

JAVA[®]

Techniki zaawansowane

WYDANIE X



CAY S. HORSTMANN

Tytuł oryginału: Core Java, Volume II-Advanced Features (10th Edition)

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-3479-3

Authorized translation from the English language edition, entitled CORE JAVA, VOLUME II – ADVANCED FEATURES, 10th Edition; ISBN 0134177290; by Cay S. Horstmann; published by Pearson Education, Inc, publishing as Prentice Hall.
Copyright © Copyright © 2017 Oracle and/or its affiliates.

Portions © 2017 Cay S. Horstmann

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education Inc.
Polish language edition published by HELION S.A. Copyright © 2017.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/javtx>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

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

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Przedmowa	11
Podziękowania	15
Rozdział 1. Biblioteka strumieni Java SE 8	17
1.1. Od iteracji do operacji na strumieniach	18
1.2. Tworzenie strumieni	20
1.3. Metody filter, map oraz flatMap	24
1.4. Pobieranie podstrumieni i łączenie strumieni	25
1.5. Inne przekształcenia strumieni	26
1.6. Proste operacje redukcji	27
1.7. Typ Optional	29
1.7.1. Sposoby posługiwania się wartościami Optional	29
1.7.2. Jak nie należy używać wartości opcjonalnych	31
1.7.3. Tworzenie obiektów typu Optional	31
1.7.4. Łączenie funkcji zwracających wartości opcjonalne przy użyciu flatMap	32
1.8. Gromadzenie wyników	34
1.9. Gromadzenie wyników w mapach	39
1.10. Grupowanie i podział	43
1.11. Kolektory przetwarzające	44
1.12. Operacje redukcji	48
1.13. Strumień danych typów prostych	50
1.14. Strumień równoległy	55
Rozdział 2. Wejście i wyjście	61
2.1. Strumień wejścia-wyjścia	61
2.1.1. Odczyt i zapis bajtów	62
2.1.2. Zoo pełne strumieni	64
2.1.3. Łączenie filtrów strumieni wejścia-wyjścia	68
2.2. Strumień tekstowy	72
2.2.1. Zapisywanie tekstu	72
2.2.2. Wczytywanie tekstu	75
2.2.3. Zapis obiektów w formacie tekstowym	75
2.2.4. Zbiory znaków	78

2.3. Odczyt i zapis danych binarnych	81
2.3.1. Interfejsy DataInput oraz DataOutput	81
2.3.2. Strumienie plików o swobodnym dostępie	84
2.3.3. Archiwa ZIP	88
2.4. Strumienie obiektów i serializacja	91
2.4.1. Zapisywanie i wczytywanie obiektów serializowalnych	91
2.4.2. Format pliku serializacji obiektów	95
2.4.3. Modyfikowanie domyślnego mechanizmu serializacji	102
2.4.4. Serializacja singletonów i wyliczeń	104
2.4.5. Wersje	105
2.4.6. Serializacja w roli klonowania	107
2.5. Zarządzanie plikami	109
2.5.1. Ścieżki dostępu	110
2.5.2. Odczyt i zapis plików	112
2.5.3. Tworzenie plików i katalogów	114
2.5.4. Kopiowanie, przenoszenie i usuwanie plików	115
2.5.5. Informacje o plikach	117
2.5.6. Przeglądanie zawartości katalogu	118
2.5.7. Stosowanie strumieni katalogów	120
2.5.8. Systemy plików ZIP	123
2.6. Mapowanie plików w pamięci	124
2.6.1. Wydajność plików mapowanych w pamięci	124
2.6.2. Struktura bufora danych	131
2.6.3. Blokowanie plików	133
2.7. Wyrażenia regularne	135
Rozdział 3. Język XML	149
3.1. Wprowadzenie do języka XML	150
3.1.1. Struktura dokumentu XML	152
3.2. Parsowanie dokumentów XML	155
3.3. Kontrola poprawności dokumentów XML	166
3.3.1. Definicje typów dokumentów	167
3.3.2. XML Schema	174
3.3.3. Praktyczny przykład	176
3.4. Wyszukiwanie informacji i XPath	189
3.5. Przestrzenie nazw	195
3.6. Parsery strumieniowe	198
3.6.1. Wykorzystanie parsera SAX	198
3.6.2. Wykorzystanie parsera StAX	203
3.7. Tworzenie dokumentów XML	207
3.7.1. Dokumenty bez przestrzeni nazw	207
3.7.2. Dokumenty z przestrzenią nazw	208
3.7.3. Zapisywanie dokumentu	209
3.7.4. Przykład: tworzenie pliku SVG	209
3.7.5. Tworzenie dokumentu XML za pomocą parsera StAX	213
3.8. Przekształcenia XSL	220
Rozdział 4. Programowanie aplikacji sieciowych	231
4.1. Połączenia z serwerem	231
4.1.1. Stosowanie programu telnet	231
4.1.2. Nawiązywanie połączenia z serwerem z wykorzystaniem Javy	234
4.1.3. Limity czasu gniazd	235
4.1.4. Adresy internetowe	237

4.2. Implementacja serwerów	238
4.2.1. Gniazda serwera	239
4.2.2. Obsługa wielu klientów	241
4.2.3. Połączenia częściowo zamknięte	244
4.3. Przerwywanie działania gniazd sieciowych	246
4.4. Połączenia wykorzystujące URL	252
4.4.1. URL i URI	252
4.4.2. Zastosowanie klasy URLConnection do pobierania informacji	254
4.4.3. Wysyłanie danych do formularzy	262
4.5. Wysyłanie poczty elektronicznej	270

Rozdział 5. Programowanie baz danych: JDBC 275

5.1. Architektura JDBC	276
5.1.1. Typy sterowników JDBC	276
5.1.2. Typowe zastosowania JDBC	278
5.2. Język SQL	278
5.3. Instalacja JDBC	284
5.3.1. Adresy URL baz danych	284
5.3.2. Pliki JAR zawierające sterownik	285
5.3.3. Uruchamianie bazy danych	285
5.3.4. Rejestracja klasy sterownika	286
5.3.5. Nawiązywanie połączenia z bazą danych	287
5.4. Stosowanie poleceń SQL	289
5.4.1. Wykonywanie poleceń SQL	290
5.4.2. Zarządzanie połączeniami, poleceniami i zbiorami wyników	293
5.4.3. Analiza wyjątków SQL	294
5.4.4. Wypełnianie bazy danych	296
5.5. Wykonywanie zapytań	300
5.5.1. Polecenia przygotowane	300
5.5.2. Odczyt i zapis dużych obiektów	306
5.5.3. Sekwencje sterujące	308
5.5.4. Zapytania o wielu zbiorach wyników	309
5.5.5. Pobieranie wartości kluczy wygenerowanych automatycznie	310
5.6. Przewijalne i aktualizowalne zbiory wyników zapytań	311
5.6.1. Przewijalne zbiory wyników	311
5.6.2. Aktualizowalne zbiory rekordów	313
5.7. Zbiory rekordów	318
5.7.1. Tworzenie zbiorów rekordów	318
5.7.2. Buforowane zbiory rekordów	319
5.8. Metadane	322
5.9. Transakcje	331
5.9.1. Programowanie transakcji w JDBC	332
5.9.2. Punkty kontrolne	332
5.9.3. Aktualizacje wsadowe	333
5.10. Zaawansowane typy języka SQL	335
5.11. Zaawansowane zarządzanie połączeniami	336

Rozdział 6. API dat i czasu 339

6.1. Oś czasu	340
6.2. Daty lokalne	343
6.3. Modyfikatory dat	346
6.4. Czas lokalny	347

6.5. Czas strefowy	348
6.6. Formatowanie i parsowanie	352
6.7. Współdziałanie ze starym kodem	356
Rozdział 7. Internacjonalizacja	359
7.1. Lokalizatory	360
7.2. Formaty liczb	365
7.3. Waluty	370
7.4. Data i czas	372
7.5. Porządek alfabetyczny i normalizacja	378
7.6. Formatowanie komunikatów	385
7.6.1. Formatowanie liczb i dat	385
7.6.2. Formatowanie z wariantami	387
7.7. Wczytywanie i wyświetlanie tekstów	389
7.7.1. Pliki tekstowe	389
7.7.2. Znaki końca wiersza	389
7.7.3. Konsola	390
7.7.4. Pliki dzienników	391
7.7.5. BOM — znacznik kolejności bajtów UTF-8	391
7.7.6. Kodowanie plików źródłowych	392
7.8. Kompletne zasobów	392
7.8.1. Wyszukiwanie kompletów zasobów	393
7.8.2. Pliki właściwości	394
7.8.3. Klasy kompletów zasobów	395
7.9. Kompletny przykład	397
Rozdział 8. Skrypty, kompilacja i adnotacje	413
8.1. Skrypty na platformie Java	413
8.1.1. Wybór silnika skryptów	414
8.1.2. Wykonywanie skryptów i wiązania zmiennych	415
8.1.3. Przekierowanie wejścia i wyjścia	417
8.1.4. Wywoływanie funkcji i metod skryptów	418
8.1.5. Kompilacja skryptu	420
8.1.6. Przykład: skrypty i graficzny interfejs użytkownika	420
8.2. Interfejs kompilatora	425
8.2.1. Kompilacja w najprostszy sposób	426
8.2.2. Stosowanie zadań kompilacji	426
8.2.3. Przykład: dynamiczne tworzenie kodu w języku Java	432
8.3. Stosowanie adnotacji	436
8.3.1. Wprowadzenie do stosowania adnotacji	437
8.3.2. Przykład: adnotacje obsługi zdarzeń	438
8.4. Składnia adnotacji	443
8.4.1. Interfejsy adnotacji	443
8.4.2. Adnotacje	445
8.4.3. Adnotacje deklaracji	446
8.4.4. Adnotacje zastosowań typów	447
8.4.5. Adnotacje i this	449
8.5. Adnotacje standardowe	450
8.5.1. Adnotacje kompilacji	451
8.5.2. Adnotacje zarządzania zasobami	451
8.5.3. Metaadnotacje	452

8.6. Przetwarzanie adnotacji w kodzie źródłowym	455
8.6.1. Procesory adnotacji	455
8.6.2. Interfejs programowy modelu języka	455
8.6.3. Stosowanie adnotacji do generacji kodu źródłowego	456
8.7. Inżynieria kodu bajtowego	459
8.7.1. Modyfikowanie plików klasowych	459
8.7.2. Modyfikacja kodu bajtowego podczas ładowania	464

Rozdział 9. Bezpieczeństwo 467

9.1. Ładowanie klas	468
9.1.1. Proces wczytywania plików klas	468
9.1.2. Hierarchia klas ładowania	469
9.1.3. Zastosowanie procedur ładujących w roli przestrzeni nazw	471
9.1.4. Implementacja własnej procedury ładującej	473
9.1.5. Weryfikacja kodu maszyny wirtualnej	478
9.2. Menedżery bezpieczeństwa i pozwolenia	483
9.2.1. Sprawdzanie uprawnień	483
9.2.2. Bezpieczeństwo na platformie Java	484
9.2.3. Pliki polityki bezpieczeństwa	487
9.2.4. Tworzenie własnych klas pozwoleń	495
9.2.5. Implementacja klasy pozwoleń	496
9.3. Uwierzelnianie użytkowników	502
9.3.1. Framework JAAS	502
9.3.2. Moduły JAAS	507
9.4. Podpis cyfrowy	516
9.4.1. Skrótóy wiadomości	517
9.4.2. Podpisywanie wiadomości	520
9.4.3. Weryfikacja podpisu	522
9.4.4. Problem uwierzelniania	524
9.4.5. Podpisywanie certyfikatów	526
9.4.6. Żądania certyfikatu	527
9.4.7. Podpisywanie kodu	528
9.5. Szyfrowanie	534
9.5.1. Szyfrowanie symetryczne	534
9.5.2. Generowanie klucza	536
9.5.3. Strumień szyfrujący	541
9.5.4. Szyfrowanie kluczem publicznym	542

Rozdział 10. Zaawansowane możliwości pakietu Swing 547

10.1. Listy	547
10.1.1. Komponent JList	548
10.1.2. Modele list	553
10.1.3. Wstawianie i usuwanie	558
10.1.4. Odrysowywanie zawartości listy	559
10.2. Tabele	563
10.2.1. Najprostsze tabele	563
10.2.2. Modele tabel	568
10.2.3. Wiersze i kolumny	571
10.2.4. Rysowanie i edycja komórek	586

10.3. Drzewa	598
10.3.1. Najprostsze drzewa	599
10.3.2. Modyfikacje drzew i ścieżek drzew	606
10.3.3. Przeglądanie węzłów	613
10.3.4. Rysowanie węzłów	615
10.3.5. Nasłuchiwanie zdarzeń w drzewach	618
10.3.6. Własne modele drzew	625
10.4. Komponenty tekstowe	633
10.4.1. Śledzenie zmian zawartości komponentów tekstowych	634
10.4.2. Sformatowane pola wejściowe	637
10.4.3. Komponent JSpinner	653
10.4.4. Prezentacja HTML za pomocą JEditorPane	661
10.5. Wskaźniki postępu	667
10.5.1. Paski postępu	667
10.5.2. Monitory postępu	670
10.5.3. Monitorowanie postępu strumieni wejścia	673
10.6. Organizatory komponentów i dekoratory	678
10.6.1. Panele dzielone	678
10.6.2. Panele z kartami	681
10.6.3. Panele pulpitu i ramki wewnętrzne	687
10.6.4. Warstwy	703

