirectory(new File("."));
         int r = chooser.showOpenDialog(ZipTestFrame.this);
         if (r == JFileChooser.APPROVE_OPTION)
         {
            zipname = chooser.getSelectedFile().getPath();
            fileCombo.removeAllItems();
            scanZipFile();
         }
      }
   });

JMenuItem exitItem = new JMenuItem("Exit");
menu.add(exitItem);
exitItem.addActionListener(new ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         System.exit(0);
      }
   });

menuBar.add(menu);
setJMenuBar(menuBar);

// dodaje obszar tekstowy i listę
fileText = new JTextArea();
fileCombo = new JComboBox();
fileCombo.addActionListener(new ActionListener()
```

```
      {
         public void actionPerformed(ActionEvent event)
         {
            loadZipFile((String) fileCombo.getSelectedItem());
         }
      });

   add(fileCombo, BorderLayout.SOUTH);
   add(new JScrollPane(fileText), BorderLayout.CENTER);
}

/**
 * Skanuje zawartość archiwum ZIP i wypełnia listę.
 */
public void scanZipFile()
{
   new SwingWorker<Void, String>()
      {
         protected Void doInBackground() throws Exception
         {
            ZipInputStream zin = new ZipInputStream(new
            ↪FileInputStream(zipname));
            ZipEntry entry;
            while ((entry = zin.getNextEntry()) != null)
            {
               publish(entry.getName());
               zin.closeEntry();
            }
            zin.close();
            return null;
         }

         protected void process(List<String> names)
         {
            for (String name : names)
               fileCombo.addItem(name);

         }
      }.execute();
}

/**
 * Ładuje zawartość pliku z archiwum ZIP
 * do obszaru tekstowego
 * @param name nazwa pliku w archiwum

 */
public void loadZipFile(final String name)
{
   fileCombo.setEnabled(false);
   fileText.setText("");
   new SwingWorker<Void, Void>()
      {
         protected Void doInBackground() throws Exception
         {
            try
            {
```

```
                ZipInputStream zin = new ZipInputStream(new
                ↪FileInputStream(zipname));
                ZipEntry entry;

                // znajduje element archiwum o odpowiedniej nazwie
                while ((entry = zin.getNextEntry()) != null)
                {
                   if (entry.getName().equals(name))
                   {
                      // wczytuje go do obszaru tekstowego
                      Scanner in = new Scanner(zin);
                      while (in.hasNextLine())
                      {
                         fileText.append(in.nextLine());
                         fileText.append("\n");
                      }
                   }
                   zin.closeEntry();
                }
                zin.close();
             }
             catch (IOException e)
             {
                e.printStackTrace();
             }
             return null;
          }

          protected void done()
          {
             fileCombo.setEnabled(true);
          }
       }.execute();
   }

   public static final int DEFAULT_WIDTH = 400;
   public static final int DEFAULT_HEIGHT = 300;

   private JComboBox fileCombo;
   private JTextArea fileText;
   private String zipname;
}
```

API `java.util.zip.ZipInputStream 1.1`

- `ZipInputStream(InputStream in)`

  tworzy obiekt typu `ZipInputStream` umożliwiający dekompresję danych z podanego strumienia `InputStream`.

- `ZipEntry getNextEntry()`

  zwraca obiekt typu `ZipEntry` opisujący następną pozycję archiwum lub `null`, jeżeli archiwum nie ma więcej pozycji.

- `void closeEntry()`

  zamyka aktualnie otwartą pozycję archiwum ZIP. Dzięki temu możemy odczytać następną pozycję, wywołując metodę `getNextEntry()`.

API java.util.zip.ZipOutputStream 1.1

- `ZipOutputStream(OutputStream out)`

  tworzy obiekt typu `ZipOutputStream`, który umożliwia kompresję i zapis danych w podanym strumieniu `OutputStream`.

- `void putNextEntry(ZipEntry ze)`

  zapisuje informacje podanej pozycji `ZipEntry` do strumienia i przygotowuje strumień do odbioru danych. Dane mogą zostać zapisane w strumieniu przy użyciu metody `write()`.

- `void closeEntry()`

  zamyka aktualnie otwartą pozycję archiwum ZIP. Aby otworzyć następną pozycję, wywołujemy metodę `putNextEntry`.

- `void setLevel(int level)`

  określa domyślny stopień kompresji następnych pozycji archiwum o trybie `DEFLATED`. Domyślną wartością jest `Deflater.DEFAULT_COMPRESSION`. Wyrzuca wyjątek `IllegalArgumentException`, jeżeli podany stopień jest nieprawidłowy.

  Parametry: `level` stopień kompresji, od 0 (`NO_COMPRESSION`) do 9 (`BEST_COMPRESSION`).

- `void setMethod(int method)`

  określa domyślną metodę kompresji dla danego `ZipOutputStream` dla wszystkich pozycji archiwum, dla których metoda kompresji nie została określona.

  Parametry: `method` metoda kompresji, `DEFLATED` lub `STORED`.

API java.util.zip.ZipEntry 1.1

- `ZipEntry(String name)`

  Parametry: `name` nazwa elementu.

- `long getCrc()`

  zwraca wartość sumy kontrolnej CRC32 danego elementu.

- `String getName()`

  zwraca nazwę elementu.

- `long getSize()`

  zwraca rozmiar danego elementu po dekompresji lub –1, jeżeli rozmiar nie jest znany.

- `boolean isDirectory()`

  zwraca wartość logiczną, która określa, czy dany element archiwum jest katalogiem.

- `void setMethod(int method)`

  Parametry: `method` metoda kompresji danego elementu, `DEFLATED` lub `STORED`.

- `void setSize(long rozmiar)`

  określa rozmiar elementu. Wymagana, jeżeli metodą kompresji jest `STORED`.

  Parametry: `rozmiar` rozmiar nieskompresowanego elementu.

- `void setCrc(long crc)`

  określa sumę kontrolną CRC32 dla danego elementu. Aby obliczyć tę sumę używamy klasy `CRC32`. Wymagana, jeżeli metodą kompresji jest `STORED`.

  Parametry: `crc` suma kontrolna elementu.

API `java.util.ZipFile` **1.1**

- `ZipFile(String name)`

  ten konstruktor tworzy obiekt typu `ZipFile`, otwarty do odczytu, na podstawie podanego łańcucha.

- `ZipFile(File file)`

  tworzy obiekt typu `ZipFile`, otwarty do odczytu, na podstawie podanego łańcucha lub obiektu typu `File`.

- `Enumeration entries()`

  zwraca obiekt typu `Enumeration`, wyliczający obiekty `ZipEntry` opisujące elementy archiwum `ZipFile`.

- `ZipEntry getEntry(String name)`

  zwraca element archiwum o podanej nazwie lub `null`, jeżeli taki element nie istnieje.

  Parametry: `name` nazwa elementu.

- `InputStream getInputStream(ZipEntry ze)`

  zwraca obiekt `InputStream` dla podanego elementu.

  Parametry: `ze` element `ZipEntry` w pliku ZIP.

- `String getName()` zwraca ścieżkę dostępu do pliku ZIP.

Strumienie obiektów i serializacja

Korzystanie z rekordów o stałej długości jest dobrym rozwiązaniem, pod warunkiem że zapisujemy dane tego samego typu. Jednak obiekty, które tworzymy w programie zorientowanym obiektowo, rzadko należą do tego samego typu. Dla przykładu: możemy używać tablicy o nazwie `staff`, której nominalnym typem jest `Employee`, ale która zawiera obiekty będące instancjami klas pochodnych, np. klasy `Manager`.

Z pewnością można zaprojektować format danych, który pozwoli przechowywać takie polimorficzne kolekcje, ale na szczęście ten dodatkowy wysiłek nie jest konieczny. Język Java obsługuje bowiem bardzo ogólny mechanizm zwany serializacją obiektów. Pozwala on na wysłanie do strumienia dowolnego obiektu i umożliwia jego późniejsze wczytanie (w dalszej części tego rozdziału wyjaśnimy, skąd wziął się termin „serializacja").

Aby zachować dane obiektu, musimy najpierw otworzyć strumień `ObjectOutputStream`:

```
ObjectOutputStream out = new ObjectOutputStream(new
↪FileOutputStream("employee.dat"));
```

Teraz, aby zapisać obiekt, wywołujemy metodę `writeObject klasy ObjectOutputStream`:

```
Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
out.writeObject(harry);
out.writeObject(boss);
```

Aby z powrotem załadować obiekty, używamy strumienia `ObjectInputStream`:

```
ObjectInputStream in = new ObjectInputStream(new FIleInputStream("employee.dat"));
```

Następnie pobieramy z niego obiekty w tym samym porządku, w jakim zostały zapisane, korzystając z metody `readObject`:

```
Employee p1 = (Employee)in.readObject();
Employee p2 = (Employee)in.readObject();
```

Jeśli chcemy zapisywać i odtwarzać obiekty za pomocą strumieni obiektów, to konieczne jest wprowadzenie jednej modyfikacji w klasie tych obiektów. Klasa ta musi implementować interfejs `Serializable`:

```
class Employee implements Serializable { . . . }
```

Interfejs Serializable nie posiada metod, nie musimy zatem wprowadzać żadnych innych modyfikacji naszych klas. Pod tym względem Serializable jest podobny do interfejsu Cloneable, który omówiliśmy w rozdziale 6. książki *Java 2. Podstawy.* Aby jednak móc klonować obiekty, musimy przesłonić metodę clone klasy Object. Aby móc serializować, nie należy robić nic poza dopisaniem powyższych słów.

Jednakże musimy rozważyć jeszcze jedną sytuację. Co się stanie, jeżeli dany obiekt jest współdzielony przez kilka innych obiektów jako element ich stanu?

Aby zilustrować ten problem, zmodyfikujemy trochę klasę `Manager`. Załóżmy, że każdy menedżer ma asystenta:

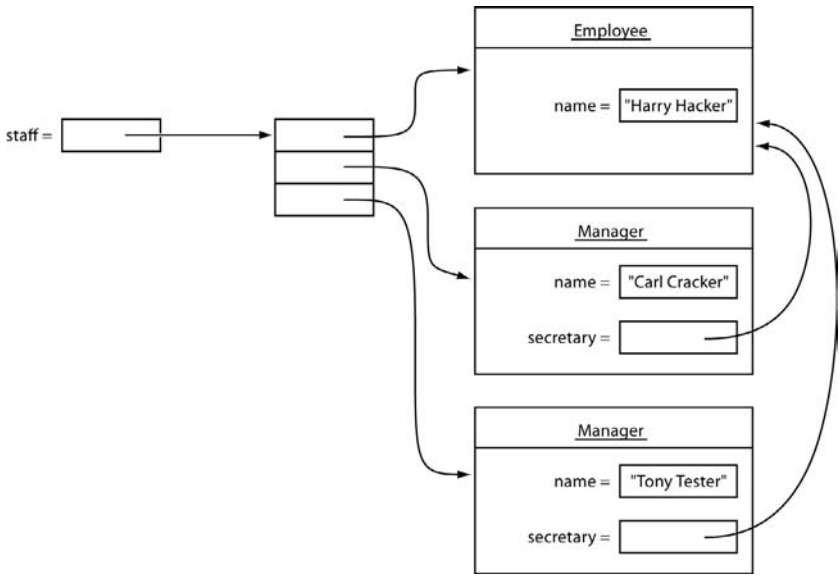
```
class Manager extends Employee
{

   . . .
   private Employee secretary;
}
```

Każdy obiekt typu `Manager` przechowuje teraz referencję do obiektu klasy `Employee` opisującego asystenta, a nie osobną kopię tego obiektu.

Oznacza to, że dwóch menedżerów może mieć tego samego asystenta, tak jak zostało to przedstawione na rysunku 1.6 i w poniższym kodzie:

