WYDANIE XIII

JAVA

Techniki zaawansowane

ORACLE

Cay S. Horstmann

Tytuł oryginału: Core Java, Volume II: Advanced Features, 13th Edition

Tłumaczenie: Piotr Rajca

ISBN: 978-83-289-3004-9

Authorized translation from the English language edition, entitled CORE JAVA VOLUME II: ADVANCED FEATURES, 13th Edition 9780135371749 by Cay S. Horstmann published by Pearson Education, Inc, publishing as Oracle Press Copyright © 2025 Pearson Education, Inc.

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/jatz13
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

2

Wejście i wyjście

W tym rozdziale omówimy interfejsy programowe związane z obsługą wejścia i wyjścia programów. Przedstawimy sposoby dostępu do plików i katalogów oraz sposoby 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. Rozdział zakończymy przedstawieniem problematyki wyrażeń regularnych, mimo że nie jest ona bezpośrednio związana z zagadnieniami wejścia-wyjścia. 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 „nowej wersji" obsługi wejścia i wyjścia.

2.1. Strumienie wejścia-wyjścia

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 `Output`
`↪Stream` stanowią bazę hierarchii klas opisujących wejście i wyjście programów Java.

Te strumienie wejścia i wyjścia nie są powiązane ze strumieniami, które zostały opisane w poprzednim rozdziale. Dla jasności w przypadku opisywania strumieni związanych z operacjami wejścia i wyjścia zawsze będziemy używali terminów „strumień wejścia", „strumień wyjścia" lub „strumień wejścia-wyjścia".

Ponieważ binarne strumienie wejścia-wyjścia 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 wartościach `char` (czyli znaków UTF-16), nie przydają się natomiast do operacji na wartościach typu `byte`.

2.1.1. Odczyt i zapis bajtów

Klasa `InputStream` ma 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 przesłaniają 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 `InputStream`, pozwalający pobierać informacje ze „standardowego wejścia”, czyli konsoli lub przekierowanego pliku.

Klasa `InputStream` ma także bardzo wygodne metody pozwalające wczytać wszystkie bajty w strumieniu:

```
byte[] bytes = in.readAllBytes();
```

Dostępne są też metody umożliwiające wczytanie określonej liczby bajtów — informacje o nich znajdziesz w uwagach dotyczących API.

Metody te wywołują abstrakcyjną metodę `read`, więc klasy pochodne muszą przesłaniać tylko tę jedną metodę.

Analogicznie, klasa `OutputStream` definiuje metodę abstrakcyjną

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

która wysyła jeden bajt do aktualnego wyjścia.

Jeśli dysponujemy tablicą bajtów, to możemy zapisać całą jej zawartość, używając jednego wywołania:

```
byte[] values = . . .;
out.write(values);
```

Metoda `transferTo` przenosi wszystkie bajty ze strumienia wejściowego do wyjściowego:

```
in.transferTo(out);
```

Metody `read` i `write` potrafią *zablokować* wątek, dopóki dany bajt nie zostanie wczytany lub zapisany. Oznacza to, że jeżeli strumień wejścia 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 bytesAvailable = in.available();
if (bytesAvailable > 0)
{
 var data = new byte[bytesAvailable];
 in.read(data);
}
```

Gdy skończymy odczytywać albo zapisywać dane do strumienia wejścia-wyjścia, zamykamy go, wywołując metodę `close`. Metoda ta uwalnia zasoby systemu operacyjnego, do tej pory udostępnione wątkowi. Jeżeli aplikacja otworzy zbyt wiele strumieni wejścia-wyjścia, nie zamykając ich, zasoby systemu mogą zostać naruszone. Co więcej, zamknięcie strumienia wyjścia powoduje *opróżnienie* bufora używanego przez ten strumień — wszystkie bajty, 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 wejścia-wyjścia 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.

Zamiast operować na bajtach, można skorzystać z jednej z wielu klas strumieni pochodzących od podstawowych klas `InputStream` i `OutputStream`.

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 wejścia.

- `int read(byte[] b)`

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

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

- `int readNBytes(byte[] b, int off, int len)` 9

  wczytuje bez blokownia (`read`) do tablicy co najwyżej `len` bajtów, o ile są one dostępne, ewentualnie blokuje operację aż do momentu, kiedy wartości staną się dostępne (`readNBytes`). Wartości są umieszczane w tablicy `b`, zaczynając do jej elementu o indeksie `off`. Zwraca faktyczą liczbę wczytanych bajtów, a jeżeli natrafi na koniec strumienia wejścia, zwraca –1.

- `byte[] readAllBytes()` **9**

  zwraca tablicę zawierającą wszystkie bajty, które można odczytać z tego strumienia.

- `lont transferTo(OutputStream out)` **9**

  przekazuje wszystkie bajty z tego strumienia wejściowego do podanego strumienia wyjściowego i zwraca liczbę przekazanych bajtów. Żaden ze strumieni nie jest zamykany.

- `long skip(long n)`

  ignoruje `n` bajtów w strumieniu wejściowym, zwraca faktyczną liczbę zignorowanych bajtów (która może być mniejsza niż `n`).

- `long skipNBytes(long n)` **12**

  ignoruje `n` bajtów w strumieniu wejściowym, zwraca faktyczną liczbę zignorowanych bajtów (która może być mniejsza od `n`, jeśli przed zadaną liczbą bajtów został osiągnięty 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 wejścia nie zostanie zresetowany.

- `boolean markSupported()`

  zwraca `true`, jeżeli strumień wejścia obsługuje znaczniki.

- `static InputStream nullInputStream()` **11**

  zwraca strumień wejściowy, który nie zawiera żadnych bajtów.

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.

- `void close()`

  opróżnia i zamyka strumień wyjścia.

- `void flush()`

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

- `static OutputStream nullOutputStream()` **11**

  zwraca strumień wyjściowy, który nie zawiera żadnych bajtów.

2.1.2. Zoo pełne strumieni

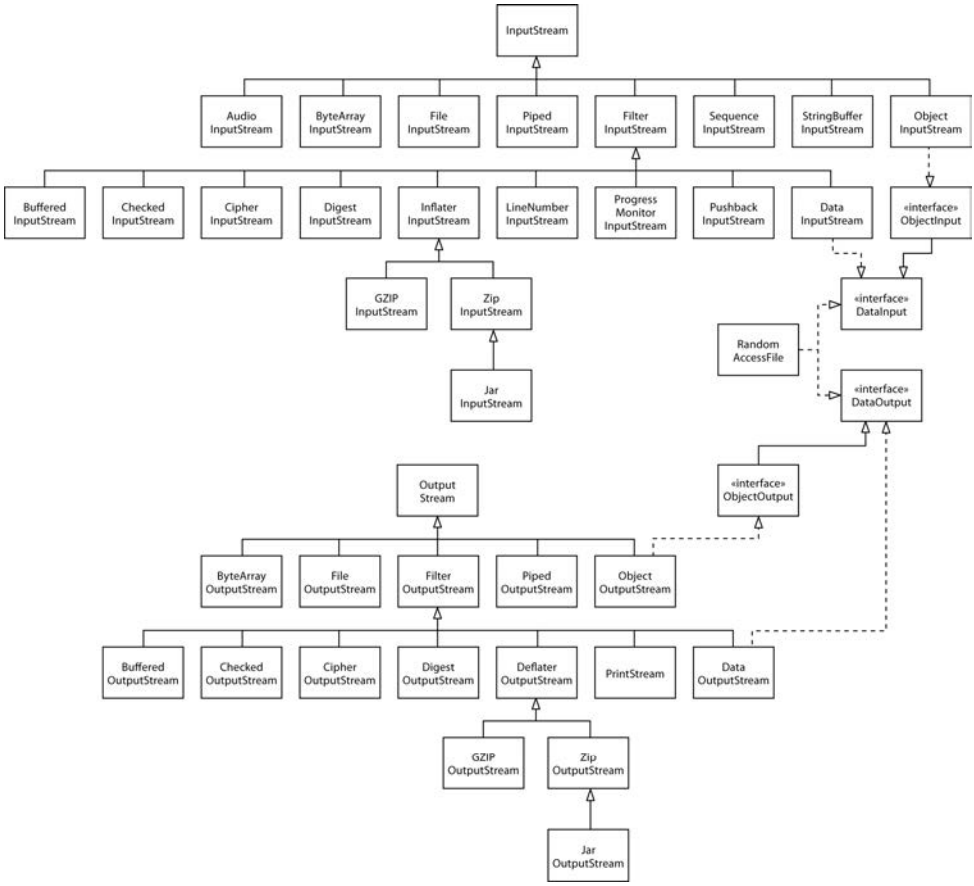
W przeciwieństwie do języka C, który w zupełności zadowala się jednym typem `FILE*`, Java ma prawdziwą menażerię ponad 60 (!) różnych klas i interfejsów związanych ze strumieniami wejścia i wyjścia.

Podzielmy gatunki należące do zoo strumieni zależnie od ich przeznaczenia. Istnieją osobne hierarchie klas przetwarzających bajty i znaki. Jak już o tym wspomnieliśmy, klasy `Input`
`↪Stream` i `OutputStream` pozwalają pobierać i wysyłać jedynie pojedyncze bajty oraz tablice bajtów. Klasy te stanowią bazę hierarchii pokazanej na rysunku 2.1. Do odczytu i zapisu liczb i łańcuchów znakowych używamy ich klas pochodnych. Na przykład `DataInputStream` i `DataOutputStream` pozwalają wczytywać i zapisywać wszystkie podstawowe typy Javy w postaci binarnej. I w końcu istnieje także wiele klas strumieni służących do wyspecjalizowanych zadań, na przykład `ZipInputStream` i `ZipOutputStream`, pozwalające odczytywać i zapisywać dane w plikach skompresowanych w formacie ZIP.

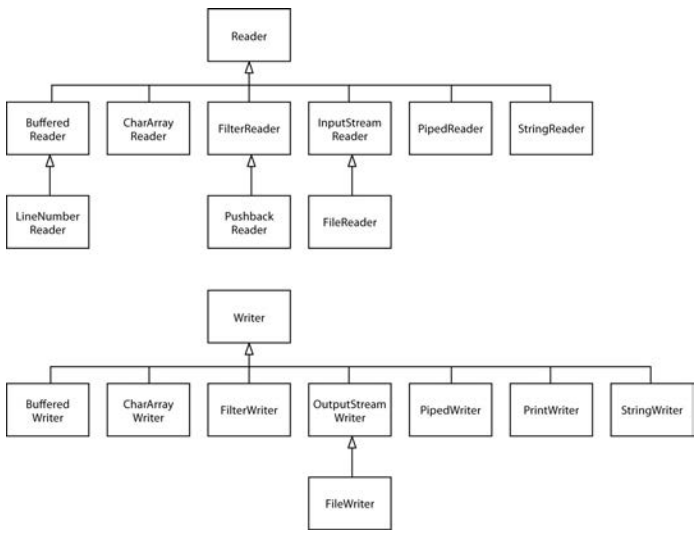
Z drugiej strony, o czym już wspominaliśmy, do obsługi tekstu Unicode używamy klas pochodzących od klas abstrakcyjnych `Reader` i `Writer` (patrz rysunek 2.2) Podstawowe metody klas `Reader` i `Writer` są podobne do tych należących do `InputStream` i `OutputStream`.

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

Metoda `read` zwraca albo kod znaku UTF-16 (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 znajdziesz w rozdziale 3. książki *Java. Podstawy*).



Rysunek 2.1. Hierarchia strumieni wejścia i wyjścia



Rysunek 2.2. Hierarchia klas Reader i Writer

Dostępne są również cztery dodatkowe interfejsy: `Closeable`, `Flushable`, `Readable` i `Appendable` (patrz rysunek 2.3). 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`.

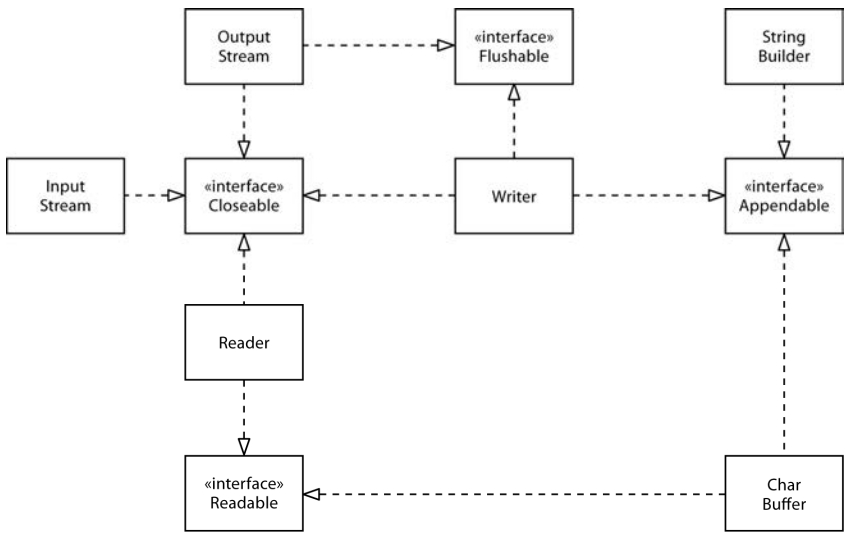
Interfejs `java.io.Closeable` stanowi rozszerzenie interfejsu `java.lang.AutoCloseable`. Dzięki temu dla każdego obiektu implementującego interfejs `Closeable` możemy użyć wersji instrukcji `try` zarządzającej zasobami. Ale po co nam dwa interfejsy? Metoda `close` interfejsu `Closeable` może zgłaszać jedynie wyjątek `IOException`, natomiast metoda `AutoCloseable.close` może zgłaszać wyjątek dowolnej klasy.

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 2.5.2).



Rysunek 2.3. Interfejsy Closeable, Flushable, Readable i Appendable

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

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

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

Spośród klas strumieni wejścia-wyjścia jedynie klasa `Writer` implementuje interfejs `Appendable`.

API *java.io.Closeable* 5.0

- `void close()`

  zamyka obiekt implementujący interfejs `Closeable`. Może zgłaszać 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.

2.1.3. Łączenie filtrów strumieni wejścia-wyjścia

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

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

spróbuje odszukać plik o nazwie *employee.dat*.

Ścieżki relatywne zaczynają się od *katalogu roboczego*. Katalog ten możesz pobrać przy użyciu wywołania `System.getProperty("user.dir")`. Jest on określany podczas uruchamiania maszyny wirtualnej, a po jej uruchomieniu już nie można go zmienić. Jeśli uruchomisz program z poziomu wiersza poleceń, to katalogiem roboczym będzie bieżący katalog wybrany w terminalu. W przypadku korzystania ze zintegrowanego środowiska programistycznego (IDE) katalog ten będzie można określić w opcjach konfiguracji uruchomieniowej.

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 wywołań systemu obsługi plików Windows interpretuje znaki / jako separatory ścieżki dostępu. Jednakże nie zalecamy tego rozwiązania — zachowanie funkcji systemu Windows może się zmienić. Jeżeli piszemy aplikację przenośną, powinniśmy używać separatora odpowiedniego dla danego systemu operacyjnego. Jego znak jest przechowywany jako stała `java.io.File.separator`.

Tak jak klasy abstrakcyjne `InputStream` i `OutputStream`, powyższe klasy obsługują jedynie 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 x = 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 wejścia (takie jak `FileInputStream` i strumień wejścia zwracany przez metodę `openStream` klasy `URL`) mogą udostępniać bajty z plików i innych, bardziej egzotycznych lokalizacji. Inne strumienie wejścia (takie jak `DataInputStream`) 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`.

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

Wróćmy do rysunku 2.1, gdzie przedstawione są klasy `FilterInputStream` i `FilterOutput`
`↪Stream`. Ich klasy pochodne możemy wykorzystać do zwiększania możliwości strumieni wejścia-wyjścia służących do operowania na zwykłych bajtach.

Różne funkcjonalności możemy dodawać poprzez zagnieżdżanie filtrów. Na przykład — domyślnie strumienie wejścia 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:

```
var 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`.

```
var 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);
```

Metoda `unread` w rzeczywistości nie modyfikuje źródła, z którego strumień odczytuje bajty stanowiące jego zawartość. Wstawiane ponownie bajty są umieszczane w buforze, po czym zostają zwrócone przez następną operację odczytu (`read`). Domyślnie bufor ten ma długość wynoszącą 1, jednak jego wielkość można zmienić w konstruktorze klasy PushbackInputStream.

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

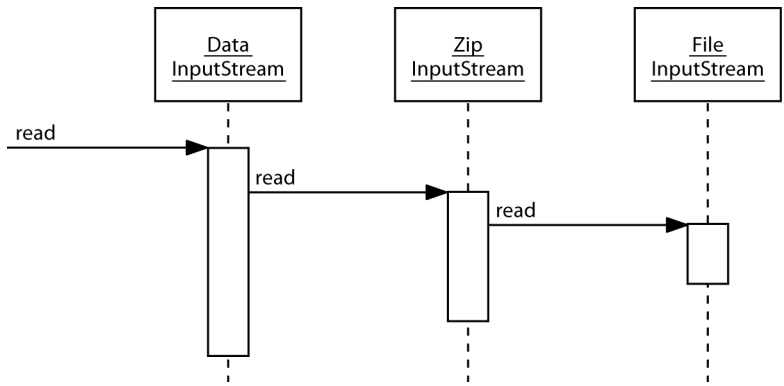
```
var pbin = new PushbackInputStream(
   new BufferedInputStream(
```

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

Oczywiście w bibliotekach wejścia-wyjścia 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 wejścia-wyjścia daje nam niespotykaną elastyczność. Na przykład, korzystając z poniższej sekwencji strumieni, możemy wczytywać liczby ze skompresowanego pliku ZIP (patrz rysunek 2.4).

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

Rysunek 2.4.
Sekwencja filtrowanych strumieni



(Aby dowiedzieć się więcej o obsłudze formatu ZIP, zajrzyj do punktu 2.3.3, poświęconego strumieniom plików ZIP.)

API java.io.FileInputStream 1.0

- `FileInputStream(String name)`
- `FileInputStream(File file)`

  tworzy nowy obiekt typu `FileInputStream`, używając pliku, którego ścieżkę dostępu zawiera parametr `name`, lub używając informacji zawartych w obiekcie klasy `File`.

API java.io.FileOutputStream 1.0

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

  tworzy nowy strumień wyjściowy pliku określonego za pomocą łańcucha `name` 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 buforowany strumień wejściowy typu `BufferedInputStream`. Wejściowy 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 buforowany strumień wyjściowy typu `BufferedOutputStream`. 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)`
- `PushbackInputStream(InputStream in, int size)`

  tworzą strumień wejściowy 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.

2.1.4. Tekstowe operacje wejścia-wyjścia

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, a odczyt i zapis danych binarnych omówimy w podrozdziale 2.2.

Zapisując łańcuchy znakowe, musimy uwzględnić sposób *kodowania znaków*. W przypadku kodowania UTF-16, którego Java używa wewnętrznie, łańcuch `"José"` zostanie zakodowany

jako 00 4A 00 6F 00 73 00 E9 (w notacji szesnastkowej). Jednakże wiele programów oczekuje, że pliki tekstowe będą zapisane przy wykorzystaniu innych sposobów kodowania. W przypadku kodowania UTF-8, obecnie najczęściej stosowanego w internecie, ten sam łańcuch znaków został zapisany jako następująca sekwencja bajtów: 4A 6F 73 C3 A9, czyli bez bajtów 00 dla pierwszych trzech znaków oraz bez używania dwóch bajtów do zapisu znaku é.

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

Od udostępnienia Javy 18 klasy te domyślnie używają kodowania UTF-8. Przed Javą 18 lub w przypadkach gdy konieczne było zapewnienie obsługi jakiegoś archaicznego sposobu kodowania, sposób kodowania można było określać w konstruktorze, jak w poniższym przykładzie:

```
var in = new InputStreamReader(new FileInputStream("data.txt"),
StandardCharsets.UTF_8);
```

Więcej informacji na temat kodowania znaków znajdziesz w punkcie 2.1.8.

Klasy `Reader` i `Writer` dysponują jedynie podstawowymi metodami do odczytu i zapisu pojedynczych znaków. Podobnie jak w przypadku strumieni, do odczytu łańcuchów i liczb używane są ich klasy pochodne.

2.1.5. Wczytywanie tekstu

Najprostszym sposobem wczytywania i przetwarzania tekstu o dowolnej postaci jest wykorzystanie klasy `Scanner`, której bardzo często używaliśmy w pierwszym tomie tej książki. Obiekt klasy `Scanner` możemy utworzyć, jeśli dysponujemy dowolnym strumieniem wejściowym.

Ewentualnie krótki plik tekstowy można wczytać w formie łańcucha znaków w następujący sposób:

```
String content = Files.readString(path, charset);
```

Gdybyśmy natomiast chcieli wczytać plik jako sekwencję wierszy, należałoby to zrobić przy użyciu następującego wywołania:

```
List<String> lines = Files.readAllLines(path, charset);
```

W przypadku obsługi dużego pliku jego poszczególne wiersze można przetwarzać jako daną typu `Stream<String>`:

```
try (Stream<String> lines = Files.lines(path, charset)
{
 ...
}
```

Jeśli w czasie wczytywania wierszy tekstu do strumienia wystąpi wyjątek `IOException`, to zostanie on opakowany w wyjątku typu UncheckedIOException, który zostanie zgłoszony przez operację na strumieniu. Takie działanie jest konieczne, gdyż operacje na strumieniach nie nie mogą zgłaszać wyjątków sprawdzanych (ang. *checked exceptions*).

Aby wczytać wiersze tekstu z dowolnego strumienia wejściowego, można użyć poniższego fragmentu kodu:

```
InputStream inputStream = ...;
try (var reader = new BufferedReader(new InputStreamReader(inputStream, charset)))
{
   Stream<String> lines = reader.lines();
   ...
}
```

Istnieje także możliwość zastosowania skanera do odczytywania tak zwanych *tokenów* — czyli fragmentów oddzielonych od siebie określonym separatorem. Jako separatora można także użyć dowolnego wyrażenia regularnego. Na przykład poniżej można zobaczyć, jak skonfigurować skaner, by używał jako separatorów wyłącznie liter nienależących do Unicode:

```
Scanner in = ...;
in.useDelimiter("\\PL+");
```

W tym przypadku separatorem mogą być dowolne litery nienależące do Unicode. Taki skaner będzie akceptował tokeny składające się wyłącznie z liter należących do Unicode.