Rozdział 11. Zaawansowane możliwości biblioteki AWT 709

11.1. Potokowe tworzenie grafiki	710
11.2. Figury	712
11.2.1. Hierarchia klas Shape	713
11.2.2. Wykorzystanie klas obiektów graficznych	714
11.3. Pola	727
11.4. Ślad pędzla	728
11.5. Wypełnienia	735
11.6. Przekształcenia układu współrzędnych	737
11.7. Przycinanie	743
11.8. Przezroczystość i składanie obrazów	745
11.9. Wskazówki operacji graficznych	753
11.10. Czytanie i zapisywanie plików graficznych	758
11.10.1. Wykorzystanie obiektów zapisu i odczytu plików graficznych	759
11.10.2. Odczyt i zapis plików zawierających sekwencje obrazów	763
11.11. Operacje na obrazach	768
11.11.1. Dostęp do danych obrazu	769
11.11.2. Filtrowanie obrazów	775
11.12. Drukowanie	783
11.12.1. Drukowanie grafiki	784
11.12.2. Drukowanie wielu stron	792
11.12.3. Podgląd wydruku	794
11.12.4. Usługi drukowania	802
11.12.5. Usługi drukowania za pośrednictwem strumieni	806
11.12.6. Atrybuty drukowania	807
11.13. Schowek	813
11.13.1. Klasy i interfejsy umożliwiające przekazywanie danych	814
11.13.2. Przekazywanie tekstu	815
11.13.3. Interfejs Transferable i formaty danych	818

11.13.4. Przekazywanie obrazów za pomocą schowka	820
11.13.5. Wykorzystanie schowka systemowego do przekazywania obiektów Java	824
11.13.6. Zastosowanie lokalnego schowka do przekazywania referencji obiektów	827
11.14. Mechanizm „przeciągnij i upuść”	828
11.14.1. Przekazywanie danych pomiędzy komponentami Swing	829
11.14.2. Źródła przeciąganych danych	833
11.14.3. Cele upuszczanych danych	835
11.15. Integracja z macierzystą platformą	844
11.15.1. Ekran powitalny	844
11.15.2. Uruchamianie macierzystych aplikacji pulpitu	849
11.15.3. Zasobnik systemowy	853
Rozdział 12. Metody macierzyste	859
12.1. Wywołania funkcji języka C z programów w języku Java	860
12.2. Numeryczne parametry metod i wartości zwracane	866
12.3. Łańcuchy znaków jako parametry	868
12.4. Dostęp do składowych obiektu	873
12.4.1. Dostęp do pól instancji	874
12.4.2. Dostęp do pól statycznych	877
12.5. Sygnatury	878
12.6. Wywoływanie metod języka Java	880
12.6.1. Wywoływanie metod obiektów	880
12.6.2. Wywoływanie metod statycznych	883
12.6.3. Konstruktory	884
12.6.4. Alternatywne sposoby wywoływania metod	885
12.7. Tablice	886
12.8. Obsługa błędów	890
12.9. Interfejs programowy wywołań języka Java	895
12.10. Kompletny przykład: dostęp do rejestru systemu Windows	900
12.10.1. Rejestr systemu Windows	900
12.10.2. Interfejs dostępu do rejestru na platformie Java	902
12.10.3. Implementacja dostępu do rejestru za pomocą metod macierzystych	902
Skorowidz	917

2

Wejście i wyjście

W tym rozdziale:

- 2.1. Strumień wejścia-wyjścia.
- 2.2. Strumień tekstowe.
- 2.3. Odczyt i zapis danych binarnych.
- 2.4. Strumień obiektów i serializacja.
- 2.5. Zarządzanie plikami.
- 2.6. Pliki mapowane w pamięci.
- 2.7. Wyrażenia regularne.

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. Następnie zajmiemy się zagadnieniami związanymi z obsługą plików i katalogów. 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. Strumień 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 `OutputStream` 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` posiada metodę abstrakcyjną:

```
abstract int read()
```

Metoda ta wczytuje jeden bajt i zwraca jego wartość lub `-1`, jeżeli natrafi na koniec źródła danych. Projektanci konkretnych klas strumieni wejścia 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` posiada również nieabstrakcyjne metody pozwalające pobrać lub zignorować tablicę bajtów. Metody te wywołują abstrakcyjną metodę `read`, tak więc podklasy muszą przesłaniać tylko tę jedną metodę.

Analogicznie, klasa `OutputStream` definiuje metodę abstrakcyjną

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

która wysyła jeden bajt do aktualnego wyjścia.

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

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)`

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

Parametry:

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

■ `long skip(long n)`

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

■ `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.

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.

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

- `void close()`

opróżnia i zamyka strumień wyjścia.

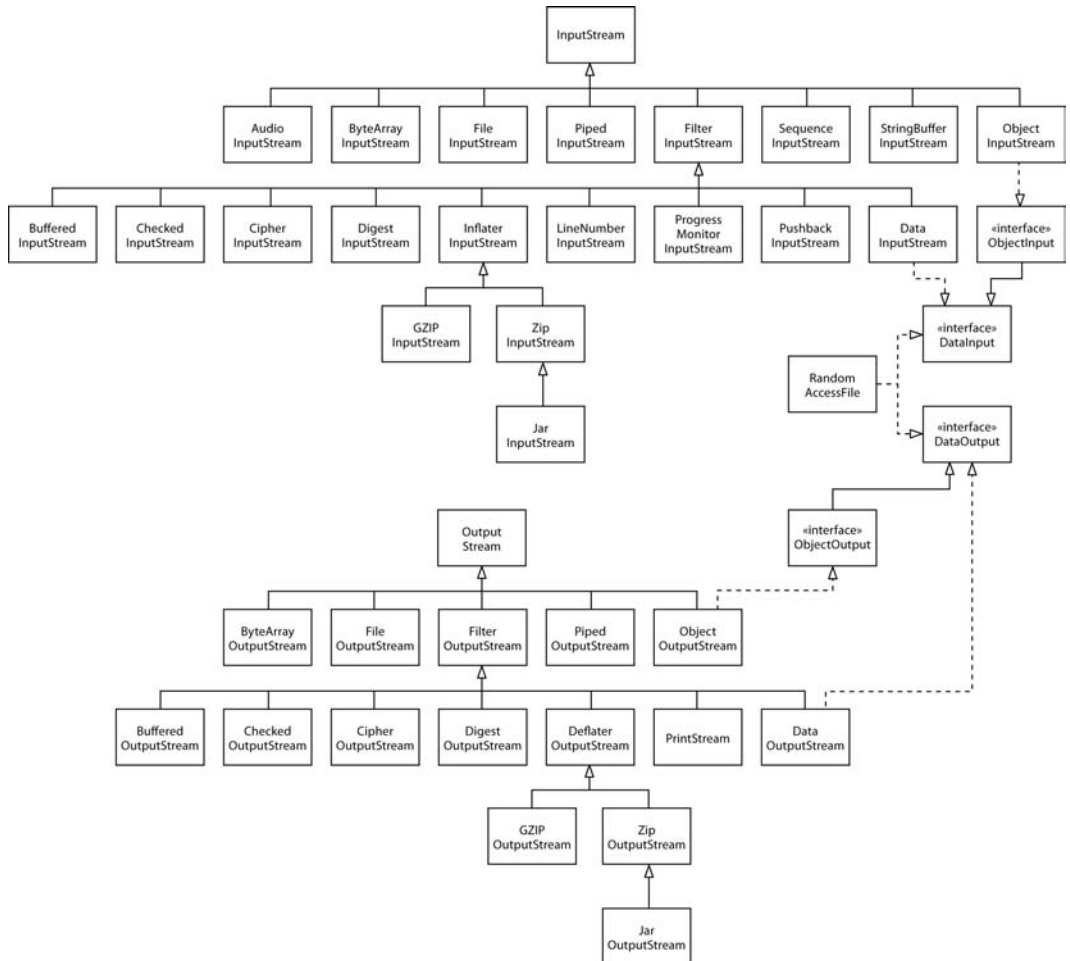
- `void flush()`

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

2.1.2. Zoo pełne strumieni

W przeciwieństwie do języka C, który w zupełności zadowala się jednym typem `FILE*`, Java posiada istne zoo ponad 60 (!) różnych typów strumieni wejścia i wyjścia (patrz rysunki 2.1 i 2.2).

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



Rysunek 2.1. Hierarchia strumieni wejścia i wyjścia

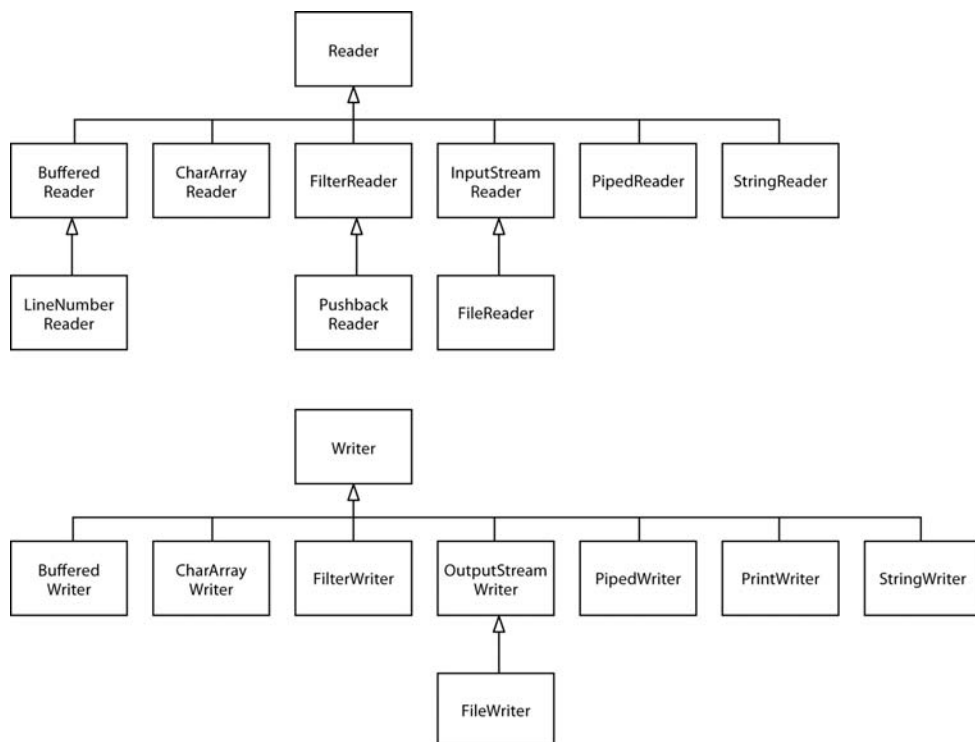
Z drugiej strony, o czym już wspominaliśmy, do obsługi tekstu Unicode używamy klas pochodzących od klas abstrakcyjnych `Reader` i `Writer` (patrz rysunek 2.2) Podstawowe metody klas `Reader` i `Writer` są podobne do tych należących do `InputStream` i `OutputStream`.

```

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

```

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 2.2. 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*.



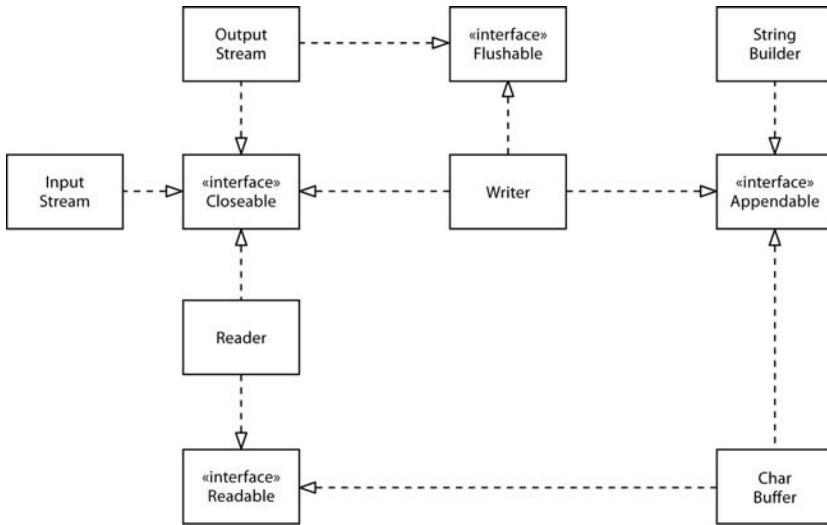
Interfejs `java.io.Closeable` stanowi rozszerzenie interfejsu `java.lang.AutoCloseable`. Dzięki temu dla każdego obiektu implementującego interfejs `Closeable` możemy użyć wersji instrukcji `try` zarządzającej zasobami. Ale po co nam dwa interfejsy? Metoda `close` interfejsu `Closeable` może wyrzucać jedynie wyjątek `IOException`, podczas gdy metoda `AutoCloseable.close` może wyrzucać 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 2.6.2, „Struktura bufora danych”).



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*.

Pośród klas strumieni wejścia-wyjścia jedynie klasa *Writer* implementuje interfejs *Appendable*.

API *java.io.Closeable* 5.0

■ void close()

zamyka obiekt implementujący interfejs *Closeable*. Może wyrzucić wyjątek *IOException*.

API *java.io.Flushable* 5.0

■ void flush()

opóżnia bufor danych związany z obiektem implementującym interfejs *Flushable*.