```
harry = new Employee("Harry Hacker", . . .);
Manager carl = new Manager("Carl Cracker", . . .);
carl.setSecretary(harry);
Manager tony = new Manager("Tony Tester", . . .);
tony.setSecretary(harry);
```



Rysunek 1.6. *Dwóch menedżerów może mieć wspólnego asystenta*

Teraz załóżmy, że zapisujemy dane pracowników na dysk.

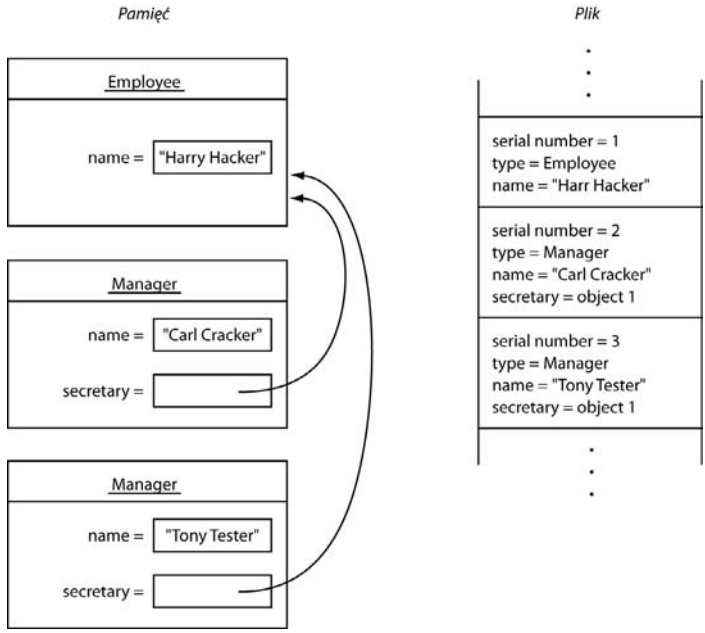
Oczywiście nie możemy zapisać i przywrócić adresów obiektów asystentów, ponieważ po ponownym załadowaniu obiekt asystenta najprawdopodobniej znajdzie się w zupełnie innym miejscu pamięci.

Zamiast tego każdy obiekt zostaje zapisany z numerem seryjnym i stąd właśnie pochodzi określenie serializacja. Oto jej algorytm:

- Wszystkim napotkanym referencjom do obiektów nadawane są numery seryjne (patrz rysunek 1.7).
- Jeśli referencja do obiektu została napotkana po raz pierwszy, obiekt zostaje zapisany w strumieniu.
- Jeżeli obiekt został już zapisany, Java zapisuje, że w danym miejscu znajduje się „ten sam obiekt, co pod numerem seryjnym x”.

Wczytując obiekty z powrotem, Java odwraca całą procedurę.

- Gdy obiekt pojawia się w strumieniu po raz pierwszy, Java tworzy go, inicjuje danymi ze strumienia i zapamiętuje związek pomiędzy numerem i referencją do obiektu.



Rysunek 1.7. *Przykład serializacji obiektów*

- Gdy natrafi na znacznik „ten sam obiekt, co pod numerem seryjnym x", sprawdza, gdzie znajduje się obiekt o danym numerze, i nadaje referencji do obiektu adres tego miejsca.

W tym rozdziale korzystamy z serializacji, aby zapisać zbiór obiektów na dysk, a później z powrotem je wczytać. Innym bardzo ważnym zastosowaniem serializacji jest przesyłanie obiektów przez sieć na inny komputer. Podobnie jak adresy pamięci są bezużyteczne dla pliku, tak samo są bezużyteczne dla innego rodzaju procesora. Ponieważ serializacja zastępuje adresy pamięci numerami seryjnymi, możemy transportować zbiory danych z jednej maszyny na drugą. Omówimy to zastosowanie przy okazji wywoływania zdalnych metod w rozdziale 5.

Listing 1.4 zawiera program zapisujący i wczytujący sieć powiązanych obiektów klas Employee i Manager (niektóre z nich mają referencję do tego samego asystenta). Zwróć uwagę, że po wczytaniu istnieje tylko jeden obiekt każdego asystenta — gdy pracownik newStaff[1] dostaje podwyżkę, znajduje to odzwierciedlenie za pomocą pól secretary obiektów klasy Manager.

Listing 1.4. *ObjectStreamTest.java*

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

/**
 * @version 1.10 17 Aug 1998
 * @author Cay Horstmann
 */
class ObjectStreamTest
{
   public static void main(String[] args)
```

```
   {
      Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      Manager carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
      carl.setSecretary(harry);
      Manager tony = new Manager("Tony Tester", 40000, 1990, 3, 15);
      tony.setSecretary(harry);

      Employee[] staff = new Employee[3];

      staff[0] = carl;
      staff[1] = harry;
      staff[2] = tony;

      try
      {
         // zapisuje rekordy wszystkich pracowników w pliku employee.dat
         ObjectOutputStream out = new ObjectOutputStream(new
         ↪FileOutputStream("employee.dat"));
         out.writeObject(staff);
         out.close();

         // wczytuje wszystkie rekordy do nowej tablicy
         ObjectInputStream in = new ObjectInputStream(new
         ↪FileInputStream("employee.dat"));
         Employee[] newStaff = (Employee[]) in.readObject();
         in.close();

         // podnosi wynagrodzenie asystenta
         newStaff[1].raiseSalary(10);

         // wyświetla wszystkie rekordy
         for (Employee e : newStaff)
            System.out.println(e);
      }
      catch (Exception e)
      {
         e.printStackTrace();
      }
   }
}

class Employee implements Serializable
{
   public Employee()
   {
   }

   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()
   {
```

```
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   public String toString()
   {
      return getClass().getName() + "[name=" + name + ",salary=" + salary +
      ↪",hireDay=" + hireDay
            + "]";
   }

   private String name;
   private double salary;
   private Date hireDay;
}

class Manager extends Employee
{
   /**
    * Tworzy obiekt klasy Manager nie inicjując pola secretary
    * @param n nazwisko pracownika
    * @param s wynagrodzenie
    * @param year rok zatrudnienia
    * @param month miesiąc zatrudnienia

    * @param day dzień zatrudnienia

    */
   public Manager(String n, double s, int year, int month, int day)
   {
      super(n, s, year, month, day);
      secretary = null;
   }

   /**
    * Przypisuje asystenta menedżerowi.
    * @param s asystent
    */
   public void setSecretary(Employee s)
   {
      secretary = s;
   }
```

```
    public String toString()
    {
       return super.toString() + "[secretary=" + secretary + "]";
    }

    private Employee secretary;
}
```

API java.io.ObjectOutputStream 1.1

- `ObjectOutputStream(OutputStream wy)`
  tworzy obiekt `ObjectOutputStream`, dzięki któremu możesz zapisywać obiekty do podanego strumienia wyjścia.
- `void writeObject(Object ob)`
  zapisuje podany obiekt do `ObjectOutputStream`. Metoda ta zachowuje klasę obiektu, sygnaturę klasy oraz wartości wszystkich niestatycznych, nieprzechodnich pól składowych tej klasy, a także jej nadklas.

API java.io.ObjectInputStream 1.1

- `ObjectInputStream(InputStream we)`
  tworzy obiekt `ObjectInputStream`, dzięki któremu możesz odczytywać informacje z podanego strumienia wejścia.
- `Object readObject()`
  wczytuje obiekt z `ObjectInputStream`. Pobiera klasę obiektu, sygnaturę klasy oraz wartości wszystkich niestatycznych, nieprzechodnich pól składowych tej klasy, a także jej nadklas. Przeprowadza deserializację, pozwalając na przyporządkowanie obiektów referencjom.

Format pliku serializacji obiektów

Serializacja obiektów powoduje zapisanie danych obiektu w określonym formacie. Oczywiście, możemy używać metod `writeObject/readObject`, nie wiedząc nawet, która sekwencja bajtów reprezentuje dany obiekt w pliku. Niemniej jednak doszliśmy do wniosku, że poznanie formatu danych będzie bardzo pomocne, ponieważ daje wgląd w proces obsługi obiektów przez strumienie. Ponieważ poniższy tekst jest pełen technicznych detali, to jeśli nie jesteś zainteresowany implementacją serializacji, możesz pominąć lekturę tego podrozdziału.

Każdy plik zaczyna się dwubajtową „magiczną liczbą”:

```
AC ED
```

po której następuje numer wersji formatu serializacji obiektów, którym aktualnie jest

```
00 05
```

(w tym podrozdziale do opisywania bajtów będziemy używać notacji szesnastkowej). Później następuje sekwencja obiektów, w takiej kolejności, w jakiej zostały one zapisane.

Łańcuchy zapisywane są jako

```
74      długość (2 bajty)      znaki
```

Dla przykładu, łańcuch `"Harry"` będzie wyglądał tak:

```
74 00 05 Harry
```

Znaki Unicode zapisywane są w zmodyfikowanym formacie UTF-8.

Wraz z obiektem musi zostać zapisana jego klasa. Opis klasy zawiera:

- nazwę klasy,
- *unikalny numer ID* stanowiący „odcisk" wszystkich danych składowych i sygnatur metod,
- zbiór flag opisujący metodę serializacji,
- opis pól składowych.

Java tworzy wspomniany „odcisk" klasy, pobierając opisy klasy, klasy bazowej, interfejsów, typów pól danych oraz sygnatury metod w postaci kanonicznej, a następnie stosuje do nich algorytm SHA (Secure Hash Algorithm).

SHA to szybki algorytm, tworzący „odciski palców" dla dużych bloków danych. Niezależnie od rozmiaru oryginalnych danych, „odciskiem" jest zawsze pakiet 20 bajtów. Jest on tworzony za pomocą sekwencji operacji binarnych, dzięki którym możemy mieć stuprocentową pewność, że jeżeli zachowana informacja zmieni się, zmianie ulegnie również jej „odcisk palca". SHA jest amerykańskim standardem, rekomendowanym przez Narodowy Instytut Nauki i Technologii (*National Institute of Science and Technology — NIST*; aby dowiedzieć się więcej o SHA, zajrzyj np. do *Cryptography and Network Security: Principle and Practice*, autorstwa Williama Stallingsa, wydanej przez Prentice Hall). Jednakże Java korzysta jedynie z pierwszych ośmiu bajtów kodu SHA. Mimo to nadal jest bardzo prawdopodobne, że „odcisk" zmieni się, jeżeli ulegną zmianie pola składowe lub metody.

W chwili odczytu danych Java sprawdza, używając „odcisku" klasy, czy definicja klasy nie uległa zmianie. Oczywiście, w praktyce klasy ulegają zmianie i może się okazać, że program będzie musiał wczytać starsze wersje obiektów. Zagadnienie to omówimy w punkcie „Wersje" na stronie 71.

Oto, w jaki sposób przechowywany jest identyfikator klasy:

```
72
```

długość nazwy klasy (2 bajty)

nazwa klasy

„odcisk" klasy (8 bajtów)

zbiór flag (1 bajt)

liczba deskryptorów pól składowych (2 bajty)

deskryptory pól składowych

78 (znacznik końca)

typ klasy bazowej (70, jeśli nie istnieje)

Bajt flag składa się z trzybitowych masek, zdefiniowanych w java.io.ObjectStreamCon
↪stants:

```
java.io.ObjectStreamConstants:
static final byte SC_WRITE_METHOD = 1;
 // klasa posiada metodę writeObject zapisującą dodatkowe dane
static final byte SC_SERIALIZABLE = 2;
 // klasa implementuje interfejs Serializable
static final byte SC_EXTERNALIZABLE = 4;
 // klasa implementuje interfejs Externalizable
```

Interfejs Externalizable omówimy w dalszej części rozdziału. Klasy implementujące Exter
↪nalizable udostępniają własne metody wczytujące i zapisujące, które przejmują obsługę nad swoimi polami składowymi. Klasy, które budujemy, implementują interfejs Serializable i będą mieć bajt flag o wartości 02. Jednak np. klasa java.util.Date implementuje Externa
↪lizable i jej bajt flag ma wartość 03.

Każdy deskryptor pola składowego składa się z następujących elementów:

- kod typu (1 bajt),
- długość nazwy pola (2 bajty),
- nazwa pola,
- nazwa klasy (jeżeli pole składowe jest obiektem).

Kod typu może mieć jedną z następujących wartości:

B	byte
C	char
D	double
F	float
I	int
J	long
L	obiekt
S	short
Z	boolean
[	tablica

Jeżeli kodem typu jest L, zaraz za nazwą pola składowego znajdzie się nazwa jego typu. Łańcuchy nazw klas i pól składowych nie zaczynają się od 74, w przeciwieństwie do typów pól składowych. Typy pól składowych używają trochę innego sposobu kodowania nazw, a dokładniej — formatu używanego przez metody macierzyste.

Dla przykładu, pole pensji klasy Employee zostanie zapisane jako:

```
D 00 06 salary
```

A oto kompletny opis klasy `Employee`:

`72 00 08 Employee`	
`E6 D2 86 7D AE AC 18 1B 02`	„Odcisk” oraz flagi
`00 03`	Liczba pól składowych
`D 00 06 salary`	Typ i nazwa pola składowego
`L 00 07 hireDay`	Typ i nazwa pola składowego
`74 00 10 Ljava/util/Date;`	Nazwa klasy pola składowego — `String`
`L 00 04 name`	Typ i nazwa pola składowego
`74 00 12 Ljava/lang/String;`	Nazwa klasy pola składowego — `String`
`78`	Znacznik końca
`70`	Brak nadklasy

Opisy te są dość długie. Jeżeli w pliku jeszcze raz musi się znaleźć opis tej samej klasy, zostanie użyta forma skrócona:

`71`	numer seryjny (4 bajty)

Numer seryjny wskazuje na poprzedni opis danej klasy. Schemat numerowania omówimy później.

Obiekt jest przechowywany w następującej postaci:

`73`	opis klasy	dane obiektu

Dla przykładu, oto zapis obiektu klasy `Employee`:

`40 E8 6A 00 00 00 00`	Wartość pola `salary` — `double`
`73`	Wartość pola `hireDay` — nowy obiekt
`71 00 7E 00 08`	Istniejąca klasa `java.util.Date`
`77 08 00 00 00 91 1B 4E B1 80 78`	Zawartość zewnętrzna — szczegóły poniżej
`74 00 0C Harry Hacker`	Wartość pola `name` — `String`

Jak widzimy, plik danych zawiera informacje wystarczające do odtworzenia obiektu klasy Employee.

Tablice są zapisywane w następujący sposób:

`75`	opis klasy	liczba elementów (4 bajty)	elementy

Nazwa klasy tablicy jest zachowywana w formacie używanym przez metody macierzyste (różni się on trochę od formatu nazw innych klas). W tym formacie nazwy klas zaczynają się od `L`, a kończą średnikiem.

Dla przykładu, tablica trzech obiektów typu `Employee` zaczyna się tak:

75	Tablica
72 00 0C [LEmployee;	Nowa klasa, długość łańcucha, nazwa klasy `Employee[]`
FC BF 36 11 C5 91 11 C7 02	„Odcisk” oraz flagi
00 00	Liczba pól składowych
78	Znacznik końca
70	Brak nadklasy
00 00 00 03	Liczba komórek tablicy

Zauważmy, że „odcisk” tablicy obiektów `Employee` różni się od „odcisku” samej klasy `Employee`.

Wszystkie obiekty (łącznie z tablicami i łańcuchami) oraz wszystkie opisy klas w chwili zapisywania do pliku otrzymują numery seryjne. Numery seryjne zaczynają się od wartości `00 7E 00 00`.

Przekonaliśmy się już, że pełny opis jest wykonywany tylko raz dla każdej klasy. Następne opisy po prostu wskazują na pierwszy. W poprzednim przykładzie kolejna referencja klasy `Date` została zakodowana w następujący sposób:

```
71 00 7E 00 08
```

Ten sam mechanizm jest stosowany dla obiektów. Jeżeli zapisywana jest referencja obiektu, który został już wcześniej zapisany, nowa referencja zostanie zachowana w dokładnie ten sam sposób, jako `71` plus odpowiedni numer seryjny. Z kontekstu zawsze jasno wynika, czy dany numer seryjny dotyczy opisu klasy, czy obiektu.

Referencja `null` jest zapisywana jako

```
70
```

Oto plik zapisany przez program `ObjectRefTest` z poprzedniego podrozdziału, wraz z komentarzami. Jeśli chcesz, uruchom program, spójrz na zapis pliku *employee.dat* w notacji szesnastkowej i porównaj z poniższymi komentarzami. Zwróć uwagę na wiersze zamieszczone pod koniec pliku, zawierające referencje zapisanego wcześniej obiektu.