Kolejny token można pobrać, wywołując metodę `next`:

```
while (in.hasNext())
{
   String word = in.next();
   ...
}
```

Można także pobrać strumień zawierający wszystkie tokeny:

```
Stream<String> words = in.tokens();
```

W zastanym kodzie możesz spotkać się ze sposobem wczytywania tekstów polegającym na wielokrotnym wywoływaniu metody `readLine` obiektu klasy `BufferedReader`. Obecnie nie ma już dobrego uzasadnienia, by tak robić.

2.1.6. Zapisywanie tekstu

W celu zapisania tekstu korzystamy z klasy `PrintWriter`. Dysponuje ona metodami umożliwiającymi zapis łańcuchów i liczb w formacie tekstowym. Aby zapisać tekst w pliku, należy utworzyć strumień `PrintStream`, podając nazwę pliku i sposób kodowania:

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

Co ciekawe, nie można utworzyć obiektu `PrintWriter` na podstawie obiektu `Path`.

Jeśli nie używasz Javy 18, to w wywołaniu konstruktora klasy PrintWriter musisz podawać drugi argument określający sposób kodowania, jak w poniższym przykładzie:

```
var out = PrintWriter("employee.txt", StandardCharsets.UTF_8);
```

W przeciwnym razie zostanie zastosowany systemowy sposób kodowania, co sprawi, że program będzie zależny od konfiguracji systemu operacyjnego, w którym będzie on uruchamiany.

Do zapisywania danych za pomocą obiektu klasy `PrintWriter` używamy tych samych metod `print`, `println` i `printf`, 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 `Print`
`↪Writer` zawsze są buforowane). Domyślnie automatyczne opróżnianie jest *wyłączone*. Automatyczne opróżnianie możemy włączać i wyłączać przy użyciu konstruktora `Print`
`↪Writer(Writer writer, boolean autoFlush)`:

```
var out = new PrintWriter(
   new OutputStreamWriter(
      new FileOutputStream("employee.txt"),
   true); // Automatyczne opróżnianie
```

Metody `print` nie zgłaszają wyjątków. Aby sprawdzić, czy ze strumieniem wyjścia 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 (wówczas Unicode był jeszcze kodem 16-bitowym). 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 z istniejącym kodem `System.in`, `System.out` i `System.err` wciąż są strumieniami wejścia-wyjścia, 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(OutputStream out)`
- `PrintWriter(OutputStream out, boolean autoFlush)`
- `PrintWriter(Writer writer)`
- `PrintWriter(Writer writer, boolean autoFlush)`

  tworzy nowy obiekt klasy `PrintWriter` zapisujący dane do przekazanego obiektu. W przypadku przekazania wartości `true` jako drugiego parametru `autoFlush` wywołania metod `println` i `printf` powodują opróżnienie i zapis bufora.

- `PrintWriter(String filename, String encoding)`
- `PrintWriter(File file, String encoding)`

  tworzy nowy obiekt klasy `PrintWriter` zapisujący dane do podanego pliku z wykorzystaniem określonego sposobu kodowania.

- `void print(Object obj)`

  zapisuje łańcuch zwracany przez metodę `toString` danego obiektu.

- `void print(String s)`

  zapisuje łańcuch Unicode.

- `void println(String s)`

  zapisuje łańcuch zakończony znakiem końca wiersza. Jeżeli automatyczne opróżnianie jest włączone, opróżnia bufor strumienia.

- `void print(char[] s)`

  zapisuje tablicę znaków Unicode.

- `void print(char c)`

  zapisuje znak Unicode.

- `void print(int i)`

- `void print(long l)`
- `void print(float f)`
- `void print(double d)`
- `void print(boolean b)`

  zapisuje podaną wartość w formacie tekstowym.

- `void printf(String format, Object... args)`

  zapisuje podane wartości według łańcucha formatującego. Specyfikację łańcucha formatującego znajdziesz w rozdziale 3. książki *Java. 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`.

2.1.7. 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|35500.0|1989-10-01
Carl Cracker|75000.0|1987-12-15
Tony Tester|38000.0|1990-03-15
```

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:

```
public static void writeEmployee(PrintWriter out, Employee e)
{
   out.println(e.getName() + "|" + e.getSalary() + "|" + e.getHireDay());
}
```

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 static Employee readEmployee(Scanner in)
{
   String line = in.nextLine();
   String[] tokens = line.split("\\|");
```

```
      String name = tokens[0];
      double salary = Double.parseDouble(tokens[1]);
      LocalDate hireDate = LocalDate.parse(tokens[2]);
      int year = hireDate.getYear();
      int month = hireDate.getMonthValue();
      int day = hireDate.getDayOfMonth();
      return new Employee(name, salary, year, month, day);
   }
```

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 2.1. Metoda statyczna

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

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

```
Employee[] readData(Scanner 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
var employees = new Employee[n];
for (int i = 0; i < n; i++)
{
   employees[i] = readEmployee(in);
}
```

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

Listing 2.1. textfile/TextFileTest.java

```
package textFile;

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

/**
 * @version 1.16 2023-08-16
 * @author Cay Horstmann
 */
public class TextFileTest
{
   public static void main(String[] args) throws IOException
   {
      var 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);

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

      // Wczytuje wszystkie rekordy do nowej tablicy
      try (var in = new Scanner(new FileInputStream("employee.dat")))
      {
         Employee[] newStaff = readData(in);

         // Wyświetla wszystkie wczytane rekordy
         for (Employee e : newStaff)
            System.out.println(e);
      }
   }

   /**
    * Zapisuje dane wszystkich obiektów klasy Employee umieszczonych w tablicy
    * @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)
         writeEmployee(out, e);
   }

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

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

   /**
    * Zapisuje dane obiektu klasy Employee do obiektu klasy PrintWriter
    * @param out obiekt klasy PrintWriter
    * @param e obiekt klasy Employee
    */
```

```
   public static void writeEmployee(PrintWriter out, Employee e)
   {
      out.println(e.getName() + "|" + e.getSalary() + "|" + e.getHireDay());
   }

   /**
    * Wczytuje dane obiektu klasy Employee
    * @param in obiekt Scanner
    * @return the obiekt Employee
    */
   public static Employee readEmployee(Scanner in)
   {
      String line = in.nextLine();
      String[] tokens = line.split("\\|");
      String name = tokens[0];
      double salary = Double.parseDouble(tokens[1]);
      LocalDate hireDate = LocalDate.parse(tokens[2]);
      int year = hireDate.getYear();
      int month = hireDate.getMonthValue();
      int day = hireDate.getDayOfMonth();
      return new Employee(name, salary, year, month, day);
   }
}
```

2.1.8. Zbiory znaków

Każdy znak, nazywany także „punktem kodowym", jest skojarzony z 21-bitową liczbą całkowitą. Istnieje kilka różnych *sposobów kodowania znaków*, czyli metod zapisu tych 21-bitowych wartości w formie bajtów.

Najczęściej stosowanym sposobem kodowania jest UTF-8, który każdy punkt kodowy Unicode zapisuje jako sekwencję o długości od jednego do czterech bajtów (patrz tabela 2.1). UTF-8 ma tę zaletę, że znaki należące do tradycyjnego zbioru znaków ASCII, czyli wszystkie znaki języka angielskiego, są zapisywane przy użyciu jednego bajta.

Tabela 2.1. Kodowanie UTF-8

Zakres znaku	Sposób kodowania
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	$11110a_{20}a_{19}a_{18}\ 10a_{17}a_{16}a_{15}a_{14}a_{13}a_{12}\ 10a_{11}a_{10}a_9a_8a_7a_6\ 10a_5a_4a_3a_2a_1a_0$

Kolejnym popularnym sposobem kodowania jest UTF-16, w którym punkty kodowe Unicode są zapisywane przy wykorzystaniu jednej lub dwóch wartości 16-bitowych (patrz tabela 2.2). To właśnie ten sposób kodowania jest używany do zapisu łańcuchów znaków w Javie. W rzeczywistości istnieją dwa rodzaje kodowania UTF-16, określane — odpowiednio —

jako *big-endian* oraz *little-endian*. W ramach przykładu przeanalizujemy 16-bitową wartość `0x2122`. W przypadku wykorzystania formatu *big-endian* jako pierwszy jest zapisywany bardziej znaczący bajt tej wartości: `0x21`, a za nim `0x22`. Z kolei w przypadku formatu *little-endian* wartość ta zostanie zapisana jako: `0x22 0x21`. Aby możliwe było określenie, który z tych dwóch rodzajów kodowania UTF-16 jest używany, na początku pliku umieszcza się tak zwany znak BOM (ang. *byte order mark*[1]) — 16-bitową wartość `0xFEFF`. Obiekt odczytujący może użyć tej wartości do określenia kolejności zapisu bajtów w pliku bądź też ją zignorować.

Tabela 2.2. Kodowanie UTF-16

Zakres znaku	Sposób kodowania
`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`	$110110b_{19}b_{18}b_{17}b_{16}a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}\ 110111a_9a_8\ a_7a_6a_5a_4a_3a_2a_1a_0$ gdzie $b_{19}b_{18}b_{17}b_{16} = a_{20}a_{19}a_{18}a_{17}a_{16} - 1$

Niektóre programy, takie jak Microsoft Notepad, dodają znacznik kolejność bajtów na początku plików zapisywanych z użyciem kodowania UTF-8. Oczywiście jest to zupełnie niepotrzebne, gdyż w przypadku kodowania UTF-8 nie ma żadnych problemów z określaniem kolejności bajtów. Niemniej jednak standard Unicode pozwala na takie stosowanie znacznika BOM, co więcej, sugeruje nawet, że jest to dobre rozwiązanie, gdyż nie pozostawia żadnych wątpliwości odnośnie do wykorzystywanego sposobu kodowania. Znacznik BOM należy usuwać podczas odczytywania pliku zapisanego z użyciem kodowania UTF-8. Niestety Java tego nie robi, a zgłoszenia błędów dotyczące tego problemu są zamykane i oznaczane jako „will not fix", czyli problem, który nie będzie rozwiązywany. A zatem najlepszym wyjściem jest po prostu usuwanie wszelkich sekwencji `\uFEFF` odnalezionych na początku danych wejściowych.

Oprócz UTF istnieją także kody częściowe, które obejmują zakresy znaków przydatne dla konkretnych populacji użytkowników. Na przykład ISO 8859-1 to jednobajtowy kod zawierający znaki z akcentami stosowane w krajach Europy Zachodniej. Z kolei *Shift-JIS* to kod o zmiennej długości zawierający znaki języka japońskiego. Wiele takich kodowań wciąż jest w powszechnym użyciu.

Nie ma możliwości niezawodnego automatycznego wykrywania sposobu kodowania zastosowanego w strumieniu bajtów. Czasami kodowanie to jest określane w innym miejscu. Na przykład w przypadku odczytywania strony WWW należy sprawdzić nagłówek `Content-Type`.

Klasa `StandardCharsets` definiuje statyczne zmienne typu `Charset`, reprezentujące wszystkie kodowania znaków, które muszą być obsługiwane przez każdą wirtualną maszynę Javy. Są to:

```
StandardCharsets.UTF_8
StandardCharsets.UTF_16
```

[1] „Znacznik kolejności bajtów" — *przyp. tłum.*

```
StandardCharsets.UTF_16BE
StandardCharsets.UTF_16LE
StandardCharsets.ISO_8859_1
StandardCharsets.US_ASCII
```

Aby pobrać obiekt `Charset` dla innego kodowania, należy posłużyć się statyczną metodą `forName`:

```
Charset shiftJIS = Charset.forName("Shift-JIS");
```

Statyczna metoda `Charset.availableCharsets` zwraca wszystkie dostępne kodowania znaków w formie mapy kojarzącej ich kanoniczne nazwy z obiektami `Charset`.

Obiektów `Charset` należy używać podczas odczytywania i zapisywania tekstów. Na przykład tablicę znaków można przekształcić na łańcuch znaków w następujący sposób:

```
var str = new String(bytes, StandardCharsets.ISO_8859_1);
```

Wiele metod pozwala określać sposoby kodowania przy użyciu łańcucha znaków, a nie tylko obiektu `Charset`. W przypadku standardowych sposobów kodowania warto jednak używać stałych klasy `StandardCharsets`, aby wszelkie błędy w zapisie były wykrywane na etapie kompilacji.

Aż do Javy 17 wywołanie metody `Charset.forName` z argumentem `"default"` zwracało obiekt `StandardCharsets.US_ASCII`. Takie działanie już od bardzo długiego czasu było złym pomysłem. Obecnie takie wywołanie zgłasza wyjątek `UnsupportedCharset`
`↪Exception`.

Jeśli nie podasz kodowania znaków, a operacja musi konwertować bajty i łańcuchy znaków, to zostanie wybrane domyślne kodowanie. Dla wielu metod w pakiecie `java.nio.file` domyślnym kodowaniem jest UTF-8. Ale inne metody, gdy nie określono żadnego sposobu kodowania znaków, używają *domyślnego zestawu znaków*. Do takich metod należą:

- Metody czytelników (`reader`) i pisarzy (`writer`) z pakietu `java.io`.
- Konstruktor `String(byte[])` i metoda `getBytes` klasy `String`.
- Metody klas `Scanner` i `Formatter` z pakietu `java.util`.
- Metody klas `URLEncoder` i `URLDecoder` z pakietu `java.net`.

W języku Java 18 domyślnym zbiorem znaków jest UTF-8. W starszych wersjach języka było to *kodowanie rodzime* (ang. *native encoding*), określane przez system operacyjny, w którym była uruchamiana wirtualna maszyna Javy. W systemach Linux i MacOS jest to zazwyczaj (choć nie zawsze) kodowanie UTF-8. Jednak w przypadku systemu Windows tym kodowaniem rodzimym jest, w przeważającej większości przypadków, archaiczne kodowanie takie jak Windws-1252.

Przed udostępnieniem języka Java 18 implementacja Javy firmy Oracle udostępniała właściwość systemową o nazwie `file.encoding`, pozwalającą przesłonić domyślne ustawienia platformy systemowej. Nie była to właściwość oficjalna, choć stosowano ją powszechnie.

W Javie 18 ta właściwość systemowa została ustandaryzowana. Podczas stosowania jej z poziomu wiersza poleceń można jej przypisywać następujące wartości:

- `COMPAT` (dostępna w Javie 18) — ustawia domyślny zestaw znaków na rodzimy sposób kodowania, a we właściwości `file.encoding` zapisuje jego nazwę.
- `UTF-8` — ustawia domyślny zestaw znaków na UTF-8.
- Inne kodowanie znaków, takie jak `US-ASCII` — w przypadku nieokreślonych starych nazw zestawów znaków może ustawiać domyślny zestaw znaków na zestaw o podanej nazwie.

Właściwość `file.encoding` musi być ustawiana z poziomu wiersza poleceń. Określanie jej wartości za pośrednictwem metody `System.setProperty` nie powoduje zmiany domyślnego zestawu znaków.

Faktycznie używany domyślny zestaw znaków jest zwracany przez statyczną metodę `Charset.defaultCharset`. W Javie 17 nazwa rodzimego sposobu kodowania jest zapisana we właściwości systemowej `native.encoding`, niezależnie od tego, czy kodowanie to jest używane, czy nie.

Kiedy uruchamiasz program napisany w Javie z poziomu terminala Windows, kodowanie, wykorzystywane przez strumienie `System.out` i `System.err`, jest określane na podstawie „strony kodowej" (ang. *code page*) terminala, która może się różnić od domyślnego zestawu znaków.

Najlepszym rozwiązaniem jest ustawienie strony kodowej UTF-8, co można zrobić przy użyciu następującego polecenia:

```
Chcp 65001
```

Alternatywnym rozwiązaniem dostępnym w języku Java 18 jest podanie kodowania `UTF-8` we właściwościach systemowych `stdout.encoding` i `stderr.encoding`. Jeśli strona kodowa terminala będzie inna niż UTF-8, to wyświetlane w nim komunikaty tekstowe mogą być nieczytelne. Jednak przekierowania do pliku będą działały prawidłowo.

Nazwy tych dwóch właściwości systemowych mogą być nieco mylące. Kodowanie nie odnosi się do `stdout` i `stderr`, które są strumieniami bajtów, lecz do obiektów `System.out` i `System.err`. Oba są instancjami klasy `PrintStream`, której metody print i println do przekształcania znaków na bajty muszą korzystać z jakiegoś sposobu kodowania.

Aby w prosty i szybki sposób wyświetlić bieżące ustawienia związane z domyślnym zbiorem znaków i różnymi sposobami kodowania, wykonaj poniższe polecenie:

```
java -XshoSettings:properties
```

W wynikach poszukaj właściwości, których nazwy kończą się na `.encoding`.

Konsola systemowa, której można używać do odczytywania łańcuchów znaków i haseł z terminala, stosuje zestaw znaków odpowiadający ustawieniom terminala. Można go odczytać przy użyciu wywołania `System.console().charset()`. Jedynym sposobem pozwalającym zmienić ten zestaw znaków jest zmiana strony kodowej terminala.

Obiekt zwracany przez wywołanie `new Scanner(System.in)` używa domyślnego zestawu znaków, który nie musi być tym samym, którego używa konsola. Problem ten występuje podczas uruchamiania programów Javy z poziomu terminala systemu Windows korzystającego z archaicznych stron kodowych. Najlepszym rozwiązaniem jest zmiana strony kodowej terminala na UTF-8. Jeśli jednak nie będzie to możliwe, można utworzyć obiekt Scanner w następujący sposób:

```
C[onsole console = System.console();
if (console != null) {
   var in = new Scanner(console.reader()):
   ...
}
```

Zabezpieczenie się przed brakiem konsoli systemowej jest konieczne — taka sytuacja występuje w przypadku uruchamiania programu w niektórych środowiskach programistycznych.

2.1.9. Wczytywanie znaków

Podczas wczytywania pliku strukturalnego, zapisanego na przykład w formacie JSON lub XML, będziesz używać gotowego parsera rozumiejącego szczegóły konkretnego formatu. Takie parsery zazwyczaj odczytują zawartość pliku znak po znaku.

W sporadycznych przypadkach kiedy trzeba napisać taki parser samodzielnie, aby poprawić wydajność jego działania, zastosuj klasę `BufferedReader`. Wywołuj metodę `read`, która zwraca wartość typu `char` lub `-1` (w przypadku dotarcia do końca strumienia). Czytelnik `BufferedReader` konwertuje kodowanie strumienia wejściowego na UTF-16. Konieczne jest zatem odpowiednie przetworzenie tego kodowania:

```
int ch = reader.read();
if (ch != -1)
{
   int codePoint;
   if (Character.isHighSurrogate((char) ch))
   {
      int ch2 = reader.read();
      if (Character.isLowSurrogate((char) ch2))
         codePoint = Character.toCodePoint(ch, ch2);
      else
         throw new MalformedInputException(2);
      }
   else
      codePoint = ch;
}
```

Klasa `Character` udostępnia metody zwracające informacje o tym, czy przekazany punkt kodowy ma określone cechy. Na przykład wywołanie:

```
Character.isLetter(codePoint)
```

zwraca `true`, jeśli `codePoint` jest literą jakiegoś języka. Dostępnych jest także kilka innych metod klasyfikujących, takich jak:

```
isUpperCase
isLowerCase
isDigit
isSpaceChar
isEmoji
```

Wszystkie te metody stosują reguły standardu Unicode. Natomiast poniższe metody korzystają z reguł określanych przez Javę:

```
isJavaIdentifierStart
isJavaIdentifierPart
isWhiteSpace
```

Po przeanalizowaniu punktów kodowych często trzeba je zapisywać w łańcuchach znaków, co wymaga wcześniejszego ponownego skonwertowania ich do UTF-16. Metoda `appendCodePoint` klasy `StringBuilder` przekształca punkt kodowy na jedną lub dwie wartości `char`, które są dołączane do obiektu.

2.2. Odczyt i zapis danych binarnych

Format tekstowy jest wygodny w przypadku testowania i debugowania, gdyż jest zrozumiały dla ludzi; niemniej jednak jeśli chodzi o przesyłanie danych, nie jest on tak wydajny jak format binarny. W tym podrozdziale powiemy, jak można wczytywać i zapisywać dane binarne.

2.2.1. Interfejsy DataInput i DataOutput

Interfejs DataOutput definiuje następujące metody służące do zapisywania liczb, znaków, wartości logicznych oraz łańcuchów znaków w formacie binarnym:

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

Na przykład metoda `writeInt` zawsze zapisuje liczbę całkowitą jako 4-bajtową wartość binarną, niezależnie od liczby cyfr, a metoda `writeDouble` zawsze zapisuje wartość typu `double` jako 8-bajtową wartość binarną. Wyniki zwracane przez te metody nie nadają się do odczytu przez ludzi, jednak ilość miejsca zajmowanego przez tak zapisane dane zawsze będzie taka sama dla wartości konkretnego typu, a ich odczytywanie będzie szybsze niż analiza tekstu.

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 2.1), znaki łańcucha są najpierw reprezentowane w kodzie UTF-16 (patrz tabela 2.2), 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.

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 firmy Intel. Może to powodować problemy z przenoszeniem nawet najprostszych plików danych pomiędzy różnymi platformami, gdyż programy w C lub C++ zapisują dane w sposób typowy dla danego procesora. W języku Java zawsze stosowany jest pierwszy sposób, niezależnie od procesora. Dzięki temu pliki danych programów w języku Java są niezależne od platformy.

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`.

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`:

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

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

```
var 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.

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

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

- `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.

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.

2.2.2. 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).

```
var in = new RandomAccesFile("employee.dat", "r");
var 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. Metody `seek` można używać do zmiany położenia wskaźnika poprzez określenie numeru bajta, na który ma on wskazywać. 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.