API *java.lang.Readable* 5.0

■ int read(CharBuffer cb)

próbuję 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)`

doписuje 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 strumienia wejścia-wyjścia

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

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

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



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



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 `DataInputStream` 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`.

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

Wróćmy do rysunku 2.1, gdzie przedstawione są klasy `FilterInputStream` i `FilterOutputStream`. Ich podklasy 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:

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

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

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

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

Teraz możemy odczytać wartość następnego bajta:

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

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

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

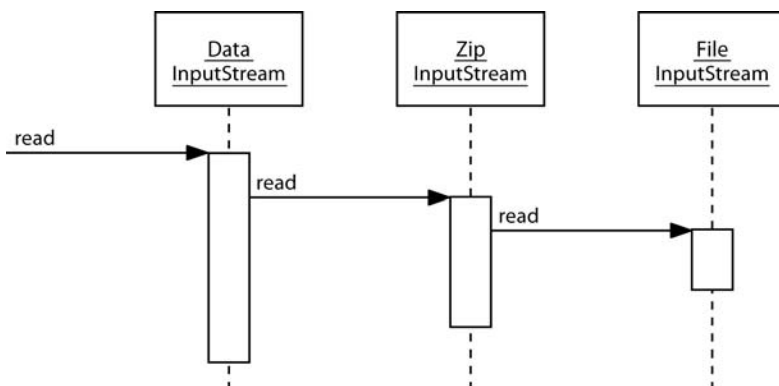
Ale wczytywanie i powtórne wstawianie bajtów to *jedyne* metody obsługiwane przez klasę `PushbackInputStream`. Jeżeli chcemy podejrzeć bajty, a także wczytywać liczby, potrzebujemy referencji zarówno do `PushbackInputStream`, jak i do `DataInputStream`.

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

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).

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

Rysunek 2.4.
Sekwencja
filtrowanych
strumieni



(Aby dowiedzieć się więcej o obsłudze formatu ZIP, zajrzyj do punktu 2.3.3, „Archiwa ZIP”, 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 `file` (klasa `File` zostanie omówiona pod koniec tego rozdziału). Ścieżki dostępu są podawane względem katalogu roboczego skonfigurowanego podczas uruchamiania maszyny wirtualnej Java.

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

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

Parametry: `b` zwracany bajt

2.2. Strumienie tekstowe

Zapisując dane, możemy wybierać pomiędzy formatem binarnym i tekstowym. Dla przykładu: jeżeli liczba całkowita 1234 zostanie zapisana w postaci binarnej, w pliku pojawi się sekwencja bajtów 00 00 04 D2 (w notacji szesnastkowej). W formacie tekstowym liczba ta zostanie zapisana jako łańcuch "1234". Mimo iż zapis danych w postaci binarnej jest szybki i efektywny, to uzyskany wynik jest kompletnie nieczytelny dla ludzi. W poniższym podrozdziale skoncentrujemy się na *tekstowym* wejściu-wyjściu, a odczyt i zapis danych binarnych omówimy w podrozdziale 2.3, „Odczyt i zapis danych binarnych”.

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

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

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

Obiekt wejścia korzysta z domyślnego kodowania lokalnego systemu. W przypadku systemów operacyjnych przeznaczonych dla komputerów stacjonarnych może to być jakiś archaiczny sposób kodowania, taki jak Windows-1250 lub Mac OS Roman. Zawsze należy określać wybrany sposób kodowania, podając jego nazwę w konstruktorze `InputStreamReader`, na przykład:

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

Więcej informacji na temat kodowania znaków znajdziesz w punkcie 2.2.4, „Zbiory znaków”.

2.2.1. Zapisywanie tekstu

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

```
PrintWriter out = new PrintWriter("employee.txt", "UTF-8");
```

stanowi odpowiednik instrukcji

```
PrintWriter out = new PrintWriter(  
    new FileOutputStream("employee.txt"), "UTF-8");
```

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 `PrintWriter` zawsze są buforowane). Domyślnie automatyczne opróżnianie jest *wyłączone*. Automatyczne opróżnianie możemy włączać i wyłączać przy użyciu konstruktora `PrintWriter(Writer writer, boolean autoFlush)`:

```
PrintWriter out = new PrintWriter(
    new OutputStreamWriter(
        new FileOutputStream("employee.txt"), "UTF-8"),
    true); // automatyczne opróżnianie
```

Metody `print` nie wyrzucają 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[])`.

API `java.io.PrintWriter 1.1`

- `PrintWriter(Writer out)`
- `PrintWriter(Writer writer)`

tworzy nowy obiekt klasy `PrintWriter` zapisujący dane w przekazanym obiekcie zapisu.
- `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`.

2.2.2. 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 = new String(Files.readAllBytes(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))
{
    ...
}
```

W początkowych wersjach języka Java jedynym sposobem przetwarzania wczytywanego tekstu było wykorzystanie klasy `BufferedReader`. Metoda `readLine` tej klasy zwraca wiersz tekstu lub wartość `null`, jeśli żadne dane wejściowe nie są już dostępne. W przypadku zastosowania tej klasy typowa pętla przetwarzania odczytywanych danych wejściowych miała następującą postać:

```
InputStream inputStream = ...;
try (BufferedReader in = new BufferedReader(new InputStreamReader(inputStream,
    StandardCharsets.UTF_8))
{
    String line;
    while ((line = in.readLine()) != null)
    {
        // wykonanie jakichś operacji na zmiennej line
    }
}
```

Obecnie jednak klasa `BufferedReader` udostępnia także metodę `lines`, która zwraca obiekt typu `Stream<String>`. W odróżnieniu od klasy `Scanner` klasa `BufferedReader` nie dysponuje żadnymi metodami do wczytywania liczb.

2.2.3. Zapis obiektów w formacie tekstowym

W tym podrozdziale przeanalizujemy działanie przykładowego programu, który będzie zapisywać tablicę obiektów typu `Employee` w pliku tekstowym. Dane każdego obiektu zostaną zapisane w osobnym wierszu. Wartości pól składowych zostaną oddzielone od siebie separatorami. Jako separatora używamy pionowej kreski (`|`) (innym popularnym separatorem jest dwukropek (`:`), zabawa polega na tym, że każdy programista używa innego separatora).

Naturalnie, taki wybór stawia przed nami pytanie, co będzie, jeśli znak | znajdzie się w jednym z zapisywanych przez nas łańcuchów?

Oto przykładowy zbiór danych obiektów:

```
Harry Hacker|35500|1989-10-01
Carl Cracker|75000|1987-12-15
Tony Tester|38000|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 `writeEmployee`, którą dodamy do klasy `Employee`.

```
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(BufferedReader in)
```

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

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

```

        employees[i] = new Employee();
        employees[i].readData(in);
    }

```

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

Listing 2.1. *textfile/TextFileTest.java*

```

1 package textfile;
2
3 import java.io.*;
4 import java.time.*;
5 import java.util.*;
6
7 /**
8  * @version 1.14 2016-07-11
9  * @author Cay Horstmann
10 */
11 public class TextFileTest
12 {
13     public static void main(String[] args) throws IOException
14     {
15         Employee[] staff = new Employee[3];
16
17         staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
18         staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
19         staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
20
21         // zapisuje wszystkie rekordy pracowników w pliku employee.dat
22         try (PrintWriter out = new PrintWriter("employee.dat", "UTF-8"))
23         {
24             writeData(staff, out);
25         }
26
27         // wczytuje wszystkie rekordy do nowej tablicy
28         try (Scanner in = new Scanner(
29             new FileInputStream("employee.dat"), "UTF-8"))
30         {
31             Employee[] newStaff = readData(in);
32
33             // wyświetla wszystkie wczytane rekordy
34             for (Employee e : newStaff)
35                 System.out.println(e);
36         }
37     }
38
39     /**
40     * Zapisuje dane wszystkich obiektów klasy Employee umieszczonych
41     * w tablicy do obiektu klasy PrintWriter
42     * @param employees tablica obiektów klasy Employee
43     * @param out obiekt klasy PrintWriter
44     */
45     private static void writeData(Employee[] employees, PrintWriter out)
46         throws IOException
47     {
48         // zapisuje liczbę obiektów
49         out.println(employees.length);

```

```
50
51     for (Employee e : employees)
52         writeEmployee(out, e);
53 }
54
55 /**
56  * Wczytuje tablicę obiektów klasy Employee
57  * @param in obiekt klasy Scanner
58  * @return tablica obiektów klasy Employee
59  */
60 private static Employee[] readData(Scanner in)
61 {
62     //pobiera rozmiar tablicy
63     int n = in.nextInt();
64     in.nextLine(); //pobiera znak nowego wiersza
65
66     Employee[] employees = new Employee[n];
67     for (int i = 0; i < n; i++)
68     {
69         employees[i] = readEmployee(in);
70     }
71     return employees;
72 }
73
74 /**
75  * Zapisuje dane obiektu klasy Employee do obiektu klasy PrintWriter
76  * @param out obiekt klasy PrintWriter
77  */
78 public static void writeEmployee(PrintWriter out, Employee e)
79 {
80     out.println(e.getName() + "|" + e.getSalary() + "|" + e.getHireDay());
81 }
82
83 /**
84  * Wczytuje dane obiektu klasy Employee
85  * @param in obiekt klasy Scanner
86  */
87 public static Employee readEmployee(Scanner in)
88 {
89     String line = in.nextLine();
90     String[] tokens = line.split("\\|");
91     String name = tokens[0];
92     double salary = Double.parseDouble(tokens[1]);
93     LocalDate hireDate = LocalDate.parse(tokens[2]);
94     int year = hireDate.getYear();
95     int month = hireDate.getMonthValue();
96     int day = hireDate.getDayOfMonth();
97     return new Employee(name, salary, year, month, day);
98 }
99 }
```

2.2.4. Zbiory znaków

Strumienie wejściowe i wyjściowe służą do wczytywania i zapisywania sekwencji bajtów, jednak często konieczne jest wykonywanie operacji na tekście — czyli na sekwencjach znaków. Właśnie w takich przypadkach ogromną rolę zaczyna odgrywać sposób, w jaki znaki są kodowane na bajty.

W Javie znaki są kodowane przy wykorzystaniu standardu Unicode. 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	0a6a5a4a3a2a1a0
80 ... 7FF	110a10a9a8a7a6 10a5a4a3a2a1a0
800 ... FFFF	1110a15a14a13a12 10a11a10a9a8a7a6 10a5a4a3a2a1a0
10000 ... 10FFFF	11110a20a19a18 10a17a16a15a14a13a12 10a11a10a9a8a7a6 10a5a4a3a2a1a0

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	a15a14a13a12a11a10a9a8 a7a6a5a4a3a2a1a0
10000 ... 10FFFF	110110b19b18b17b16a15a14a13a12a11a10 110111a9a8 a7a6a5a4a3a2a1a0 gdzie b19b18b17b16 = a20a19a18a17a16 - 1

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.

¹ „Znacznik kolejności bajtów” — *przyp. tłum.*



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.

Nie ma możliwości automatycznego wykrywania sposobu kodowania zastosowanego w strumieniu bajtów. Niektóre metody API pozwalają na wykorzystywanie „domyślnego zbioru znaków”, czyli sposobu kodowania preferowanego przez używany system operacyjny. Czy jednak będzie to ten sam sposób kodowania, który był stosowany przez źródło bajtów? Te bajty mogły przecież zostać wygenerowane w zupełnie innym miejscu świata. I właśnie z tego powodu zawsze należy jawnie określać kodowanie. Na przykład w przypadku odczytywania strony WWW należy sprawdzić nagłówek `Content-Type`.



Kodowanie używane przez daną platformę systemową jest zwracane przez statyczną metodę `Charset.defaultCharset`. Z kolei statyczna metoda `Charset.availableCharsets` zwraca wszystkie dostępne instancje klasy `Charset` — są one zwracane w formie mapy kojarzącej przyjęte nazwy z obiektami `Charset`.



W implementacji Javy firmy Oracle dostępna jest systemowa właściwość `file.encoding`, która zmienia domyślny, systemowy sposób kodowania. Właściwość ta nie jest jednak obsługiwana oficjalnie, a implementacja biblioteki Javy firmy Oracle stosuje ją niekonsekwentnie. Dlatego też nie należy z niej korzystać.