AC ED 00 05	Nagłówek pliku
75	Tablica `staff` (nr #1)
72 00 0C [LEmployee;	Nowa klasa, długość łańcucha, nazwa klasy `Employee[]` (nr #0)
FC BF 36 11 C5 91 11 C7 02	„Odcisk” oraz flagi
00 00	Liczba pól składowych
78	Znacznik końca
70	Brak klasy bazowej
00 00 00 03	Liczba elementów tablicy
73	`staff[0]` — nowy obiekt (nr #7)

`72 00 08 Manager`	Nowa klasa, długość łańcucha, nazwa klasy (nr #2)
`36 06 AE 13 63 8F 59 B7 02`	„Odcisk” oraz flagi
`00 01`	Liczba pól składowych
`L 00 08 secretary`	Typ i nazwa pola składowego
`74 00 0B LEmployee;`	Nazwa klasy pola składowego — `String` (nr #3)
`78`	Znacznik końca
`72 00 09 Employee`	Nadklasa — nowa klasa, długość łańcucha, nazwa klasy (nr #4)
`E6 D2 86 7D AE AC 18 1B 02`	„Odcisk” oraz flagi
`00 03`	Liczba pól składowych
`D 00 06 salary`	Typ i nazwa pola składowego
`L 00 11 hireDay`	Typ i nazwa pola składowego
`74 00 10 Ljava/util/Date;`	Nazwa klasy pola składowego — `String` (nr #5)
`L 00 08 name`	Typ i nazwa pola składowego
`74 00 12 Ljava/lang/String;`	Nazwa klasy pola składowego — `String` (nr #6)
`78`	Znacznik końca
`70`	Brak klasy bazowej
`40 F3 88 00 00 00 00 00`	Wartość pola `salary` — `double`
`73`	Wartość pola `hireDay` — nowy obiekt (nr #9)
`72 00 0E java.util.Date`	Nowa klasa, długość łańcucha, nazwa klasy (nr #8)
`68 6A 81 01 4B 59 74 19 03`	„Odcisk” oraz flagi
`00 00`	Brak zmiennych składowych
`78`	Znacznik końca
`70`	Brak klasy bazowej
`77 08`	Zawartość zewnętrzna, liczba bajtów
`00 00 00 83 E9 39 E0 00`	Data
`78`	Znacznik końca
`74 00 0C Carl Cracker`	Wartość pola `name` — `String` (nr #10)
`73`	Wartość pola `secretary` — nowy obiekt (nr #11)
`71 00 7E 00 04`	Istniejąca klasa (użyj nr #4)
`40 E8 6A 00 00 00 00 00`	Wartość pola `pensja` — `double`

```
        73                                  Wartość pola dzienZatrudnienia
                                            — nowy obiekt (nr #12)
           71 00 7E 00 08                   Istniejąca klasa (użyj nr #8)
           77 08                            Zawartość zewnętrzna, liczba bajtów
           00 00 00 91 1B 4E B1 80          Data
           78                               Znacznik końca
        74 00 0C Harry Hacker               Wartość pola name — String (nr #13)
  71 00 7E 00 0B                            staff[1] — istniejący obiekt
                                            (użyj nr #11)
  73                                        obsluga[2] — nowy obiekt (nr #14)
     71 00 7E 00 08                         Istniejąca klasa (użyj nr #4)
     40 E3 88 00 00 00 00 00                Wartość pola salary — double
     73                                     Wartość pola hireDay
                                            — nowy obiekt (nr #15)
        71 00 7E 00 08                      Istniejąca klasa (użyj nr #8)
        77 08                               Zawartość zewnętrzna, liczba bajtów
        00 00 00 94 6D 3E EC 00 00          Data
        78                                  Znacznik końca
     74 00 0D Tony Tester                   Wartość pola name — String (nr #16)

71 00 7E 00 0B                              Wartość pola secretary — istniejący
                                            obiekt (użyj nr #11)
```

Zazwyczaj znajomość dokładnego format pliku nie jest ważna (o ile nie próbujesz dokonać zmian w samym pliku).

Powinniśmy jednak pamiętać, że:

- strumień obiektów zapisuje typy i pola składowe wszystkich obiektów,
- każdemu obiektowi zostaje przypisany numer seryjny,
- powtarzające się odwołania do tego samego obiektu są przechowywane jako referencje jego numeru seryjnego.

Modyfikowanie domyślnego mechanizmu serializacji

Niektóre dane nie powinny być serializowane — np. wartości typu `integer` reprezentujące uchwyty plików lub okien, czytelne wyłącznie dla metod rodzimych. Gdy wczytamy takie dane ponownie lub przeniesiemy je na inną maszynę, najczęściej okażą się bezużyteczne. Co gorsza, nieprawidłowe wartości tych zmiennych mogą spowodować błędy w działaniu metod rodzimych. Dlatego Java obsługuje prosty mechanizm zapobiegający serializacji takich danych. Wystarczy oznaczyć je słowem kluczowym `transient`. Słowem tym należy również oznaczyć pola, których klasy nie są serializowalne. Pola oznaczone jako `transient` są zawsze pomijane w procesie serializacji.

Mechanizm serializacji na platformie Java udostępnia sposób, dzięki któremu indywidualne klasy mogą sprawdzać prawidłowość danych lub w jakikolwiek inny sposób wpływać na zachowanie strumienia podczas domyślnych operacji odczytu i zapisu. Klasa implementująca interfejs `Serializable` może zdefiniować metody o sygnaturach

```
private void readObject(ObjectInputStream in)
 throws IOException, ClassNotFoundException;
private void writeObject(ObjectOutputStream out)
 throws IOException;
```

Dzięki temu obiekty nie będą automatycznie serializowane — Java wywoła dla nich powyższe metody.

A oto typowy przykład. Wielu klas należących do pakietu `java.awt.geom`, takich jak na przykład klasa `Point2D.Double`, nie da się serializować. Przypuśćmy zatem, że chcemy serializować klasę `LabeledPoint` zawierającą pola typu `String` i `Point2D.Double`. Najpierw musimy oznaczyć pole `Point2D.Double` słowem kluczowym `transient`, aby uniknąć wyjątku `NotSerializableEx`
`↪ception`.

```
public class LabeledPoint implements Serializable
{
   . . .
   private String label;
   private transient Point2D.Double point;
}
```

W metodzie `writeObject` najpierw zapiszemy opis obiektu i pole typu `String`, wywołując metodę `defaultWriteObject`. Jest to specjalna metoda klasy `ObjectOutputStream`, która może być wywoływana jedynie przez metodę `writeObject` klasy implementującej interfejs `Seria`
`↪lizable`. Następnie zapiszemy współrzędne punktu, używając standardowych wywołań klasy `DataOutput`.

```
private void writeObject(ObjectOutputStream out)
   throws IOException
{
   out.defaultWriteObject();
   out.writeDouble(point.getX());
   out.writeDouble(point.getY());
}
```

Implementując metodę `readObject`, odwrócimy cały proces:

```
private void readObject(ObjectInputStream in)
   throws IOException
{
   in.defaultReadObject();
   double x = in.readDouble();
   double y = in.readDouble();
   point = new Point2D.Double(x, y);
}
```

Innym przykładem jest klasa `java.util.Date`, która dostarcza własnych metod `readObject` i `writeObject`. Metody te zapisują datę jako liczbę milisekund, które upłynęły od północy 1 stycznia 1970 roku, czasu UTC. Klasa `Date` stosuje skomplikowaną reprezentację wewnętrzną,

która przechowuje zarówno obiekt klasy `Calendar`, jak i licznik milisekund, co pozwala zoptymalizować operacje wyszukiwania. Stan obiektu klasy `Calendar` jest wtórny i nie musi być zapisywany.

Metody `readObject` i `writeObject` odczytują i zapisują jedynie dane własnej klasy. Nie zajmują się przechowywaniem i odtwarzaniem danych klasy bazowej bądź jakichkolwiek innych informacji o klasie.

Klasa może również zdefiniować własny mechanizm zapisywania danych, nie oglądając się na serializację. Aby tego dokonać, klasa musi zaimplementować interfejs `Externalizable`. Oznacza to implementację dwóch metod:

```
public void readExternal(ObjectInputSream in)
 throws IOException, ClassNotFoundException;
public void writeExternal(ObjectOutputStream out)
 throws IOException;
```

W przeciwieństwie do omówionych wcześniej metod `readObject` i `writeObject`, te metody są całkowicie odpowiedzialne za zapisanie i odczytanie obiektu, *łącznie z danymi klasy bazowej*. Mechanizm serializacji zapisuje jedynie klasę obiektu w strumieniu. Odtwarzając obiekt implementujący interfejs `Externalizable`, strumień obiektów wywołuje domyślny konstruktor, a następnie metodę `readExternal`. Oto jak możemy zaimplementować te metody w klasie `Employee`:

```
public void readExternal(ObjectInput s)
   throws IOException
{
   name = s.readUTF();
   salary = s.readDouble();
   hireDay = new Date(s.readLong());
}

public void writeExternal(ObjectOutput s)
   throws IOException
{
  s.writeUTF(name);
  s.writeDouble(salary);
  s.writeLong(hireDay.getTime());
}
```

Serializacja jest dość powolnym mechanizmem, ponieważ maszyna wirtualna musi rozpoznać strukturę każdego obiektu. Jeżeli zależy nam na efektywności działania, a jednocześnie chcemy wczytywać i zapisywać dużą liczbę obiektów pewnej klasy, powinniśmy rozważyć możliwość zastosowania interfejsu `Externalizable`. Artykuł na stronie *http://java.sun.com/developer/TechTips/2000/tt0425.html* stwierdza, że w przypadku klasy reprezentującej pracowników skorzystanie z własnych metod zapisu i odczytu było o 35 – 40% szybsze niż domyślna serializacja.

W przeciwieństwie do metod `writeObject` i `readObject`, które są prywatne i mogą zostać wywołane wyłącznie przez mechanizm serializacji, metody `writeExternal` i `read`
`↪External` są publiczne. W szczególności metoda `readExternal` potencjalnie może być wykorzystana do modyfikacji stanu istniejącego obiektu.

Serializacja singletonów i wyliczeń

Szczególną uwagę należy zwrócić na serializację obiektów, które z założenia mają być unikalne. Ma to miejsce w przypadku implementacji singletonów i wyliczeń.

Jeśli używamy w programach konstrukcji `enum` wprowadzonej w Java SE 5.0, to nie musimy przejmować się serializacją — wszystko będzie działać poprawnie. Załóżmy jednak, że mamy starszy kod, który tworzy typy wyliczeniowe w następujący sposób:

```
public class Orientation
{
   public static final Orientation HORIZONTAL = new Orientation(1);
   public static final Orientation VERTICAL  = new Orientation(2);
   private Orientation(int v) { value = v; }
   private int value;
}
```

Powyższy sposób zapisu był powszechnie stosowany, zanim wprowadzono typ wyliczeniowy w języku Java. Zwróćmy uwagę, że konstruktor klasy `Orientation` jest prywatny. Dzięki temu powstaną jedynie obiekty `Orientation.HORIZONTAL` i `Orientation.VERTICAL`. Obiekty tej klasy możemy porównywać za pomocą operatora ==:

```
if (orientation == Orientation.HORIZONTAL) . . .
```

Jeśli taki typ wyliczeniowy implemenuje interfejs `Serializable`, to domyślny sposób serializacji okaże się w tym przypadku niewłaściwy. Przypuśćmy, że zapisaliśmy wartość typu `Orientation` i wczytujemy ją ponownie:

```
Orientation original = Orientation.HORIZONTAL;
ObjectOutputStream out = . . .;
out.write(value);
out.close();
ObjectInputStream in = . . .;
Orientation saved = (Orientation) in.read();
```

Okaże się, że porównanie

```
if (saved == Orientation.HORIZONTAL) . . .
```

da wynik negatywny. W rzeczywistości bowiem wartość `saved` jest zupełnie nowym obiektem typu `Orientation` i nie jest ona równa żadnej ze stałych wstępnie zdefiniowanych przez tę klasę. Mimo że konstruktor klasy jest prywatny, mechanizm serializacji może tworzyć zupełnie nowe obiekty tej klasy!

Aby rozwiązać ten problem, musimy zdefiniować specjalną metodę serializacji o nazwie `readResolve`. Jeśli metoda `readResolve` jest zdefiniowana, zostaje wywołana po deserializacji obiektu. Musi ona zwrócić obiekt, który następnie zwróci metoda `readObject`. W naszym przykładzie metoda `readResolve` sprawdzi pole `value` i zwróci odpowiednią stałą:

```
protected Object readResolve() throws ObjectStreamException
{
   if (value == 1) return Orientation.HORIZONTAL;
   if (value == 2) return Orientation.VERTICAL;
   return null; // to nie powinno się zdarzyć
}
```

Musimy zatem pamiętać o zdefiniowaniu metody `readResolve` dla wszystkich wyliczeń konstruowanych w tradycyjny sposób i wszystkich klas implementujących wzorzec singletonu.

Wersje

Jeśli używamy serializacji do przechowywania obiektów, musimy zastanowić się, co się z nimi stanie, gdy powstaną nowe wersje programu. Czy wersja 1.1 będzie potrafiła czytać starsze pliki? Czy użytkownicy wersji 1.0 będą mogli wczytywać pliki tworzone przez nową wersję?

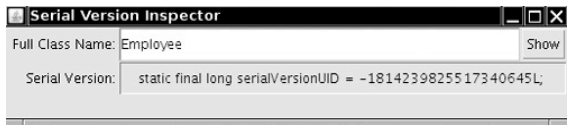
Na pierwszy rzut oka wydaje się to niemożliwe. Wraz ze zmianą definicji klasy zmienia się kod SHA, a strumień obiektów nie odczyta obiektu o innym „odcisku palca". Jednakże klasa może zaznaczyć, że jest *kompatybilna* ze swoją wcześniejszą wersją. Aby tego dokonać, musimy pobrać „odcisk palca" *wcześniejszej* wersji tej klasy. Do tego celu użyjemy `serialver`, programu będącego częścią JDK. Na przykład, uruchamiając

```
serialver Employee
```

otrzymujemy:

```
Employee:      static final long serialVersionUID =
-8768751229047793l1L
```

Jeżeli uruchomimy `serialver` z opcją `-show`, program wyświetli okno dialogowe (rysunek 1.8).