Przeanalizujmy 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);
var 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
{
   var b = new StringBuilder(size);
   int i = 0;
   var done = false;
   while (!done && i < size)
   {
      char ch = in.readChar();
      i++;
      if (ch == 0) done = true;
      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.

```
DataIO.writeFixedString(e.getName(), Employee.NAME_SIZE, out);
out.writeDouble(e.getSalary());
LocalDate hireDay = e.getHireDay();
out.writeInt(hireDay.getYear());
out.writeInt(hireDay.getMonthValue());
out.writeInt(hireDay.getDayOfMonth());
```

Odczyt rekordu jest równie prosty.

```
String name = DataIO.readFixedString(NAME_SIZE, in);
double salary = in.readDouble();
int y = in.readInt();
int m = in.readInt();
int d = in.readInt();
```

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 2.2 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 2.2. randomAccess/RandomAccessTest.java

```
package randomAccess;

import java.io.*;
import java.time.*;

/**
 * @version 1.14 2023-10-05
 * @author Cay Horstmann
 */
public class RandomAccessTest
{
   public static final int NAME_SIZE = 40;
   public static final int RECORD_SIZE = 2 * NAME_SIZE + 8 + 4 + 4 + 4;

   public static void main(String[] args) throws IOException
   {
      var 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 (var out = new DataOutputStream(new FileOutputStream("employee.dat")))
      {
         // Zapisuje rekordy wszystkich pracowników w pliku employee.dat
         for (Employee e : staff)
            writeData(out, e);
      }

      try (var in = new RandomAccessFile("employee.dat", "r"))
      {
         // Wczytuje wszystkie rekordy do nowej tablicy

         // Oblicza rozmiar tablicy
         int n = (int) (in.length() / RECORD_SIZE);
         var newStaff = new Employee[n];

         // Wczytuje rekordy pracowników w odwrotnej kolejności
         for (int i = 0; i < n; i++)
         {
            in.seek((n - 1 - i) * RECORD_SIZE);
            newStaff[i] = readData(in);
         }

         // Wyświetla wczytane rekordy
         for (Employee e : newStaff)
```

```
            System.out.println(e);
         }
      }

      /**
       * Zapisuje dane pracownika do wynikowego strumienia danych
       * @param out obiekt typu DataOutput
       * @param e   pracownik
       */
      public static void writeData(DataOutput out, Employee e) throws IOException
      {
         DataIO.writeFixedString(e.getName(), NAME_SIZE, out);
         out.writeDouble(e.getSalary());

         LocalDate hireDay = e.getHireDay();
         out.writeInt(hireDay.getYear());
         out.writeInt(hireDay.getMonthValue());
         out.writeInt(hireDay.getDayOfMonth());
      }

      /**
       * Wczytuje dane pracownika
       * @param in obiekt typu DataInput
       * @return pracownik
       */
      public static Employee readData(DataInput in) throws IOException
      {
         String name = DataIO.readFixedString(NAME_SIZE, in);
         double salary = in.readDouble();
         int y = in.readInt();
         int m = in.readInt();
         int d = in.readInt();
         return new Employee(name, salary, y, m, d);
      }
   }
```

API java.io.RandomAccessFile 1.0

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

  otwiera plik w podanym trybie. Tryb `"r"` oznacza tryb samego odczytu, `"rw"` to tryb odczytu i zapisu, `"rws"` to tryb odczytu i zapisu danych wraz z synchronicznym zapisem danych i metadanych dla każdej aktualizacji, `"rwd"` to tryb 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.

2.2.3. Archiwa 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 `getNextEntry` zwraca obiekt typu `ZipEntry` opisujący pozycję archiwum. Strumień należy odczytać do końca, który jest jednocześnie końcem bieżącej pozycji archiwum. Następnie należy wywołać metodę `closeEntry`, by przejść do następnej pozycji. Nie należy zamykać strumienia `ZipInputStream`, zanim zostanie odczytana ostatnia pozycja. Oto typowa sekwencja wywołań służąca do odczytu zawartości pliku ZIP:

```
var zin = new ZipInputStream(new FileInputStream(zipname));
boolean done = false;
while (!done)
{
   ZipEntry entry = zin.getNextEntry();
   if (entry == null) done = true;
   else
   {
      wczytaj zawartość zin;
      zin.closeEntry();
   }
}
zin.close();
```

Aby zapisać dane do pliku ZIP, używamy strumienia `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 wyjściowego strumienia ZIP. Po zakończeniu zapisu pliku wywołujemy metodę `closeEntry`. Wymienione operacje powtarzamy dla wszystkich plików, które chcemy skompresować w archiwum. Oto schemat kodu:

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

Pliki JAR (omówione w rozdziale 4. książki *Java. Podstawy*) są po prostu plikami ZIP, zawierającymi specjalny rodzaj pliku, tzw. manifest. Do wczytania i zapisania manifestu używamy klas `JarInputStream` i `JarOutputStream`.

Strumienie wejściowe 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.

W punkcie 2.4.8 zobaczymy, jak korzystać z archiwów ZIP bez zastosowania specjalnego API, używając klasy `FileSystem`.

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`. Zgłasza wyjątek `IllegalArgumentException`, jeżeli podany stopień jest nieprawidłowy.

- `void setMethod(int method)`

  określa domyślną metodę kompresji dla danego `ZipOutputStream` dla wszystkich pozycji archiwum, dla których metoda kompresji nie została określona. Parametr może przyjować wartości `DEFLATED` lub `STORED`.

API java.util.zip.ZipEntry 1.1

- `ZipEntry(String name)`

  tworzy pozycję archiwum o podanej nazwie.

- `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)`

  ustawia metodę kompresji danego elementu, `DEFLATED` lub `STORED`.

- `void setSize(long rozmiar)`

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

- `void setCrc(long crc)`

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

API java.util.zip.ZipFile 1.1

- `ZipFile(String name)`
- `ZipFile(File file)`
- `ZipFile(String name, Charset charset)` **1.7**
- `ZipFile(File file, Charset charset)` **1.7**

  tworzy obiekt typu `ZipFile`, otwarty do odczytu, na podstawie podanego łańcucha znaków lub obiektu typu `File`. W przypadku przekazania obiektu `Charset` określa on sposób kodowania nazw poszczególnych elementów pliku ZIP. Jeśli argument ten nie zostanie przekazany, domyślnie będzie używane kodowanie UTF-8.

- `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.

- `InputStream getInputStream(ZipEntry ze)`

  zwraca obiekt `InputStream` dla podanego elementu.

- `String getName()`

  zwraca ścieżkę dostępu do pliku ZIP.

2.3. 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 wyjściowego dowolnego obiektu i umożliwia jego późniejsze wczytanie (w dalszej części tego rozdziału wyjaśnimy, skąd wziął się termin „serializacja”).

2.3.1. Zapisywanie i wczytywanie obiektów serializowalnych

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

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

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

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

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

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

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

```
var e1 = (Employee)in.readObject();
var e2 = (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. Podstawy*. Aby jednak móc klonować obiekty, musimy przesłonić metodę `clone` klasy `Object`. Aby móc serializować, nie należy robić nic poza dopisaniem powyższych słów.

Za pomocą metod `writeObject/readObject` możemy zapisywać i odczytywać wyłącznie *obiekty*. W przypadku wartości należących do typów podstawowych języka Java należy stosować metody takie jak `writeInt/readInt` czy `writeDouble/readDouble`. (Klasy strumieni wejścia-wyjścia służących do zapisu i odczytu obiektów implementują interfejsy, odpowiednio, `DataOutput` i `DataInput`).

Działanie strumienia `ObjectOutputStream` polega na przeglądaniu wszystkich pól obiektu i zapisie ich wartości. Na przykład podczas zapisu obiektu klasy `Employee` w strumieniu wyjściowym zapisane zostają wartości pól `name`, `hireDay` i `salary`.

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

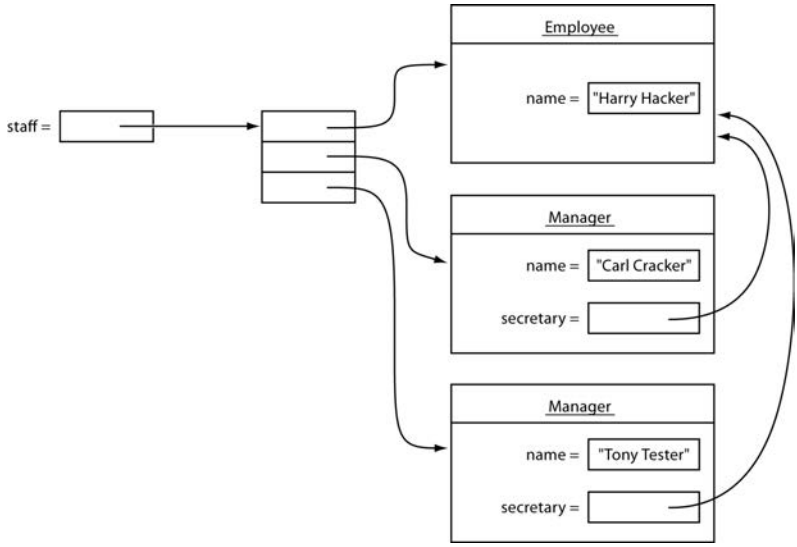
Aby zilustrować ten problem, zmodyfikujemy trochę klasę `Manager`. Załóżmy, że każdy menedżer ma asystenta:

```
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 2.5 i w poniższym kodzie:

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

Rysunek 2.5.
Dwóch menedżerów może mieć wspólnego asystenta

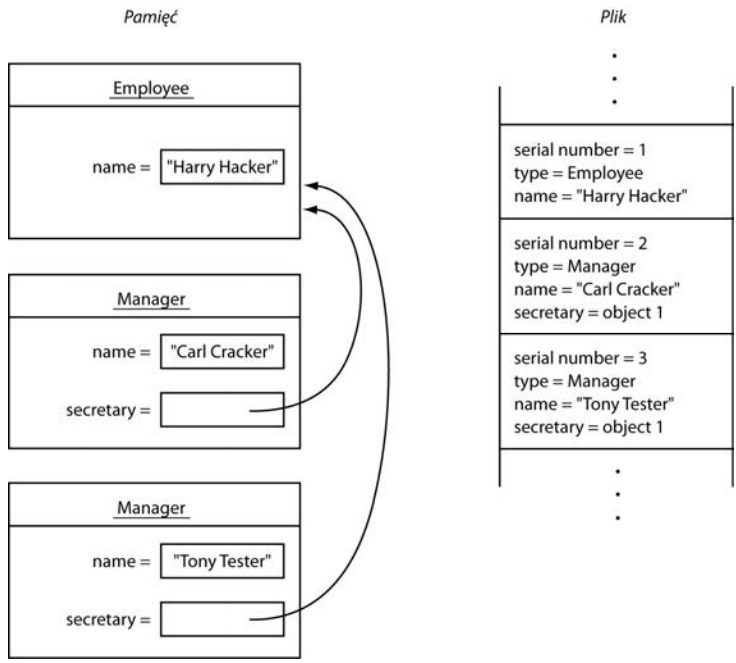


Teraz załóżmy, że zapisujemy dane pracowników na dysk. Oczywiście nie możemy zapisać i przywrócić adresów obiektów asystentów w pamięci, ponieważ po ponownym załadowaniu obiekt asystenta najprawdopodobniej znajdzie się w zupełnie innym miejscu pamięci.

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

1. Wszystkim napotkanym referencjom do obiektów nadawane są numery seryjne (patrz rysunek 2.6).

Rysunek 2.6.
Przykład serializacji obiektów



2. Jeśli referencja do obiektu została napotkana po raz pierwszy, obiekt zostaje zapisany w strumieniu wyjściowym.
3. 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.

1. Gdy obiekt pojawia się w strumieniu wejściowym po raz pierwszy, Java tworzy go, inicjuje danymi ze strumienia i zapamiętuje związek pomiędzy numerem i referencją do obiektu.
2. 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 jednego komputera do drugiego.

Jeśli klasa bazowa klasy zapewniającej możliwość serializacji sama takiej możliwości nie daje, musi udostępniać konstruktor bezargumentowy. Przeanalizujmy następujący przykład:

```
class Persona // Nie obsługuje serializacji
class Employee extends Person implements Serializable
```

Podczas deserializacji obiektu `Employee` jego pola składowe są odczytywane z obiektowego strumienia wejściowego, jednak pola składowe obiektu Person są ustawiane przez konstruktor `Person`.

Listing 2.3 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 2.3. serial/ObjectStreamTest.java

```
package serial;

import java.io.*;

/**
 * @version 1.12 2021-09-10
 * @author Cay Horstmann
 */
class ObjectStreamTest
```

```
{
   public static void main(String[] args) throws IOException, ClassNotFoundException
   {
      var harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      var carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
      carl.setSecretary(harry);
      var tony = new Manager("Tony Tester", 40000, 1990, 3, 15);
      tony.setSecretary(harry);

      var staff = new Employee[3];

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

      // Zapisuje rekordy wszystkich pracowników w pliku employee.dat
      try (var out = new ObjectOutputStream(new FileOutputStream("employee.ser")))
      {
         out.writeObject(staff);
      }

      try (var in = new ObjectInputStream(new FileInputStream("employee.ser")))
      {
         // Wczytuje wszystkie rekordy do nowej tablicy

         var newStaff = (Employee[]) in.readObject();

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

         // Wyświetla wszystkie wczytane rekordy
         for (Employee e : newStaff)
            System.out.println(e);
      }
   }
}
```

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 klas nadrzędnych.

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 klas nadrzędnych. Przeprowadza deserializację, pozwalając na przyporządkowanie obiektów referencjom.

2.3.2. 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, gdyż da nam wgląd w proces serializacji obiektów. 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 zapisana w dwóch bajtach),
- znaki w formacie „zmodyfikowanego UTF-8”.

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

```
74 00 05 Harry
```

Wraz z obiektem musi zostać zapisana także jego klasa. Deskryptor klasy ma następujący format:

- 72,
- długość nazwy klasy (2 bajty),
- nazwa klasy,
- „odcisk” klasy (8 bajtów),
- flaga (1 bajt),
- liczba deskryptorów pól składowych (2 bajty),

- deskryptory pól składowych,
- 78 (znacznik końca),
- deskryptor klasy bazowej (70, jeśli go nie ma).

Java tworzy wspomniany „odcisk palca" (ang. *fingerprint*) klasy, pobierając opisy klasy, klasy bazowej, interfejsów, typów pól danych oraz sygnatury metod w postaci kanonicznej, a następnie stosuje do nich algorytm SHA w wersji 1 (SHA-1, *Secure Hash Algorithm*).

SHA-1 to szybki algorytm tworzący odciski dla bloków danych. Niezależnie od rozmiaru oryginalnych danych odciskiem jest zawsze pakiet 20 bajtów. Jest on tworzony za pomocą sprytnej sekwencji operacji binarnych, dzięki którym możemy mieć niemal stuprocentową pewność, że jeżeli zachowana informacja zmieni się, zmianie ulegnie również jej odcisk palca. Mechanizm serializacji używa jako odcisku palca jedynie pierwszych 8 bajtów kodu SHA-1. Pomimo to prawdopodobieństwo tego, że zmiana pól lub metod obiektu spowoduje także zmianę odcisku palca, jest bardzo wysokie.

W chwili odczytu obiektu jego odcisk zostaje porównany z aktualnym odciskiem klasy. Jeśli różni się, oznacza to, że definicja klasy uległa zmianie po zapisaniu obiektu. Oczywiście w praktyce klasy ulegają zmianie i może się okazać, że program będzie musiał wczytać starsze wersje obiektów. Zagadnienie to omówię w punkcie 2.3.7.

Bajt `flag` składa się z trzybitowych masek, zdefiniowanych w interfejsie `java.io.Object` ↪`StreamConstants` i przedstawionych w tabeli 2.3.

Tabela 2.3. Maski bitowe

Nazwa	Wartość	Opis
`SC_WRITE_METHOD`	1	Klasa definiuje metodę `writeObject`.
`SC_SERIALIZABLE`	2	Klasa implementuje interfejs `Serializable`.
`SC_EXTERNALIZABLE`	4	Klasa implementuje interfejs `Externalizable`.
`SC_BLOCK_DATA`	8	Nieprzezroczyste dane są zapisywane jako „dane blokowe" (dostępna w JDK 1.2).
`SC_ENUM`	16	Klasa jest wyliczeniem.

Klasy, które piszemy, implementują interfejs `Serializable` i będą mieć bajt flag o wartości `02`. Z kolei serializowalna klasa `java.util.Date` będzie mieć bajt flag o wartości `03`, gdyż definiuje ona swoją własną metodę `writeObject` (patrz punkt 2.3.4). Interfejs `Externalizable` został dokładniej opisany w punkcie 2.3.5.

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 salary klasy Employee zostanie zapisane jako:

```
D 00 06 salary
```

A oto kompletny deskryptor zamieszczonej w przykładach klasy Employee, należącej do pakietu serial:

72 00 08 Employee	
74 1E C8 E6 C3 F8 B8 77 02	„Odcisk" i 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 — java.util.Date
L 00 04 name	Typ i nazwa pola składowego
74 00 12 Ljava/lang/String;	Nazwa klasy pola składowego — java.lang.String
78	Znacznik końca
70	Brak klasy bazowej

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. (W tym przykładzie datę zatrudnienia zapisujemy jako obiekt typu `Date`, a nie `LocalDate`, gdyż serializacja obiektów klasy `LocalDate` jest nieco trudniejsza).

Jak się właśnie przekonałeś, strumień obiektów zawiera nazwy klas, klas bazowych i pól wszystkich serializowanych obiektów. W przypadku klas wewnętrznych niektóre z tych nazw są tworzone przez kompilator, a stosowane przy tym konwencje nazewnicze mogą się zmieniać w poszczególnych implementacjach Javy. Jeśli tak się stanie, deserializacja nie powiedzie się. Dlatego istnieje niewielkie ryzyko związane z serializowaniem klas wewnętrznych, jeśli zamierzasz deserializować je przy użyciu innej implementacji.

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 0B [LEmployee;`	Nowa klasa, długość łańcucha, nazwa klasy `Employee[]`
`40 05 D6 7E 3B BB F2 50 02`	„Odcisk” i flagi
`00 00`	Liczba pól składowych

78	Znacznik końca
70	Brak klas nadrzędnych
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.

Niektóre klasy (takie jak `Date`) przechowują stan obiektu w formie nieprzezroczystego bloku danych binarnych, zakodowanego w następujący sposób:

- `77` (krótki blok) lub `7A` (długi blok),
- długość bloku (odpowiednio 1 bajt lub 4 bajty),
- zawartość bloku,
- `78`.

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.ser* w notacji szesnastkowej i porównaj z poniższymi komentarzami. Zwróć uwagę na najważniejsze wiersze, które są zamieszczone pod koniec pliku — zawierają one referencje zapisanego wcześniej obiektu.

AC ED 00 05	Nagłówek pliku
75	Tablica `staff` (nr 1)
72 00 0B [Lserial.Employee;	Nowa klasa, długość łańcucha, nazwa klasy `Employee[]` (nr 0)
40 05 D6 7E 3B BB F2 50 02	Odcisk i 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 0E serial.Manager             Nowa klasa, długość łańcucha, nazwa klasy
                                       (nr 2)
      2F FF 1F 1D E8 7A A9 74 02       Odcisk i flagi
      00 01                            Liczba pól składowych
      L 00 09 secretary                Typ i nazwa pola składowego
      74 00 11 Lserial/Employee;       Nazwa klasy pola składowego
                                       — String (nr 3)
      78                               Znacznik końca
   72 00 0F serial.Employee            Klasa nadrzędna — nowa klasa, długość
                                       łańcucha, nazwa klasy (nr 4)
      74 1E C8 E6 C3 F8 B8 77 02       Odcisk i 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 (nr 5)
      L 00 04 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 i flagi
         00 00                         Brak zmiennych składowych
         78                            Znacznik końca
      70                               Brak klasy bazowej
      77 08                            Zawartość zewnętrzna, liczba bajtów
         00 00 00 83 E7 4B 7D 80       Nieprzezroczyste bajty
         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	Nieprzezroczyste bajty
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 02	Istniejąca klasa (użyj nr 2)
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	Nieprzezroczyste bajty
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)

Studiowanie tych kodów nie jest oczywiście zbyt ciekawym zajęciem. Zazwyczaj znajomość dokładnego formatu pliku nie jest potrzebna (o ile nie próbujesz celowo dokonać zmian w samym pliku). Warto jednak wiedzieć, że format serializacji obiektów zawiera szczegółowy opis wszystkich obiektów, które zostały zapisane w strumieniu, co pozwala mu odtwarzać zarówno te obiekty, jak i tablice obiektów.

Powinniśmy zapamiętać, że:

- format serializacji obiektów zawiera 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.

2.3.3. Pola pomijane podczas serializacji

Niektóre pola nigdy nie powinny być serializowane — na przykład połączenia z bazą danych, które, po odtworzeniu obiektu, przestaną mieś sens. Dodatkowo, jeśli obiekt przechowuje pamięć podręczną wartości, lepszym rozwiązaniem może być jej usunięcie i ponowne przeliczenie zamiast przechowywania.

Aby zapobiec serializacji pola, należy oznaczyć je modyfikatorem `transient`. Zawsze oznaczaj pola jako `transient`, jeśli zawierają instancje klas, które nie są serializowalne. Takie pola są po prostu pomijane podczas serializacji obiektów.

2.3.4. Metody readObject i writeObject

W sporadycznych sytuacjach może się okazać, że konieczne będzie zmodyfikowanie domyślnego sposobu działania mechanizmu serializacji. Klasa zapewniająca możliwość serializacji może dodawać do domyślnego sposobu odczytu i zapisu dowolne operacje, używając do tego dwóch metod, których sygnatury zostały przedstawione poniżej:

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

W takim przypadku nagłówki obiektu będą zapisywane w standardowy sposób, jednak jego pola nie będą już automatycznie serializowane — zamiast tego będą wywoływane przedstawione metody.

Zwróć uwagę na adnotację `@Serial`. Metody służące do dostosowywania sposobu serializacji nie należą do żadnego interfejsu. Dlatego w ich przypadku nie można użyć adnotacji `@Override`, aby kompilator sprawdzał deklaracje tych metod. Adnotacja `@Serial` ma umożliwić wykonanie podobnego sprawdzenia dla metod związanych z serializacją. Do wersji 21 Javy kompilator `javac` nie dokonuje takiego sprawdzania, jednak w przyszłości może się to zmienić. Zintegrowane środowisko programistyczne IntelliJ wykorzystuje tę adnotację.