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
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");
```

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

```
String str = new String(bytes, StandardCharsets.UTF_8);
```



Niektóre metody pozwalają na określanie sposobów kodowania przy wykorzystaniu obiektów `Charset` lub łańcuchów znaków. Warto jednak wybierać stałe klasy `StandardCharsets`, tak by nie trzeba było zwracać uwagi na ich poprawny zapis. Na przykład użycie wywołania `new String(bytes, "UTF 8")` jest niedopuszczalne i spowoduje zgłoszenie błędu czasu wykonania.



Niektóre metody (takie jak konstruktor `String(byte[])` w przypadku pominięcia jawnego określenia żadanego sposobu kodowania użyją domyślnego kodowania platformy systemowej; z kolei inne metody (takie jak `Files.readLine()`) skorzystają z kodowania UTF-8.

2.3. 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.

2.3.1. Interfejsy `DataInput` oraz `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 1.4), znaki łańcucha są najpierw reprezentowane w kodzie UTF-16 (patrz tabela 1.5), a dopiero potem przekodowywane na UTF-8. Wynik takiego kodowania różni się dla znaków o kodach większych od `0xFFFF`. Kodowanie takie stosuje się dla zachowania zgodności z maszynami wirtualnymi powstałymi, gdy Unicode zadowalał się tylko 16 bitami.



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 ($1234 = 4 \times 256 + 13 \times 16 + 2$). Może ona zostać przechowana w ten sposób, że pierwszym z czterech bajtów pamięci będzie bajt najbardziej znaczący (ang. *most significant byte*, *MSB*): 00 00 04 D2. Albo w taki sposób, że będzie to bajt najmłodszy (ang. *least significant byte*, *LSB*): D2 04 00 00. Pierwszy sposób stosowany jest przez maszyny SPARC, a drugi przez procesory Pentium. Może to powodować problemy z przenoszeniem nawet najprostszych plików danych pomiędzy różnymi platformami, 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`:

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

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

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

java.io.DataInput 1.0

- `boolean readBoolean()`
- `byte readByte()`
- `char readChar()`
- `double readDouble()`
- `float readFloat()`
- `int readInt()`
- `long readLong()`

- `short readShort()`
wczytuje wartość określonego typu.
- `void readFully(byte[] b)`
wczytuje bajty do tablicy `b`, blokując wątek, dopóki wszystkie bajty nie zostaną wczytane.
Parametry: `b` bufor, do którego zapisywane są dane.
- `void readFully(byte[] b, int off, int len)`
wczytuje bajty do tablicy `b`, blokując wątek, dopóki wszystkie bajty nie zostaną wczytane.
Parametry: `b` bufor, do którego zapisywane są dane.
 `off` indeks pierwszego bajta.
 `len` maksymalna ilość odczytanych bajtów.
- `String readUTF()`
wczytuje łańcuch znaków zapisanych w zmodyfikowanym formacie UTF-8.
- `int skipBytes(int n)`
ignoruje `n` bajtów, blokując wątek, dopóki wszystkie bajty nie zostaną zignorowane.
Parametry: `n` liczba ignorowanych bajtów.

`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.

2.3.2. 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).

```
RandomAccessFile in = new RandomAccessFile("employee.dat", "r");
RandomAccessFile inOut = new RandomAccessFile("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);
Employee e = new Employee();
e.readData(in);
```

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

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

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

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

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

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

```

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

```

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

```

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

```

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

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

```

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();

```

Wyznamy 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*

```
1 package randomAccess;
2
3 import java.io.*;
4 import java.util.*;
5 import java.time.*;
6
7 /**
8  * @version 1.13 2016-07-11
9  * @author Cay Horstmann
10 */
11 public class RandomAccessTest
12 {
13     public static void main(String[] args) throws IOException
14     {
15         Employee[] staff = new Employee[3];
16
17         staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
18         staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
19         staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
20
21         try (DataOutputStream out = new DataOutputStream(new
22             ↪FileOutputStream("employee.dat")))
23         {
24             // zapisuje rekordy wszystkich pracowników w pliku employee.dat
25             for (Employee e : staff)
26                 writeData(out, e);
27         }
28
29         try (RandomAccessFile in = new RandomAccessFile("employee.dat", "r"))
30         {
31             // wczytuje wszystkie rekordy do nowej tablicy
32
33             // oblicza rozmiar tablicy
34             int n = (int)(in.length() / Employee.RECORD_SIZE);
35             Employee[] newStaff = new Employee[n];
36
37             // wczytuje rekordy pracowników w odwrotnej kolejności
38             for (int i = n - 1; i >= 0; i--)
39             {
40                 newStaff[i] = new Employee();
41                 in.seek(i * Employee.RECORD_SIZE);
42                 newStaff[i] = readData(in);
43             }
44
45             // wyświetla wczytane rekordy
46             for (Employee e : newStaff)
47                 System.out.println(e);
48         }
49     }
50 }
```

```

51  * Zapisuje dane pracownika
52  * @param out obiekt typu DataOutput
53  * @param e pracownik
54  */
55  public static void writeData(DataOutput out, Employee e) throws IOException
56  {
57      DataIO.writeFixedString(e.getName(), Employee.NAME_SIZE, out);
58      out.writeDouble(e.getSalary());
59
60      LocalDate hireDay = e.getHireDay();
61      out.writeInt(hireDay.getYear());
62      out.writeInt(hireDay.getMonthValue());
63      out.writeInt(hireDay.getDayOfMonth());
64  }
65
66  /**
67   * Wczytuje dane pracownika
68   * @param in obiekt typu DataInput
69   * @return pracownik
70   */
71  public static Employee readData(DataInput in) throws IOException
72  {
73      String name = DataIO.readFixedString(Employee.NAME_SIZE, in);
74      double salary = in.readDouble();
75      int y = in.readInt();
76      int m = in.readInt();
77      int d = in.readInt();
78      return new Employee(name, salary, y, m - 1, d);
79  }
80 }

```

API java.io.RandomAccessFile 1.0

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

<i>Parametry:</i>	file	plik, który ma zostać otwarty.
	tryb	"r" dla samego odczytu, "rw" dla odczytu i zapisu, "rws" dla odczytu i zapisu danych wraz z synchronicznym zapisem danych i metadanych dla każdej aktualizacji, "rwd" dla odczytu i zapisu danych wraz z synchronicznym zapisem tylko samych danych.
- long getFilePointer()

zwraca aktualne położenie wskaźnika pliku.
- void seek(long pos)

zmienia położenie wskaźnika pliku, przesuując go o pos bajtów od początku pliku.
- long length()

zwraca długość pliku w bajtach.

2.3.3. 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. Aby uzyskać strumień wejściowy pozwalający na odczyt pozycji, reprezentujący ją obiekt `ZipEntry` należy przekazać do metody `getInputStream` obiektu `ZipInputStream`. Następnie, by odczytać kolejną pozycję archiwum, należy wywołać metodę `closeEntry`. Oto typowa sekwencja wywołań służąca do odczytu zawartości pliku ZIP:

```
ZipInputStream zin = ZipInputStream(new FileInputStream(zipname));
ZipEntry entry;
while ((entry = zin.getNextEntry()) != null)
{
    InputStream in = zin.getInputStream(entry);
    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:

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



Pliki JAR (omówione w rozdziale 13. książki *Java. Podstawy*) są po prostu plikami ZIP, zawierającymi specjalny rodzaj pliku, tzw. manifest. Do wczytania i zapisania manifestu używamy klas `JarInputStream` i `JarOutputStream`.

Strumień wejściowy ZIP są dobrym przykładem potęgi abstrakcji strumieni. Odczytując dane przechowywane w skompresowanej postaci, nie musimy zajmować się ich dekompresją. Źródło bajtów formatu ZIP nie musi być plikiem — dane ZIP mogą być ściągane przez połączenie sieciowe. Na przykład za każdym razem, gdy mechanizm ładowania klas jakiegoś apletu wczytuje plik JAR, tak naprawdę wczytuje i dekompresuje dane pochodzące z sieci.



W punkcie 2.5.8, „Systemy plików ZIP”, zobaczymy, w jaki sposób można korzystać z archiwów ZIP bez korzystania ze specjalnego interfejsu programowego, używając w tym celu klasy `FileSystem` wprowadzonej w Java SE 7.

API | `java.util.zip.ZipInputStream` 1.1

- `ZipInputStream(InputStream in)`
tworzy obiekt typu `ZipInputStream` umożliwiający dekompresję danych z podanego strumienia `InputStream`.
- `ZipEntry getNextEntry()`
zwraca obiekt typu `ZipEntry` opisujący następną pozycję archiwum lub `null`, jeżeli archiwum nie ma więcej pozycji.
- `void closeEntry()`
zamyka aktualnie otwartą pozycję archiwum ZIP. Dzięki temu możemy odczytać następną pozycję, wywołując metodę `getNextEntry()`.

API | `java.util.zip.ZipOutputStream` 1.1

- `ZipOutputStream(OutputStream out)`
tworzy obiekt typu `ZipOutputStream`, który umożliwia kompresję i zapis danych w podanym strumieniu `OutputStream`.
- `void putNextEntry(ZipEntry ze)`
zapisuje informacje podanej pozycji `ZipEntry` do strumienia i przygotowuje strumień do odbioru danych. Dane mogą zostać zapisane w strumieniu przy użyciu metody `write()`.
- `void closeEntry()`
zamyka aktualnie otwartą pozycję archiwum ZIP. Aby otworzyć następną pozycję, wywołujemy metodę `putNextEntry`.
- `void setLevel(int level)`
określa domyślny stopień kompresji następnych pozycji archiwum o trybie `DEFLATED`. Domyślną wartością jest `Deflater.DEFAULT_COMPRESSION`. Wyrzuca wyjątek `IllegalArgumentException`, jeżeli podany stopień jest nieprawidłowy.
Parametry: `level` stopień kompresji, od 0 (`NO_COMPRESSION`) do 9 (`BEST_COMPRESSION`).
- `void setMethod(int method)`
określa domyślną metodę kompresji dla danego `ZipOutputStream` dla wszystkich pozycji archiwum, dla których metoda kompresji nie została określona.
Parametry: `method` metoda kompresji, `DEFLATED` lub `STORED`.

API `java.util.zip.ZipEntry` **1.1**

- `ZipEntry(String name)`
tworzy pozycję archiwum o podanej nazwie.
Parametry: `name` nazwa elementu.
- `long getCrc()`
zwraca wartość sumy kontrolnej CRC32 danego elementu.
- `String getName()`
zwraca nazwę elementu.
- `long getSize()`
zwraca rozmiar danego elementu po dekompresji lub `-1`, jeżeli rozmiar nie jest znany.
- `boolean isDirectory()`
zwraca wartość logiczną, która określa, czy dany element archiwum jest katalogiem.
- `void setMethod(int method)`
Parametry: `method` metoda kompresji danego elementu, `DEFLATED` lub `STORED`.
- `void setSize(long rozmiar)`
określa rozmiar elementu. Wymagana, jeżeli metodą kompresji jest `STORED`.
Parametry: `rozmiar` rozmiar nieskompresowanego elementu.
- `void setCrc(long crc)`
określa sumę kontrolną CRC32 dla danego elementu. Aby obliczyć tę sumę używamy klasy `CRC32`. Wymagana, jeżeli metodą kompresji jest `STORED`.
Parametry: `crc` suma kontrolna elementu.

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

- `ZipFile(String name)`
- `ZipFile(File file)`
tworzy obiekt typu `ZipFile`, otwarty do odczytu, na podstawie podanego łańcucha lub obiektu typu `File`.
- `Enumeration entries()`
zwraca obiekt typu `Enumeration`, wyliczający obiekty `ZipEntry` opisujące elementy archiwum `ZipFile`.
- `ZipEntry getEntry(String name)`
zwraca element archiwum o podanej nazwie lub `null`, jeżeli taki element nie istnieje.
Parametry: `name` nazwa elementu.

- `InputStream getInputStream(ZipEntry ze)`
zwraca obiekt `InputStream` dla podanego elementu.
Parametry: `ze` element `ZipEntry` w pliku ZIP.
- `String getName()`
zwraca ścieżkę dostępu do pliku ZIP.

2.4. 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”).

2.4.1. Zapisywanie i wczytywanie obiektów serializowalnych

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

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

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

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

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

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

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

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

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

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

Interfejs `Serializable` nie posiada metod, nie musimy zatem wprowadzać żadnych innych modyfikacji naszych klas. Pod tym względem `Serializable` jest podobny do interfejsu `Cloneable`, który omówiliśmy w rozdziale 6. książki *Java. 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:

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

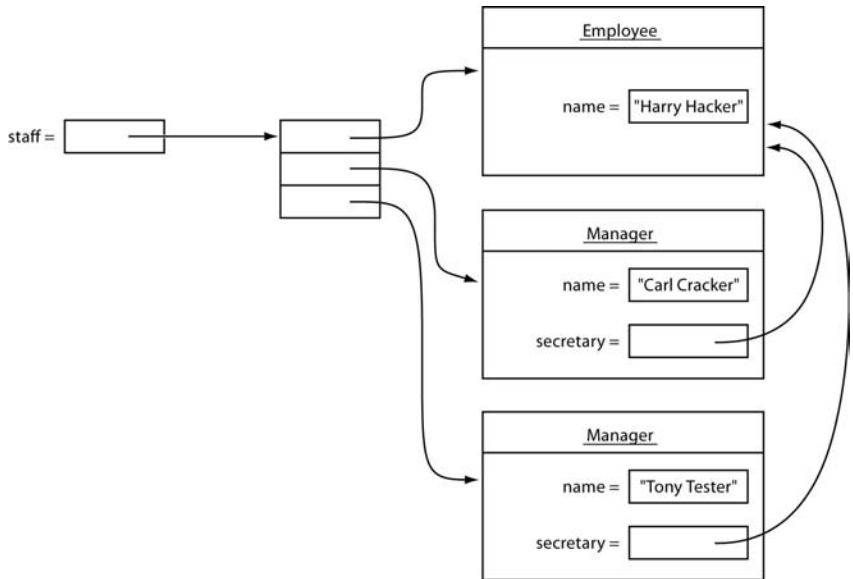
Teraz założmy, że zapisujemy dane pracowników na dysk. Oczywiście nie możemy zapisać i przywrócić adresów obiektów asystentów w pamięci, ponieważ po ponownym załadowaniu obiekt asystenta najprawdopodobniej znajdzie się w zupełnie innym miejscu pamięci.