Rysunek 1.8. *Wersja graficzna programu serialver*

Wszystkie późniejsze wersje tej klasy muszą definiować stałą `serialVersionUID` o tym samym „odcisku palca", co wersja oryginalna.

```
class Employee implements Serializable // wersja 1.1
{
   . . .
   public static final long serialVersionUID = -1814239825517340645L;
}
```

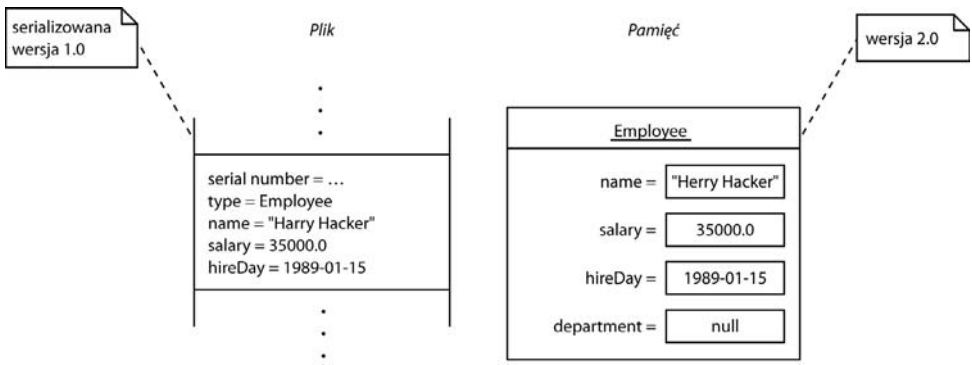
Klasa posiadająca statyczne pole składowe o nazwie `serialVersionUID` nie obliczy własnego „odcisku palca", ale skorzysta z już istniejącej wartości.

Od momentu, gdy w danej klasie umieścisz powyższą stałą, system serializacji będzie mógł odczytywać różne wersje obiektów tej samej klasy.

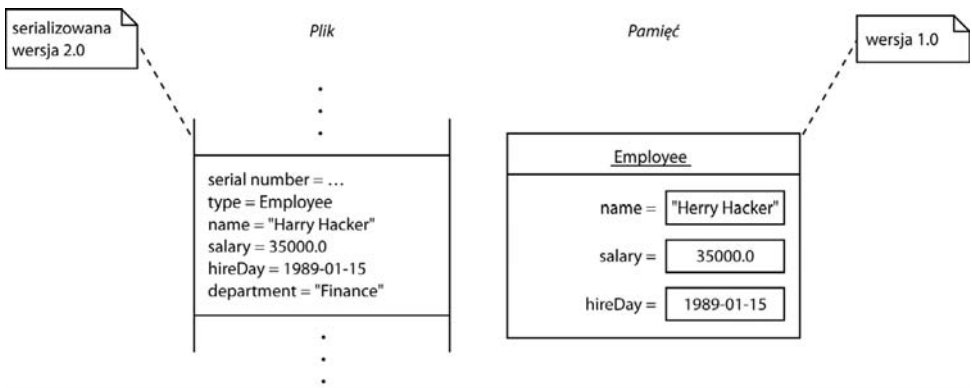
Jeśli zmienią się tylko metody danej klasy, sposób odczytu danych nie ulegnie zmianie. Jednakże jeżeli zmieni się pole składowe, możemy mieć pewne problemy. Dla przykładu, stary obiekt może posiadać więcej lub mniej pól składowych niż aktualny, albo też typy danych mogą się różnić. W takim wypadku strumień obiektów spróbuje skonwertować obiekt na aktualną wersję danej klasy.

Strumień obiektów porównuje pola składowe aktualnej wersji klasy z polami składowymi wersji znajdującej się w strumieniu. Oczywiście, strumień bierze pod uwagę wyłącznie niestatyczne, nieulotne pola składowe. Jeżeli dwa pola mają te same nazwy, lecz różne typy, strumień nawet nie próbuje konwersji — obiekty są niekompatybilne. Jeżeli obiekt w strumieniu posiada pola składowe nieobecne w aktualnej wersji, strumień ignoruje te dodatkowe dane. Jeżeli aktualna wersja posiada pola składowe nieobecne w zapisanym obiekcie, dodatkowe zmienne otrzymują swoje domyślne wartości (`null` dla obiektów, `0` dla liczb i `false` dla wartości logicznych).

Oto przykład. Załóżmy, że zapisaliśmy na dysku pewną liczbę obiektów klasy `Employee`, używając przy tym oryginalnej (1.0) wersji klasy. Teraz wprowadzamy nową wersję 2.0 klasy `Employee`, dodając do niej pole składowe `department`. Rysunek 1.9 przedstawia, co się dzieje, gdy obiekt wersji 1.0 jest wczytywany przez program korzystający z obiektów 2.0. Pole department otrzymuje wartość `null`. Rysunek 1.10 ilustruje odwrotną sytuację — program korzystający z obiektów 1.0 wczytuje obiekt 2.0. Dodatkowe pole `department` jest ignorowane.



Rysunek 1.9. *Odczytywanie obiektu o mniejszej liczbie pól*



Rysunek 1.10. *Odczytywanie obiektu o większej liczbie pól*

Czy ten proces jest bezpieczny? To zależy. Opuszczanie pól składowych wydaje się być bezbolesne — odbiorca wciąż posiada dane, którymi potrafi manipulować. Nadawanie wartości `null` nie jest już tak bezpieczne. Wiele klas inicjalizuje wszystkie pola składowe, nadając im w konstruktorach niezerowe wartości, tak więc metody mogą być nieprzygotowane do obsłu-

giwania wartości `null`. Od projektanta klasy zależy, czy zaimplementuje w metodzie `readObject` dodatkowy kod poprawiający wyniki wczytywania różnych wersji danych, czy też dołączy do metod obsługę wartości `null`.

Serializacja w roli klonowania

Istnieje jeszcze jedno, ciekawe zastosowanie mechanizmu serializacji — umożliwia on łatwe klonowanie obiektów klas implementujących interfejs `Serializable`. Aby sklonować obiekt, po prostu zapisujemy go w strumieniu, a następnie odczytujemy z powrotem. W efekcie otrzymujemy nowy obiekt, będący dokładną kopią istniejącego obiektu. Nie musisz zapisywać tego obiektu do pliku — możesz skorzystać z `ByteArrayOutputStream` i zapisać dane do tablicy bajtów.

Kod z listingu 1.5 udowadnia, że aby otrzymać metodę `clone` „za darmo”, wystarczy rozszerzyć klasę `SerialCloneable`.

Listing 1.5. *SerialCloneTest.java*

```
/**
   @version 1.20 17 Aug 1998
   @author Cay Horstmann
*/

import java.io.*;
import java.util.*;

public class SerialCloneTest
{
   public static void main(String[] args)
   {
      Employee harry = new Employee("Harry Hacker", 35000, 1989, 10, 1);
      // klonuje obiekt harry
      Employee harry2 = (Employee) harry.clone();

      // modyfikuje obiekt harry
      harry.raiseSalary(10);

      // teraz obiekt harry i jego klon są różne
      System.out.println(harry);
      System.out.println(harry2);
   }
}

/**
   Klasa, której metoda clone wykorzystuje serializację.
*/
class SerialCloneable implements Cloneable, Serializable
{
   public Object clone()
   {
      try
      {
         // zapisuje obiekt w tablicy bajtów
         ByteArrayOutputStream bout = new ByteArrayOutputStream();
```

```
            ObjectOutputStream out = new ObjectOutputStream(bout);
            out.writeObject(this);
            out.close();

            // wczytuje klon obiektu z tablicy bajtów
            ByteArrayInputStream bin = new ByteArrayInputStream(bout.toByteArray());
            ObjectInputStream in = new ObjectInputStream(bin);
            Object ret = in.readObject();
            in.close();

            return ret;
         }
         catch (Exception e)
         {
            return null;
         }
      }
}

/**
   Znana już klasa Employee,
   tym razem jako pochodna klasy SerialCloneable.
*/
class Employee extends SerialCloneable
{
   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   public String toString()
   {
      return getClass().getName()
```

```
            + "[name=" + name
            + ",salary=" + salary
            + ",hireDay=" + hireDay
            + "]";
      }

      private String name;
      private double salary;
      private Date hireDay;
   }
```

Należy jednak być świadomym, że opisany sposób klonowania, jakkolwiek sprytny, zwykle okaże się znacznie wolniejszy niż metoda clone jawnie tworząca nowy obiekt i kopiująca lub klonująca pola danych.

Zarządzanie plikami

Potrafimy już zapisywać i wczytywać dane z pliku. Jednakże obsługa plików to coś więcej niż tylko operacje zapisu i odczytu. Klasa `File` zawiera metody potrzebne do obsługi systemu plików na komputerze użytkownika. Na przykład, możemy wykorzystać klasę `File`, aby sprawdzić, kiedy nastąpiła ostatnia modyfikacja danego pliku, oraz usunąć lub zmienić nazwę tego pliku. Innymi słowy, klasy strumieni zajmują się zawartością plików, natomiast klasa `File` interesuje się sposobem przechowywania pliku na dysku.

Jak to ma często miejsce w języku Java, klasa `File` przyjmuje metodę najmniejszego wspólnego mianownika. Na przykład, w systemie Windows możemy sprawdzić (lub zmienić) flagę „tylko do odczytu" dla danego pliku, ale mimo iż możemy również sprawdzić, czy plik jest ukryty, nie możemy ukryć go samemu, jeśli nie zastosujemy w tym celu odpowiedniej metody macierzystej.

Najprostszy konstruktor obiektu `File` pobiera pełną nazwę pliku. Jeżeli nie podamy ścieżki dostępu, Java używa bieżącego katalogu. Na przykład:

```
File p = new File("test.txt");
```

tworzy obiekt pliku o podanej nazwie, znajdującego się w bieżącym katalogu (bieżącym katalogiem jest katalog, w którym program jest uruchamiany).

Ponieważ znak \ w łańcuchach na platformie Java jest traktowany jako początek sekwencji specjalnej, musimy pamiętać, aby w ścieżkach dostępu do plików systemu Windows używać sekwencji \\ (np. *C:\\Windows\\win.ini*). W systemie Windows możemy również korzystać ze znaku / (np. *C:/Windows/win.ini*), ponieważ większość systemów obsługi plików Windows interpretuje znaki / jako separatory ścieżki dostępu. Jednakże nie zalecamy tego rozwiązania — zachowanie funkcji systemu Windows może się zmieniać, a inne systemy operacyjne mogą używać jeszcze innych separatorów. Jeżeli piszemy aplikację przenośną, powinniśmy używać separatora odpowiedniego dla danego systemu operacyjnego. Jego znak jest przechowywany jako stała `File.separator`.

Wywołanie tego konstruktora *nie tworzy nowego pliku, jeżeli plik o danej nazwie nie istnieje*. Tworzenie nowego pliku na podstawie obiektu File odbywa się przy użyciu jednego z konstruktorów klas strumieni lub metody `createNewFile` klasy `File`. Metoda `createNewFile` tworzy nowy plik tylko pod warunkiem, że plik o podanej nazwie nie istnieje i zwraca wartość logiczną, by poinformować, czy operacja się udała.

Z drugiej strony, gdy utworzymy obiekt typu `File`, dzięki metodzie `exists` możemy sprawdzić, czy plik o danej nazwie istnieje. Dla przykładu, poniższy program prawie na pewno wydrukuje „false", równocześnie podając ścieżkę dostępu do nieistniejącego pliku.

```
import java.io.*;

public class Test
{
 public static void main(String args[])
 {
  File f = new File("plikktóryprawdopodobnieistnieje");
  System.out.println(f.getAbsolutePath());
  System.our.println(f.exists());
 }
}
```

Klasa `File` posiada jeszcze dwa inne konstruktory:

```
File(String path, String name)
```

tworzące obiekt typu `File` o podanej nazwie i katalogu określonym przez parametr path (jeżeli `path` ma wartość `null`, konstruktor tworzy obiekt `File`, korzystając z katalogu bieżącego).

Oprócz tego możemy również posłużyć się istniejącym już obiektem `File`, wywołując konstruktor:

```
File(File dir, String name)
```

gdzie obiekt `File` reprezentuje katalog i, tak jak poprzednio, jeżeli `dir` ma wartość `null`, konstruktor tworzy nowy obiekt w katalogu bieżącym.

Co ciekawe, obiekt `File` może reprezentować zarówno plik, jak i katalog (być może dlatego, że system operacyjny, na którym pracowali projektanci języka Java, pozwalał traktować katalogi tak, jakby były plikami). Aby sprawdzić, czy obiekt reprezentuje katalog, czy plik, korzystamy z metod `isDirectory` i `isFile`. Dziwne — w zorientowanym obiektowo systemie mogliśmy się spodziewać osobnej klasy `Directory`, być może rozszerzającej klasę `File`.

Aby obiekt reprezentował katalog, musimy podać konstruktorowi nazwę tego katalogu:

```
File tempDir = new File(File.separator + "temp");
```

Jeżeli katalog nie istnieje, możemy utworzyć go przy pomocy metody `mkdir`:

```
tempDir.mkdir();
```

Jeżeli dany obiekt reprezentuje katalog, metoda `list()` zwróci nam tablicę nazw plików znajdujących się w tym katalogu. Program zamieszczony na listingu 1.6 korzysta ze wszystkich omówionych metod, aby wydrukować strukturę dowolnego katalogu przekazanego jako argument wywołania w wierszu poleceń (łatwo można zmienić ten program w klasę zwracającą listę podkatalogów do dalszego przetwarzania).

Listing 1.6. *FindDirectories.java*

```
import java.io.*;

/**
 * @version 1.00 05 Sep 1997
 * @author Gary Cornell
 */
public class FindDirectories
{
   public static void main(String[] args)
   {
      // jeśli brak parametru wywołania,
      // rozpoczyna od katalogu nadrzędnego
      if (args.length == 0) args = new String[] { ".." };

      try
      {
         File pathName = new File(args[0]);
         String[] fileNames = pathName.list();

         // wyświetla wszystkie pliki w katalogu
         for (int i = 0; i < fileNames.length; i++)
         {
            File f = new File(pathName.getPath(), fileNames[i]);

            // jeśli kolejny katalog,
            // wywołuje rekurencyjnie metodę main
            if (f.isDirectory())
            {
               System.out.println(f.getCanonicalPath());
               main(new String[] { f.getPath() });
            }
         }
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
   }
}
```

Zamiast otrzymywać listę wszystkich plików w danym katalogu, możemy użyć obiektu klasy `FileNameFilter` jako parametru metody `list`, aby zmniejszyć rozmiar wynikowej listy plików. Na listę będą wtedy wpisywane tylko te obiekty, które spełniają warunki postawione przez interfejs `FilenameFilter`.