Oto typowy przykład dostosowania serializacji. Wiele klas w pakiecie `java.awt.geom`, takich jak `Point2D.Double`, nie zapewnia możliwości serializacji. Załóżmy, że chcesz serializować klasę `LabeledPoint`, która przechowuje obiekty `String` i `Point2D.Double`. Najpierw musisz oznaczyć pole `Point2D.Double` jako `transient`, aby uniknąć wystąpienia wyjątku `NotSerializableException`.

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

W metodzie `writeObject` najpierw zapisz deskryptor obiektu oraz pole typu `String`, czyli `label`, używając wywołania metody `defaultWriteObject`. Jest to specjalna metoda klasy

ObjectOutputStream, która może być wywoływana tylko wewnątrz metody writeObject klasy umożliwiającej serializację. Następnie zapisz współrzędne punktu, korzystając ze standardowych wywołań DataOutput.

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

Natomiast w metodzie readObject powyższy proces należy wykonać w odwrotnej kolejności:

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

Innym przykładem może być klasa HashSet, która dostarcza własne metody readObject i writeObject. Zamiast zapisywać wewnętrzną strukturę tablicy mieszającej, metoda write
↪Object zapisuje jedynie pojemność, współczynnik wypełnienia, rozmiar oraz poszczególne elementy. Metoda readObject wczytuje pojemność i współczynnik wypełnienia, tworzy nową tablicę i wstawia do niej elementy.

Metody readObject i writeObject muszą jedynie zapisywać i odczytywać swoje dane. Nie zajmują się danymi klasy bazowej ani żadnymi innymi informacjami dotyczącymi klasy.

To samo podejście wykorzystuje klasa Date. Jej metoda writeObject zapisuje liczbę milisekund, które upłynęły od „epoki" (czyli 1 stycznia 1970 roku). Struktura danych, która przechowuje pamięć podręczną danych kalendarza, nie jest zapisywana. Z tego powodu zserializowane instancje klasy Date przedstawione w podpunkcie 2.3.2. składają się z nieprzejrzystego bloku danych.

Metoda readObject, podobnie jak konstruktor, operuje na częściowo zainicjalizowanych obiektach. Jeśli wewnątrz metody readObject wywołasz niesfinalizowaną metodę, która jest przesłonięta w klasie pochodnej, może się okazać, że będziesz operować na niezainicjalizowanych danych.

Jeśli klasa, którą można serializować, definiuje pole:

```
@Serial private static final ObjectStreamField[] serialPersistentFields
```

to proces serializacji zastosuje podane w nim deskryptory pól, zamiast serializować pola, które nie są statyczne i nie zostały oznaczone modyfikatorem transient. Dostępne jest także API pozwalające określać wartości pól przed przeprowadzeniem serializacji bądź odczytanie ich przed przeprowadzeniem deserializacji. Przydaje się ono w przypadkach, gdy konieczne jest zachowanie starszego układu po zmodyfikowaniu klasy. Na przykład klasa BigDecimal używa tego mechanizmu do serializowania swoich instancji w formacie, który nie odzwierciedla już pól składowych.

2.3.5. Metody readExternal i writeExternal

Klasa może także definiować swój własny mechanizm serializacji i odtwarzania, zamiast korzystać z mechanizmu standardowego. Można to wykorzystać na przykład do szyfrowania danych albo stosowania formatu bardziej efektywnego niż standardowy.

Aby skorzystać z tych możliwości, klasa musi implementować interfejs `Externalizable`, który wymaga zdefiniowania dwóch metod:

```
public void readExternal(ObjectInput in)
      throws IOException, ClassNotFoundException;
public void writeExternal(ObjectOutput out) throws IOException;
```

W odróżnieniu od metod `readObject` i `writeObject` te dwie metody są w pełni odpowiedzialne za zapisywanie i odtwarzanie całego obiektu, włączając w to *dane klasy nadrzędnej*. Podczas zapisywania obiektu mechanizm serializacji jedynie podaje klasę obiektu w strumieniu wyjściowym. Przy odczytywaniu obiektu z zewnętrznej reprezentacji obiektowy strumień wejściowy tworzy obiekt, korzystając z konstruktora bezargumentowego, a następnie wywołuje metodę `readExternal`.

W poniższym przykładzie klasa `LabeledPixel` rozszerza serializowalną klasę `Point`, lecz przejmuje kontrolę nad serializacją zarówno jej, jak i jej klasy nadrzędnej. Pola obiektu nie są przechowywane w standardowym formacie serializacji. Zamiast tego informacje z obiektu zostają zapisane w nieprzejrzystym bloku danych.

```
public class LabeledPixel extends Point implements Externalizable
{
   private String label;

   public LabeledPixel() {} // Wymagany przez klasy implementujące Externalizable

   public void writeExternal(ObjectOutput out)
         throws IOException
   {
      out.writeInt((int) getX());
      out.writeInt((int) getY());
      out.writeUTF(label);
   }

   public void readExternal(ObjectInput in)
         throws IOException, ClassNotFoundException
   {
      int x = in.readInt();
      int y = in.readInt();
      setLocation(x, y);
      label = in.readUTF();
   }
   ...
}
```

Metod `readExternal` i `writeExternal` nie należy poprzedzać adnotacją `@Serial`. Ponieważ metody te są zdefiniowane w interfejsie `Externalizable`, wystarczy poprzedzić je adnotacją `@Override`.

W odróżnieniu od metod `readObject` i `writeObject`, które są prywatne i mogą być wywoływane wyłącznie przez mechanizm serializacji, metody `readExternal` i `write`
`↪External` są publiczne. W szczególności warto zwrócić uwagę, że metoda `readExternal` może pozwalać na zmianę stanu istniejącego obiektu.

2.3.6. Metody readResolve i writeReplace

Przyjmujemy za oczywiste, że obiekty można tworzyć jedynie przy użyciu konstruktora. Jednak obiekt odtwarzany przez mechanizm deserializacji *nie jest konstruowany* w taki sposób. Jego pola są po prostu przywracane ze strumienia obiektów.

Może to stanowić problem, jeśli konstruktor wymusza pewne warunki. Na przykład obiekt singleton może być zaimplementowany w taki sposób, by konstruktor był wywołany tylko raz. Innym przykładem mogą być encje baz danych, które mogą być tworzone w taki sposób, że zawsze pochodzą z puli zarządzanych instancji.

Nie należy tworzyć własnego mechanizmu dla singletonów. Jeśli potrzebujesz singletona, utwórz typ wyliczeniowy z jedną instancją, która zwyczajowo będzie nosić nazwę jest `INSTANCE`.

```
public enum PersonDatabase {
   INSTANCE;

   public Person findById(int id) { . . . }
   ...
}
```

To rozwiązanie działa, ponieważ poprawna deserializacja instancji typów wyliczeniowych (`enum`) jest zagwarantowana.

A teraz załóżmy, że znalazłeś się w bardzo sporadycznej sytuacji, która zmusza Cię do kontrolowania tożsamości każdej deserializowanej instancji. W ramach przykładu załóżmy, że instancje klasy `Person` mają być deserializowane z bazy danych. W takim przypadku nie należy serializować samego obiektu, lecz zażądać zapisania instancji obiektu pośredniczącego (ang. *proxy*). Później, podczas deserializacji, ten obiekt pośredniczący zlokalizuje i utworzy pożądany obiekt. W takiej sytuacji Twoja klasa powinna definiować metodę `writeReplace`, która będzie zwracać obiekt pośredniczący:

```
public class Person implements Serializable {
   private int id;
   // Inne pola
   ...
   @Serial private Object writeReplace() {
      return new PersonProxy(id);
   }
}
```

Podczas serializacji takiego obiektu `Person` nie jest zapisywane żadne z jego pól. Zamiast tego zostaje wywołana metoda `writeReplace`, a do strumienia zostanie zapisany *zwrócony przez nią wynik*.

Z kolei klasa pośrednicząca powinna implementować metodę `readResolve`, która zwraca instancję klasy `Person`:

```
class PersonProxy implements Serializable {
   private int id;

   public PersonProxy(int id) {
      this.id = id;
   }

   @Serial private Object readResolve() {
      return PersonDatabase.INSTANCE.findById(id);
   }
}
```

Kiedy metoda `readObject` odnajdzie w strumieniu `ObjectInputStream` instancję `PersonProxy`, zdeserializuje ją, wywoła na rzecz utworzonego obiektu metodę `readResolve` i zwróci jej wynik.

W odróżnieniu od metod `readObject` i `writeObject` metody `readResolve` i `writeReplace` muszą być prywatne.

Metody `readObject` i `writeObject` i `readExternal` i `writeExternal` nie są stosowane podczas serializacji wartości typów wyliczeniowych i rekordów. Metoda `writeReplace` jest natomiast używana w procesie serializacji rekordów, lecz nie wartości typów wyliczeniowych.

2.3.7. 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 wejściowy 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:

```
serial.Employee:      static final long serialVersionUID = 8367346051156850807L;
```

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

```
class Employee implements Serializable // Wersja 1.1
{
```

```
    . . .
    @Serial public static final long serialVersionUID = 8367346051156850807L;
}
```

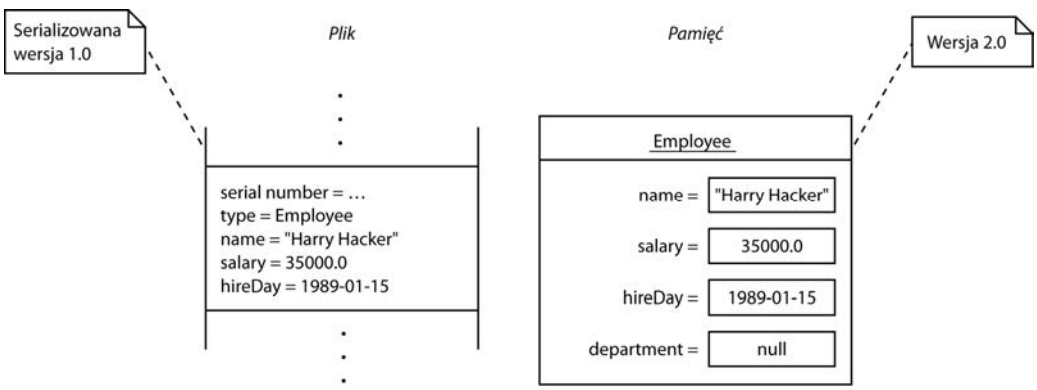
Klasa posiadająca statyczne pole składowe o nazwie `serialVersionUID` nie obliczy własnego „odcisku palca", ale skorzysta z już istniejącej wartości.

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

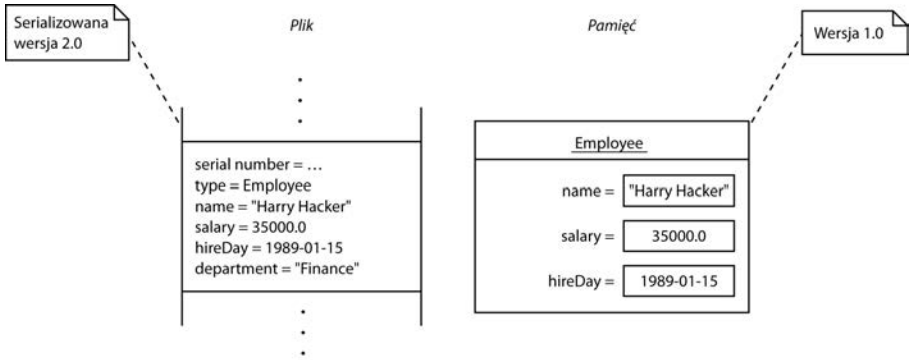
Jeśli zmienią się tylko metody danej klasy, sposób odczytu danych nie ulegnie zmianie. Jednakże jeżeli zmieni się pole składowe, możemy mieć pewne problemy. Na przykładu stary obiekt może mieć więcej lub mniej pól składowych niż aktualny albo typy pól mogą się różnić. W takim wypadku wejściowy strumień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

Wejściowy strumień obiektów porównuje pola składowe aktualnej wersji klasy z polami składowymi serializowanej wersji klasy 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 wejściowy nawet nie próbuje konwersji — obiekty są niekompatybilne. Jeżeli serializowany obiekt zapisany w strumieniu ma pola składowe nieobecne w aktualnej wersji, strumień ignoruje te dodatkowe dane. Jeżeli aktualna wersja ma pola składowe nieobecne w serializowanym 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 2.7 pokazuje, co się dzieje, gdy obiekt wersji 1.0 jest wczytywany przez program korzystający z obiektów 2.0. Pole `department` otrzymuje wartość `null`. Rysunek 2.8 ilustruje odwrotną sytuację — program korzystający z obiektów 1.0 wczytuje obiekt 2.0. Dodatkowe pole `department` jest ignorowane.



Rysunek 2.7. Odczytywanie obiektu o mniejszej liczbie pól



Rysunek 2.8. Odczytywanie obiektu o większej liczbie pól

Czy ten proces jest bezpieczny? To zależy. Opuszczanie pól składowych wydaje się bezbolesne — odbiorca wciąż posiada dane, którymi potrafi operować. 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 obsługiwania 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`.

Przed dodaniem do klasy pola `serialVersionUID` należy zadać sobie pytanie, dlaczego dana klasa zapewnia możliwość serializacji. Jeśli serializacja ma służyć jedynie do zapewnienia obiektom trwałości na krótki okres czasu, na przykład na potrzeby wywołań rozproszonych w obrębie serwera aplikacji, to nie ma potrzeby przejmować się wersjami i polem `serialVersionUID`. To samo dotyczy sytuacji, kiedy rozszerzona zostanie klasa, która już zapewnia możliwość serializacji, lecz nie zależy nam na trwałym przechowywaniu jej instancji. Jeśli stosowane zintegrowane środowisko programistyczne zgłasza jakieś natrętne ostrzeżenia związane z trwałością, to można je wyłączyć w ustawieniach IDE bądź przy użyciu adnotacji `@SuppressWarnings("serial")`. To bezpieczniejsze rozwiązanie niż dodawanie pola `serialVersionUID`, o którego aktualizacji łatwo można później zapomnieć.

Typy wyliczeniowe i rekordy ignorują pole `serialVersionUID`. W przypadku typu wyliczeniowego pole to zawsze ma wartość 0L. W przypadku rekordów wartość pola serialVersionUID można zadeklarować, jednak podczas deserializacji ich wartości nie muszą być zgodne.

W tym punkcie wyjaśniłem, co się dzieje, gdy dana wersja klasy czytelnika ma pola składowe, których nie ma w strumieniu obiektów. Może się także zdarzyć, że podczas ewolucji klasy zostanie do niej dodana klasa bazowa. W takim przypadku czytnik używający nowej wersji może natrafić na strumień obiektów, w którym pola składowe danej klasy bazowej nie będą ustawione. Domyślnie takie pola składowe są ustawiane na ich domyślne wartości zerowe lub `null`. Może to sprawić, że po deserializacji klasa bazowa znajdzie się w niebezpiecznym stanie. Klasa bazowa może się chronić przed takimi sytuacjami poprzez zdefiniowanie metody inicjalizującej:

```
@S[erial private void readObjectNoData() throws ObjectStreamException
```

Metoda ta powinna ustawiać obiekt w takim samym stanie, w jakim zwraca go konstruktor bezargumentowy, bądź zgłaszać wyjątek `InvalidObjectException`. Jest ona wywoływana jedynie w niezwykłych okolicznościach, gdy podczas odczytu strumienia wejściowego natrafiono na instancję klasy pochodnej z brakującymi danymi klasy bazowej.

2.3.8. 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 2.4 udowadnia, że aby otrzymać metodę `clone` „za darmo”, wystarczy rozszerzyć klasę `SerialCloneable`.

Listing 2.4. serialClone/SerialCloneTest.java

```
package serialClone;

/**
 * @version 1.23 2024-01-12
 * @author Cay Horstmann
 */

import java.io.*;
import java.time.*;

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

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

      // Teraz obiekt harry i jego klon różnią się
      System.out.println(harry);
      System.out.println(harry2);
   }
}

/**
 * Klasa, której metoda clone wykorzystuje serializację
 */
class SerialCloneable implements Cloneable, Serializable
{
   public Object clone() throws CloneNotSupportedException
   {
```

```
      try
      {
         // Zapisuje obiekt w tablicy bajtów
         var bout = new ByteArrayOutputStream();
         try (var out = new ObjectOutputStream(bout))
         {
            out.writeObject(this);
         }

         // Wczytuje klon obiektu z tablicy bajtów
         var bin = new ByteArrayInputStream(bout.toByteArray());
         try (var in = new ObjectInputStream(bin))
         {
            return in.readObject();
         }
      }
      catch (IOException | ClassNotFoundException e)
      {
         var e2 = new CloneNotSupportedException();
         e2.initCause(e);
         throw e2;
      }
   }
}

/**
 * Znana już klasa Employee, tym razem jako pochodna klasy SerialCloneable
 */
class Employee extends SerialCloneable
{
   private String name;
   private double salary;
   private LocalDate hireDay;

   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      hireDay = LocalDate.of(year, month, day);
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public LocalDate getHireDay()
   {
      return hireDay;
   }

   /**
    * Metoda zwiększająca wynagrodzenie tego pracownika
    *
```

```
     * @param byPercent procentowa wartość wzrostu wynagrodzenia
     */
    public void raiseSalary(double byPercent)
    {
       double raise = salary * byPercent / 100;
       salary += raise;
    }

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

2.3.9. Deserializacja a bezpieczeństwo

Podczas deserializacji klasy, która zapewnia tę możliwość, jej obiekty są tworzone bez wywoływania jakiegokolwiek konstruktora tej klasy. Nawet jeśli klasa ma konstruktor bezargumentowy, nie jest on używany. Wartości pól są ustawiane bezpośrednio na podstawie wartości odczytywanych z wejściowego strumienia obiektów.

> W przypadku deserializacji *rekordów* zapewniających tę możliwość proces deserializacji będzie wywoływał konstruktor kanoniczny i przekazywał do niego wartości komponentów odczytywane z wejściowego strumienia obiektów. (Oznacza to, że odwołania cykliczne występujące w takich rekordach nie będą odtwarzane).

Pominięcie konstruktorów jest zagrożeniem bezpieczeństwa. Atakujący może przygotować bajty opisujące nieprawidłowy obiekt, który normalnie nie mógłby zostać utworzony. Na przykład załóżmy, że konstruktor klasy `Employee` zgłasza wyjątek, gdy zostanie wywołany z ujemną pensją. Chcielibyśmy móc założyć, że oznacza to, iż żaden obiekt `Employee` nie może mieć ujemnej pensji. Jednak jak wiesz z poprzedniego punktu rozdziału, całkiem łatwo można przeanalizować bajty serializowanego obiektu i odpowiednio zmodyfikować niektóre z nich. Dzięki temu można przygotować bajty obiektu pracownika, który po deserializacji będzie mieć ujemną pensję.

Serializowalna klasa czasami może implementować interfejs `ObjectInputValidation` i definiować metodę `validateObject`, aby sprawdzać, czy obiekt został prawidłowo odtworzony podczas procesu deserializacji. Na przykład klasa `Employee` może sprawdzać, czy pensja pracownika nie jest ujemna.

```
public void validateObject() throws InvalidObjectException
{
   System.out.println("validateObject");
```

```
      if (salary < 0)
         throw new InvalidObjectException("salary < 0");
   }
```

Niestety metoda `validateObject` nie jest wywoływana automatycznie. Aby była wywoływana podczas deserializacji, należy także zaimplementować poniższą metodę:

```
@Serial private void readObject(ObjectInputStream in)
      throws IOException, ClassNotFoundException
{
   in.registerValidation(this, 0);
   in.defaultReadObject();
}
```

Dzięki temu zostanie zaplanowana walidacja obiektu i wywołana metoda `validateObject`, gdy dany obiekt oraz wszystkie obiekty zależne zostaną odtworzone. Drugi parametr pozwala określić priorytet. Żądania walidacji z wyższymi priorytetami są wykonywane w pierwszej kolejności.

Istnieją także inne zagrożenia bezpieczeństwa. Osoby wrogo nastawione mogą tworzyć struktury danych, które będą zużywać wystarczająco dużo zasobów, by spowodować awarię wirtualnej maszyny Javy. Jeszcze bardziej podstępna jest możliwość deserializacji każdej klasy dostępnej w ścieżce klas. Pomysłowi hakerzy tworzyli „łańcuchy gadżetów", sekwencje operacji na różnych klasach narzędziowych korzystających z mechanizmu refleksji, które w efekcie pozwalały wywoływać metody takie jak `Runtime.exec` z dowolnym łańcuchem znaków.

Każda aplikacja, która przeprowadza deserializację danych pochodzących z niezaufanych źródeł z internetu, jest narażona na takie ataki. Na przykład niektóre serwery serializują dane sesji i deserializują dane zwracane w ciasteczku HTTP sesji.

Należy unikać sytuacji, w których dowolne dane pochodzące z niezaufanych źródeł są deserializowane. Na przykład w przypadku danych sesji serwer powinien podpisywać dane i deserializować tylko te, które mają ważny podpis.

Propozycje JEP 290 i 415 przedstawiają mechanizm *filtrów serializacji*, przeznaczonych do zabezpieczenia aplikacji przed takimi atakami. Filtry te mogą sprawdzać nazwy deserializowanych klas oraz kilka parametrów (rozmiar strumienia, rozmiary tablic, całkowitą liczbę referencji, najdłuższy łańcuch referencji) i na ich podstawie przerywać proces deserializacji danych.

W najprostszej postaci takiego filtra podajemy wzorzec opisujący poprawne i niepoprawne klasy. Na przykład jeśli uruchomimy nasz prosty przykład serializacji, używając następującego polecenia:

```
java -Djdk.serialFilter='serial.*;java.**;!*' serial.ObjectStreamTest
```

to obiekty zostaną wczytane. Zastosowany filtr pozwala na wczytywanie wszystkich klas należących do pakietu `serial` oraz wszystkich klas, których nazwy pakietów zaczynają się od `java`, lecz żadnych innych. Jeśli nie zezwolimy na deserializację obiektów z pakietów

java.**, lub choćby z pakietu java.util.Date, proces deserializacji zakończy się niepowodzeniem.

Taki filtr deserializacji można umieścić w pliku konfiguracyjnym i przygotować wiele podobnych na różne sytuacje. Można także implementować własne filtry. Więcej informacji na ten temat znajdziesz na stronie *https://docs.oracle.com/en/java/javase/21/core/serialization-filtering1.html*.

2.4. Praca z plikami

Potrafimy już zapisywać i wczytywać dane z pliku. Jednakże obsługa plików to coś więcej niż tylko operacje zapisu i odczytu. Interfejs Path oraz klasa Files zawierają metody potrzebne do obsługi systemu plików na komputerze użytkownika. Na przykład możemy wykorzystać klasę Files, aby sprawdzić, kiedy nastąpiła ostatnia modyfikacja danego pliku, oraz usunąć plik lub zmienić jego nazwę. Innymi słowy, klasy strumieni wejścia-wyjścia zajmują się zawartością plików, a klasy omawiane w tym podrozdziale związane są z organizacją przechowywania plików na dysku.

Interfejs Path i klasa Files są znacznie wygodniejsze w użyciu niż klasa File pamiętająca jeszcze czasy JDK 1.0. Klasy File nie należy używać, chyba że w razie konieczności współdziałania z zastanym kodem.

2.4.1. Ścieżki dostępu

Ścieżka dostępu reprezentowana przez klasę Path jest sekwencją nazw katalogów zakończoną opcjonalnie nazwą pliku. Pierwszym elementem ścieżki może być *komponent korzenia*, taki jak na przykład / czy C:\. Dozwolone komponenty korzenia zależą od używanego systemu plików. Ścieżka zaczynająca się od komponentu korzenia jest ścieżką *bezwzględną*. W przeciwnym razie jest ścieżką *względną*. Poniżej konstruujemy przykładową ścieżkę bezwzględną i ścieżkę względną. W przypadku ścieżki bezwzględnej założyliśmy, że komputer wykorzystuje system plików zorganizowany na wzór systemu UNIX.