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

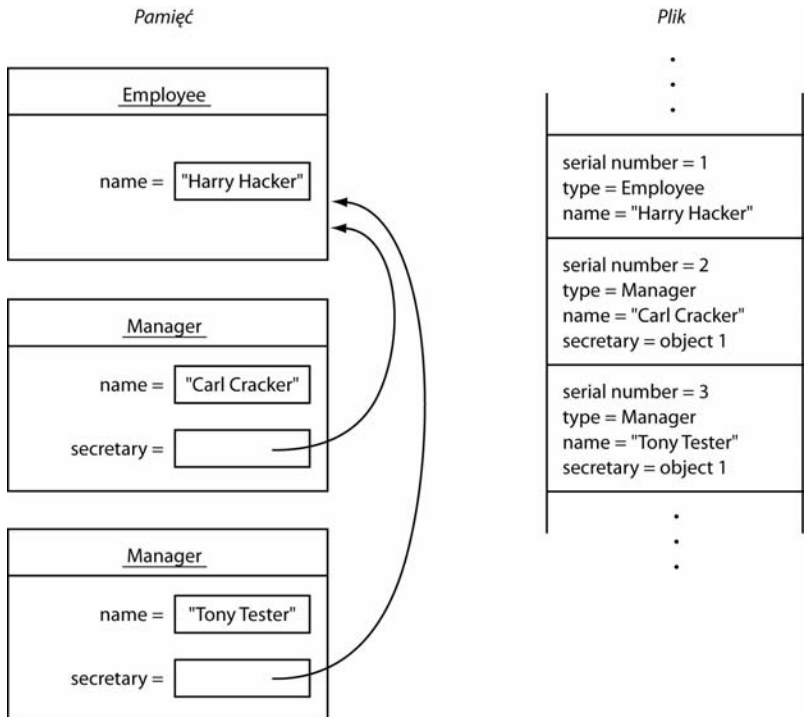
1. Wszystkim napotkanym referencjom do obiektów nadawane są numery seryjne (patrz rysunek 2.6).
2. Jeśli referencja do obiektu została napotkana po raz pierwszy, obiekt zostaje zapisany w strumieniu wyjściowym.
3. Jeżeli obiekt został już zapisany, Java zapisuje, że w danym miejscu znajduje się „ten sam obiekt, co pod numerem seryjnym x”.

Rysunek 2.5.

Dwóch
menedżerów
może mieć
wspólnego
asystenta

**Rysunek 2.6.**

Przykład serializacji
obiektów



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 beużyteczne dla pliku, tak samo są beużyteczne dla innego rodzaju procesora. Ponieważ serializacja zastępuje adresy pamięci numerami seryjnymi, możemy transportować zbiory danych z jednego komputera do drugiego.

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 sekretarzy obiektów klasy `Manager`.

Listing 2.3. *objectStream/ObjectStreamTest.java*

```

1 package objectStream;
2
3 import java.io.*;
4
5 /**
6  * @version 1.10 17 Aug 1998
7  * @author Cay Horstmann
8  */
9 class ObjectStreamTest
10 {
11     public static void main(String[] args)
12         throws IOException, ClassNotFoundException
13     {
14         Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
15         Manager carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
16         carl.setSecretary(harry);
17         Manager tony = new Manager("Tony Tester", 40000, 1990, 3, 15);
18         tony.setSecretary(harry);
19
20         Employee[] staff = new Employee[3];
21
22         staff[0] = carl;
23         staff[1] = harry;
24         staff[2] = tony;
25
26         // zapisuje rekordy wszystkich pracowników w pliku employee.dat
27         try (ObjectOutputStream out =
28             new ObjectOutputStream(new FileOutputStream("employee.dat")))
29         {
30             out.writeObject(staff);
31         }
32
33         try (ObjectInputStream in =
34             new ObjectInputStream(new FileInputStream("employee.dat")))
35         {
36             // wczytuje wszystkie rekordy do nowej tablicy

```

```

37
38     Employee[] newStaff = (Employee[]) in.readObject();
39
40     // podnosi wynagrodzenie asystenta
41     newStaff[1].raiseSalary(10);
42
43     // wyświetla wszystkie wczytane rekordy
44     for (Employee e : newStaff)
45         System.out.println(e);
46     }
47 }
48 }

```

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, nieprzechodnych pól składowych tej klasy, a także jej nadklas.

API java.io.ObjectInputStream 1.1

■ ObjectInputStream(InputStream we)

tworzy obiekt ObjectInputStream, dzięki któremu możesz odczytywać informacje z podanego strumienia wejścia.

■ Object readObject()

wczytuje obiekt z ObjectInputStream. Pobiera klasę obiektu, sygnaturę klasy oraz wartości wszystkich niestatycznych, nieprzechodnych pól składowych tej klasy, a także jej nadklas. Przeprowadza deserializację, pozwalając na przyporządkowanie obiektów referencjom.

2.4.2. 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 (2 bajty) znaki

Dla przykładu, łańcuch "Harry" będzie wyglądał tak:

74 00 05 Harry

Znaki Unicode zapisywane są w zmodyfikowanym formacie UTF-8.

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

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

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

SHA to szybki algorytm, tworzący „odciski palców” dla dużych bloków danych. Niezależnie od rozmiaru oryginalnych danych, „odciskiem” jest zawsze pakiet 20 bajtów. Jest on tworzony za pomocą sekwencji operacji binarnych, dzięki którym możemy mieć stuprocentową pewność, że jeżeli zachowana informacja zmieni się, zmianie ulegnie również jej „odcisk palca”. (Aby dowiedzieć się więcej o SHA, zajrzyj na przykład do książki *Cryptography and Network Security: Seventh Edition* autorstwa Williama Stallingsa, wydanej przez Prentice Hall w 2016 r.). Jednakże Java korzysta jedynie z pierwszych ośmiu bajtów kodu SHA. Mimo to nadal jest bardzo prawdopodobne, że „odcisk” zmieni się, jeżeli ulegną zmianie pola składowe lub metody.

W chwili odczytu 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 omówimy w punkcie 2.4.5, „Wersje”.

Oto w jaki sposób przechowywany jest identyfikator klasy:

- 72
- długość nazwy klasy (2 bajty)
- nazwa klasy
- „odcisk” klasy (8 bajtów)
- zbiór flag (1 bajt)

- liczba deskryptorów pól składowych (2 bajty)
- deskryptory pól składowych
- 78 (znacznik końca)
- typ klasy bazowej (70, jeśli nie istnieje)

Bajt flag składa się z trzybitowych masek, zdefiniowanych w `java.io.ObjectStreamConstants`:

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

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

Każdy deskryptor pola składowego składa się z następujących elementów:

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

Kod typu może mieć jedną z następujących wartości:

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

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

Dla przykładu, pole `salary` klasy `Employee` zostanie zapisane jako:

D 00 06 salary

A oto kompletny opis klasy `Employee`:

72 00 08 Employee

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

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

71 numer seryjny (4 bajty)

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

Obiekt jest przechowywany w następującej postaci:

73 opis klasy dane obiektu

Dla przykładu, oto zapis obiektu klasy `Employee`:

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

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

Tablice są zapisywane w następujący sposób:

75 opis klasy liczba elementów (4 bajty) elementy

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

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

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

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

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

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

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

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

Referencja `null` jest zapisywana jako

```
70
```

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

AC ED 00 05	Nagłówek pliku
75	Tablica <code>staff</code> (nr #1)
72 00 0B [LEmployee;	Nowa klasa, długość łańcucha, nazwa klasy <code>Employee[]</code> (nr #0)
FC BF 36 11 C5 91 11 C7 02	„Odcisk” oraz flagi

00 00	Liczba pól składowych
78	Znacznik końca
70	Brak klasy bazowej
00 00 00 03	Liczba elementów tablicy
73	staff[0] — nowy obiekt (nr #7)
72 00 07 Manager	Nowa klasa, długość łańcucha, nazwa klasy (nr #2)
36 06 AE 13 63 8F 59 B7 02	„Odcisk” oraz flagi
00 01	Liczba pól składowych
L 00 09 secretary	Typ i nazwa pola składowego
74 00 0A LEmployee;	Nazwa klasy pola składowego — String (nr #3)
78	Znacznik końca
72 00 08 Employee	Nadklasa — nowa klasa, długość łańcucha, nazwa klasy (nr #4)
E6 D2 86 7D AE AC 18 1B 02	„Odcisk” oraz flagi
00 03	Liczba pól składowych
D 00 06 salary	Typ i nazwa pola składowego
L 00 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” oraz flagi
00 00	Brak zmiennych składowych

78	Znacznik końca
70	Brak klasy bazowej
77 08	Zawartość zewnętrzna, liczba bajtów
00 00 00 83 E9 39 E0 00	Data
78	Znacznik końca
74 00 0C Car1 Cracker	Wartość pola name — String (nr #10)
73	Wartość pola secretary — nowy obiekt (nr #11)
71 00 7E 00 04	Istniejąca klasa (użyj nr #4)
40 E8 6A 00 00 00 00 00	Wartość pola pensja — double
73	Wartość pola dzienZatrudnienia — nowy obiekt (nr #12)
71 00 7E 00 08	Istniejąca klasa (użyj nr #8)
77 08	Zawartość zewnętrzna, liczba bajtów
00 00 00 91 1B 4E B1 80	Data
78	Znacznik końca
74 00 0C Harry Hacker	Wartość pola name — String (nr #13)
71 00 7E 00 0B	staff[1] — istniejący obiekt (użyj nr #11)
73	obsługa[2] — nowy obiekt (nr #14)
71 00 7E 00 08	Istniejąca klasa (użyj nr #4)
40 E3 88 00 00 00 00 00	Wartość pola salary — double
73	Wartość pola hireDay — nowy obiekt (nr #15)
71 00 7E 00 08	Istniejąca klasa (użyj nr #8)
77 08	Zawartość zewnętrzna, liczba bajtów
00 00 00 94 6D 3E EC 00 00	Data
78	Znacznik końca
74 00 0D Tony Tester	Wartość pola name — String (nr #16)
71 00 7E 00 0B	Wartość pola secretary — istniejący obiekt (użyj nr #11)

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.

2.4.3. Modyfikowanie domyślnego mechanizmu serializacji

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

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

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

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

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

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

W metodzie `writeObject` najpierw zapiszemy opis obiektu i pole typu `String`, wywołując metodę `defaultWriteObject`. Jest to specjalna metoda klasy `ObjectOutputStream`, która może być wywoływana jedynie przez metodę `writeObject` klasy implementującej interfejs

Serializable. Następnie zapiszemy współrzędne punktu, używając standardowych wywołań klasy `DataOutput`.

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

Implementując metodę `readObject`, odwrócimy cały proces:

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

Innym przykładem jest klasa `java.util.Date`, która dostarcza własnych metod `readObject` i `writeObject`. Metody te zapisują datę jako liczbę milisekund, które upłynęły od północy 1 stycznia 1970 roku, czasu UTC. Klasa `Date` stosuje skomplikowaną reprezentację wewnętrzną, która przechowuje zarówno obiekt klasy `Calendar`, jak i licznik milisekund, co pozwala zoptymalizować operacje wyszukiwania. Stan obiektu klasy `Calendar` jest wtórny i nie musi być zapisywany.

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

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

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

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

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

```

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

```



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

2.4.4. Serializacja singletonów i wyliczeń

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

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

```

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

```

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

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

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

```

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

```

Okaże się, że porównanie

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

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

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

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

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

2.4.5. 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

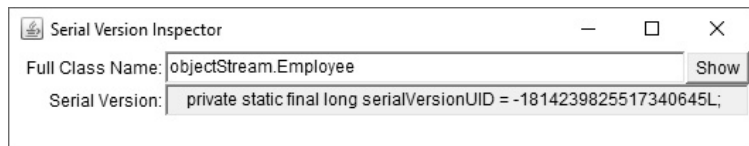
```
serialver Employee
```

otrzymujemy:

```
Employee:      static final long serialVersionUID = -1814239825517340645L;
```

Jeżeli uruchomimy `serialver` z opcją `-show`, program wyświetli okno dialogowe (rysunek 2.7).

Rysunek 2.7.
Wersja graficzna
programu `serialver`



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

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

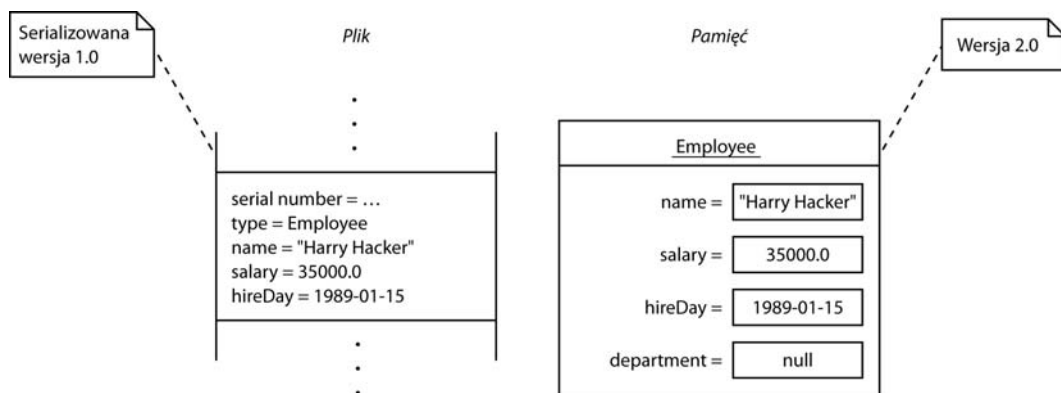
Klasa posiadająca statyczne pole składowe o nazwie `serialVersionUID` nie obliczy własnego „odcisku palca”, ale skorzysta z już istniejącej wartości.