Aby zaimplementować interfejs `FilenameFilter`, klasa musi zdefiniować metodę o nazwie `accept`. Oto przykład prostej klasy implementującej `FilenameFilter`, która akceptuje wyłącznie pliki o określonym rozszerzeniu:

```
public class ExtensionFilter implements FilenameFilter

{

   public ExtensionFilter(String ext)
```

```
    {
       extension = "." + ext;
    }

    public boolean accept(File dir, String name)
    {
       return name.endsWith(extension);
    }

    private String extension;
}
```

W przypadku programów przenośnych wyspecyfikowanie nazwy pliku znajdującego się w określonym katalogu jest znacznie trudniejsze. Jak już o tym wspominaliśmy, okazuje się, że nawet w systemie Windows możesz używać znaku / jako separatora katalogów (mimo iż jest to separator systemu Unix), ale inne systemy mogą na to nie pozwolić, dlatego nie polecamy tego sposobu.

Jeżeli tworząc obiekt klasy `File`, używasmy znaków / jako separatorów katalogów, metoda `getAbsolutePath` zwróci ścieżkę dostępu również zawierającą znaki /, co w systemie Windows będzie wyglądać dość dziwnie. W takim przypadku warto skorzystać z metody `getCanonicalPath` — zamienia ona znaki / na znaki \.

Lepszym pomysłem jest wykorzystanie informacji o używanym separatorze, przechowywanej przez statyczne pole składowe `separator` klasy `File` (w środowisku Windows będzie to znak \, w środowisku Unix znak /). Na przykład:

```
File foo = new File("Documents" + File.separator + "data.txt")
```

Oczywiście, jeżeli skorzystamy z drugiej wersji konstruktora `File`:

```
File foo = new File("Documents", "data.txt");
```

Java automatycznie wstawi odpowiednie separatory.

Poniższe notki API opisują (naszym zdaniem) najważniejsze spośród pozostałych metod klasy `File`; sposób ich użycia powinien być oczywisty.

`java.io.File` **1.0**

- `boolean canRead()`
- `boolean canWrite()`
- `boolean canExecute()` **6**

  sprawdza, czy na pliku mogą być wykonywane operacje odczytu i zapisu oraz czy plik może być wykonywany.

- `boolean setReadable(boolean state, boolean ownerOnly)` **6**
- `boolean setWritable(boolean state, boolean ownerOnly)` **6**
- `boolean setExecutable(boolean state, boolean ownerOnly)` **6**

  nadaje plikowi odpowiednią właściwość. Jeśli parametr `ownerOnly` ma wartość `true`, to właściwość ta jest dostępna jedynie dla właściciela pliku. W przeciwnym razie dotyczy wszystkich użytkowników. Metoda ta zwraca wartość `true`, jeśli operacja nadania właściwości powiodła się.

- `static boolean createTempFile(String prefi, String sufix)` **1.2**
- `static boolean createTempFile(String prefix, String sufix, File directory)` **1.2**

  tworzy tymczasowy plik w domyślnym katalogu `tymczasowym` systemu operacyjnego lub w katalogu podanym przez użytkownika, używając przedrostka i przyrostka do utworzenia tymczasowej nazwy.

  Parametry:

`prefix`	łańcuch przedrostka o długości co najmniej trzech znaków.
`sufix`	opcjonalny przyrostek. Jeżeli ma wartość `null`, używane jest rozszerzenie `.tmp`.
`directory`	katalog, w którym plik ma się znajdować. Jeżeli ma wartość `null`, plik jest tworzony w bieżącym katalogu roboczym.

- `boolean delete()`

  próbuje skasować plik; zwraca `true`, jeżeli plik został skasowany; w przeciwnym wypadku zwraca `false`.

- `void deleteOnExit()`

  żąda, aby plik został skasowany, gdy zostanie wyłączona maszyna wirtualna.

- `boolean exists()`

  zwraca `true`, jeżeli dany plik lub katalog istnieje; w przeciwnym wypadku zwraca `false`.

- `String getAbsolutePath()`

  zwraca łańcuch zawierający absolutną ścieżkę dostępu. Wskazówka: zamiast tej funkcji lepiej korzystać z `getCanonicalPath`.

- `File getCanonicalFile()` **1.2**

  zwraca obiekt klasy `File` zawierający kanoniczną ścieżkę dostępu do danego pliku. Oznacza to, że usuwane są zbędne katalogi `"."`, używany jest odpowiedni separator katalogów oraz zależna od systemu operacyjnego obsługa wielkich i małych liter.

- `String getCanonicalPath()`

  zwraca łańcuch zawierający kanoniczną formę ścieżki dostępu. Oznacza to, że usuwane są zbędne katalogi ".", używany jest odpowiedni separator katalogów oraz zależna od systemu operacyjnego obsługa wielkich i małych liter.

- `String getName()`

  zwraca łańcuch zawierający nazwę pliku danego obiektu `File` (łańcuch ten nie zawiera ścieżki dostępu).

- `String getParent()`

  zwraca łańcuch zawierający nazwę katalogu, w którym znajduje się „rodzic" danego obiektu `File`. Jeżeli obiekt jest plikiem, „rodzicem" jest po prostu katalog, w którym dany plik się znajduje. Jeżeli obiekt jest katalogiem, jego „rodzicem" jest jego katalog bazowy lub `null`, jeżeli katalog bazowy nie istnieje.

- `File getParentFile()` **1.2**

  zwraca obiekt klasy `File` „rodzica" danego pliku. W notce o `getParent` znajdziesz definicję „rodzica".

- `String getPath()`

  zwraca łańcuch zawierający ścieżkę dostępu do pliku.

- `boolean isDirectory()`

  zwraca `true`, jeżeli obiekt reprezentuje katalog; w przeciwnym wypadku zwraca `false`.

- `boolean isFile()`

  zwraca `true`, jeżeli obiekt reprezentuje plik — pozostałe opcje to katalog lub urządzenie.

- `boolean isHidden()` **1.2**

  zwraca `true`, jeżeli obiekt reprezentuje plik lub katalog ukryty.

- `long lastModified()`

  zwraca datę ostatniej modyfikacji pliku (liczba milisekund od północy 1 stycznia 1970 GMT) lub 0, jeżeli plik nie istnieje. Aby zmienić tę wartość w obiekt `Date`, używamy konstruktora `Date(long)`.

- `long length()`

  zwraca długość pliku w bajtach lub 0, jeżeli plik nie istnieje.

- `String[] list()`

  zwraca tablicę łańcuchów, zawierających nazwy plików i katalogów znajdujących się w danym obiekcie `File`, lub `null`, jeżeli dany obiekt nie reprezentuje katalogu.

- `String[] list(FilenameFilter filter)`

  zwraca tablicę nazw plików i katalogów, znajdujących się w danym obiekcie i spełniających warunki filtra, lub `null`, jeżeli nie ma takich elementów.

  Parametry: `filter` używany obiekt typu `FilenameFilter`.

- `File[] listFiles()` **1.2**

  zwraca tablicę obiektów klasy `File`, odpowiadających plikom i katalogom znajdującym się w danym obiekcie klasy `File`, lub `null`, jeżeli dany obiekt nie reprezentuje katalogu.

- `File[] listFiles(FilenameFilter filter)` **1.2**

  zwraca tablicę obiektów klasy `File`, odpowiadających plikom i katalogom znajdującym się w danym obiekcie klasy `File` i spełniającym warunki filtra, lub `null`, jeżeli nie ma takich elementów.

  Parametry: `filter` używany obiekt typu `FilenameFilter`.

- `static File[] listRoots()` **1.2**

  zwraca tablicę obiektów klasy `File` odpowiadającą dostępnym katalogom najwyższego poziomu (np. w systemie Windows otrzymasz obiekty klasy `File` reprezentujące zainstalowane dyski — zarówno dyski lokalne, jak i mapowane dyski sieciowe; w systemie Unix otrzymasz po prostu `"/"`).

- `boolean createNewFile()` **1.2**

  jeżeli plik o nazwie podanej przez obiekt pliku nie istnieje, automatycznie tworzy taki plik. Oznacza to, że sprawdzanie nazwy oraz tworzenie nowego pliku nie zostanie zakłócone przez inną działalność systemu. Jeżeli udało się utworzyć nowy plik, metoda zwraca `true`.

- `boolean mkdir()`

  tworzy podkatalog, którego nazwa została podana przez obiekt pliku. Zwraca `true`, jeżeli udało się utworzyć katalog; w przeciwnym wypadku zwraca `false`.

- `boolean mkdirs()`

  w przeciwieństwie do `mkdir`, ta metoda tworzy również wymagane katalogi pośrednie. Zwraca `false`, jeżeli którykolwiek z wymaganych katalogów nie mógł zostać utworzony.

- `boolean renameTo(File newName)`

  zwraca `true`, jeżeli nazwa została zmieniona; w przeciwnym wypadku zwraca `false`.

  Parametry: `newName` obiekt klasy `File` określający nową nazwę pliku.

- `boolean setLastModified(long time)` **1.2**

  określa datę ostatniej modyfikacji pliku. Zwraca `true`, jeżeli zmiana się powiodła, w przeciwnym wypadku zwraca `false`.

  Parametry: `time` liczba typu `long` reprezentująca ilość milisekund, jakie upłynęły od północy 1 stycznia 1970 GMT. Metoda `getTime` klasy `Date` pozwala obliczyć tę wartość.

- `boolean setReadOnly()` **1.2**

  zmienia tryb pliku na „tylko do odczytu”. Zwraca `true`, jeżeli operacja się powiodła, w przeciwnym wypadku zwraca `false`.

- `URL toURL()` **1.2**

  konwertuje obiekt klasy `File` na plik `URL`.

- `long getTotalSpace()` **6**

- `long getFreeSpace()` **6**

- `long getUsableSpace()` **6**

  zwraca całkowity rozmiar, liczbę nieprzydzielonych bajtów i liczbę dostępnych bajtów partycji reprezentowanej przez obiekt klasy `File`. Jeśli obiekt klasy `File` nie reprezentuje partycji, metoda ta zwraca wartość 0.

API `java.io.FilenameFilter` **1.0**

- `boolean accept(File dir, String name)`

  powinna zostać tak zdefiniowana, aby zwracała `true`, jeżeli dany plik spełnia warunki filtra.

  Parametry: `dir` obiekt typu `File` reprezentujący katalog, w którym znajduje się dany plik.

  `name` nazwa danego pliku.

Ulepszona obsługa wejścia i wyjścia

Java SE 1.4 udostępnia w pakiecie `java.nio` szereg nowych rozwiązań poprawiających obsługę wejścia i wyjścia w programach.

Wspomniany pakiet obsługuje następujące rozwiązania:

- kodery i dekodery zbiorów znaków,
- nieblokujące operacje wejścia i wyjścia,
- pliki mapowane w pamięci,
- blokowanie dostępu do plików.

Kodowanie i dekodowanie znaków omówiliśmy już w punkcie „Zbiory znaków" na stronie 35. Nieblokujące operacje wejścia i wyjścia zostaną omówione w rozdziale 3., ponieważ mają one szczególne znaczenie w przypadku komunikacji w sieci. W następnych podrozdziałach zajmiemy się zatem omówieniem mapowania plików w pamięci oraz blokowaniem plików.

Mapowanie plików w pamięci

Większość systemów operacyjnych oferuje możliwość wykorzystania pamięci wirtualnej do stworzenia „mapy" pliku lub jego fragmentu w pamięci. Dostęp do pliku odbywa się wtedy znacznie szybciej niż w tradycyjny sposób.

Na końcu tego podrozdziału zamieściliśmy program, który oblicza sumę kontrolną CRC32 dla pliku, używając standardowych operacji wejścia i wyjścia, a także pliku mapowanego w pamięci. Na jednej i tej samej maszynie otrzymaliśmy wyniki jego działania przedstawione w tabeli 1.6 dla pliku *rt.jar* (37 MB) znajdującego się w katalogu *jre/lib* pakietu JDK.

Tabela 1.6. *Czasy wykonywania operacji na pliku*

Metoda	Czas
Zwykły strumień wejściowy	110 sekund
Buforowany strumień wejściowy	9,9 sekundy
Plik o swobodnym dostępie	162 sekundy
Mapa pliku w pamięci	7,2 sekundy

Jak łatwo zauważyć, na naszym komputerze mapowanie pliku dało nieco lepszy wynik niż zastosowanie buforowanego wejścia i znacznie lepszy niż użycie klasy `RandomAccessFile`.

Oczywiście dokładne wartości pomiarów będą się znacznie różnić dla innych komputerów, ale łatwo domyślić się, że w przypadku swobodnego dostępu do pliku zastosowanie mapowania da zawsze poprawę efektywności działania programu. Natomiast w przypadku sekwencyjnego odczytu plików o umiarkowanej wielkości zastosowanie mapowania nie ma sensu.

Pakiet `java.nio` znakomicie upraszcza stosowanie mapowania plików. Poniżej podajemy przepis na jego zastosowanie.

Najpierw musimy uzyskać *kanał* dostępu do pliku. Kanał jest abstrakcją stworzoną dla plików dyskowych, pozwalającą na korzystanie z takich możliwości systemów operacyjnych jak mapowanie plików w pamięci, blokowanie plików czy szybki transfer danych pomiędzy plikami. Kanał uzyskujemy, wywołując metodę `getChannel` dodaną do klas `FileInputStream`, `FileOutputStream` i `RandomAccessFile`.

```
FileInputStream in = new FileInputStream(. . .);
FileChannel channel = in.getChannel();
```

Następnie uzyskujemy z kanału obiekt klasy `MappedByteBuffer`, wywołując metodę `map` klasy `FileChannel`. Określamy przy tym interesujący nas obszar pliku oraz *tryb mapowania*. Dostępne są trzy tryby mapowania:

- `FileChannel.MapMode.READ_ONLY`: otrzymany bufor umożliwia wyłącznie odczyt danych. Jakakolwiek próba zapisu do bufora spowoduje wyrzucenie wyjątku `ReadOnlyBufferException`.
- `FileChannel.MapMode.READ_WRITE`: otrzymany bufor umożliwia zapis danych, które w pewnym momencie zostaną również zaktualizowane w pliku dyskowym. Należy pamiętać, że modyfikacje mogą nie być od razu widoczne dla innych programów, które mapują ten sam plik. Dokładny sposób działania równoległego mapowania tego samego pliku przez wiele programów zależy od systemu operacyjnego.
- `FileChannel.MapMode.PRIVATE`: otrzymany bufor umożliwia zapis danych, ale wprowadzone w ten sposób modyfikacje pozostają lokalne i nie są propagowane do pliku dyskowego.