```
Path absolute = Paths.get("/home", "harry");
Path relative = Paths.get("myprog", "conf", "user.properties");
```

Metoda statyczna Paths.get otrzymuje jeden lub więcej łańcuchów znakowych, które łączy separatorem ścieżki dla domyślnego systemu plików (/ w przypadku systemu UNIX i pokrewnych, \ w przypadku systemu Windows). Następnie parsuje wynik i tworzy obiekt Path. W przypadku gdy wynik połączenia argumentów metody nie stanowi poprawnej ścieżki w danym systemie plików, metoda zgłasza wyjątek InvalidPathException.

Metoda get może również otrzymać pojedynczy łańcuch znaków zawierający wiele komponentów. Na przykład możemy wczytać ścieżkę z pliku konfiguracyjnego w poniższy sposób:

```
String baseDir = props.getProperty("base.dir")
// Może zawierać łańcuch o postaci /opt/myprog lub c:\Program Files\myprog
Path basePath = Paths.get(baseDir); // OK, baseDir zawiera separatory
```

Ścieżka nie musi odpowiadać istniejącemu plikowi. Stanowi raczej abstrakcyjną sekwencję nazw. W następnym podrozdziale pokażemy, że chcąc utworzyć plik, najpierw konstruujemy ścieżkę, a dopiero potem wywołujemy metodę tworzącą plik odpowiadający tej ścieżce.

Często wykonywana operacja polega na łączeniu bądź inaczej *rozwiązywaniu* ścieżek. Wykonywane w tym celu wywołanie `p.resolve(q)` zwraca ścieżkę zgodnie z poniższymi regułami:

- Jeśli `q` jest ścieżką bezwzględną, wynikiem jest `q`.
- Jeśli `q` nie określa katalogu głównego, to wynik będzie stanowić połączenie `p` i `q`.
- W przeciwnym razie wynik będzie zależny od reguł stosowanych w używanym systemie plików.

Załóżmy na przykład, że nasza aplikacja musi określić swój katalog roboczy względem danego katalogu bazowego wczytanego z pliku konfiguracyjnego jak w poprzednim przykładzie.

```
Path workRelative = Path.of("work");
Path workPath = basePath.resolve(workRelative);
```

Istnieje szybszy sposób wykonania tego zadania dzięki wersji metody `resolve`, której parametrem jest łańcuch znaków zamiast obiektu `Path`:

```
Path workPath = basePath.resolve("work");
```

Metoda resolve nie upraszcza wyniku. Na przykład wywołanie:

```
Path.of("/usr/bin").resolve("../local")
```

zwraca ścieżkę o postaci: `/usr/bin/../local`.

Istnieje również metoda `resolveSibling` pozwalająca utworzyć ścieżkę bliźniaczą na podstawie ścieżki nadrzędnej. Na przykład jeśli `workPath` reprezentuje ścieżkę */opt/myapp/work*, to wywołanie

```
Path tempPath = workPath.resolveSibling("temp")
```

utworzy ścieżkę */opt/myapp/temp*.

Działaniem odwrotnym do rozwiązywania ścieżki jest jej *relatywizacja*. Wywołanie `p.relativize(r)` daje w wyniku ścieżkę `q`, gdy `r` jest wynikiem rozwiązania `p` względem `q`. Na przykład wynikiem relatywizacji ścieżki */home/harry* względem */home/fred/input.txt* jest ścieżka *../fred/input.txt*. W tym przykładzie zakładamy, że .. oznacza katalog nadrzędny w danym systemie plików.

Zastosowanie metody `normalize` pozwala usunąć powtarzające się komponenty `.` i `..` (lub inne, w zależności od systemu plików). Na przykład wynikiem normalizacji ścieżki */home/harry/../fred/./input.txt* będzie ścieżka */home/fred/input.txt*.

Metoda `toAbsolutePath` daje w wyniku bezwzględną ścieżkę dla ścieżki podanej, rozpoczynającą się komponentem korzenia, na przykład */home/fred/input.txt* lub *c:\Users\fred\input.txt*.

Interfejs `Path` zawiera wiele przydatnych metod wykonujących różne operacje na ścieżkach. Poniżej kilka przykładów:

```
Path p = Paths.get("/home", "fred", "myprog.properties");
Path parent = p.getParent(); // Ścieżka /home/fred
Path file = p.getFileName(); // Ścieżka myprog.properties
Path root = p.getRoot(); // Ścieżka /
```

Metoda getFileName zwraca ostatni komponent danej ścieżki. Jeśli ścieżka ta reprezentuje katalog, będzie to nazwa katalogu, a nie pliku.

Jak już wiesz z pierwszego tomu książki, obiekt Scanner można utworzyć na podstawie obiektu Path:

```
var in = new Scanner(Path.of("/home/fred/input.txt"));
```

Jeśli zdarzy się konieczność współdziałania z tradycyjnym kodem wykorzystującym klasę `File`, warto wiedzieć, że interfejs `Path` udostępnia metodę `toFile`, a klasa `File` metodę `toPath`.

API *java.nio.file.Path* 7

- `static Path of(String first, String... more)` 11

  tworzy ścieżkę poprzez połączenie przekazanych łańcuchów znaków.

- `Path resolve(Path other)`
- `Path resolve(String other)`

  jeśli `other` jest ścieżką bezwzględną, zwraca `other`; w przeciwnym razie zwraca ścieżkę powstałą z połączenia `this` i `other`.

- `Path resolveSibling(Path other)`
- `Path resolveSibling(String other)`

  jeśli `other` jest ścieżką bezwzględną, zwraca `other`; w przeciwnym razie zwraca ścieżkę powstałą z połączenia ścieżki nadrzędnej dla `this` i `other`.

- `Path relativize(Path other)`

  zwraca ścieżkę względną, która rozwiązana względem `this` daje w wyniku `other`.

- `Path normalize()`

  usuwa nadmiarowe elementy ścieżki, takie jak `.` i `..`

- `Path toAbsolutePath()`

  zwraca daną ścieżkę, jeśli jest ona ścieżką bezwzględną, a w przeciwnym razie ścieżkę względną wyznaczoną względem katalogu roboczego.

- `Path getParent()`

  zwraca ścieżkę nadrzędną lub `null`, gdy ścieżka nadrzędna nie istnieje.

- `Path getFileName()`

  zwraca ostatni komponent ścieżki lub `null`, gdy ścieżka nie ma komponentów.

- `Path getRoot()`

  zwraca komponent korzenia danej ścieżki lub `null`, gdy ścieżka nie ma takiego komponentu.

- `toFile()`

  tworzy obiekt `File` dla danej ścieżki.

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

- `Path toPath()` **7**

  tworzy obiekt `Path` dla danego pliku.

2.4.2. Odczyt i zapis plików

Klasa `Files` pozwala przyspieszyć programowanie typowych operacji na plikach. Na przykład poniższe wywołanie umożliwia wczytanie całej zawartości pliku w postaci tablicy bajtów, łańcucha znaków, listy lub strumienia wierszy:

```
byte[] bytes = Files.readAllBytes(path);
String content = Files.readString(path, charset);
List<String> lines = Files.readAllLines(path, charset);
Stream<String> lineStream = Files.lines(path, charset);
```

A oto analogiczne operacje zapisu:

```
Files.write(path, bytes);
Files.writeString(path, content, charset);
Files.write(path, lines, charset);
```

Aby dopisać łańcuch na końcu pliku, zastosujemy następujące wywołanie:

```
Files.write(path, content, charset, StandardOpenOption.APPEND);
```

To wywołanie:

```
long pos = Files.mismatch(path1, path2);
```

zwraca położenie pierwszego bajtu, na którym zawartości dwóch podanych plików różnią się od siebie.

Typ MIME pliku (taki jak `"text/html"` lub `"image/png"`) można pobrać przy użyciu następującego wywołania:

```
String mimeType = Files.probeContentType(path);
```

Dokładny sposób działania może się różnić w zależności od konkretnej implementacji Javy. Każda implementacja dysponuje zbiorem instancji typu `FileTypeDetector`, które są sprawdzane w nieokreślonej kolejności. Każdy z takich detektorów może sprawdzać rozszerzenie pliku, metadane charakterystyczne dla systemu operacyjnego lub kilka początkowych bajtów zawartości pliku.

Poniższe metody umożliwiają korzystanie z API, które pozwala używać poznanych wcześniej strumieni wejścia-wyjścia oraz obiektów `Reader/Writer`:

```
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
Reader in = Files.newBufferedReader(path, charset);
Writer out = Files.newBufferedWriter(path, charset);
```

Powyższe metody uwalniają nas od korzystania z klas `FileInputStream`, `FileOutputStream`, `BufferedReader` lub `BufferedWriter`.

API `java.nio.file.Files` 7

- `static byte[] readAllBytes(Path path)`
- `static String readString(Path path, Charset charset)`
- `static List<String> readAllLines(Path path, Charset charset)`

  wczytują zawartość pliku.

- `static Path write(Path path, byte[] contents, OpenOption... options)`
- `string Path write(Path path, String contents, Charset charset, OpenOption... options)`
- `static Path writeString(Path path, CharSequence contents, Charset cs, OpenOption... options)` **11**
- `static Path write(Path path, Iterable<? extends CharSequence> contents, Charset cs, OpenOption options)`

  zapisują podaną zawartość w pliku i zwracają ścieżkę.

- `static long mismatch(Path path, Path path2)` **12**
- zwraca położenie pierwszego bajtu, na którym zawartości dwóch plików różnią się od siebie.
- `static String probeContentType(Path path)` **12**

  zwraca wynik próby odgadnięcia typu MIME zawartości pliku wykonanej przy użyciu zainstalowanych detektorów typów, a jeśli tej próby nie udało się pomyślnie wykonać, wartość `null`.

- `static InputStream newInputStream(Path path, OpenOption... options)`
- `static OutputStream newOutputStream(Path path, OpenOption... options)`
- `static BufferedReader newBufferedReader(Path path, Charset charset)`

- `static BufferedWriter newBufferedWriter(Path path, Charset charset, OpenOption... options)`

  otwierają plik do odczytu lub zapisu.

2.4.3. Tworzenie plików i katalogów

Aby utworzyć nowy katalog, wywołamy:

```
Files.createDirectory(path);
```

Wszystkie komponenty ścieżki oprócz ostatniego muszą już istnieć. Jeśli chcemy utworzyć również katalogi pośrednie, użyjemy wywołania:

```
Files.createDirectories(path);
```

Pusty plik tworzymy, wywołując:

```
Files.createFile(path);
```

Powyższa metoda zgłasza wyjątek, jeśli plik już istnieje. Operacja sprawdzenia istnienia pliku i jego utworzenia jest atomowa. Jeśli plik nie istnieje, na pewno zostanie utworzony, zanim ktokolwiek inny uzyska szansę wykonania takiej samej operacji.

Istnieją również metody przydatne do tworzenia plików lub katalogów tymczasowych w podanej lokalizacji lub lokalizacji specyficznej dla danego systemu.

```
Path newPath = Files.createTempFile(dir, prefix, suffix);
Path newPath = Files.createTempFile(prefix, suffix);
Path newPath = Files.createTempDirectory(dir, prefix);
Path newPath = Files.createTempDirectory(prefix);
```

W powyższych przykładach `dir` jest obiektem `Path`, a argumenty `prefix` i `suffix` są łańcuchami znaków i mogą również mieć wartość `null`. Na przykład wynikiem wywołania `Files.createTempFile(null, ".txt")` może być ścieżka postaci */tmp/1234405522364837194.txt*.

Tworząc plik lub katalog, możemy określić jego atrybuty, takie jak właściciele pliku i uprawnienia. Ponieważ jednak szczegóły zależą od wykorzystywanego systemu plików, nie będziemy ich tutaj omawiać.

API java.nio.file.Files 7

- `static Path createFile(Path path, FileAttribute<?>... attrs)`
- `static Path createDirectory(Path path, FileAttribute<?>... attrs)`
- `static Path createDirectories(Path path, FileAttribute<?>... attrs)`

  tworzą plik lub katalog. Metoda `createDirectories` tworzy również katalogi pośrednie.

- `static Path createTempFile(String prefix, String suffix, FileAttribute<?>... attrs)`

- `static Path createTempFile(Path parentDir, String prefix, String suffix, FileAttribute<?>... attrs)`
- `static Path createTempDirectory(String prefix, FileAttribute<?>... attrs)`
- `static Path createTempDirectory(Path parentDir, String prefix, FileAttribute<?>... attrs)`

  tworzą plik lub katalog tymczasowy w lokalizacji przewidzianej dla takich plików lub w podanym katalogu nadrzędnym. Zwracają ścieżkę dostępu do utworzonego pliku lub katalogu.

2.4.4. Kopiowanie, przenoszenie i usuwanie plików

Aby skopiować plik z jednej lokalizacji do innej, wywołamy:

```
Files.copy(fromPath, toPath);
```

Aby przenieść plik, użyjemy następującego wywołania:

```
Files.move(fromPath, toPath);
```

Operacje kopiowania lub przenoszenia pliku zakończą się niepowodzeniem, jeśli plik docelowy już istnieje. Jeśli chcemy go zastąpić, powinniśmy użyć opcji `REPLACE_EXISTING`. Opcja `COPY_ATTRIBUTES` przydaje się, jeśli chcemy skopiować wszystkie atrybuty pliku. Opcji tych używamy w poniższy sposób:

```
Files.copy(fromPath, toPath, StandardCopyOption.REPLACE_EXISTING,
➥StandardCopyOption.COPY_ATTRIBUTES);
```

Możemy zażądać, aby operacja przeniesienia pliku była atomowa. Dzięki temu mamy gwarancję, że zakończy się ona powodzeniem lub w przeciwnym razie oryginalny plik nie zostanie usunięty. W tym celu używamy opcji `ATOMIC_MOVE`:

```
Files.move(fromPath, toPath, StandardCopyOption.ATOMIC_MOVE);
```

Możemy także skopiować strumień wejściowy do obiektu typu `Path`, co jest równoznaczne z zapisaniem zawartości strumienia na dysku. Analogicznie można też skopiować obiekt `Path` do strumienia wyjściowego. W tym celu należy użyć następujących wywołań:

```
Files.copy(inputStream, toPath);
Files.copy(fromPath, outputStream);
```

Podobnie jak przy pozostałych wywołaniach metody `copy`, także w tym przypadku można określać dodatkowe opcje.

Aby usunąć plik, wywołujemy po prostu:

```
Files.delete(path);
```

Metoda ta zgłosi wyjątek, jeśli plik nie istnieje. Dlatego zamiast niej możemy użyć również wywołania:

```
boolean deleted = Files.deleteIfExists(path);
```

Metody te możemy również wykorzystać do usunięcia pustego katalogu.

Lista wszystkich opcji, których można używać w operacjach na plikach, została podana w tabeli 2.4.

Tabela 2.4. Standardowe opcje stosowane w operacjach na plikach

Opcja	Opis
`StandardOpenOption` — do stosowania w metodach `newBufferedWriter`, `newInputStream`, `newOutputStream`, `write`	
`READ`	Otwiera plik do odczytu.
`WRITE`	Otwiera plik do zapisu.
`APPEND`	Jeśli plik otworzono do zapisu, dane będą dopisywane na jego końcu.
`TRUNCATE_EXISTING`	Jeśli plik otworzono do zapisu, jego dotychczasowa zawartość zostanie usunięta.
`CREATE_NEW`	Tworzy nowy plik lub zgłasza błąd, jeśli plik istnieje.
`CREATE`	Automatycznie tworzy nowy plik, jeśli jeszcze nie istnieje.
`DELETE_ON_CLOSE`	Środowisko postara się usunąć plik po jego zamknięciu.
`SPARSE`	Podpowiedź dla systemu operacyjnego, że plik będzie rzadki.
`DSYNC|SYNC`	Wymaga, by wszelkie zmiany w danych pliku, jego zawartości oraz metadanych były synchronicznie zapisywane na nośniku.
`StandardCopyOption` — do stosowania w metodach `copy`, `move`	
`ATOMIC_MOVE`	Przesunięcie plików ma być operacją atomową.
`COPY_ATTRIBUTES`	Skopiowanie atrybutów pliku.
`REPLACE_EXISTING`	Zastąpienie pliku docelowego, jeśli ten istnieje.
`LinkOption` — do stosowania we wszystkich metodach wymienionych powyżej, jak również w metodach `exists`, `isDirectory`, `isRegularFile`	
`NOFOLLOW_LINKS`	Nie należy używać łączy symbolicznych.
`FileVisitOption` — do stosowania w metodach `find`, `walk`, `walkFileTree`	
`FOLLOW_LINKS`	Należy używać łączy symbolicznych.

API **`java.nio.file.Files` 7**

- `static Path copy(Path from, Path to, CopyOption... options)`
- `static Path move(Path from, Path to, CopyOption... options)`

  kopiują lub przenoszą `from` do lokalizacji `to`. Zwracają `to`.
- `static long copy(InputStream from, Path to, CopyOption... options)`
- `static long copy(Path from, OutputStream to, CopyOption... options)`

  kopiuje dane ze strumienia wejściowego do pliku lub z pliku do strumienia wyjściowego, zwracając przy tym liczbę skopiowanych bajtów.

- `static void delete(Path path)`
- `static boolean deleteIfExists(Path path)`

  zwracają podany plik lub pusty katalog. Metoda `delete` zgłasza wyjątek, jeśli plik lub katalog nie istnieje. W takim przypadku metoda `deleteIfExists` zwraca wartość `false`.

2.4.5. Informacje o plikach

Poniższe metody statyczne zwracają wartość `boolean` pozwalającą sprawdzić właściwość ścieżki:

- `exists`
- `isHidden`
- `isReadable, isWritable, isExecutable`
- `isRegularFile, isDirectory, isSymbolicLink`

Metoda `size` zwraca liczbę bajtów w pliku.

```
long fileSize = Files.size(path);
```

Metoda `getOwner` zwraca właściciela pliku jako instancję klasy `java.nio.file.attribute.UserPrincipal`.

Wszystkie systemy plików raportują pewien podstawowy zbiór atrybutów reprezentowany przez interfejs `BasicFileAttributes`. Należą do nich:

- czasy: utworzenia pliku, ostatniego dostępu do pliku i ostatniej jego modyfikacji — reprezentowane przez instancje klasy `java.nio.file.attribute.FileTime`;
- typ pliku — zwykły, katalog, łącze lub inny;
- rozmiar pliku;
- klucz pliku — obiekt pewnej klasy specyficzny dla systemu plików, który może identyfikować plik w unikatowy sposób.

Atrybuty te pobieramy za pomocą następującego wywołania:

```
BasicFileAttributes attributes = Files.readAttributes(path, BasicFileAttributes.class);
```

Jeśli wiemy, że system plików jest zgodny z POSIX, możemy pobrać instancję `PosixFileAttributes`:

```
PosixFileAttributes attributes = Files.readAttributes(path, PosixFileAttributes.class);
```

Następnie możemy również dowiedzieć się, jakie uprawnienia ma właściciel pliku, grupa użytkowników i reszta świata. Nie będziemy tutaj zagłębiać się w szczegóły, ponieważ większość tych informacji nie jest przenośna pomiędzy różnymi systemami.

API `java.nio.file.Files` 7

- `static boolean exists(Path path)`
- `static boolean isHidden(Path path)`
- `static boolean isReadable(Path path)`
- `static boolean isWritable(Path path)`
- `static boolean isExecutable(Path path)`
- `static boolean isRegularFile(Path path)`
- `static boolean isDirectory(Path path)`
- `static boolean isSymbolicLink(Path path)`

  sprawdzają wybraną właściwość pliku określonego przez ścieżkę `path`.
- `static long size(Path path)`

  zwraca rozmiar pliku w bajtach.
- `A readAttributes(Path path, Class<A> type, LinkOption... options)`

  wczytuje atrybuty pliku typu `A`.

API *`java.nio.file.attribute.BasicFileAttributes`* 7

- `FileTime creationTime()`
- `FileTime lastAccessTime()`
- `FileTime lastModifiedTime()`
- `boolean isRegularFile()`
- `boolean isDirectory()`
- `boolean isSymbolicLink()`
- `long size()`
- `Object fileKey()`

  zwracają żądany atrybut.

API `java.nio.file.attribute.FileTime` 7

- `Instant toInstant()` 8

  zwraca instancję typu `Instant` reprezentującą ten sam czas co instancja `FileTime`.

2.4.6. Przeglądanie zawartości katalogu

Statyczna metoda `Files.list` miała metodę zwracającą obiekt `Stream<Path>`, pozwalający na odczytywanie zawartości katalogu. Zawartość katalogu jest odczytywana w sposób leniwy, dzięki czemu nawet przetwarzanie katalogów zawierających ogromną liczbę plików można wykonywać efektywnie.

Ponieważ odczyt zawartości katalogu wiąże się z wykorzystaniem zasobów systemowych, które następnie trzeba zamykać, operację tę należy wykonywać w bloku `try`:

```
try (Stream<Path> entries = Files.list(pathToDirectory))
{
   ...
}
```

Wywołanie metody `list` nie powoduje przejścia do katalogów podrzędnych. Aby przeanalizować także ich zawartość, trzeba skorzystać z metody `Files.walk`.