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

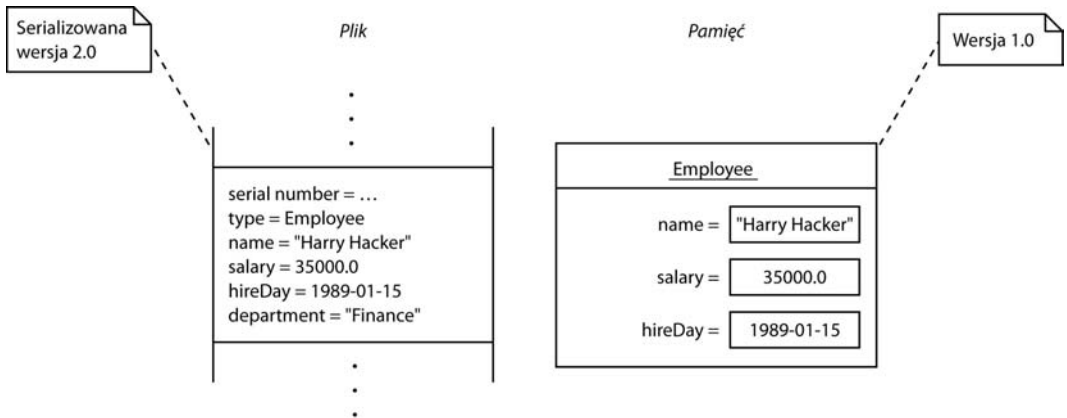
Jeśli zmieniają się tylko metody danej klasy, sposób odczytu danych nie ulegnie zmianie. Jednakże jeżeli zmieni się pole składowe, możemy mieć pewne problemy. Dla przykładu, stary obiekt może posiadać więcej lub mniej pól składowych niż aktualny albo też typy danych mogą się różnić. W takim wypadku wejściowy 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 niestyczne, 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 posiada pola składowe nieobecne w aktualnej wersji, strumień ignoruje te dodatkowe dane. Jeżeli aktualna wersja posiada 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 2.8 pokazuje, co się dzieje, gdy obiekt wersji 1.0 jest wczytywany przez program korzystający z obiektów 2.0. Pole `department` otrzymuje wartość `null`. Rysunek 2.9 ilustruje odwrotną sytuację — program korzystający z obiektów 1.0 wczytuje obiekt 2.0. Dodatkowe pole `department` jest ignorowane.



Rysunek 2.8. Odczytywanie obiektu o mniejszej liczbie pól



Rysunek 2.9. Odczytywanie obiektu o większej liczbie pól

Czy ten proces jest bezpieczny? To zależy. Opuszczanie pól składowych wydaje się być bezbolesne — odbiorca wciąż posiada dane, którymi potrafi manipulować. Nadawanie wartości `null` nie jest już tak bezpieczne. Wiele klas inicjalizuje wszystkie pola składowe, nadając im w konstruktorach niezerowe wartości, tak więc metody mogą być nieprzygotowane do obsługi wartości `null`. Od projektanta klasy zależy, czy zaimplementuje w metodzie `readObject` dodatkowy kod poprawiający wyniki wczytywania różnych wersji danych, czy też dołączy do metod obsługujących wartości `null`.

2.4.6. 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 `Serializable`.

Listing 2.4. `serialClone/SerailCloneTest.java`

```

1 package serialClone;
2
3 /**
4  * @version 1.21 13 Jul 2016
5  * @author Cay Horstmann
6  */
7
8 import java.io.*;
9 import java.util.*;
10 import java.time.*;
11
12 public class SerialCloneTest

```

```
13 {
14     public static void main(String[] args) throws CloneNotSupportedException
15     {
16         Employee harry = new Employee("Harry Hacker", 35000, 1989, 10, 1);
17         // klonuje obiekt harry
18         Employee harry2 = (Employee) harry.clone();
19
20         // modyfikuje obiekt harry
21         harry.raiseSalary(10);
22
23         // teraz obiekt harry i jego klon różnią się
24         System.out.println(harry);
25         System.out.println(harry2);
26     }
27 }
28
29 /**
30  * Klasa, której metoda clone wykorzystuje serializację.
31  */
32 class SerialCloneable implements Cloneable, Serializable
33 {
34     public Object clone() throws CloneNotSupportedException
35     {
36         try {
37             // zapisuje obiekt w tablicy bajtów
38             ByteArrayOutputStream bout = new ByteArrayOutputStream();
39             try (ObjectOutputStream out = new ObjectOutputStream(bout))
40             {
41                 out.writeObject(this);
42             }
43
44             // wczytuje klon obiektu z tablicy bajtów
45             try (InputStream bin = new ByteArrayInputStream(bout.toByteArray()))
46             {
47                 ObjectInputStream in = new ObjectInputStream(bin);
48                 return in.readObject();
49             }
50         }
51         catch (IOException | ClassNotFoundException e)
52         {
53             CloneNotSupportedException e2 = new CloneNotSupportedException();
54             e2.initCause(e);
55             throw e2;
56         }
57     }
58 }
59
60 /**
61  * Znana już klasa Employee,
62  * tym razem jako pochodna klasy SerialCloneable.
63  */
64 class Employee extends SerialCloneable
65 {
66     private String name;
67     private double salary;
68     private LocalDate hireDay;
```

```
69
70 public Employee(String n, double s, int year, int month, int day)
71 {
72     name = n;
73     salary = s;
74     hireDay = LocalDate.of(year, month, day);
75 }
76
77 public String getName()
78 {
79     return name;
80 }
81
82 public double getSalary()
83 {
84     return salary;
85 }
86
87 public LocalDate getHireDay()
88 {
89     return hireDay;
90 }
91
92 /**
93  * Metoda zwiększająca wynagrodzenie tego pracownika.
94  * @byPercent procentowa wartość wzrostu wynagrodzenia
95  */
96 public void raiseSalary(double byPercent)
97 {
98     double raise = salary * byPercent / 100;
99     salary += raise;
100 }
101
102 public String toString()
103 {
104     return getClass().getName()
105         + "[name=" + name
106         + ",salary=" + salary
107         + ",hireDay=" + hireDay
108         + "]\n";
109 }
110 }
```

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

2.5. Zarządzanie 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.

Klasy `Path` i `Files` wprowadzono w wersji Java SE 7. Posługiwanie się nimi jest znacznie wygodniejsze niż klasą `File` pamiętającą jeszcze czasy pakietu JDK 1.0. Spodziewamy się, że zyskają one sporą popularność wśród programistów i dlatego omawiamy je szczegółowo.

2.5.1. 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 *względą*. 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 wyrzuca 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 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 odpowiadającą 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`.
- W przeciwnym razie wynik ma postać połączenia „`p` potem `q`” wykonanego zgodnie z regułami systemu plików.

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

```
Path workRelative = Paths.get("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");
```

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.relativeTo(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, rozpoczynając 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 /
```



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.Paths` 7

- `static Path get(String first, String... more)`
tworzy ścieżkę, łącząc podane łańcuchy.

API | `java.nio.file.Path` 7

- `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 ścieżkę bezwzględną odpowiadającą danej ścieżce.

- `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.

2.5.2. Odczyt i zapis plików

Klasa `Files` pozwala przyspieszyć programowanie typowych operacji na plikach. Na przykład poniższe wywołanie umożliwia wczytanie całej zawartości pliku:

```
byte[] bytes = Files.readAllBytes(path);
```

Jeśli chcemy uzyskać tę zawartość w postaci łańcucha znaków, to po wywołaniu metody `readAllBytes` wystarczy wykonać poniższą instrukcję.

```
String content = new String(bytes, charset);
```

Jeśli natomiast wolelibyśmy zawartość pliku w postaci sekwencji wierszy, wtedy posłużymy się następującym wywołaniem:

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

Jeśli z kolei chcemy zapisać łańcuch znaków w pliku, wywołamy:

```
Files.write(path, content.getBytes(charset));
```

Aby dopisać łańcuch do pliku, zastosujemy poniższe wywołanie.

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

Kolekcję wierszy możemy zapisać w następujący sposób:

```
Files.write(path, lines);
```

Te proste metody zostały udostępnione z myślą o obsłudze plików tekstowych o umiarkowanych rozmiarach. Jeśli przetwarzane pliki są znacznych rozmiarów lub zawierają dane binarne, to nadal z powodzeniem możemy 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`.

`java.nio.file.Files` 7

- `static byte[] readAllBytes(Path path)`
- `static List<String> readAllLines(Path path, Charset charset)`
wczytują zawartość pliku.
- `static Path write(Path path, byte[] contents, OpenOption... options)`
- `static Path write(Path path, Iterable<? extends CharSequence> contents, OpenOption options)`
zapisują podaną zawartość w pliku i zwracają ścieżkę.
- `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.

2.5.3. 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 wyrzuci 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.

2.5.4. Kopiowanie, przenoszenie i usuwanie plików

Aby skopiować plik z jednej lokalizacji do innej, wywołamy:

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

Aby przenieść plik (czyli skopiować go i usunąć oryginał), 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 wyrzuci 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 2.3.

Tabela 2.3. 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	Podpowieź 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 wyrzuca wyjątek, jeśli plik lub katalog nie istnieje. W takim przypadku metoda deleteIfExists zwraca wartość false.

2.5.5. 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.

2.5.6. 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 odwiedzeniem 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, łącznie 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.

2.5.7. 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 2.4.

Tabela 2.4. 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 Testx.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.newDirectoryStream(dir, "C:\\\\")`.

Jeśli chcemy odwiedzić 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 odwiedzenia 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 odwiedzanie następnego pliku: `FileVisitResult.CONTINUE;`
- kontynuować przeglądanie, ale bez odwiedzania kolejnych elementów danego katalogu: `FileVisitResult.SKIP_SUBTREE;`
- kontynuować przeglądanie, ale bez odwiedzania rodzeństwa danego pliku: `FileVisitResult.SKIP_SIBLINGS;`
- zakończyć przeglądanie: `FileVisitResult.TERMINATE.`

Jeśli którakolwiek z metod wyrzuci wyjątek, przeglądanie zostanie zakończone, a metoda `walkFileTree` wyrzuci 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`. Metody — oprócz `visitFileFailed` — nie podejmują żadnych działań, powodując tym samym kontynuację przeglądania. Metoda `visitFileFailed` wyrzuca wyjątek będący przyczyną błędu i w ten sposób kończy przeglądanie.

Poniżej przedstawiamy fragment kodu wyświetlający wszystkie podkatalogi danego katalogu.

```
Files.walkFileTree(Paths.get("/"), 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 exc)
    {
        return FileVisitResult.CONTINUE;
    }
    public FileVisitResult visitFileFailed(Path path, IOException exc)
        throws IOException
    {
        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 `preVisitDirectory` oraz `visitFile` jako ich parametr. 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 odwiedzając lub opuszczając katalog, 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)`
odwiedza 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 odwiedzany 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, BasicFileAttributes attrs)`
wywoływane przed i po wizycie w katalogu. Domyślna implementacja nie podejmuje żadnych działań, tym samym powodując kontynuację przeglądania.
- `static FileVisitResult visitFileFailed(T path, IOException exc)`
wywoływana, gdy podczas próby uzyskania informacji o danym pliku został wyrzucony wyjątek. Domyślna implementacja ponownie wyrzuca ten wyjątek, powodując zakończenie przeglądania. Jeśli przeglądanie ma być kontynuowane, należy zastąpić metodę własną implementacją.