Gdy mamy już bufor, możemy czytać i zapisywać dane, stosując w tym celu metody klasy `ByteBuffer` i jej klasy bazowej `Buffer`.

Bufory obsługują zarówno dostęp sekwencyjny, jak i swobodny. *Pozycja* w buforze zmienia się na skutek wykonywania operacji `get` i `put`. Wszystkie bajty bufora możemy przejrzeć sekwencyjnie na przykład w poniższy sposób:

```
while (buffer.hasRemaining())
{
   byte b = buffer.get();
   . . .
}
```

Alternatywnie możemy również wykorzystać dostęp swobodny:

```
for (int i = 0; i < buffer.limit(); i++)
{
   byte b = buffer.get(i);
   . . .
}
```

Możemy także czytać tablice bajtów, stosując metody:

```
get(byte[] bytes)
get(byte[], int offset, int length)
```

Dostępne są również poniższe metody:

```
getInt
getLong
getShort
getChar
getFloat
getDouble
```

umożliwiające odczyt wartości typów podstawowych zapisanych w pliku w postaci binarnej. Jak już wyjaśniliśmy wcześniej, Java zapisuje dane w postaci binarnej, począwszy od najbardziej znaczącego bajta. Jeśli musimy przetworzyć plik, który zawiera dane zapisane od najmniej znaczącego bajta, to wystarczy zastosować poniższe wywołanie:

```
buffer.order(ByteOrder.LITTLE_ENDIAN);
```

Aby poznać bieżący sposób uporządkowania bajtów w buforze, wywołujemy:

```
ByteOrder b = buffer.order()
```

Ta para metod nie stosuje konwencji nazw set/get.

Aby zapisać wartości typów podstawowych w buforze, używamy poniższych metod:

```
putInt
putLong
putShort
putChar
putFloat
putDouble
```

Program przedstawiony na listingu 1.7 oblicza sumę kontrolną CRC32 pliku. Suma taka jest często używana do kontroli naruszenia zawartości pliku. Uszkodzenie zawartości pliku powo-

duje zwykle zmianę wartości jego sumy kontrolnej. Pakiet `java.util.zip` zawiera klasę `CRC32` pozwalającą wyznaczyć sumę kontrolną sekwencji bajtów przy zastosowaniu następującej pętli:

```
CRC32 crc = new CRC32();
while (więcej bajtów)
   crc.update(następny bajt)
long checksum = crc.getValue();
```

Listing 1.7. *NIOTest.java*

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.zip.*;

/**
 * Program obliczający sumę kontrolną CRC pliku.
 * Uruchamianie: java NIOTest nazwapliku
 * @version 1.01 2004-05-11
 * @author Cay Horstmann
 */
public class NIOTest
{
   public static long checksumInputStream(String filename) throws IOException
   {
      InputStream in = new FileInputStream(filename);
      CRC32 crc = new CRC32();

      int c;
      while ((c = in.read()) != -1)
         crc.update(c);
      return crc.getValue();
   }

   public static long checksumBufferedInputStream(String filename) throws
   ↪IOException
   {
      InputStream in = new BufferedInputStream(new FileInputStream(filename));
      CRC32 crc = new CRC32();

      int c;
      while ((c = in.read()) != -1)
         crc.update(c);
      return crc.getValue();
   }

   public static long checksumRandomAccessFile(String filename) throws IOException
   {
      RandomAccessFile file = new RandomAccessFile(filename, "r");
      long length = file.length();
      CRC32 crc = new CRC32();

      for (long p = 0; p < length; p++)
      {
         file.seek(p);
         int c = file.readByte();
```

```
            crc.update(c);
         }
         return crc.getValue();
      }

      public static long checksumMappedFile(String filename) throws IOException
      {
         FileInputStream in = new FileInputStream(filename);
         FileChannel channel = in.getChannel();

         CRC32 crc = new CRC32();
         int length = (int) channel.size();
         MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ_ONLY, 0,
         ↪length);

         for (int p = 0; p < length; p++)
         {
            int c = buffer.get(p);
            crc.update(c);
         }
         return crc.getValue();
      }

      public static void main(String[] args) throws IOException
      {
         System.out.println("Input Stream:");
         long start = System.currentTimeMillis();
         long crcValue = checksumInputStream(args[0]);
         long end = System.currentTimeMillis();
         System.out.println(Long.toHexString(crcValue));
         System.out.println((end - start) + " milliseconds");

         System.out.println("Buffered Input Stream:");
         start = System.currentTimeMillis();
         crcValue = checksumBufferedInputStream(args[0]);
         end = System.currentTimeMillis();
         System.out.println(Long.toHexString(crcValue));
         System.out.println((end - start) + " milliseconds");

         System.out.println("Random Access File:");
         start = System.currentTimeMillis();
         crcValue = checksumRandomAccessFile(args[0]);
         end = System.currentTimeMillis();
         System.out.println(Long.toHexString(crcValue));
         System.out.println((end - start) + " milliseconds");

         System.out.println("Mapped File:");
         start = System.currentTimeMillis();
         crcValue = checksumMappedFile(args[0]);
         end = System.currentTimeMillis();
         System.out.println(Long.toHexString(crcValue));
         System.out.println((end - start) + " milliseconds");
      }
   }
```

Opis działania algorytmu CRC znajdziesz na stronie *http://www.relisoft.com/Science/*
↪CrcMath.html.

Szczegóły obliczeń sumy kontrolnej CRC nie są dla nas istotne. Stosujemy ją jedynie jako przykład pewnej praktycznej operacji na pliku.

Program uruchamiamy w następujący sposób:

```
java NIOTest nazwapliku
```

`java.io.FileInputStream` **1.0**

- `FileChannel getChannel()` **1.4**
  zwraca kanał dostępu do strumienia.

`java.io.FileOutputStream` **1.0**

- `FileChannel getChannel()` **1.4**
  zwraca kanał dostępu do strumienia.

`java.io.RandomAccessFile` **1.0**

- `FileChannel getChannel()` **1.4**
  zwraca kanał dostępu do pliku.

`java.nio.channels.FileChannel` **1.4**

- `MappedByteBuffer map(FileChannel.MapMode mode, long position, long size)`
  tworzy w pamięci mapę fragmentu pliku.

  Parametry:

`mode`	jedna ze stałych `READ_ONLY`, `READ_WRITE` lub `PRIVATE` zdefiniowanych w klasie `FileChannel.MapMode`
`position`	początek mapowanego fragmentu
`size`	rozmiar mapowanego fragmentu

`java.nio.Buffer` **1.4**

- `boolean hasRemaining()`
  zwraca wartość `true`, jeśli bieżąca pozycja bufora nie osiągnęła jeszcze jego końca.
- `int limit()`
  zwraca pozycję końcową bufora; jest to pierwsza pozycja, na której nie są już dostępne kolejne dane bufora.

API `java.nio.ByteBuffer` **1.4**

- `byte get()`

  pobiera bajt z bieżącej pozycji bufora i przesuwa pozycję do kolejnego bajta.

- `byte get(int index)`

  pobiera bajt o podanym indeksie.

- `ByteBuffer put(byte b)`

  umieszcza bajt na bieżącej pozycji bufora i przesuwa pozycję do kolejnego bajta.

- `ByteBuffer put(int index, byte b)`

  umieszcza bajt na podanej pozycji bufora. Zwraca referencję do bufora.

- `ByteBuffer get(byte[] destination)`

- `ByteBuffer get(byte[] destination, int offset, int length)`

  wypełnia tablicę bajtów lub jej zakres bajtami z bufora i przesuwa pozycję bufora o liczbę wczytanych bajtów. Jeśli bufor nie zawiera wystarczającej liczby bajtów, to nie są one w ogóle wczytywane i zostaje wyrzucony wyjątek `BufferUnderflowException`. Zwracają referencję do bufora.

  Parametry: `destination` wypełniana tablica bajtów
  `offset` początek wypełnianego zakresu
  `length` rozmiar wypełnianego zakresu

- `ByteBuffer put(byte[] source)`

- `ByteBuffer put(byte[] source, int offset, int length)`

  umieszcza w buforze wszystkie bajty z tablicy lub jej zakresu i przesuwa pozycję bufora o liczbę umieszczonych bajtów. Jeśli w buforze nie ma wystarczającego miejsca, to nie są zapisywane żadne bajty i zostaje wyrzucony wyjątek `BufferOverflowException`. Zwraca referencję do bufora.

  Parametry: `source` tablica stanowiąca źródło bajtów zapisywanych w buforze
  `offset` początek zakresu źródła
  `length` rozmiar zakresu źródła

- *Xxx* `get`*Xxx*`()`

- *Xxx* `get`*Xxx*`(int index)`

- `ByteBuffer put`*Xxx*`(`*xxx* `value)`

- `ByteBuffer put`*Xxx*`(int index,` *xxx* `value)`

  pobiera lub zapisuje wartość typu podstawowego. *Xxx* może być typu `Int`, `Long`, `Short`, `Char`, `Float` lub `Double`.

- `ByteBuffer order(ByteOrder order)`

- `ByteOrder order()`

  określa lub pobiera uporządkowanie bajtów w buforze. Wartością parametru `order` jest stała `BIG_ENDIAN` lub `LITTLE_ENDIAN` zdefiniowana w klasie `ByteOrder`.

Struktura bufora danych

Gdy używamy mapowania plików w pamięci, tworzymy pojedynczy bufor zawierający cały plik lub interesujący nas fragment pliku. Buforów możemy również używać podczas odczytu i zapisu mniejszych porcji danych.

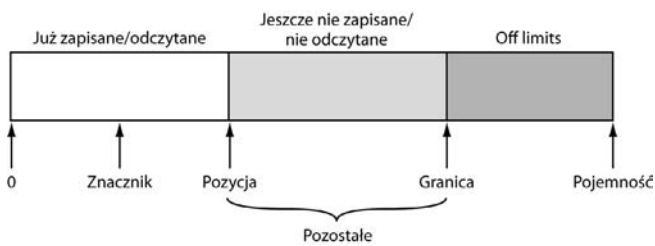
W tym podrozdziale omówimy krótko podstawowe operacje na obiektach typu `Buffer`. Bufor jest w rzeczywistości tablicą wartości tego samego typu. Abstrakcyjna klasa `Buffer` posiada klasy pochodne `ByteBuffer`, `CharBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer` i `ShortBuffer`.

Klasa `StringBuffer` nie jest związana z omawianą tutaj hierarchią klas.

W praktyce najczęściej używane są klasy `ByteBuffer` i `CharBuffer`. Na rysunku 1.11 pokazaliśmy, że bufor jest scharakteryzowany przez:

- *pojemność*, która nigdy nie ulega zmianie;
- *pozycję* wskazującą następną wartość do odczytu lub zapisu;
- *granicę*, poza którą odczyt i zapis nie mają sensu;
- opcjonalny *znacznik* dla powtarzających się operacji odczytu lub zapisu.

Rysunek 1.11. *Bufor*



Wymienione wartości spełniają następujący warunek:

$$0 \leq znacznik \leq pozycja \leq granica \leq pojemność$$

Podstawowa zasada funkcjonowania bufora brzmi: „najpierw zapis, potem odczyt". Na początku pozycja bufora jest równa 0, a granicą jest jego pojemność. Następnie bufor jest wypełniany danymi za pomocą metody `put`. Gdy dane skończą się lub wypełniony zostanie cały bufor, pora przejść do operacji odczytu.

Metoda `flip` przenosi granicę bufora do bieżącej pozycji, a następnie zeruje pozycję. Teraz możemy wywoływać metodę `get`, dopóki metoda `remaining` zwraca wartość większą od zera (metoda ta zwraca różnicę *granica – pozycja*). Po wczytaniu wszystkich wartości z bufora wywołujemy metodę `clear`, aby przygotować bufor do następnego cyklu zapisu. Jak łatwo się domyślić, metoda ta przywraca pozycji wartość 0, a granicy nadaje wartość pojemności bufora.

Jeśli chcemy ponownie odczytać bufor, używamy metody rewind lub metod `mark`/`reset`. Więcej szczegółów na ten temat w opisie metod zamieszczonym poniżej.

API java.nio.Buffer 1.4

- `Buffer clear()`

  przygotowuje bufor do zapisu, nadając pozycji wartość 0, a granicy wartość równą pojemności bufora; zwraca `this`.

- `Buffer flip()`

  przygotowuje bufor do zapisu, nadając granicy wartość równą pozycji, a następnie zerując wartość pozycji; zwraca `this`.

- `Buffer rewind()`

  przygotowuje bufor do ponownego odczytu tych samych wartości, nadając pozycji wartość 0 i pozostawiając wartość granicy bez zmian; zwraca `this`.

- `Buffer mark()`

  nadaje znacznikowi wartość pozycji; zwraca `this`.

- `Buffer reset()`

  nadaje pozycji bufora wartość znacznika, umożliwiając w ten sposób ponowny odczyt lub zapis danych; zwraca `this`.

- `int remaining()`

  zwraca liczbę wartości pozostających do odczytu lub zapisu; jest to różnica pomiędzy wartością granicy i pozycji.

- `int position()`

  zwraca pozycję bufora.

- `int capacity()`

  zwraca `pojemność` bufora.

API java.nio.CharBuffer 1.4

- `char get()`
- `CharBuffer get(char[] destination)`
- `CharBuffer get(char[] destination, int offset, int length)`

  zwraca jedną wartość typu `char` lub zakres wartości typu `char`, począwszy od bieżącej pozycji bufora, która w efekcie zostaje przesunięta za ostatnią wczytaną wartość. Ostatnie dwie wersje zwracają `this`.

- `CharBuffer put(char c)`
- `CharBuffer put(char[] source)`
- `CharBuffer put(char[] source, int offset, int length)`
- `CharBuffer put(String source)`
- `CharBuffer put(CharBuffer source)`

  zapisuje w buforze jedną wartość typu `char` lub zakres wartości typu `char`, począwszy od bieżącej pozycji bufora, która w efekcie zostaje przesunięta za ostatnią zapisaną wartość. Wszystkie wersje zwracają `this`.

- `CharBuffer read(CharBuffer destination)`
  pobiera wartości typu `char` z bufora i umieszcza je w buforze `destination`, do momentu aż zostanie osiągnięta granica bufora `destination`. Zwraca `this`.

Blokowanie plików

Rozważmy sytuację, w której wiele równocześnie wykonywanych programów musi zmodyfikować ten sam plik. Jeśli pomiędzy programami nie będzie mieć miejsca pewien rodzaj komunikacji, to bardzo prawdopodobne jest, że plik zostanie uszkodzony.

Blokady plików pozwalają kontrolować dostęp do plików lub pewnego zakresu bajtów w pliku. Jednak implementacja blokad plików różni się istotnie w poszczególnych systemach operacyjnych i dlatego wcześniejsze wersje JDK nie umożliwiały stosowania takich blokad.

Blokowanie plików nie jest wcale tak powszechne w przypadku typowych aplikacji. Wiele z nich przechowuje swoje dane w bazach danych, które dysponują mechanizmami pozwalającymi na równoległy dostęp. Jeśli nasz program przechowuje dane w plikach, a problem równoległego dostępu zaczyna zaprzątać naszą uwagę, to często łatwiej będzie właśnie skorzystać z bazy danych zamiast projektować skomplikowany system blokowania plików.

Istnieją jednak sytuacje, w których blokowanie plików jest niezbędne. Załóżmy na przykład, że nasza aplikacja zapisuje preferencje użytkownika w pliku konfiguracyjnym. Jeśli uruchomi on dwie instancje aplikacji, to może się zdarzyć, że obie będą chciały zapisać dane w pliku konfiguracyjnym w tym samym czasie. W takiej sytuacji pierwsza instancja powinna zablokować dostęp do pliku. Gdy druga instancja natrafi na blokadę, może zaczekać na odblokowanie pliku lub po prostu pominąć zapis danych.

Aby zablokować plik, wywołujemy metodę `lock` lub `tryLock` klasy `FileChannel`:

```
FileLock lock = channel.lock();
```

lub

```
FileLock lock = channel.tryLock();
```

Pierwsze wywołanie blokuje wykonanie programu do momentu, gdy blokada pliku będzie dostępna. Drugie wywołanie nie powoduje blokowania, lecz natychmiast zwraca blokadę lub wartość `null`, jeśli blokada nie jest dostępna. Plik pozostaje zablokowany do momentu zamknięcia kanału lub wywołania metody `release` dla danej blokady.

Można również zablokować dostęp do fragmentu pliku za pomocą wywołania

```
FileLock lock(long start, long size, boolean exclusive)
```

lub

```
FileLock tryLock(long start, long size, boolean exclusive)
```

Parametrowi `flag` nadajemy wartość `true`, aby zablokować dostęp do pliku zarówno dla operacji odczytu, jak i zapisu. W przypadku blokady współdzielonej parametr `flag` otrzymuje wartość `false`, co umożliwia wielu procesom odczyt pliku, zapobiegając jednak uzyskaniu przez którykolwiek z nich wyłącznej blokady pliku. Nie wszystkie systemy operacyjne obsługują

jednak blokady współdzielone. W takim przypadku możemy uzyskać blokadę wyłączną, nawet jeśli żądaliśmy jedynie blokady współdzielonej. Metoda `isShared` klasy `FileLock` pozwala nam dowiedzieć się, którą z blokad otrzymaliśmy.

Jeśli zablokujemy dostęp do końcowego fragmentu pliku, a rozmiar pliku zwiększy się poza granicę zablokowanego fragmentu, to dostęp do dodatkowego obszaru nie będzie zablokowany. Aby zablokować dostęp do wszystkich bajtów, należy parametrowi `size` nadać wartość `Long.MAX_VALUE`.

Należy pamiętać, że możliwości blokad zależą w znacznej mierze od konkretnego systemu operacyjnego. Poniżej wymieniamy kilka aspektów tego zagadnienia, na które warto zwrócić szczególną uwagę:

- W niektórych systemach blokady plików mają jedynie charakter pomocniczy. Nawet jeśli aplikacji nie uda się zdobyć blokady, to może zapisywać dane w pliku „zablokowanym" wcześniej przez inną aplikację.
- W niektórych systemach nie jest możliwe zablokowanie dostępu do mapy pliku w pamięci.
- Blokady plików są przydzielane na poziomie maszyny wirtualnej Java. Jeśli zatem dwa programy działają na tej samej maszynie wirtualnej, to nie mogą uzyskać blokady tego samego pliku. Metody `lock` i `tryLock` wyrzucą wyjątek `OverlappingFile` ↪`LockException` w sytuacji, gdy maszyna wirtualna jest już w posiadaniu blokady danego pliku.
- W niektórych systemach zamknięcie kanału zwalnia wszystkie blokady pliku będące w posiadaniu maszyny wirtualnej Java. Dlatego też należy unikać wielu kanałów dostępu do tego samego, zablokowanego pliku.
- Działanie blokad plików w sieciowych systemach plików zależy od konkretnego systemu i dlatego należy unikać stosowania blokad w takich systemach.

API `java.nio.channels.FileChannel` 1.4

- `FileLock lock()`

  uzyskuje wyłączną blokadę pliku. Blokuje działanie programu do momentu uzyskania blokady.

- `FileLock tryLock()`

  uzyskuje wyłączną blokadę całego pliku lub zwraca `null`, jeśli nie może uzyskać blokady.

- `FileLock lock(long position, long size, boolean shared)`
- `FileLock tryLock(long position, long size, boolean shared)`

  uzyskuje blokadę dostępu do fragmentu pliku. Pierwsza wersja blokuje działanie programu do momentu uzyskania blokady, a druga zwraca natychmiast wartość `null`, jeśli nie może uzyskać od razu blokady.

  Parametry: `position` początek blokowanego fragmentu
  `size` rozmiar blokowanego fragmentu

`shared`	wartość `true` dla blokady współdzielonej, `false` dla wyłącznej

API `java.nio.channels.FileLock` **1.4**

- `void release()`
  zwalnia blokadę.

Wyrażenia regularne

Wyrażenia regularne stosujemy do określenia wzorców występujących w łańcuchach znaków. Używamy ich najczęściej wtedy, gdy potrzebujemy odnaleźć łańcuchy zgodne z pewnym wzorcem. Na przykład jeden z naszych przykładowych programów odnajdywał w pliku HTML wszystkie hiperłącza, wyszukując łańcuchy zgodne ze wzorcem `<a href= "... ">`.

Oczywiście zapis `...` nie jest wystarczająco precyzyjny. Specyfikując wzorzec, musimy dokładnie określić znaki, które są dopuszczalne. Dlatego też opis wzorca wymaga zastosowania odpowiedniej składni.

Oto prosty przykład. Z wyrażeniem regularnym

```
[Jj]ava.+
```

może zostać uzgodniony dowolny łańcuch znaków następującej postaci:

- Pierwszą jego literą jest `J` lub `j`.
- Następne trzy litery to `ava`.
- Pozostała część łańcucha może zawierać jeden lub więcej dowolnych znaków.

Na przykład łańcuch `"javanese"` zostanie dopasowany do naszego wyrażenia regularnego, `"Core Java"` już nie.

Aby posługiwać się wyrażeniami regularnymi, musimy nieco bliżej poznać ich składnię. Na szczęście na początek wystarczy kilka dość oczywistych konstrukcji.

- Przez *klasę znaków* rozumiemy zbiór alternatywnych znaków ujęty w nawiasy kwadratowe, na przykład `[Jj]`, `[0-9]`, `[A-Za-z]` czy `[^0-9]`. Znak `-` oznacza zakres (czyli wszystkie znaki, których kody Unicode leżą w podanych granicach), a znak `^` oznacza dopełnienie (wszystkie znaki oprócz podanych).
- Istnieje wiele wstępnie zdefiniowanych klas znaków, takich jak `\d` (cyfry) czy `\p{Sc}` (symbol waluty w Unicode). Patrz przykłady w tabelach 1.7 i 1.8.
- Większość znaków oznacza samą siebie, tak jak znaki `ava` w poprzednim przykładzie.
- Symbol `.` oznacza dowolny znak (z wyjątkiem, być może, znaków końca wiersza, co zależy od stanu odpowiedniego znacznika).

Tabela 1.7. *Składnia wyrażeń regularnych*

Składnia	Objaśnienie
Znaki	
c	Znak *c*.
\unnnn, \xnn, \0n, \0nn, \0nnn	Znak o kodzie, którego wartość została podana w notacji szesnastkowej lub ósemkowej.
\t, \n, \r, \f, \a, \e	Znaki sterujące tabulatora, nowego wiersza, powrotu karetki, końca strony, alertu i sekwencji sterującej.
\cc	Znak sterujący odpowiadający znakowi c.
Klasy znaków	
[C_1C_2. . .]	Dowolny ze znaków reprezentowanych przez C_1C_2. . ., gdzie C_i jest znakiem, zakresem znaków (c_1-c_2) lub klasą znaków.
[^. . .]	Dopełnienie klasy znaków.
[. . .&&. . .]	Część wspólna (przecięcie) dwóch klas znaków.
Wstępnie zdefiniowane klasy znaków	
.	Dowolny znak oprócz kończącego wiersz (lub dowolny znak, jeśli znacznik `DOTALL` został ustawiony).
\d	Cyfra [`0-9`].
\D	Znak, który nie jest cyfrą [`^0-9`].
\s	Znak odstępu [`\t\n\r\f\x0B`].
\S	Znak, który nie jest odstępem.
\w	Znak słowa [`a-zA-Z0-9_`].
\W	Znak inny niż znak słowa.
\p{nazwa}	Klasa znaków o podanej nazwie (patrz tabela 1.8).
\P{nazwa}	Dopełnienie klasy znaków o podanej nazwie.
Granice dopasowania	
^ $	Początek, koniec wejścia (lub początek, koniec wiersza w trybie wielowierszowym).
\b	Granica słowa.
\B	Granica inna niż słowa.
\A	Początek wejścia.
\z	Koniec wejścia.
\Z	Koniec wejścia oprócz ostatniego zakończenia wiersza.
\G	Koniec poprzedniego dopasowania.
Kwantyfikatory	
X?	Opcjonalnie X.

Tabela 1.7. *Składnia wyrażeń regularnych — ciąg dalszy*

Składnia	Objaśnienie
X*	X, 0 lub więcej razy.
X+	X, 1 lub więcej razy.
X{n} X{n,} X{n,m}	X *n* razy, co najmniej *n* razy, pomiędzy *n* i *m* razy.
Przyrostki kwantyfikatora	
?	Powoduje dopasowanie najmniejszej liczby wystąpień.
+	Powoduje dopasowanie największej liczby wystąpień, nawet kosztem ogólnego powodzenia dopasowania.
Operacje na zbiorach	
XY	Dowolny łańcuch z *X*, po którym następuje dowolny łańcuch z *Y*.
X|Y	Dowolny łańcuch z *X* lub *Y*.
Grupowanie	
(X)	Grupa.
\n	Dopasowanie n-tej grupy.
Sekwencje sterujące	
\c	Znak c (nie może być znakiem alfabetu).
\Q...\E	Cytat… dosłownie.
(?...)	Specjalna konstrukcja — patrz opis klasy Pattern.

Tabela 1.8. *Wstępnie zdefiniowane nazwy klas znaków*

Nazwa klasy znaków	Objaśnienie
`Lower`	Małe litery ASCII [`a-z`]
`Upper`	Duże litery ASCII [`A-Z`]
`Alpha`	Litery alfabetu ASCII [`A-Za-z`]
`Digit`	Cyfry ASCII [`0-9`]
`Alnum`	Litery alfabetu bądź cyfry ASCII [`A-Za-z0-9`]
`XDigit`	Cyfry szesnastkowe [`0-9A-Fa-f`]
`Print lub Graph`	Znaki ASCII posiadające reprezentację graficzną (na wydruku) [`\x21-\x7E`]
`Punct`	Znaki, które nie należą do znaków alfanumerycznych, bądź cyfry [`\p{Print}&&\P{Alnum}`]
`ASCII`	Wszystkie znaki ASCII [`\x00-\x7F`]
`Cntrl`	Znaki sterujące ASCII [`\x00-\x1F`]
`Blank`	Spacja lub tabulacja [`\t`]
`Space`	Odstęp [`\t\n\r\f\0x0B`]

Tabela 1.8. *Wstępnie zdefiniowane nazwy klas znaków — ciąg dalszy*

Nazwa klasy znaków	Objaśnienie
`javaLowerCase`	Mała litera, zgodnie z wynikiem metody Character.isLowerCase()
`javaUpperCase`	Duża litera, zgodnie z wynikiem metody Character.isUpperCase()
`javaWhitespace`	Odstęp, zgodnie z wynikiem metody Character.isWhiteSpace()
`javaMirrored`	Ekwiwalent wyniku metody Character.isMirrored()
`InBlok`	Blok jest nazwą bloku znaków Unicode z usuniętymi spacjami, na przykład BasicLatin lub Mongolian. Listę nazw bloków znajdziesz na stronach witryny *http://www.unicode.org*
Kategoria lub *InKategoria*	*Kategoria* jest nazwą kategorii znaków Unicode, na przykład L (litera) czy Sc (symbol waluty). Listę nazw kategorii znajdziesz na stronach witryny *http://www.unicode.org*

- `\` spełnia rolę znaku specjalnego, na przykład `\.` oznacza znak kropki, a `\\` znak lewego ukośnika.
- `^` i `$` oznaczają odpowiednio początek i koniec wiersza.
- Jeśli *X* i *Y* są wyrażeniami regularnymi, to *XY* oznacza „dowolne dopasowanie do *X*, po którym następuje dowolne dopasowanie do *Y*", a *X* | *Y* „dowolne dopasowanie do *X* lub *Y*".
- Do wyrażenia regularnego *X* możemy stosować *kwantyfikatory X*+ (raz lub więcej), *X** (0 lub więcej) i *X*? (0 lub 1).
- Domyślnie kwantyfikator dopasowuje największą możliwą liczbę wystąpień, która gwarantuje ogólne powodzenie dopasowania. Zachowanie to możemy zmodyfikować za pomocą przyrostka `?` (dopasowanie najmniejszej liczby wystąpień) i przyrostka `+` (dopasowanie największej liczby wystąpień, nawet jeśli nie gwarantuje ono ogólnego powodzenia dopasowania).

  Na przykład łańcuch `cab` może zostać dopasowany do wyrażenia `[a-z]*ab`, ale nie do `[a-z]*+ab`. W pierwszym przypadku wyrażenie `[a-z]*` dopasuje jedynie znak `c`, wobec czego znaki `ab` zostaną dopasowane do reszty wzorca. Jednak wyrażenie `[a-z]*+` dopasuje znaki `cab`, wobec czego reszta wzorca pozostanie bez dopasowania.
- Grupy pozwalają definiować podwyrażenia. Grupy ujmujemy w znaki nawiasów `( )`; na przykład `([+-]?)([0-9]+)`. Możemy następnie zażądać dopasowania do wszystkich grup lub do wybranej grupy, do której odwołujemy się przez `\`*n*, gdzie *n* jest numerem grupy (numeracja rozpoczyna się od `\1`).

A oto przykład nieco skomplikowanego, ale potencjalnie użytecznego wyrażenia regularnego, które opisuje liczby całkowite zapisane dziesiętnie lub szesnastkowo:

```
[+-]?[0-9]+|0[Xx][0-9A-Fa-f]+
```

Niestety, składnia wyrażeń regularnych nie jest całkowicie ustandaryzowana. Istnieje zgodność w zakresie podstawowych konstrukcji, ale diabeł tkwi w szczegółach. Klasy języka Java związane z przetwarzaniem wyrażeń regularnych używają składni podobnej do zastosowanej w języku Perl. Wszystkie konstrukcje tej składni zostały przedstawione w tabeli 1.7. Więcej

informacji na temat składni wyrażeń regularnych znajdziesz w dokumentacji klasy `Pattern` lub książce *Wyrażenia regularne* autorstwa J.E.F. Friedla (Wydawnictwo Helion, 2001).

Najprostsze zastosowanie wyrażenia regularnego polega na sprawdzeniu, czy dany łańcuch znaków pasuje do tego wyrażenia. Oto w jaki sposób zaprogramować taki test w języku Java. Najpierw musimy utworzyć obiekt klasy `Pattern` na podstawie łańcucha opisującego wyrażenie regularne. Następnie pobrać obiekt klasy `Matcher` i wywołać jego metodę `matches`:

```
Pattern pattern = Pattern.compile(patternString);
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) . . .
```

Wejście obiektu `Matcher` stanowi obiekt dowolnej klasy implementującej interfejs `Char`
`↪Sequence`, na przykład `String`, `StringBuilder` czy `CharBuffer`.

Kompilując wzorzec, możemy skonfigurować jeden lub więcej znaczników, na przykład:

```
Pattern pattern = Pattern.compile(patternString,
   Pattern.CASE_INSENSITIVE + Pattern.UNICODE_CASE);
```

Obsługiwanych jest sześć następujących znaczników:

- `CASE_INSENSITIVE` — dopasowanie niezależnie od wielkości liter. Domyślnie dotyczy to tylko znaków US ASCII.
- `UNICODE_CASE` — zastosowany w połączeniu z `CASE_INSENSITIVE`, dotyczy wszystkich znaków Unicode.
- `MULTILINE` — `^` i `$` oznaczają początek i koniec wiersza, a nie całego wejścia.
- `UNIX_LINES` — tylko `'\n'` jest rozpoznawany jako zakończenie wiersza podczas dopasowywania do `^` i `$` w trybie wielowierszowym.
- `DOTALL` — symbol `.` oznacza wszystkie znaki, w tym końca wiersza.
- `CANON_EQ` — bierze pod uwagę kanoniczny odpowiednik znaków Unicode. Na przykład znak `u`, po którym następuje znak `¨` (diareza), zostanie dopasowany do znaku `ü`.

Jeśli wyrażenie regularne zawiera grupy, obiekt `Matcher` pozwala ujawnić granice grup. Metody:

```
int start(int groupIndex)
int end(int groupIndex)
```

zwracają indeks początkowy i końcowy podanej grupy.

Dopasowany łańcuch możemy pobrać, wywołując

```
String group(int groupIndex)
```

Grupa 0 oznacza całe wejście; indeks pierwszej grupy równy jest 1. Metoda `groupCount` zwraca całkowitą liczbę grup.

Grupy zagnieżdżone są uporządkowane według nawiasów otwierających. Na przykład wzorzec opisany wyrażeniem

```
((1?[0-9]):([0-5][0-9]))[ap]m
```

dla danych

```
11:59am
```

spowoduje, że obiekt klasy `Matcher` będzie raportować grupy w poniższy sposób:

Indeks grupy	Początek	Koniec	Łańcuch
0	0	7	11:59am
1	0	5	11:59
2	0	2	11
3	3	5	59

Program przedstawiony na listingu 1.8 umożliwia wprowadzenie wzorca, a następnie łańcucha, którego dopasowanie zostanie sprawdzone. Jeśli łańcuch pasuje do wzorca zawierającego grupy, to program wyświetla granice grup w postaci nawiasów, na przykład:

```
((11):(59))am
```

Listing 1.8. *RegExTest.java*

```
import java.util.*;
import java.util.regex.*;

/**
   Program testujący zgodność z wyrażeniem regularnym.
   Wprowadź wzorzec i dopasowywany łańcuch.
   Jeśli wzorzec zawiera grupy, program
   wyświetli ich granice po uzgodnieniu łańcucha ze wzorcem.
   @version 1.01 2004-05-11
   @author Cay Horstmann
*/
public class RegExTest
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.println("Enter pattern: ");
      String patternString = in.nextLine();

      Pattern pattern = null;
      try
      {
         pattern = Pattern.compile(patternString);
      }
      catch (PatternSyntaxException e)
      {
         System.out.println("Pattern syntax error");
         System.exit(1);
      }

      while (true)
      {
         System.out.println("Enter string to match: ");
         String input = in.nextLine();
         if (input == null || input.equals("")) return;
         Matcher matcher = pattern.matcher(input);
         if (matcher.matches())
```

```
            {
               System.out.println("Match");
               int g = matcher.groupCount();
               if (g > 0)
               {
                  for (int i = 0; i < input.length(); i++)
                  {
                     for (int j = 1; j <= g; j++)
                        if (i == matcher.start(j))
                           System.out.print('(');
                     System.out.print(input.charAt(i));
                     for (int j = 1; j <= g; j++)
                        if (i + 1 == matcher.end(j))
                           System.out.print(')');
                  }
                  System.out.println();
               }
            }
            else
               System.out.println("No match");
         }
      }
   }
```

Zwykle nie chcemy dopasowywać do wzorca całego łańcucha wejściowego, lecz jedynie odnaleźć jeden lub więcej podłańcuchów. Aby znaleźć kolejne dopasowanie, używamy metody `find` klasy `Matcher`. Jeśli zwróci ona wartość `true`, to stosujemy metody `start` i `end` w celu odnalezienia dopasowanego podłańcucha.

```
while (matcher.find())
{
   int start = matcher.start();
   int end = matcher.end();
   String match = input.substring(start, end);
   . . .
}
```

Program przedstawiony na listingu 1.9 wykorzystuje powyższy mechanizm. Odnajduje on wszystkie hiperłącza na stronie internetowej i wyświetla je. Uruchamiając program, podajemy adres URL jako parametr w wierszu poleceń, na przykład:

```
java HrefMatch http://www.horstmann.com
```

Listing 1.9. *HrefMatch.java*

```
import java.io.*;
import java.net.*;
import java.util.regex.*;

/**
 * Program wyświetlający wszystkie adresy URL na stronie WWW
 * poprzez dopasowanie wyrażenia regularnego
 * opisującego znacznik <a href=...> języka HTML.
 * Uruchamianie: java HrefMatch adresURL
 * @version 1.01 2004-06-04
 * @author Cay Horstmann
 */
```

```
public class HrefMatch
{
   public static void main(String[] args)
   {
      try
      {
         // pobiera URL z wiersza poleceń lub używa domyślnego
         String urlString;
         if (args.length > 0) urlString = args[0];
         else urlString = "http://java.sun.com";

         // otwiera InputStreamReader dla podanego URL
         InputStreamReader in = new InputStreamReader(new
         ↪URL(urlString).openStream());

         // wczytuje zawartość do obiektu klasy StringBuilder
         StringBuilder input = new StringBuilder();
         int ch;
         while ((ch = in.read()) != -1)
            input.append((char) ch);

         // poszukuje wszystkich wystąpień wzorca
         String patternString = "<a\\s+href\\s*=\\s*(\"[^\"]*\"|[^\\s>])\\s*>";
         Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);
         Matcher matcher = pattern.matcher(input);

         while (matcher.find())
         {
            int start = matcher.start();
            int end = matcher.end();
            String match = input.substring(start, end);
            System.out.println(match);
         }
      }
      catch (IOException e)
      {
         e.printStackTrace();
      }
      catch (PatternSyntaxException e)
      {
         e.printStackTrace();
      }
   }
}
```

Metoda `replaceAll` klasy `Matcher` zastępuje wszystkie wystąpienia wyrażenia regularnego podanym łańcuchem. Na przykład poniższy kod zastąpi wszystkie sekwencje cyfr znakiem #:

```
Pattern pattern = Pattern.compile("[0-9]+");
Matcher matcher = pattern.matcher(input);
String output = matcher.replaceAll("#");
```

Łańcuch zastępujący może zawierać referencje grup wzorca: `$n` zostaje zastąpione przez *n*-tą grupę. Sekwencja `\$` pozwala umieścić znak `$` w zastępującym tekście.

Metoda `replaceFirst` zastępuje jedynie pierwsze wystąpienie wzorca.

Klasa Pattern dysponuje również metodą split, która dzieli łańcuch wejściowy na tablicę łańcuchów, używając dopasowań wyrażenia regularnego jako granic podziału. Na przykład poniższy kod podzieli łańcuch wejściowy na tokeny na podstawie znaków interpunkcyjnych otoczonych opcjonalnym odstępem.

```
Pattern pattern = Pattern.compile("\\s*\\p{Punct}\\s*");
String[] tokens = pattern.split(input);
```

API java.util.regex.Pattern 1.4

- static Pattern compile(String expression)
- static Pattern compile(String expression, int flags)

  kompiluje łańcuch wyrażenia regularnego, tworząc obiekt wzorca przyspieszający przetwarzanie.

  Parametry: expression — wyrażenie regularne
  flags — jeden lub więcej znaczników CASE_INSENSITIVE, UNICODE_CASE, MULTILINE, UNIX_LINES, DOTALL i CANON_EQ.

- Matcher matcher(CharSequence input)

  tworzy obiekt pozwalający odnajdywać dopasowania do wzorca w łańcuchu wejściowym.

- String[] split(CharSequence input)
- String[] split(CharSequence input, int limit)

  rozbija łańcuch wejściowy na tokeny, stosując wzorzec do określenia granic podziału. Zwraca tablicę tokenów, które nie zawierają granic podziału.

  Parametry: input — łańcuch rozbijany na tokeny
  limit — maksymalna liczba utworzonych łańcuchów. Jeśli dopasowanych zostało limit - 1 granic podziału, to ostatni element zwracanej tablicy zawiera niepodzieloną resztę łańcucha wejściowego. Jeśli limit jest równy lub mniejszy od 0, to zostanie podzielony cały łańcuch wejściowy. Jeśli limit jest równy 0, to puste łańcuchy kończące dane wejściowe nie są umieszczane w tablicy

API java.util.regex.Matcher 1.4

- boolean matches()

  zwraca true, jeśli łańcuch wejściowy pasuje do wzorca.

- boolean lookingAt()

  zwraca true, jeśli początek łańcucha wejściowego pasuje do wzorca.

- boolean find()

- `boolean find(int start)`

  próbuje odnaleźć następne dopasowanie i zwraca `true`, jeśli próba się powiedzie.

  Parametry: `start` indeks, od którego należy rozpocząć poszukiwanie

- `int start()`
- `int end()`

  zwraca pozycję początkową dopasowania lub następną pozycję za dopasowaniem.

- `String group()`

  zwraca bieżące dopasowanie.

- `int groupCount()`

  zwraca liczbę grup we wzorcu wejściowym.

- `int start(int groupIndex)`
- `int end(int groupIndex)`

  zwraca pozycję początkową grupy lub następną pozycję za grupą dla danej grupy bieżącego dopasowania.

  Parametry: `groupIndex` indeks grupy (wartości indeksu rozpoczynają się od 1) lub 0 dla oznaczenia całego dopasowania

- `String group(int groupIndex)`

  zwraca łańcuch dopasowany do podanej grupy.

  Parametry: `groupIndex` indeks grupy (wartości indeksu rozpoczynają się od 1) lub 0 dla oznaczenia całego dopasowania

- `String replaceAll(String replacement)`
- `String replaceFirst(String replacement)`

  zwracają łańcuch powstały przez zastąpienie podanym łańcuchem wszystkich dopasowań lub tylko pierwszego dopasowania.

  Parametry: `replacement` łańcuch zastępujący może zawierać referencje do grup wzorca postaci `$`*n*. Aby umieścić w łancuchu symbol `$`, stosujemy sekwencję `\$`.

- `Matcher reset()`
- `Matcher reset(CharSequence input)`

  resetuje stan obiektu `Matcher`. Druga wersja powoduje przejście obiektu `Matcher` do pracy z innymi danymi wejściowymi. Obie wersje zwracają `this`.

W tym rozdziale omówiliśmy metody obsługi plików i katalogów, a także metody zapisywania informacji do plików w formacie tekstowym i binarnym i wczytywania informacji z plików w formacie tekstowym i binarnym, jak również szereg ulepszeń, które do obsługi wejścia i wyjścia wprowadził pakiet java.nio. W następnym rozdziale omówimy możliwości biblioteki języka Java związane z przetwarzaniem języka XML.