```
try (Stream<Path> entries = Files.walk(pathToRoot))
{
   // Przetwarzana jest także zawartość podkatalogów, w kolejności
   // odpowiadającej przeszukiwaniu w głąb
}
```

Poniżej przedstawiona została przykładowa analiza drzewa rozpakowanego pliku *src.zip*:

```
java
java/nio
java/nio/DirectCharBufferU.java
java/nio/ByteBufferAsShortBufferRL.java
java/nio/MappedByteBuffer.java
...
java/nio/ByteBufferAsDoubleBufferB.java
java/nio/charset
java/nio/charset/CoderMalfunctionError.java
java/nio/charset/CharsetDecoder.java
java/nio/charset/UnsupportedCharsetException.java
java/nio/charset/spi
java/nio/charset/spi/CharsetProvider.java
java/nio/charset/StandardCharsets.java
java/nio/charset/Charset.java
...
java/nio/charset/CoderResult.java
java/nio/HeapFloatBufferR.java
...
```

Jak widać, gdy tylko podczas analizy zostanie odnaleziony katalog, to jego zawartość zostanie przeanalizowana przed przetworzeniem innych plików lub katalogów na tym samym poziomie.

Głębokość, na jaką będzie analizowane drzewo katalogów, można ograniczyć, wywołując metodę `Files.walk(pathToRoot, depth)`. Obie metody `walk` dysponują parametrem typu `FileVisitOptions...`, umożliwiającym przekazywanie wielu opcji, choć obecnie dostępna jest tylko jedna: `FOLLOW_LINKS`, pozwalająca na stosowanie łączy symbolicznych.

Jeśli ścieżki zwracane przez metodę `walk` są filtrowane i jeśli kryterium filtrowania obejmuje atrybuty przechowywane w katalogu, takie jak rozmiar, czas utworzenia lub typ (plik, katalog, łącze symboliczne), to zamiast metody `walk` należy użyć metody `find`. Należy ją wywołać, przekazując do niej funkcję predykatu pobierającą ścieżkę oraz obiekt `BasicFileAttributes`. Jedyną zaletą takiego rozwiązania jest jego wydajność. Ponieważ zawartość katalogu i tak musi zostać odczytana, wszelkie atrybuty będą łatwo dostępne.

Poniższy fragment kodu używa metody `Files.walk` do skopiowania jednego katalogu do drugiego:

```
Files.walk(source).forEach(p ->
   {
      try
      {
         Path q = target.resolve(source.relativize(p));
         if (Files.isDirectory(p))
            Files.createDirectory(q);
         else
            Files.copy(p, q);
      }
      catch (IOException ex)
      {
         throw new UncheckedIOException(ex);
      }
   });
```

Niestety metody `Files.walk` nie można w prosty sposób używać do usuwania drzewa katalogów, gdyż przed usunięciem katalogu nadrzędnego trzeba usunąć wszystkie jego podkatalogi. W kolejnym punkcie rozdziału pokazano, jak można rozwiązać ten problem.

Metoda `Files.walk` zgłasza wyjątek, jeśli nie może odczytać zawartości któregokolwiek z katalogów. Chcąc odczytywać zawartości wyłącznie katalogów, które na to pozwalają, użyj metody `walkFileTree`, omówionej w następnym punkcie rozdziału.

2.4.7. Stosowanie strumieni katalogów

Jak przekonaliśmy się w poprzednim punkcie rozdziału, metoda `Files.walk` zwraca obiekt `Stream<Path>`, pozwalający na przeglądanie zawartości podkatalogów. Jednak czasami konieczne będzie uzyskanie bardziej szczegółowej kontroli nad procesem przeglądania drzewa katalogów. W takim przypadku można skorzystać z metody `Files.newDirectoryStream`. Zwraca ona obiekt typu `DirectoryStream`. Warto zauważyć, że nie jest to interfejs pochodny `java.util.stream.Stream`, lecz wyspecjalizowany interfejs przeznaczony do przeglądania drzew katalogów. Jest on typem pochodnym interfejsu `Iterable`, żeby można go było używać w rozszerzonej pętli `for`. Oto jak można to robić:

```
try (DirecotryStream<Path> entries = Files.newDirectoryStream(dir))
{
   for (Path entry : entries)
      // Przetwarzanie entries
}
```

Zastosowanie bloku try z zasobem daje gwarancję, że strumień katalogu zostanie prawidłowo zamknięty.

Poszczególne elementy analizowanego katalogu nie są zwracane w żadnej określonej kolejności.

Przeglądane pliki możemy filtrować za pomocą wzorca:

```
try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir, "*.java"))
```

Dostępne wzorce zostały przedstawione w tabeli 2.5.

Tabela 2.5. Wzorce filtrowania plików

Wzorzec	Opis	Przykład
*	Oznacza zero lub więcej znaków komponentu ścieżki	`*.java` oznacza wszystkie pliki Java w bieżącym katalogu
**	Oznacza zero lub więcej znaków, przekraczając granicę katalogu	`**.java` oznacza wszystkie pliki Java w dowolnym katalogu
?	Oznacza jeden znak	`????.java` oznacza wszystkie pliki Java, których nazwa składa się z czterech znaków (nie licząc rozszerzenia nazwy)
`[...]`	Oznacza zbiór znaków. Można stosować łącznik `[0-9]` i negację `[!0-9]`	`Test[0-9A-F].java` oznacza wszystkie pliki `Test`*x*`.java`, gdzie *x* jest jedną cyfrą szesnastkową
`{...}`	Oznacza znaki alternatywne rozdzielone przecinkami	`*.{java,class}` oznacza wszystkie pliki Java i pliki klas
\	Sekwencja specjalna pozwalająca używać znaków stosowanych w poprzednich wzorcach	`*\**` oznacza wszystkie pliki, których nazwy zawierają znak *

Jeśli używamy wzorców w systemie Windows, musimy w przypadku lewego ukośnika zastosować *podwójną* sekwencję specjalną: raz ze względu na składnię wzorca i drugi raz ze względu na składnię łańcuchów w języku Java: `Files.newDirectory ↪Stream(dir, "C:\\\\")`.

Jeśli chcemy przejrzeć wszystkie podkatalogi, to wywołujemy metodę `walkFileTree` i dostarczamy obiekt typu `FileVisitor`. Obiekt ten zostaje powiadomiony:

- gdy napotkany zostaje plik: `FileVisitResult visitFile(T path, BasicFileAttributes attrs);`
- zanim zostanie przetworzony katalog: `FileVisitResult preVisitDirectory ↪(T dir, IOException ex);`
- po przetworzeniu katalogu: `FileVisitResult postVisitDirectory(T dir, IOException ex);`

- gdy podczas próby przetworzenia pliku lub katalogu wystąpi błąd, na przykład na skutek próby otwarcia katalogu bez wystarczających uprawnień: `FileVisitResult visitFileFailed(T path, IOException ex)`.

W każdym z tych przypadków możemy określić, czy chcemy:

- kontynuować przetwarzanie następnego pliku: `FileVisitResult.CONTINUE`;
- kontynuować przetwarzanie, ale bez odwiedzania kolejnych elementów danego katalogu: `FileVisitResult.SKIP_SUBTREE`;
- kontynuować przeglądanie, ale bez przetwarzania rodzeństwa danego pliku: `FileVisitResult.SKIP_SIBLINGS`;
- zakończyć przeglądanie: `FileVisitResult.TERMINATE`.

Jeśli którakolwiek z metod zgłosi wyjątek, przeglądanie zostanie zakończone, a metoda `walkFileTree` zwróci ten wyjątek.

Interfejs `FileVisitor` jest typem generycznym, ale jest mało prawdopodobne, by zaszła potrzeba użycia go inaczej niż `FileVisitor<Path>`. Metoda `walkFileTree` jest gotowa zaakceptować `FileVisitor<? super Path>`, ale w praktyce `Path` nie ma zbyt wielu interesujących typów bazowych.

Klasa `SimpleFileVisitor` implementuje interfejs `FileVisitor` udostępniający proste metody, które kontynuują przeglądanie lub przerywają je w przypadku wystąpienia wyjątku. Można ją rozszerzać i przesłaniać jej metody w zależności od potrzeb.

Oto fragment kodu wyświetlający wszystkie podkatalogi danego katalogu:

```
Files.walkFileTree(Path.of("/"), new SimpleFileVisitor<Path>()
   {
      public FileVisitResult preVisitDirectory(Path path,
         BasicFileAttributes attrs) throws IOException
      {
         System.out.println(path);
         return FileVisitResult.CONTINUE;
      }
      public FileVisitResult postVisitDirectory(Path dir, IOException e)
      {
         return FileVisitResult.CONTINUE;
      }
      public FileVisitResult visitFileFailed(Path path, IOException e)
            throws IOException
      {
         if (e != null) throw e;
            return FileVisitResult.SKIP_SUBTREE;
         }
   });
```

Zauważmy, że w tym przypadku konieczne jest przesłonięcie metod `postVisitDirectory` oraz `visitFileFailed`. W przeciwnym razie przeglądanie zakończy się w momencie, gdy napotka katalog, którego nie mamy prawa otworzyć, bądź pliku, do którego nie mamy dostępu.

Zwróćmy również uwagę na to, że atrybuty ścieżki są przekazywane do metod `preVisit` ↪`Directory` i `visitFile` jako ich argument. Już wcześniej konieczne było wykonanie odpowiedniego wywołania systemowego w celu pobrania atrybutów, gdyż w przeciwnym razie nie można by było odróżnić plików od katalogów. Dzięki temu sami nie musimy wykonywać kolejnego wywołania.

Pozostałe metody interfejsu `FileVisitor` przydają się, gdy wchodząc do katalogu lub z niego wychodząc, musimy wykonać pewne działania. Na przykład gdy usuwamy drzewo katalogów, bieżący katalog usuwamy dopiero po usunięciu z niego wszystkich plików. Poniżej przedstawiony został fragment kodu służący do usuwania drzewa katalogu:

```
// Usuwanie drzewa katalogu, zaczynając od jego korzenia
Files.walkFileTree(root, new SimpleFileVisitor<Path>()
   {
      public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
         throws IOException
      {
         Files.delete(file);
         return FileVisitResult.CONTINUE;
      }
      public FileVisitResult postVisitDirectory(Path dir, IOException e)
         throws IOException
      {
         if (e != null) throw e;
         Files.delete(dir);
         return FileVisitResult.CONTINUE;
      }
   });
```

API java.nio.file.Files 7

- `static DirectoryStream<Path> newDirectoryStream(Path path)`
- `static DirectoryStream<Path> newDirectoryStream(Path path, String glob)`

  pobierają iterator umożliwiający przeglądanie plików i katalogów w danym katalogu. Druga z metod akceptuje jedynie elementy zgodne z podanym wzorcem `glob`.

- `static Path walkFileTree(Path start, FileVisitor<? super Path> visitor)`

  przegląda wszystkie elementy podrzędne danej ścieżki, stosując do nich obiekt `visitor`.

API java.nio.file.SimpleFileVisitor<T> 7

- `static FileVisitResult visitFile(T path, BasicFileAttributes attrs)`

  wywoływana, gdy przetwarzany jest plik lub katalog, zwraca `CONTINUE`, `SKIP_SUBTREE`, `SKIP_SIBLINGS` lub `TERMINATE`. Domyślna implementacja nie podejmuje żadnych działań, tym samym powodując kontynuację przeglądania.

- `static FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)`
- `static FileVisitResult postVisitDirectory(T dir, IOException exc)`

  wywoływane przed i po wizycie w katalogu. Domyślna implementacja nie podejmuje żadnych działań, tym samym powodując kontynuację przeglądania. Jeśli jednak wyjątek `exc` będzie różny od `null`, to zostanie on zgłoszony, co spowoduje przerwanie działania metody.

- `static FileVisitResult visitFileFailed(T path, IOException exc)`

  wywoływana, gdy podczas próby uzyskania informacji o danym pliku został zgłoszony wyjątek. Domyślna implementacja ponownie zgłasza ten wyjątek, powodując zakończenie przeglądania. Jeśli przeglądanie ma być kontynuowane, należy zastąpić metodę własną implementacją.

2.4.8. Systemy plików ZIP

Klasa `Paths` wyszukuje ścieżki w domyślnym systemie plików — na lokalnym dysku użytkownika. Możliwe jest jej wykorzystanie również dla innych systemów plików. Jednym z częściej spotykanych jest *system plików ZIP*. Jeśli `zipname` jest nazwą archiwum ZIP, to wywołanie

```
FileSystem fs = FileSystems.newFileSystem(Path.of(zipname));
```

ustanawia system plików zawierający wszystkie pliki należące do tego archiwum ZIP. Jeśli znamy nazwę pliku należącego do tego archiwum, to łatwo możemy skopiować go w poniższy sposób:

```
Files.copy(fs.getPath(sourceName), targetPath);
```

Metoda `fs.getPath` działa analogicznie do `Path.of` dla dowolnego systemu plików.

Aby wyświetlić wszystkie pliki należące do archiwum ZIP, przeglądamy drzewo plików w poniższy sposób:

```
FileSystem fs = FileSystems.newFileSystem(Paths.get(zipname), null);
Files.walkFileTree(fs.getPath("/"), new SimpleFileVisitor<Path>()
  {
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws
    ↪IOException
    {
      System.out.println(file);
      return FileVisitResult.CONTINUE;
    }
  });
```

Takie rozwiązanie jest łatwiejsze niż obsługa archiwów ZIP przedstawiona w punkcie 2.2.3, wymagająca użycia całego zestawu odpowiednich klas.

API `java.nio.file.FileSystems 7`

- `static FileSystem newFileSystem(Path path) 13`

  zwraca system plików utworzony przez pierwszego dostawcę, który akceptuje podaną ścieżkę. Domyślnie istnieje dostawca systemu plików ZIP akceptujący nazwy plików kończące się rozszerzeniem *.zip* lub *.jar*.

API `java.nio.file.FileSystem 7`

- `static Path getPath(String first, String... more)`

  tworzy ścieżkę, łącząc podane łańcuchy.

2.5. 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.

2.5.1. Wydajność plików mapowanych w pamięci

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 2.6 dla pliku *src.jar* (51 MB) znajdującego się w katalogu *lib* pakietu JDK.

Tabela 2.6. Czasy wykonywania operacji na pliku

Metoda	Czas
Zwykły strumień wejściowy	15,1 sekundy
Buforowany strumień wejściowy	0,8 sekundy
Plik o swobodnym dostępie	11,9 sekundy
Mapa pliku w pamięci	0,2 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.

Korzystanie z plików mapowanych w pamięci jest nieco mniej intuicyjne niż API plików o dostępie swobodnym. 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.

```
FileChannel channel = FileChannel.open(path, options);
```

Następnie uzyskujemy z kanału obiekt klasy `ByteBuffer`, 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 zgłoszenie 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.

Na przykład:

```
MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ_ONLY, 0,
➥channel.size());
```

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[] bytes, 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()
```

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

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

Dane z bufora zostają zapisane w pliku najpóźniej w momencie zamknięcia pliku.

Program przedstawiony na listingu 2.5 oblicza sumę kontrolną (CRC32) pliku. Suma taka jest często używana do kontroli naruszenia zawartości pliku. Uszkodzenie zawartości pliku powoduje zwykle zmianę wartości jego sumy kontrolnej. Pakiet `java.util.zip` zawiera klasę `CRC32` pozwalającą wyznaczyć sumę kontrolną sekwencji bajtów przy zastosowaniu następującej pętli:

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

Listing 2.5. memoryMap/MemoryMapTest.java

```
package memoryMap;

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

/**
 * Program obliczający sumę kontrolną pliku na cztery sposoby
 * Uruchamianie: java memoryMap.MemoryMapTest nazwapliku
 *
 * @version 1.03 2018-05-01
 * @author Cay Horstmann
 */
```

```
public class MemoryMapTest
{
   public static long checksumInputStream(Path filename) throws IOException
   {
      try (InputStream in = Files.newInputStream(filename))
      {
         var crc = new CRC32();
         boolean done = false;
         while (!done)
         {
            int c = in.read();
            if (c == -1)
               done = true;
            else
               crc.update(c);
         }
         return crc.getValue();
      }
   }

   public static long checksumBufferedInputStream(Path filename) throws IOException
   {
      try (var in = new BufferedInputStream(Files.newInputStream(filename)))
      {
         var crc = new CRC32();

         boolean done = false;
         while (!done)
         {
            int c = in.read();
            if (c == -1)
               done = true;
            else
               crc.update(c);
         }
         return crc.getValue();
      }
   }

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

         for (long p = 0; p < length; p++)
         {
            int c = file.readByte();
            crc.update(c);
         }
         return crc.getValue();
      }
   }

   public static long checksumMappedFile(Path filename) throws IOException
   {
      try (FileChannel channel = FileChannel.open(filename))
```

```
        {
           var 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("Strumień wejściowy:");
        long start = System.nanoTime();
        Path filename = Path.of(args[0]);
        long crcValue = checksumInputStream(filename);
        long end = System.nanoTime();
        System.out.println(Long.toHexString(crcValue));
        System.out.printf("%.3f sekund%n", (end - start) * 1E-9);

        System.out.println("Buforowany strumień wejściowy:");
        start = System.nanoTime();
        crcValue = checksumBufferedInputStream(filename);
        end = System.nanoTime();
        System.out.println(Long.toHexString(crcValue));
        System.out.printf("%.3f sekund%n", (end - start) * 1E-9);

        System.out.println("Plik o dostępie swobodnym:");
        start = System.nanoTime();
        crcValue = checksumRandomAccessFile(filename);
        end = System.nanoTime();
        System.out.println(Long.toHexString(crcValue));
        System.out.printf("%.3f sekund%n", (end - start) * 1E-9);

        System.out.println("Plik mapowany w pamięci:");
        start = System.nanoTime();
        crcValue = checksumMappedFile(filename);
        end = System.nanoTime();
        System.out.println(Long.toHexString(crcValue));
        System.out.printf("%.3f sekund%n", (end - start) * 1E-9);
     }
  }
```

Szczegóły obliczeń sumy kontrolnej CRC nie są dla nas istotne. Stosujemy ją jedynie jako przykład pewnej praktycznej operacji na pliku. (W praktyce zawartość plików nie byłaby odczytywana lub zapisywana bajt po bajcie, lecz większymi blokami. Dlatego różnice czasowe nie byłyby aż tak znaczące).

Program uruchamiamy w następujący sposób:

```
java memoryMap.MemoryMapTest nazwapliku
```

API `java.io.FileInputStream 1.0`

- `FileChannel getChannel()` **1.4**

  zwraca kanał dostępu do strumienia wejściowego.

API `java.io.FileOutputStream 1.0`

- `FileChannel getChannel()` **1.4**

  zwraca kanał dostępu do strumienia wyjściowego.

API `java.io.RandomAccessFile 1.0`

- `FileChannel getChannel()` **1.4**

  zwraca kanał dostępu do pliku.

API `java.nio.channels.FileChannel 1.4`

- `static FileChannel open(Path path, OpenOption... options)` **7**

  otwiera kanał dla pliku o podanej ścieżce dostępu. Domyślnie kanał zostaje otwarty dla odczytu. Parametr `options` może przyjmować jedną z następujących wartości typu wyliczeniowego `StandardOpenOption`: `WRITE`, `APPEND`,`TRUNCATE_EXISTING`, `CREATE`.

- `MappedByteBuffer map(FileChannel.MapMode mode, long position, long size)`

  tworzy w pamięci mapę fragmentu pliku. Parametr `mode` to jedna ze stałych `READ_ONLY`, `READ_WRITE` lub `PRIVATE` zdefiniowanych w klasie `FileChannel.MapMode`.

API `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 zgłoszony wyjątek `BufferUnderflowException`. Zwracają referencję do bufora.

- `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 wystarczająco miejsca, to nie są zapisywane żadne bajty i zostaje zgłoszony wyjątek `BufferOverflowException`. Zwraca referencję do bufora.

- *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`.

- `static ByteBuffer allocate(int capacity)`

  tworzy bufor o podanej pojemności.

- `static ByteBuffer wrap(byte[] values)`

  tworzy bufor w oparciu o podaną tablicę.

- `CharBuffer asCharBuffer()`

  tworzy nowy bufor na podstawie istniejącego, z którym ma wspólną zawartość, ale jednocześnie ma własną pozycję, ograniczenie i znacznik.

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`.

2.5.2. 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.

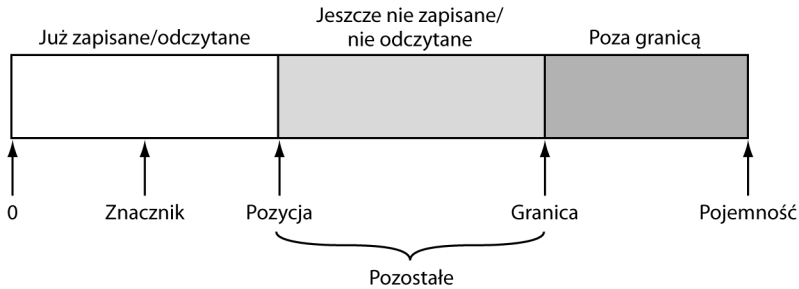
W tym podrozdziale omówimy krótko podstawowe operacje na obiektach typu `Buffer`. Bufor jest w rzeczywistości tablicą wartości tego samego typu. Abstrakcyjna klasa `Buffer` posiada klasy pochodne `ByteBuffer`, `CharBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer` i `ShortBuffer`.