2.5.8. 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(Paths.get(zipname), null);
```

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 `Paths.get` 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 2.3.3, „Archiwa ZIP”, wymagająca użycia całego zestawu odpowiednich klas.

API `java.nio.file.FileSystems` 7

- `static FileSystem newFileSystem(Path path, ClassLoader loader)`

przegląda zainstalowanych dostawców systemów plików oraz systemy plików, które może załadować `loader` (jeśli jest różny od `null`). 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.

2.6. 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.

2.6.1. 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 2.5 dla pliku *rt.jar* (37 MB) znajdującego się w katalogu *jre/lib* pakietu JDK.

Tabela 2.5. Czas wykonywania operacji na pliku

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

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

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

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

Najpierw musimy uzyskać *kanal* dostępu do pliku. Kanal 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. Kanal uzyskujemy, wywołując metodę `getChannel` dodaną do klas `FileInputStream`, `FileOutputStream` i `RandomAccessFile`.

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

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

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

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

Alternatywnie możemy również wykorzystać dostęp swobodny:

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

Możemy także czytać tablice bajtów, stosując metody:

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

Dostępne są również poniższe metody:

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

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

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

Aby poznać bieżący sposób uporządkowania bajtów w buforze, wywołujemy:

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



Ta para metod nie stosuje konwencji nazw set/get.

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

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

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:

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

Listing 2.5. `memoryMap/MemoryMapTest.java`

```
1 package memoryMap;
2
3 import java.io.*;
4 import java.nio.*;
5 import java.nio.channels.*;
6 import java.nio.file.*;
```

```
7 import java.util.zip.*;
8
9 /**
10  * Program obliczający sumę kontrolną CRC pliku.
11  * Uruchamianie: java memoryMap.MemoryMapTest nazwapliku
12  * @version 1.01 2012-05-30
13  * @author Cay Horstmann
14  */
15 public class MemoryMapTest
16 {
17     public static long checksumInputStream(Path filename) throws IOException
18     {
19         try (InputStream in = Files.newInputStream(filename))
20         {
21             CRC32 crc = new CRC32();
22
23             int c;
24             while ((c = in.read()) != -1)
25                 crc.update(c);
26             return crc.getValue();
27         }
28     }
29
30     public static long checksumBufferedInputStream(Path filename)
31         throws IOException
32     {
33         try (InputStream in =
34             new BufferedInputStream(Files.newInputStream(filename)))
35         {
36             CRC32 crc = new CRC32();
37
38             int c;
39             while ((c = in.read()) != -1)
40                 crc.update(c);
41             return crc.getValue();
42         }
43     }
44
45     public static long checksumRandomAccessFile(Path filename) throws IOException
46     {
47         try (RandomAccessFile file = new RandomAccessFile(filename.toFile(), "r"))
48         {
49             long length = file.length();
50             CRC32 crc = new CRC32();
51
52             for (long p = 0; p < length; p++)
53             {
54                 file.seek(p);
55                 int c = file.readByte();
56                 crc.update(c);
57             }
58             return crc.getValue();
59         }
60     }
61
62     public static long checksumMappedFile(Path filename) throws IOException
63     {
64         try (FileChannel channel = FileChannel.open(filename))
```

```
65     {
66         CRC32 crc = new CRC32();
67         int length = (int) channel.size();
68         MappedByteBuffer buffer = channel.map(
69             FileChannel.MapMode.READ_ONLY, 0, length);
70
71         for (int p = 0; p < length; p++)
72         {
73             int c = buffer.get(p);
74             crc.update(c);
75         }
76         return crc.getValue();
77     }
78 }
79
80 public static void main(String[] args) throws IOException
81 {
82     System.out.println("Input Stream:");
83     long start = System.currentTimeMillis();
84     Path filename = Paths.get(args[0]);
85     long crcValue = checksumInputStream(filename);
86     long end = System.currentTimeMillis();
87     System.out.println(Long.toHexString(crcValue));
88     System.out.println((end - start) + " milisekund");
89
90     System.out.println("Buffered Input Stream:");
91     start = System.currentTimeMillis();
92     crcValue = checksumBufferedInputStream(filename);
93     end = System.currentTimeMillis();
94     System.out.println(Long.toHexString(crcValue));
95     System.out.println((end - start) + " milisekund");
96
97     System.out.println("Random Access File:");
98     start = System.currentTimeMillis();
99     crcValue = checksumRandomAccessFile(filename);
100    end = System.currentTimeMillis();
101    System.out.println(Long.toHexString(crcValue));
102    System.out.println((end - start) + " milisekund");
103
104    System.out.println("Mapped File:");
105    start = System.currentTimeMillis();
106    crcValue = checksumMappedFile(filename);
107    end = System.currentTimeMillis();
108    System.out.println(Long.toHexString(crcValue));
109    System.out.println((end - start) + " milisekund");
110 }
111 }
```



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

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

Parametry: path ścieżka dostępu do pliku

options wartości WRITE, APPEND, TRUNCATE_EXISTING, CREATE należące do typu wyliczeniowego StandardOpenOption

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

tworzy w pamięci mapę fragmentu pliku.

Parametry: mode jedna ze stałych READ_ONLY, READ_WRITE lub PRIVATE zdefiniowanych w klasie FileChannel.MapMode

position początek mapowanego fragmentu

size rozmiar mapowanego fragmentu

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 wyrzucony wyjątek `BufferUnderflowException`. Zwracają referencję do bufora.

Parametry:

<code>destination</code>	wypełniana tablica bajtów
<code>offset</code>	początek wypełnianego zakresu
<code>length</code>	rozmiar wypełnianego zakresu

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

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

Parametry:

<code>source</code>	tablica stanowiąca źródło bajtów zapisywanych w buforze
<code>offset</code>	początek zakresu źródła
<code>length</code>	rozmiar zakresu źródła

■ `Xxx getXxx()`■ `Xxx getXxx(int index)`■ `ByteBuffer putXxx(xxx value)`■ `ByteBuffer putXxx(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`.

2.6.2. 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`.

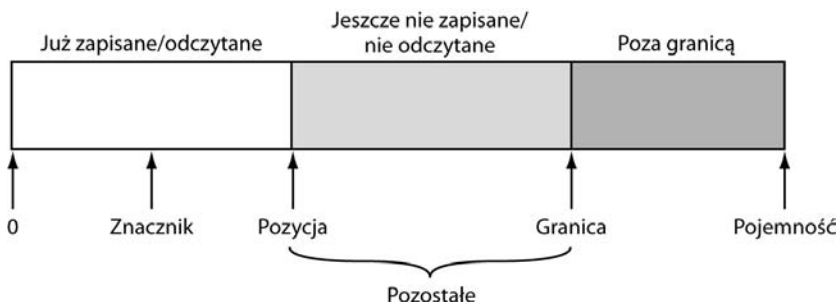


Klasa `StringBuffer` nie jest związana z omawianą tutaj hierarchią klas.

W praktyce najczęściej używane są klasy `ByteBuffer` i `CharBuffer`. Na rysunku 2.10 pokazaliśmy, że bufor jest scharakteryzowany przez:

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

Rysunek 2.10.
Bufor



Wymienione wartości spełniają następujący warunek:

$$0 \leq \text{znacznik} \leq \text{pozycja} \leq \text{granica} \leq \text{pojemność}$$

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

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

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

Aby uzyskać bufor, wywołujemy metodę statyczną taką jak `ByteBuffer.allocate` lub `ByteBuffer.wrap`.

Następnie możemy wypełnić bufor, korzystając z kanału, lub zapisać zawartość bufora do kanału. Na przykład:

```
ByteBuffer buffer = ByteBuffer.allocate(RECORD_SIZE);
channel.read(buffer);
channel.position(newpos);
buffer.flip();
channel.write(buffer);
```

Rozwiązanie takie może okazać się przydatną alternatywą pliku o dostępie swobodnym.

API java.nio.Buffer 1.4

- `Buffer clear()`
przygotowuje bufor do zapisu, nadając pozycji wartość 0, a granicy wartość równą pojemności bufora; zwraca `this`.
- `Buffer flip()`
przygotowuje bufor do zapisu, nadając granicy wartość równą pozycji, a następnie zerując wartość pozycji; zwraca `this`.
- `Buffer rewind()`
przygotowuje bufor do ponownego odczytu tych samych wartości, nadając pozycji wartość 0 i pozostawiając wartość granicy bez zmian; zwraca `this`.
- `Buffer mark()`
nadaje znacznikowi wartość pozycji; zwraca `this`.
- `Buffer reset()`
nadaje pozycji bufora wartość znacznika, umożliwiając w ten sposób ponowny odczyt lub zapis danych; zwraca `this`.
- `int remaining()`
zwraca liczbę wartości pozostających do odczytu lub zapisu; jest to różnica pomiędzy wartością granicy i pozycji.
- `int position()`
- `void position(int newValue)`
zwracają i określają pozycję bufora.
- `int capacity()`
zwraca pojemność bufora.

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

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

```
FileChannel = 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 blokady plików mają jedynie charakter *pomocniczy*. Nawet jeśli aplikacji nie uda się zdobyć blokady, to może zapisywać dane w pliku „zablokowanym” wcześniej przez inną aplikację.
- W niektórych systemach nie jest możliwe zablokowanie dostępu do mapy pliku w pamięci.
- Blokady plików są przydzielane na poziomie maszyny wirtualnej Java. Jeśli zatem dwa programy działają na tej samej maszynie wirtualnej, to nie mogą uzyskać blokady tego samego pliku. Metody `lock` i `tryLock` wyrzucą wyjątek `OverlappingFileLockException` w sytuacji, gdy maszyna wirtualna jest już w posiadaniu blokady danego pliku.

- W niektórych systemach zamknięcie kanału zwalnia wszystkie blokady pliku będące w posiadaniu maszyny wirtualnej Java. Dlatego też należy unikać wielu kanałów dostępu do tego samego, zablokowanego pliku.
- Działanie blokad plików w sieciowych systemach plików zależy od konkretnego systemu i dlatego należy unikać stosowania blokad w takich systemach.

API java.nio.channels.FileChannel 1.4

- `FileLock lock()`
uzyskuje wyłączną blokadę pliku. Blokuje działanie programu do momentu uzyskania blokady.
 - `FileLock tryLock()`
uzyskuje wyłączną blokadę całego pliku lub zwraca `null`, jeśli nie może uzyskać blokady.
 - `FileLock lock(long position, long size, boolean shared)`
 - `FileLock tryLock(long position, long size, boolean shared)`
uzyskuje blokadę dostępu do fragmentu pliku. Pierwsza wersja blokuje działanie programu do momentu uzyskania blokady, a druga zwraca natychmiast wartość `null`, jeśli nie może uzyskać od razu blokady.
- | | | |
|-------------------|-----------------------|---|
| <i>Parametry:</i> | <code>position</code> | początek blokowanego fragmentu |
| | <code>size</code> | rozmiar blokowanego fragmentu |
| | <code>shared</code> | wartość <code>true</code> dla blokady współdzielonej, <code>false</code> dla wyłączonej |

API java.nio.channels.FileLock 1.4

- `void close()` 1.7
zwalnia blokadę.

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 wzorec, musimy dokładnie określić znaki, które są dopuszczalne. Dlatego też opis wzorca wymaga zastosowania odpowiedniej składni.

Oto prosty przykład. Z wyrażeniem regularnym

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

może zostać uzgodniony dowolny łańcuch znaków następującej postaci:

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

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

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

- Przez *klasę znaków* rozumiemy zbiór alternatywnych znaków ujęty w nawiasy kwadratowe, na przykład [Jj], [0-9], [A-Za-z] czy [^0-9]. Znak - oznacza zakres (czyli wszystkie znaki, których kody Unicode leżą w podanych granicach), a znak ^ oznacza dopełnienie (wszystkie znaki oprócz podanych).
- Jeśli klasa ma zawierać znak łącznika -, to musimy umieścić go jako pierwszy lub ostatni znak w definicji klasy. W przypadku znaku [musimy umieścić go jako pierwszy. Natomiast znak ^ możemy umieścić w dowolnym miejscu definicji klasy z wyjątkiem pierwszego. Sekwencję specjalną musimy zastosować w przypadku znaku \.
- Istnieje wiele wstępnie zdefiniowanych klas znaków, takich jak \d (cyfry) czy \p{Sc} (symbol waluty w Unicode). Patrz przykłady w tabelach 2.6 i 2.7.
- Większość znaków oznacza samą siebie, tak jak znaki ava w poprzednim przykładzie.
- Symbol . oznacza dowolny znak (z wyjątkiem, być może, znaków końca wiersza, co zależy od stanu odpowiedniego znacznika).
- \ spełnia rolę znaku specjalnego, na przykład \. oznacza znak kropki, a \\ znak lewego ukośnika.
- ^ i \$ oznaczają odpowiednio początek i koniec wiersza.
- Jeśli X i Y są wyrażeniami regularnymi, to XY oznacza „dowolne dopasowanie do X , po którym następuje dowolne dopasowanie do Y ”, a $X|Y$ „dowolne dopasowanie do X lub Y ”.
- Do wyrażenia regularnego X możemy stosować *kwantyfikatory* X^+ (raz lub więcej), X^* (0 lub więcej) i $X?$ (0 lub 1).
- Domyślnie kwantyfikator dopasowuje największą możliwą liczbę wystąpień, która gwarantuje ogólne powodzenie dopasowania. Zachowanie to możemy zmodyfikować za pomocą przyrostka ? (dopasowanie najmniejszej liczby wystąpień) i przyrostka + (dopasowanie największej liczby wystąpień, nawet jeśli nie gwarantuje ono ogólnego powodzenia dopasowania).

Tabela 2.6. 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{p}	Znak Unicode o szesnastkowej wartości p.	\x{1D546}
\uhhhh, \xhh, \0o, \0oo, \0ooo	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
\cc, gdzie c należy do [A-Z] bądź jest jednym ze znaków @ [\] ^ _ ?	Znak sterujący odpowiadający znakowi c.	\cH to znak cofnięcia (ang. <i>backspace</i> ; \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 (. . .)
Klasy znaków		
[C ₁ C ₂ . . .], gdzie C _i jest znakiem, zakresem znaków c – d lub klasą znaków	Dowolny ze znaków reprezentowanych przez C ₁ C 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
Wstępnie zdefiniowane klasy znaków		
\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).	