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

W praktyce najczęściej używane są klasy `ByteBuffer` i `CharBuffer`. Na rysunku 2.9 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 2.9.
Bufor



Wymienione wartości spełniają następujący warunek:

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

Podstawowym przeznaczeniem bufora jest wykonywanie sekwencji operacji „najpierw zapisz, potem odczytuj". Na początku pozycja bufora jest równa 0, a granicą jest jego pojemność. Następnie bufor jest wypełniany danymi za pomocą metody `put` albo poprzez odczyt danych z kanału.

Kiedy dotrzesz do końca danych lub osiągniesz pojemność bufora, przechodzimy do fazy odczytu zawartości. Wywołaj metodę `flip` w celu przeniesienia limitu do bieżącej pozycji i ustawienia pozycji na 0.

Teraz możemy wywoływać metodę `get` lub zapisywać do kanału, dopóki metoda `remaining` zwraca wartość większą od zera (przy czym jej wynik jest różnicą *granica – pozycja*).

Po wczytaniu wszystkich wartości z bufora wywołujemy metodę `clear`, aby przygotować bufor do następnego cyklu zapisu. Wszystkie dane, które nie zostały odczytane z bufora, zostają przeniesione na jego początek. Pozycja jest ustawiana w miejscu, w którym te dane się kończą, a granica jest ustawiana na pojemność bufora.

Jeśli chcemy ponownie odczytać bufor, używamy metody `rewind` lub metod `mark`/`reset` (więcej na ich temat można się dowiedzieć z dokumentacji API).

Aby pobrać bufor, należy wywołać metodę statyczną, taką jak `ByteBuffer.allocate()` lub `ByteBuffer.wrap`.

Poniższa pętla używa bufora do przeniesienia danych z jednego kanału do drugiego:

```
ByteBuffer buffer = ByteBuffer.allocate(BUFFER_SIZE);
boolean doneReading = false;
boolean doneWriting = false;
while (!(doneReading && doneWriting))
{
   if (sourceChannel.read(buffer) == -1) doneReading = true;
   buffer.flip();
   destinationChannel.write(buffer);
   if (buffer.remaining() == 0) doneWriting = true;
   buffer.compact();
}
```

API java.nio.Buffer 1.4

- `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 clear()`

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

API java.nio.ByteBuffer 1.4

- `Buffer compact()`

  kopiuje wszelkie nieodczytane bajty na początek bufora oraz ustawia pozycję na następny bajt i granicę na pojemność bufora; zwraca `this`.

API java.nio.ReadableByteChannel 1.4

- `int read(ByteBuffer dst)`

  wypełnia pozostały obszar bufora dostępnymi bajtami i zwraca liczbę bajtów przeniesionych danych lub -1, jeśli zawartość kanału została w całości odczytana.

API java.nio.WritableByteChannel 1.4

- `int write(ByteBuffer src)`

  próbuje pobrać z bufora następne bajty i zwraca liczbę przeniesionych bajtów.

2.6. 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.

Załóżmy na przykład, że aplikacja zapisuje w pliku konfiguracyjnym ustawienia systemowe użytkownika. Jeśli użytkownik uruchomi dwie instancje tej aplikacji, może się zdarzyć, że obie będą chciały zapisać dane w pliku 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`:

```
FileChannel channel = FileChannel.open(path);
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 shared)
```

lub

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

Parametrowi `shared` nadajemy wartość `false`, aby zablokować dostęp do pliku zarówno dla operacji odczytu, jak i zapisu. W przypadku blokady *współdzielonej* parametr `shared` otrzymuje wartość `true`, 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`.

Zawsze upewnij się, że po wykonaniu operacji na pliku zwolniłeś blokadę. Najlepiej użyć w tym celu instrukcji `try` zarządzającej zasobami:

```
try (FileLock lock = channel.lock())
{
  dostęp do zablokowanego pliku lub segmentu
}
```

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, w tym w Linuksie, 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 obsługiwane globalnie na poziomie maszyny wirtualnej Javy. Dwa działające współbieżnie wątki nie mogą uzyskać blokady tego samego pliku. Metody `lock` i `tryLock` zgłoszą wyjątek `OverlappingFileLockException` w sytuacji, gdy inny wątek ma już blokadę danego pliku lub pokrywającego się zakresu.
- W niektórych systemach zamknięcie kanału zwalnia wszystkie blokady pliku. Dlatego też należy unikać tworzenia 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.
- Poprawność działania blokad jest gwarantowana wyłącznie dla programów napisanych w Javie. Programy napisane w innych językach mogą korzystać z innych, niezgodnych mechanizmów blokowania.

Program przedstawiony na listingu 2.6 pokazuje prosty przykład blokowania plików. Wielu użytkowników chce zarejestrować swoje hasła we wspólnym pliku. Program odczytuje plik, dodaje nazwę użytkownika oraz oblicza i zapisuje skrót hasła, a następnie zapisuje zmodyfikowany plik. Ważne jest, aby ta operacja była atomowa. W przeciwnym razie plik z hasłami może zostać uszkodzony, jeśli dwóch użytkowników będzie chciało jednocześnie go zaktualizować.

Listing 2.6. fileLock/FileLockDemo.java

```
package fileLock;

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;
import java.security.*;
import java.security.spec.*;
import java.util.*;

import javax.crypto.*;
import javax.crypto.spec.*;
```

```
/**
 * Ten program zapisuje nazwy użytkowników i hasła do pliku passwords.properties.
 * Plik jest aktualizowany w sposób atomowy. W systemie Linux spróbuj wykonać polecenie:
 *    for f in $(seq 1 100) ; do (java fileLock.FileLockDemo user$f secret$f &) ; done
 * Po zakończeniu wszystkich procesów Javy plik passwords.properties będzie zawierał wpis
 * dla każdego z nich.
 *
 * @version 1.0 2023-10-27
 * @author Cay Horstmann
 */


public class FileLockDemo
{
   public static void main(String[] args)
         throws IOException, NoSuchAlgorithmException, InvalidKeySpecException
   {
      Path path = Path.of("password.properties");
      FileChannel channel = FileChannel.open(path, StandardOpenOption.READ,
            StandardOpenOption.WRITE, StandardOpenOption.CREATE);
      try (FileLock lock = channel.lock())
      {
         // Odczyt bajtów z bufora
         ByteBuffer readBuffer = ByteBuffer.allocate((int) channel.size());
         channel.read(readBuffer);

         // Odczyt właściwości z bajtów
         Reader in = new StringReader(new String(readBuffer.array()));
         Properties props = new Properties();
         props.load(in);

         // Pobranie nazwy użytkownika i hasła z argumentów
         // lub danych wprowadzonych przez użytkownika
         String username;
         char[] password;
         if (args.length >= 2)
         {
            username = args[0];
            password = args[1].toCharArray();
         }
         else
         {
            Console console = System.console();
            username = console.readLine("Nazwa użytkownika: ");
            password = console.readPassword("Hasło: ");
         }

         // Wyznaczenie skrótu hasła
         final int ITERATIONS = 210_000;
         final int SALT_BYTES = 16;
         final int HASH_BYTES = 32;
         SecureRandom random = new SecureRandom();
         byte[] salt = new byte[SALT_BYTES];
         random.nextBytes(salt);

         var spec = new PBEKeySpec(password, salt, ITERATIONS, 8 * HASH_BYTES);
         SecretKeyFactory skf = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
         byte[] hash = skf.generateSecret(spec).getEncoded();
         Base64.Encoder encoder = Base64.getEncoder();
```

```
            // Zapisanie nazwy użytkownika i skrótu hasła we właściwości
            props.put(username, ITERATIONS + "|" + encoder.encodeToString(salt) + "|"
                  + encoder.encodeToString(hash));

            // Zapis właściwości w postaci bajtów
            StringWriter out = new StringWriter();
            props.store(out, null);
            byte[] outBytes = out.toString().getBytes();

            // Zapis bajtów do pliku
            channel.truncate(0);
            ByteBuffer writeBuffer = ByteBuffer.wrap(outBytes);
            channel.write(writeBuffer);
         }
      }
   }
```

Więcej informacji o określaniu skrótów haseł znajdziesz w rozdziale 9. Nie ma to nic wspólnego z blokowaniem plików, ale nie chciałem pokazywać programu, który zapisuje hasła bez ich zaszyfrowania.

Przedstawiony program ilustruje ograniczenia API blokowania plików. Jest ono powiązane z API klasy `FileChannel`, które opiera się na odczycie i zapisie buforów bajtów. Program musi przekształcać te bufory bajtów na obiekty `Reader` i `Writer`.

Alternatywne rozwiązanie mogłoby polegać na użyciu klasy `RandomAccessFile`, której metoda `getChannel` zwraca kanał do pliku. Jednak takie rozwiązanie jest równie niewygodne. W jego przypadku należy wczytać plik do tablicy bajtów i przygotować ją do zapisu zmodyfikowanego pliku.

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. Parametr `shared` ma wartość `true` dla blokady współdzielonej, `false` dla wyłącznej.

- `FileChannel truncate(long size)`

  przycina plik do zadanej wielkości, o ile jest większy.

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

- `void close()` **1.7**

  zwalnia blokadę.

API **java.io.RandomAccessFile 1.0**

- `FileChannel getChannel()` **1.4**

  zwraca kanał do tego pliku o dostępie swobodnym.

2.7. 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.

W dalszej części tego podrozdziału przedstawię składnię wyrażeń regularnych stosowanych w API Javy oraz opiszę sposoby korzystania z wyrażeń regularnych.

2.7.1. Składnia wyrażeń regularnych

Zacznijmy od czegoś prostego. 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.

W wyrażeniach regularnych znaki reprezentują same siebie, chyba że mamy do czynienia z jednym z wymienionych poniżej znaków specjalnych:

```
. * + ? { | ( ) [ \ ^ $
```

Na przykład wyrażenie regularne `Java` zostanie dopasowane wyłącznie do łańcucha znaków `Java`.

Symbol `.` jest dopasowywany do dowolnego znaku. Na przykład wyrażenie regularne `.a.a` zostanie dopasowane zarówno do łańcucha `Java`, jak i `data`.

Symbol * oznacza, że poprzedzający go element wyrażenia może wystąpić zero lub więcej razy, a symbol + oznacza, że musi on wystąpić co najmniej raz. Końcówka ? informuje, że dany element jest opcjonalny (może wystąpić zero lub jeden raz). Na przykład wyrażenie regularne `be+s?` pasuje do takich łańcuchów znaków jak `be`, `bee` oraz `bees`. Inne liczby powtórzeń można określić za pomocą nawiasów klamrowych `{ }`, zgodnie z opisem zamieszczonym w tabeli 2.7.

Tabela 2.7. Składnia wyrażeń regularnych

Składnia	Objaśnienie	Przykład
Znaki		
c, ale żadne z: `. * + ? { | ( ) [ \ ^ $`	Znak *c*.	`J`
.	Dowolny znak oprócz kończącego wiersz (lub dowolny znak, jeśli znacznik `DOTALL` został ustawiony).	
\X	Dowolny „rozszerzony klaster grafemów" Unicode, który jest traktowany jako znak lub symbol.	
`\x{`*p*`}`	Znak Unicode o szesnastkowej wartości *p*.	`\x{1D546}`
`\u`*hhhh*, `\x`*hh*, `\0`*o*, `\0`*oo*, `\0`*ooo*	Znak o kodzie, którego wartość została podana w notacji szesnastkowej lub ósemkowej.	`\uFEFF`
`\a`, `\e`, `\f`, `\n`, `\r`, `\t`	Znaki sterujące: alertu (`\x{7}`), sekwencji sterującej (`\x{1B}`), końca strony (`\x{B}`), nowego wiersza (`\x{A}`), powrotu karetki (`\x{D}`) i tabulacji (`\x{9}`).	`\n`
`\c`*c*, gdzie *c* należy do `[A-Z]` bądź jest jednym ze znaków `@ [ \ ] ^ _ ?`	Znak sterujący odpowiadający znakowi *c*.	`\cH` to znak cofnięcia (ang. *backspace*; `\x{8}`)
`\`*c*, gdzie *c* nie należy do `[A-Za-z0-9]`	Znak *c*.	`\\`
`\Q . . . \E`	Wszystko pomiędzy początkiem i końcem cytatu.	`\Q(...)\E` pasuje do łańcucha `(...)`

Tabela 2.7. Składnia wyrażeń regularnych — ciąg dalszy

Składnia	Objaśnienie	Przykład
Klasy znaków		
[C_1C_2. . .], gdzie C_i jest znakiem, zakresem znaków *c* – *d* lub klasą znaków	Dowolny ze znaków reprezentowanych przez C_1C_2. . ..	`[0-9+-]`
[^. . .]	Dopełnienie klasy znaków.	`[^\d\s]`
[. . .&&. . .]	Część wspólna (przecięcie) dwóch klas znaków.	`[\p{L}&&[^A-Za-z]]`
`\p{...}`, `\P{...}`	Predefiniowana klasa znaków (patrz tabela 2.7); dopełnienie predefiniowanej klasy znaków.	`\p{L}` odpowiada literze Unicode, podobnie jak `\pL` — można pominąć nawiasy klamrowe wokół pojedynczej litery
`\d`, `\D`	Cyfry (`[0-9]` lub `\p{Digit}` w przypadku użycia flagi `UNICODE_CHARACTER_CLASS`); dopełnienie, czyli znak, który nie jest cyfrą.	`\d+` — sekwencja cyfr
`\w`, `\W`	Znak słowa (`[a-zA-Z0-9_]` lub znak słowa w Unicode, jeśli została użyta flaga `UNICODE_CHARACTER_CLASS`).	
`\s`, `\S`	Znak odstępu (`[ \t\n\r\f\x0B]` lub `\p{IsWhiteSpace}`, jeśli została użyta flaga `UNICODE_CHARACTER_CLASS`); znak, który nie jest odstępem.	`\s*,\s*` — przecinek opcjonalnie otoczony znakami odstępu
`\h`, `\v`, `\H`, `\V`	Odstęp w poziomie, odstęp w pionie; dopełnienia tych znaków.	
Sekwencje i alternatywy		
XY	Dowolny łańcuch z X, po którym następuje dowolny łańcuch z Y.	`[1-9][0-9]* – liczba dodatnia bez początkowego zera`
X|Y	Dowolny łańcuch z X lub Y.	`http|ftp`
Grupowanie		
(X)	Przechwycenie dopasowania X.	`'([^']*)' – tekst w apostrofach`
\n	Dopasowanie n-tej grupy.	`(['"]).*\1 – pasuje do "Fred" lub 'Fred', lecz nie do "Fred'`
(?<*nazwa*>*X*)	Przechwytuje dopasowanie *X*, nadając mu podaną *nazwę*.	`'(?<id>[A-Za-z0-9]+)'` przechwytuje dopasowanie, nadając mu nazwę `id`

Tabela 2.7. Składnia wyrażeń regularnych — ciąg dalszy

Składnia	Objaśnienie	Przykład
Grupowanie		
`\k<`*nazwa*`>`	Grupa o podanej *nazwie*.	`\k<id>` dopasowuje grupę o nazwie `id`
`(?:`*X*`)`	Zastosowane nawiasów bez przechwytywania *X*.	W wyrażeniu regularnym `(?:http|ftp)://(.*)` dopasowanie po `://` jest grupą `\1`
`(?`$f_1f_2 \ldots$`:`*X*`)`, `(?`$f_i \ldots$`-`$f_k \ldots$`:`*X*`)`, gdzie f_i należy do `[dimsuUx]`	Dopasowuje *X*, lecz go nie przechwytuje, używając lub nie używając (jeśli zastosowano `-`) określonych flag.	`(?i:jpe?g)` — w dopasowaniu nie będzie uwzględniana wielkość liter
Inne `(? ...)`	Patrz dokumentacja API `Pattern`.	
Kwantyfikatory		
X`?`	Opcjonalnie *X*.	`\x?` to opcjonalny znak +
X`*`, *X*`+`	*X*, 0 lub więcej razy oraz *X* co najmniej jeden raz.	`[1-9][0-9]+` to liczba całkowita większa od 10.
X`{`*n*`}` *X*`{`*n*`,}` *X*`{`*n*`,`*m*`}`	X *n* razy, co najmniej *n* razy, pomiędzy *n* i *m* razy.	`[0-7]{1,3}` od jednej do trzech cyfr ósemkowych.
Q`?`, gdzie *Q* jest wyrażeniem z kwantyfikatorem	Kwantyfikator oporny; próbuje znaleźć jak najkrótsze dopasowanie, zanim spróbuje dobrać dłuższe.	`.*(<.+?>).*` — przechwytuje najkrótszą sekwencję, umieszczoną pomiędzy nawiasami kątowymi
Q`+`, gdzie *Q* jest wyrażeniem z kwantyfikatorem	Kwantyfikator zachłanny, znajdujący najdłuższe dopasowanie, które nie wymaga cofania.	`'[^']*+'` — odnajduje łańcuchy w apostrofach i szybko przerywa dopasowywanie, gdy łańcuch nie ma zamykającego apostrofu
Granice dopasowania		
`^`, `$`	Początek, koniec wejścia (lub początek, koniec wiersza w trybie wielowierszowym).	`^Java$` — pasuje do wpisanego słowa `Java` lub wiersza o tej zawartości
`\A`, `\Z`, `\z`	Początek wejścia, koniec wejścia, bezwzględny koniec wejścia (niezmienane w trybie wielowierszowym)	
`\b`, `\B`	Granica słowa, granica inna niż słowa.	`\bJava\b` — pasuje do słowa `Java`

Tabela 2.7. Składnia wyrażeń regularnych — ciąg dalszy

Składnia	Objaśnienie	Przykład
Granice dopasowania		
`\b{g}`	Granica klastra grafemów.	Przydatne w metodzie `split` do dzielenia łańcucha na klastry grafemów.
`\R`	Znak nowego wiersza Unicode	
`\G`	Koniec poprzedniego dopasowania.	

Symbol | oznacza alternatywę: wyrażenie `.(oo|ee)f` pasuje do `beef` lub `woof`. Zwróć uwagę na nawiasy — bez nich wyrażenie `.oo|eef` oznaczałoby alternatywę pomiędzy `.oo` a `eef`. Nawiasy służą również do grupowania (patrz punkt 2.7.4).

Klasa znaków to zbiór alternatywnych znaków ujętych w nawiasy kwadratowe, na przykład `[Jj]`, `[0-9]`, `[A-Za-z]` lub `[^0-9]`. W zapisie klasy znaków znak `-` (minus) oznacza zakres (wszystkie znaki, których wartości Unicode mieszczą się pomiędzy dwiema granicami). Jednak znak `-` umieszczony na samym początku lub na samym końcu klas oznacza dosłownie znak minusa. Znak `^` na początku klasy znaków oznacza dopełnienie (wszystkie znaki oprócz wymienionych z nim).

Istnieje wiele predefiniowanych klas znaków, takich jak `\d` (cyfry) czy `\s` (znaki odstępu) — zostały one opisane w tabeli 2.7. Bardziej złożone klasy znaków można opisać za pomocą składni `\p`, przedstawionej w tabeli 2.8. Na przykład `\p{Sc}` oznacza klasę znaków będących symbolami walut w Unicode.

Tabela 2.8. Wstępnie zdefiniowane nazwy klas znaków

Nazwa klasy znaków	Objaśnienie
klasaPosix	*klasaPosix* to jedna z wartości: `Lower`, `Upper`, `Alpha`, `Digit`, `Alnum`, `Punct`, `Print`, `Graph`, `Cntrl`, `Xdigit`, `Space`, `Blank` albo `ASCII`, interpretowana jako klasa POSIX lub Unicode, w zależności od tego, czy została użyta flaga `UNICODE_CHARACTER_CLASS`, czy nie.
`Is`*Skrypt*, `sc=`*Skrypt*, `script=`*Skrypt*	*Skrypt* jest nazwą skryptu Unicode, akceptowaną przez metodę `Character.UnicodeScript.forName`.
`In`*Blok*, `blk=`*Blok*, `block=`*Blok*	*Blok* jest nazwą bloku znaków Unicode, akceptowaną przez metodę `Character.UnicodeBlock.forName`.
Kategoria, `In`*Kategoria*, `gc=`*Kategoria*, general_category=*Kategoria*	Jedno lub dwuliterowa nazwa ogólnej kategorii znaków Unicode.
`Is`*Właściwość*	*Właściwość* jest jedną z wartości `Alphabetic`, `Ideographic`, `Letter`, `Lowercase`, `Uppercase`, `Titlecase`, `Punctuation`, `Control`, `White_Space`, `Digit`, `Hex_Digit`, `Join_Control`, `Noncharacter_Code_Point`, `Assigned`.
`java`*Metoda*	Wywołuje metodę `Character.is`*Metoda* (nie może być przestarzała).

Znaki ^ i $ reprezentują odpowiednio początek i koniec danych wejściowych.

Jeśli chcesz znaku ., *, +, ?, {, |, (,), [, \, ^ lub $ w dosłownym znaczeniu, poprzedź go znakiem odwrotnego ukośnika (\). Wewnątrz klasy znaków trzeba poprzedzać w ten sposób jedynie znaki [i -, pod warunkiem że zostaną one odpowiednio zapisane względem znaków], -, ^. Na przykład wyrażenie []^-] definiuje klasę zawierającą wszystkie trzy znaki umieszczone w nawiasach kwadratowych.

Kiedy wyrażenie regularne jest zapisane w literale łańcuchowym, to każdy odwrotny ukośnik należy poprzedzić drugim odwrotnym ukośnikiem. Jeśli o nim zapomnisz, zazwyczaj spowoduje to błąd, gdyż w literałach łańcuchowych nie można umieszczać takich sekwencji jak \$ lub \.. Jeśli jednak chcesz dopasować granicę słowa i przez przypadek zamiast zapisu \\b użyjesz \b, to będziesz mieć problem, ponieważ \b jest prawidłową sekwencją oznaczającą znak *backspace*.

W prawidłowym stosowaniu znaków odwrotnego ukośnika w literałach łańcuchowych może pomóc używane zintegrowane środowisko programistyczne (IDE). Wklej łańcuch wyrażenia regularnego do pustego łańcucha znaków, a wszystkie znaki odwrotnego ukośnika zostaną automatycznie powielone.

Zamiast znaków odwrotnego ukośnika można także używać sekwencji \Q i \E. Na przykład oba wyrażenia — \(\$0\.99\) oraz \Q($0.99)\E — zostaną dopasowane do łańcucha ($0.99).

Jeśli w wyrażeniu regularnym może się pojawić wiele znaków specjalnych, można zadbać o jego poprawność, używając metody `Parse.quote(str)`. Metoda ta zapisuje przekazany łańcuch znaków z wyrażeniem regularnym pomiędzy sekwencjami \Q i \E, a przy okazji zwraca uwagę na sytuację, w której sam przekazany łańcuch `str` zawiera sekwencję \E.

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 2.7. Więcej informacji na temat składni wyrażeń regularnych znajdziesz w dokumentacji klasy `Pattern` lub książce *Wyrażenia regularne. Wprowadzenie* autorstwa Michaela Fitzgeralda (Wydawnictwo Helion, 2013).

2.7.2. Testowanie dopasowania

Ogólnie rzecz biorąc, istnieją dwa sposoby stosowania wyrażeń regularnych. Pierwszy z nich sprawdza, czy łańcuch znaków pasuje do wyrażenia regularnego, a drugi pozwala stwierdzić, czy wyrażenie można dopasować do jakiegoś fragmentu łańcucha.

W pierwszym przypadku należy skorzystać ze statycznej metody `matches`, jak w poniższym przykładzie:

```
String regex = "-?\\d+";
CharSequence input = ...;
```

```
if (Pattern.matches(regex, input))
{
   ...
}
```

Jeśli tego samego wyrażenia regularnego trzeba użyć więcej niż raz, to w celu poprawienia wydajności działania kodu należy je skompilować, a następnie dla każdego testowanego łańcucha znaków utworzyć obiekt `Matcher`:

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

Jeśli wyrażenie zostanie pomyślnie dopasowane, to następnie można pobrać położenie dopasowanych grup — więcej informacji na ten temat podaję w kolejnym punkcie rozdziału.

Aby sprawdzić, czy dane wejściowe *zawierają* dopasowanie, należy skorzystać z metody `find`:

```
if (matcher.find()) ...
```

Metody `find` i `match` modyfikują stan obiektu `Matcher`. Jeśli chcemy sprawdzić, czy danemu obiektowi `Matcher` udało się już znaleźć dopasowanie, wystarczy posłużyć się metodą `hasMatch`.

Ten wzorzec można przekształcić w predykcję:

```
Pattern digits = Pattern.compile("[0-9]+");
List<String> strings = List.of("December", "31st", "1999");
List<String> matchingStrings
  = strings.stream().filter(digits.asMatchPredicate()).toList(); // ["1999"]
```

Wyniki zwrócone przez ten fragment kodu zawierają wszystkie łańcuchy znaków pasujące do wyrażenia regularnego.

Do sprawdzenia, czy łańcuch znaków zawiera fragment pasujący do wyrażenia regularnego, należy użyć metody `asPredicate`:

```
List<String> stringsContainingMatch
   = strings.stream().filter(digits.asPredicate()).toList(); // ["31st", "1999"]
```

2.7.3. Znajdowanie wszystkich dopasowań w łańcuchu

W tym punkcie rozdziału przeanalizujemy kolejny często występujący przypadek użycia wyrażeń regularnych: znajdowanie wszystkich dopasowań występujących w łańcuchu znaków. Należy do tego użyć następującej pętli:

```
String input = ...;
Matcher matcher = pattern.matcher(input);
while (matcher.find())
{
   String match = matcher.group();
   int matchStart = matcher.start();
   int matchEnd = matcher.end();
   ...
}
```

W ten sposób można obsłużyć kolejno wszystkie dopasowania. Jak widać w przykładzie, dla każdego dopasowania można pobrać zarówno pasujący fragment łańcucha znaków, jak i jego położenie w łańcuchu wejściowym.

Jeszcze bardziej eleganckim rozwiązaniem jest wywołanie metody `results`, która zwraca strumień typu `Stream<MatchResult>`. Interfejs `MatchResult`, podobnie jak klasa `Matcher`, udostępnia metody `group`, `start` oraz `end`. (W rzeczywistości okazuje się, że klasa `Matcher` implementuje ten interfejs). Oto przykład, jak można pobrać listę wszystkich dopasowań:

```
List<String> matches = pattern.matcher(input)
   .results()
   .map(MatchResult::group)
   .toList();
```

Jeśli dysponujemy plikiem zawierającym dane, to możemy użyć metody `Scanner.findAll`, aby pobrać strumień typu `Stream<MatchResult>` bez konieczności wcześniejszego wczytywania zawartości pliku do łańcucha znaków:

```
Scanner in = new Scanner(path, "UTF_8");
Stream<String> words = in.findAll("\\pL+")
   .map(MatchResult::group);
```

Program przedstawiony na listingu 2.7 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 match.HrefMatch http://horstmann.com
```

Listing 2.7. match/HrefMatch.java

```
package match;

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 match.HrefMatch adresURL
 *
 * @version 1.05 2023-08-16
 * @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 = "https://openjdk.org/";
```

```
            // Wczytuje zawartość strony
            InputStream in = URI.create(urlString).toURL().openStream();
            var input = new String(in.readAllBytes());

            // Wyszukuje wszystkie wystąpienia wzorca
            String patternString = "<a\\s+href\\s*=\\s*(\"[^\"]*\"|[^\\s>]*)\\s*>";
            Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);

pattern.matcher(input).results().map(MatchResult::group).forEach(System.out::println);
         } catch (IOException | PatternSyntaxException e)
         {
            e.printStackTrace();
         }
      }
}
```

2.7.4. Grupy

W przypadkach kiedy trzeba pobierać różne fragmenty dopasowania i na nich operować, powszechnie stosuje się grupy. W ramach przykładu załóżmy, że jednym z kilku elementów faktury jest wiersz zawierający nazwę, ilość oraz cenę jednostkową o postaci takiej jak:

```
Blackwell Toaster    USD29.95
```

Oto wyrażenie regularne zawierające grupę dla każdego z komponentów przedstawionego wiersza tekstu:

```
(\p{Alnum}+(\s+\p{Alnum}+)*)\s+([A-Z]{3})([0-9.]*)
```

Po wykonaniu dopasowania można pobrać *n*-tą grupę z obiektu `Matcher`, używając następującego kodu:

```
String contents = matcher.group(n);
```

Grupy są liczone od 1 i są zapisane w kolejności, w jakiej w wyrażeniu regularnym występowały ich nawiasy otwierające (grupą o indeksie 0 jest cały łańcuch wejściowy):

```
Matcher matcher = pattern.matcher(input);
if (matcher.matches())
{
   item = matcher.group(1);
   currency = matcher.group(3);
   price = matcher.group(4);
}
```

Grupa o indeksie 2 nas nie interesuje — została ona utworzona ze względu na użycie nawiasów koniecznych do zapewniania powtórzeń. Dla poprawienia przejrzystości wyników można użyć grupy nieprzechwytującej, przedstawionej w poniższym przykładzie:

```
(\p{Alnum}+(?:\s+\p{Alnum}+)*)\s+([A-Z]{3})([0-9.]*)
```

Jeszcze lepszym rozwiązaniem może być zastosowanie grup nazwanych:

```
(?<item>\p{Alnum}+(\s+\p{Alnum}+)*)\s+(?<currency>[A-Z]{3})(?<price>[0-9.]*)
```

W takim przypadku poszczególne dopasowane grupy można pobierać na podstawie ich nazw:

```
item = matcher.group("item");
```

Metody `start` i `end` pozwalają pobrać położenie dopasowania grupy w łańcuchu wejściowym:

```
int itemStart = matcher.start("item");
int itemEnd = matcher.end("item");
```

W języku Java 20 nazwane grupy są także obsługiwane przez klasę `MatchResult`.

Metoda `namedGroups` zwraca obiekt `Map<String, Integer>`, odwzorowujący nazwy grup na liczby.

Jeśli grupa wchodzi w skład powtórzenia, jak w powyższym przykładzie — `(\s+\p{Alnum}+)*` — to pobranie wszystkich jej dopasowań nie jest możliwe. Wówczas metoda `group` zwraca jedynie ostatnie dopasowanie, co rzadko kiedy będzie przydatne. W takim przypadku trzeba pobrać całe wyrażenie, używając innej grupy.

W wyrażeniach regularnych można umieszczać komentarze. Taki komentarz zaczyna się od znaku # i obejmuje cały tekst do końca wiersza. Komentarze szczególnie dobrze sprawdzają się w blokach tekstowych, takich jak ten:

```
var regex = """
([1-9]|1[0-2]) #hours
:([0-5][0-9]) #minutes
[ap]m""";
```

Program przedstawiony na listingu 2.8 prosi o podanie wyrażenia, a następnie łańcucha znaków, w których chcemy sprawdzić dopasowania. Program wyświetla informacje o tym, czy łańcuch został dopasowany. Jeśli dopasowanie istnieje, a wyrażenie regularne zawiera grupy, to ich granice zostają wyświetlone jak w poniższym przykładzie:

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

Listing 2.8. regex/RegexTest.java

```
package regex;

/**
 *
 * Ten program testuje dopasowanie wyrażeń regularnych. Wprowadź wzorzec i łańcuch znaków
 * do dopasowania lub naciśnij Enter, aby zakończyć. Jeśli wzorzec zawiera grupy,
 * po dopasowaniu zostaną wyświetlone ich granice.
 *
 * @version 1.03 2018-05-01
 * @author Cay Horstmann
 */

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

```
public class RegexTest
{
   public static void main(String[] args) throws PatternSyntaxException
   {
      var in = new Scanner(System.in);
      System.out.println("Wpisz wyrażenie regularne: ");
      String patternString = in.nextLine();

      Pattern pattern = Pattern.compile(patternString);

      while (true)
      {
         System.out.println("Wpisz łańcuch do dopasowania: ");
         String input = in.nextLine();
         if (input == null || input.equals(""))
            return;
         Matcher matcher = pattern.matcher(input);
         if (matcher.matches())
         {
            System.out.println("Dopasowanie!");
            int g = matcher.groupCount();
            if (g > 0)
            {
               for (int i = 0; i < input.length(); i++)
               {
                  // Wyświetlamy puste grupy
                  for (int j = 1; j <= g; j++)
                     if (i == matcher.start(j) && i == matcher.end(j))
                        System.out.print("()");
                  // Wyświetlamy ( dla niepustych grup - początek
                  for (int j = 1; j <= g; j++)
                     if (i == matcher.start(j) && i != matcher.end(j))
                        System.out.print("(");
                  System.out.print(input.charAt(i));
                  // Wyświetlamy ) dla niepustych grup - koniec
                  for (int j = 1; j <= g; j++)
                     if (i + 1 != matcher.start(j) && i + 1 == matcher.end(j))
                        System.out.print(")");
               }
               System.out.println();
            }
         }
         else
            System.out.println("Brak dopasowania");
      }
   }
}
```

2.7.5. Podział w miejscach wystąpienia separatora

Czasami konieczne jest podzielenie łańcucha wejściowego w miejscach wystąpienia określonego separatora i zachowanie wszystkich uzyskanych w ten sposób fragmentów. Zadanie to ułatwia metoda `Pattern.split`. Zwraca ona tablicę łańcuchów, z której zostaną usunięte wystąpienia separatora:

```
String input = "1, 2 , 3,4";
Pattern commas = Pattern.compile("\\s*,\\s*");
String[] tokens = commas.split(input); // ["1", "2", "3", "4"]
```

Jeśli tak uzyskanych elementów jest wiele, to można je pobrać w sposób leniwy:

```
Stream<String> tokens = commas.splitAsStream(input);
```

Aby także pobrać znaki lub sekwencje, które wyznaczały granice podziału, należy użyć metody `splitWithDelimiters`:

```
tokens = commas.splitWithDelimiters(input, -1); // ["1", ", ", "2", " , ", "3", ",", "4"]
```

Jeśli drugi argument jest liczbą dodatnią *n*, to wzorzec separatora został zastosowany co najwyżej *n*-1 razy, a ostatnim elementem jest pozostały łańcuch znaków. W przeciwnym razie wzorzec podziału zostanie zastosowany tyle razy, ile to możliwe. W przypadku przekazania wartości 0 jako drugiego argumentu końcowe puste łańcuchy są odrzucane.

Jeśli ani wstępna kompilacja wyrażenia regularnego, ani pobieranie leniwe nie mają znaczenia, można także skorzystać z metod `split` lub `splitWithDelimiters` klasy `String`:

```
tokens = input.split("\\s*,\\s*");
```

Łatwo można zapomnieć, że argument metody `split` jest wyrażeniem regularnym. Na przykład wywołanie:

```
"com.horstmann.corejava".split(".")
```

nie powoduje podzielenia łańcucha znaków w miejscach występowania kropek. Zamiast tego separatorem będzie każdy znak, więc w efekcie zostanie zwrócona pusta tablica!

Kropkę w wyrażeniu regularnym należy poprzedzić znakiem odwrotnego ukośnika, a to oznacza, że w literale łańcuchowym trzeba go zabezpieczyć, poprzedzając kolejnym ukośnikiem, jak w poniższym przykładzie:

```
"com.horstmann.corejava".split("\\.")
```

Ewentualnie można skorzystać z metody `Pattern.quote`:

```
"com.horstmann.corejava".split(Pattern.quote("."))
```

Jeśli dane wejściowe są zapisane w pliku, to można użyć obiektu klasy `Scanner`:

```
Scanner in = new Scanner(path, StandardCharsets.UTF_8);
in.useDelimiter("\\s*,\\s*");
Stream<String> tokens = in.tokens();
```

2.7.6. Zastępowanie dopasowań

Aby zastąpić wszystkie dopasowania wyrażenia regularnego podanym łańcuchem znaków, należy wywołać metodę `replaceAll` obiektu klasy `Matcher`:

```
Matcher matcher = commas.matcher(input);
String result = matcher.replaceAll(",");
   // Normalizacja przecinków
```

Jeśli nie interesuje nas wstępna kompilacja wyrażenia, można także skorzystać z metody `replaceAll` klasy `String`:

```
String result = input.replaceAll("\\s*,\\s*", ",");
```

Łańcuch zastępujący może zawierać referencje grup wzorca, `$n`, bądź ich nazwy, `${nazwa}`. Są one zastępowane zawartością odpowiednich dopasowanych grup:

```
String result = "3:45".replaceAll(
"(\\d{1,2}):(?<minutes>\\d{2})",
"$1 godzin i ${minutes} minut");
   // Wynikowy łańcuch ma postać "3 godziny i 45 minut"
```

Aby w łańcuchu zastępującym umieścić znak $ lub \, należy je poprzedzić znakiem odwrotnego ukośnika (\); ewentualnie można skorzystać z metody `Matcher.quoteReplacement`:

```
matcher.replaceAll(Matcher.quoteReplacement(str))
```

Jeśli chcemy wykonać operacje bardziej złożone od połączenia pogrupowanych dopasowań, to zamiast łańcucha zamiennika możemy przekazać funkcję. Funkcja ta pobiera obiekt `MatchResult` i zwraca łańcuch znaków. Na przykład poniższy fragment kodu zastępuje wszystkie słowa składające się z co najmniej czterech liter tymi samymi słowami zapisanymi dużymi literami:

```
String result = Pattern.compile("\\pL{4,}")
   .matcher("Czarny kot w kropki bordo")
   .replaceAll(m -> m.group().toUpperCase());
   // Zwraca "CZARNY kot w KROPKI BORDO"
```

Metoda `replaceFirst` zastępuje tylko pierwsze wystąpienie wzorca.

2.7.7. Flagi

Istnieje kilka *flag*, które zmieniają sposób działania wyrażeń regularnych. Można je określać podczas kompilacji wyrażeń:

```
Pattern pattern = Pattern.compile(regex,
   Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CHARACTER_CLASS);
```

Ewentualnie można je także podawać w wyrażeniu:

```
String regexp = "(?iU:wyrażenie)";
```

Obsługiwane są poniższe znaczniki:

- `Pattern.CASE_INSENSITIVE` lub `i` — dopasowanie niezależnie od wielkości liter. Domyślnie dotyczy to tylko znaków US ASCII.
- `Pattern.UNICODE_CASE` lub `U` — zastosowane w połączeniu z `CASE_INSENSITIVE`; dotyczy wszystkich znaków Unicode.
- `Pattern.UNICODE_CHARACTER_CLASS` — używane mają być klasy znaków UNICODE, a nie POSIX. Oznacza zastosowanie `UNICODE_CASE`.

- `Pattern.MULTILINE` lub `m` — `^` i `$` oznaczają początek i koniec wiersza, a nie całego wejścia.
- `Pattern.UNIX_LINES` lub `d` — tylko `'\n'` jest rozpoznawany jako zakończenie wiersza podczas dopasowywania do `^` i `$` w trybie wielowierszowym.
- `Pattern.DOTALL` lub `s` — symbol `.` oznacza wszystkie znaki, w tym końca wiersza.
- `Pattern.COMMENTS` lub `x` — odstępy i komentarze (od znaku `#` do końca wiersza) będą ignorowane.
- `Pattern.LITERAL` — wzorzec jest traktowany dosłownie i musi zostać dopasowany w dokładnie takiej samej postaci, z ewentualnymi różnicami w wielkości liter.
- `Pattern.CANON_EQ` — bierze pod uwagę kanoniczny odpowiednik znaków Unicode. Na przykład znak `u`, po którym następuje znak ¨ (diareza), zostanie dopasowany do znaku ü.

Ostatnich dwóch znaczników nie można używać wewnątrz wyrażeń regularnych.

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. Parametr `flags` może mieć ustawiony jeden lub kilka bitów: `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)`
- `String[] splitWithDelimiters(CharSequence input, int limit)`
- `Stream<String> splitAsStream(CharSequence input)` **8**

  rozbija łańcuch wejściowy na tokeny, stosując wzorzec do określenia granic podziału. Zwraca tablicę tokenów, które nie zawierają separatorów. Druga wersja metody ma parametr `limit`, określający maksymalną liczbę łańcuchów, które mogą zostać zwrócone. Jeśli dopasowanych zostało `limit - 1` separatorów, 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.

- `Map<String, Integer> namedGroups()` **20**

  zwraca niezmienną mapę odwzorowującą nazwy grup na ich indeksy.

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.

- `boolean hasMatch()` **20**

  zwraca `true`, jeśli w wyniku wywołania metody `matches` lub `find` obiekt `Macher` będzie umieszczony na jakimś dopasowaniu.

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

  zwracają łańcuch powstały przez zastąpienie podanym łańcuchem wszystkich dopasowań lub tylko pierwszego dopasowania. Łańcuch zastępujący może zawierać referencje do grup wzorca o postaci $*n*. Aby umieścić w łańcuchu symbol $, stosujemy sekwencję \$.

- `static String quoteReplacement(String str)` **5.0**

  cytuje wszystkie znaki \ i $ w łańcuchu `str`.

- `String replaceAll(Function<MatchResult,String> replacer)` **9**

  zastępuje każde dopasowanie wynikiem zwróconym przez funkcję `replacer`, do której został przekazany obiekt `MatchResult`.

- `Stream<MatchResult> results()` **9**

  zwraca strumień zawierający wszystkie dopasowania.

API *`java.util.regex.MatchResult`* **5**

- `string group()`
- `string group(int group)`
- `String group(String name)` **20**

  zwraca dopasowany łańcuch lub łańcuch dopasowany przez określoną grupę.

- `int start()`

- `int end()`
- `int start(int group)`
- `int start(String name)` **20**
- `int end(int group)`
- `int end(String group)` **20**

  zwraca indeks początku lub końca dopasowanego łańcucha albo łańcuch dopasowany przez podaną grupę.

- `Map<String, Integer> namedGroups()` **20**

  zwraca niezmienną mapę odwzorowującą nazwy grup na ich indeksy w wyrażeniu regularnym.

API **java.util.Scanner 5**

- `Stream<MatchResult> findAll(Pattern pattern)` **9**

  zwraca strumień wszystkich wyników dopasowanych do podanego wzorca w danych zwróconych przez dany obiekt `Scanner`.

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.

Skorowidz

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

X

Z

Ź

Ż

JAVA:

sztuka programowania na najwyższym poziomie!

To jest najlepsze kompendium wiedzy o języku Java i o całym ekosystemie Javy!
— **Andrew Binstock**, „Java Magazine"

Od ponad trzydziestu lat Java jest wybierana przez programistów, którym zależy na tworzeniu bezpiecznych aplikacji wyjątkowej jakości. Jednak pisanie programów oznacza konieczność posiadania znacznie bardziej zaawansowanej wiedzy niż ta, którą ma przeciętny użytkownik — trzeba znać język i jego wszystkie, nawet mniej oczywiste funkcjonalności, również te niedawno zaimplementowane.

Książka zawiera szczegółowe omówienie Javy 21, programowania korporacyjnego, sieciowego i bazodanowego, a także zagadnień związanych z internacjonalizacją i metodami natywnymi. Dużo miejsca poświęcono obsłudze strumieni, pracy z językiem XML, API dat i czasu, API skryptowemu czy kompilacji. Opisano też sposoby korzystania z biblioteki Swing, tworzenia graficznych interfejsów użytkownika po stronie klienta i generowania obrazów po stronie serwera. Przykłady kodu zostały starannie przetestowane, prezentują nowoczesny styl programowania w Javie i opierają się na najlepszych praktykach.

W książce ponadto:

- najlepsze praktyki pisania niezawodnego kodu w Javie
- rozszerzone API wejścia-wyjścia, serializacja obiektów i wyrażenia regularne
- usługi sieciowe, wbudowane serwery Javy i implementacja własnych serwerów
- API skryptowe i API kompilatora
- model bezpieczeństwa Javy i funkcje biblioteki bezpieczeństwa
- nowe API do korzystania z funkcji natywnych i pamięci poza stertą

Cay S. Horstmann jest profesorem informatyki na Uniwersytecie Stanowym w San José i autorem najpopularniejszych w Polsce podręczników do nauki Javy. Został wyróżniony tytułem Java Champion. Regularnie występuje na konferencjach programistycznych. Urodził się w północnej części Niemiec, obecnie mieszka w USA.

Helion

helion.pl

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

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-3004-9

Cena: 169,00 zł

Java™

Pearson