Tabela 2.6. Składnia wyrażeń regularnych — ciąg dalszy

Składnia	Objaśnienie	Przykład
Wstępnie zdefiniowane klasy znaków		
<code>\s, \S</code>	Znak odstępu (<code>[\t\n\r\f\x0B]</code> lub <code>\p{IsWhiteSpace}</code> , jeśli została użyta flaga <code>UNICODE_CHARACTER_CLASS</code>); znak, który nie jest odstępem.	<code>\s*, \s*</code> — przecinek opcjonalnie otoczony znakami odstępu
<code>\h, \v, \H, \V</code>	Odstęp w poziomie, odstęp w pionie; dopełnienia tych znaków.	
Sekwencje i alternatywy		
<code>XY</code>	Dowolny łańcuch z <i>X</i> , po którym następuje dowolny łańcuch z <i>Y</i> .	<code>[1-9][0-9]*</code> — liczba dodatnia bez początkowego zera
<code>X Y</code>	Dowolny łańcuch z <i>X</i> lub <i>Y</i> .	<code>http ftp</code>
Grupowanie		
<code>(X)</code>	Przechwycenie dopasowania <i>X</i> .	<code>'([^\']**)'</code> — tekst w apostrofach
<code>\n</code>	Dopasowanie <i>n</i> -tej grupy.	<code>(['"]).*\1</code> — pasuje do "Fred" lub 'Fred', lecz nie do "Fred"
<code>(?<nazwa>X)</code>	Przechwytuje dopasowanie <i>X</i> , nadając mu podaną nazwę.	<code>'(?<id>[A-Za-z0-9]+)'</code> przechwytuje dopasowanie, nadając mu nazwę <code>id</code>
<code>\k<nazwa></code>	Grupa o podanej nazwie.	<code>\k<id></code> dopasowuje grupę o nazwie <code>id</code>
<code>(?:X)</code>	Zastosowane nawiasów bez przechwytywania <i>X</i> .	W wyrażeniu regularnym <code>(?:http ftp):/(.*)</code> dopasowanie po <code>://</code> jest grupą <code>\1</code>
<code>(?f_1...:X), (?f_1...~f_k...:X), gdzie <i>f_i</i> należy do <code>[dimsuUx]</code></code>	Dopasowuje <i>X</i> , lecz go nie przechwytuje, używając lub nie używając (jeśli zastosowano -) określonych flag.	<code>(?i:.jpe?g)</code> — w dopasowaniu nie będzie uwzględniana wielkość liter
Inne <code>(? ...)</code>	Patrz dokumentacja API Pattern.	
Kwantyfikatory		
<code>X?</code>	Opcjonalnie <i>X</i> .	<code>\x?</code> to opcjonalny znak +
<code>X*, X+</code>	<i>X</i> , 0 lub więcej razy oraz <i>X</i> co najmniej jeden raz.	<code>[1-9][0-9]+</code> to liczba całkowita większa od 10.
<code>X{n} X{n,} X{n,m}</code>	<i>X</i> <i>n</i> razy, co najmniej <i>n</i> razy, pomiędzy <i>n</i> i <i>m</i> razy.	<code>[0-7]{1,3}</code> od jednej do trzech cyfr ósemkowych.

Tabela 2.6. Składnia wyrażeń regularnych — ciąg dalszy

Składnia	Objaśnienie	Przykład
Kwantyfikatory		
$Q?$, gdzie Q jest wyrażeniem z kwantyfikatorem	Kwantyfikator oporny; próbuje znaleźć jak najkrótsze dopasowanie, zanim spróbuje dobrac dłuższe.	<code>.*(<.+?>).*</code> — przechwytuje najkrótszą sekwencję, umieszczoną pomiędzy nawiasami kątowymi
$Q+$, gdzie Q jest wyrażeniem z kwantyfikatorem	Kwantyfikator zachłanny, znajdujący najdłuższe dopasowanie, które nie wymaga cofania.	<code>'[^']*+'</code> — 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).	<code>^Java\$</code> — pasuje do wpisanego słowa Java lub wiersza o tej zawartości
$\backslash A$, $\backslash Z$, $\backslash z$	Początek wejścia, koniec wejścia, bezwzględny koniec wejścia (niezmienane w trybie wielowierszowym)	
$\backslash b$, $\backslash B$	Granica słowa, granica inna niż słowa.	<code>\bJava\b</code> — pasuje do słowa Java
$\backslash R$	Znak nowego wiersza Unicode	
$\backslash G$	Koniec poprzedniego dopasowania.	

Tabela 2.7. Wstępnie zdefiniowane nazwy klas znaków

Nazwa klasy znaków	Objaśnienie
<i>klasaPosix</i>	<i>klasaPosix</i> 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.
<i>IsSkrypt</i> , <i>sc=Skrypt</i> , <i>script=Skrypt</i>	<i>Skrypt</i> jest nazwą skryptu Unicode, akceptowaną przez metodę <code>Character.UnicodeScript.forName</code> .
<i>InBlok</i> , <i>blk=Blok</i> , <i>block=Blok</i>	<i>Blok</i> jest nazwą bloku znaków Unicode, akceptowaną przez metodę <code>Character.UnicodeBlock.forName</code> .
<i>Kategoria</i> , <i>InKategoria</i> , <i>gc=Kategoria</i> , <i>general_category=Kategoria</i>	Jedno lub dwuliterowa nazwa ogólnej kategorii znaków Unicode.
<i>IsWłaściwość</i>	<i>Właściwość</i> jest jedną z wartości Alphabetic, Ideographic, Letter, Lowercase, Uppercase, Titlecase, Punctuation, Control, White_Space, Digit, Hex_Digit, Join_Control, Noncharacter_Code_Point, Assigned.
<i>javaMetoda</i>	Wywołuje metodę <code>Character.isMetoda</code> (nie może być przestarzała).

Na przykład łańcuch `cab` może zostać dopasowany do wyrażenia `[a-z]*ab`, ale nie do `[a-z]*+ab`. W pierwszym przypadku wyrażenie `[a-z]*` dopasuje jedynie znak `c`, wobec czego znaki `ab` zostaną dopasowane do reszty wzorca. Jednak wyrażenie `[a-z]*+` dopasuje znaki `cab`, wobec czego reszta wzorca pozostanie bez dopasowania.

- Grupy pozwalają definiować podwyrażenia. Grupy ujmujemy w znaki nawiasów `( )`; na przykład `([+-]?)([0-9]+)`. Możemy następnie zażądać dopasowania do wszystkich grup lub do wybranej grupy, do której odwołujemy się przez `\n`, gdzie `n` jest numerem grupy (numeracja rozpoczyna się od `\1`).

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

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

Niestety, składnia wyrażeń regularnych nie jest całkowicie ustandaryzowana. Istnieje zgodność w zakresie podstawowych konstrukcji, ale diabeł tkwi w szczegółach. Klasy języka Java związane z przetwarzaniem wyrażeń regularnych używają składni podobnej do zastosowanej w języku Perl. Wszystkie konstrukcje tej składni zostały przedstawione w tabeli 1.8. Więcej informacji na temat składni wyrażeń regularnych znajdziesz w dokumentacji klasy `Pattern` lub książce *Wyrażenia regularne. Wprowadzenie* autorstwa Michaela Fitzgeralda (Wydawnictwo Helion, 2013).

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

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

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

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

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

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` — odstęp i komentarze (od znaku `#` do końca wiersza) będą ignorowane.
- `Pattern.LITERAL` — wzorec jest traktowany dosłownie i musi zostać dopasowany w dokładnie takiej samej postaci, z ewentualnymi różnicami w wielkości liter.
- `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.

Aby dopasować elementy kolekcji lub strumienia, wzorec należy przekształcić na funkcję predykatu:

```
Stream<String> strings = ...;
Stream<String> result = strings.filter(pattern.asPredicate());
```

Zmienna `result` będzie zawierać wszystkie łańcuchy znaków pasujące do użytego wyrażenia regularnego.

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

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

zwracają indeks początkowy i końcowy podanej grupy.

Dopasowany łańcuch możemy pobrać, wywołując

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

Grupa 0 oznacza całe wejście; indeks pierwszej grupy równy jest 1. Metoda `groupCount` zwraca całkowitą liczbę grup. Do obsługi grup nazwanych służą następujące metody:

```
int start(String groupName)
int end(String groupName)
String group(String groupName)
```

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

```
(([1-9]|1[0-2]):([0-5][0-9]))[ap]m
```

dla danych

```
11:59am
```

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

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

Program przedstawiony na listingu 2.6 umożliwia wprowadzenie wzorca, a następnie łańcucha, którego dopasowanie zostanie sprawdzone. Jeśli łańcuch pasuje do wzorca zawierającego grupy, to program wyświetla granice grup w postaci nawiasów, na przykład:

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

Listing 2.6. *regex/RegexTest.java*

```
1 package regex;
2
3 import java.util.*;
4 import java.util.regex.*;
5
6 /**
7  * Program testujący zgodność z wyrażeniem regularnym. Wprowadź wzorec
8  * i dopasowywany łańcuch.
9  * Jeśli wzorec zawiera grupy, to po dopasowaniu program wyświetli ich granice.
10  * @version 1.02 2012-06-02
11  * @author Cay Horstmann
12  */
13 public class RegexTest
14 {
15     public static void main(String[] args) throws PatternSyntaxException
16     {
17         Scanner in = new Scanner(System.in);
18         System.out.println("Wpisz wyrażenie regularne: ");
19         String patternString = in.nextLine();
20
21         Pattern pattern = Pattern.compile(patternString);
22
23         while (true)
24         {
25             System.out.println("Wpisz łańcuch znaków: ");
26             String input = in.nextLine();
27             if (input == null || input.equals("")) return;
28             Matcher matcher = pattern.matcher(input);
29             if (matcher.matches())
30             {
31                 System.out.println("Dopasowano");
32                 int g = matcher.groupCount();
33                 if (g > 0)
34                 {
35                     for (int i = 0; i < input.length(); i++)
36                     {
37                         // Wyświetla puste grupy
38                         for (int j = 1; j <= g; j++)
39                             if (i == matcher.start(j) && i == matcher.end(j))
40                                 System.out.print("(");
41                         // Wyświetla (dla niepustych grup, które tu się zaczynają
```

```

42         for (int j = 1; j <= g; j++)
43             if (i == matcher.start(j) && i != matcher.end(j))
44                 System.out.print(' ');
45         System.out.print(input.charAt(i));
46         // Wyświetla ) dla grup kończących się tutaj
47         for (int j = 1; j <= g; j++)
48             if (i + 1 != matcher.start(j) && i + 1 == matcher.end(j))
49                 System.out.print(' ');
50     }
51     System.out.println();
52 }
53 }
54 else
55     System.out.println("Brak dopasowania");
56 }
57 }
58 }

```

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

```

while (matcher.find())
{
    int start = matcher.start();
    int end = matcher.end();
    String match = input.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*

```

1 package match;
2
3 import java.io.*;
4 import java.net.*;
5 import java.nio.charset.*;
6 import java.util.regex.*;
7
8 /**
9  * Program wyświetlający wszystkie adresy URL na stronie WWW poprzez dopasowanie
10  * wyrażenia regularnego opisującego znacznik <a href=...> języka HTML.
11  * Uruchamianie: java match.HrefMatch adresURL
12  * @version 1.02 2016-07-14
13  * @author Cay Horstmann
14  */
15 public class HrefMatch
16 {

```

```
17     public static void main(String[] args)
18     {
19         try
20         {
21             // pobiera URL z wiersza poleceń lub używa domyślnego
22             String urlString;
23             if (args.length > 0) urlString = args[0];
24             else urlString = "http://java.sun.com";
25
26             // otwiera InputStreamReader dla podanego URL
27             InputStreamReader in =
28                 new InputStreamReader(new URL(urlString).openStream(),
29                                     StandardCharsets.UTF_8);
30
31             // wczytuje zawartość do obiektu klasy StringBuilder
32             StringBuilder input = new StringBuilder();
33             int ch;
34             while ((ch = in.read()) != -1)
35                 input.append((char) ch);
36
37             // poszukuje wszystkich wystąpień wzorca
38             String patternString = "<a\\s+href\\s*=\\s*(\\\"[^\\\"]*\\\"|\\\"[^\\s>]*\\\")\\s*>";
39             Pattern pattern = Pattern.compile(patternString,
40                                             Pattern.CASE_INSENSITIVE);
41             Matcher matcher = pattern.matcher(input);
42
43             while (matcher.find())
44             {
45                 String match = matcher.group();
46                 System.out.println(match);
47             }
48         }
49         catch (IOException | PatternSyntaxException e)
50         {
51             e.printStackTrace();
52         }
53     }
54 }
```

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

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

Łańcuch zastępujący może zawierać referencje grup wzorca: `$n` zostaje zastąpione przez *n*-tą grupę, a `${nazwa}` przez grupę o podanej nazwie. Sekwencja `\\s` pozwala umieścić znak `$` w zastępującym tekście.

Jeśli mamy łańcuch zawierający znaki `$` i `\` i nie chcemy, aby były one interpretowane jako referencje grup wzorca, wywołujemy `matcher.replaceAll(Matcher.quoteReplacement(str))`.

Metoda `replaceFirst` zastępuje jedynie pierwsze wystąpienie wzorca.

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

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

Jeśli tokenów jest wiele, to można je pobierać w sposób leniwy:

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

Jeżeli jednak nie zamierzamy zwracać sobie głowy ani wstępną kompilacją wyrażenia regularnego, ani leniwym pobieraniem, to wystarczy skorzystać z metody `String.split`:

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

API `java.util.regex.Pattern` 1.4

- `static Pattern compile(String expression)`

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

kompiluje łańcuch wyrażenia regularnego, tworząc obiekt wzorca przyspieszający przetwarzanie.

Parametry:

<code>expression</code>	wyrażenie regularne
<code>flags</code>	jeden lub więcej znaczników <code>CASE_INSENSITIVE</code> , <code>UNICODE_CASE</code> , <code>MULTILINE</code> , <code>UNIX_LINES</code> , <code>DOTALL</code> i <code>CANON_EQ</code> .

- `Matcher matcher(CharSequence input)`

tworzy obiekt pozwalający odnajdywać dopasowania do wzorca w łańcuchu wejściowym.

- `String[] split(CharSequence input)`

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

- `Stream<String> splitAsStream(CharSequence input)` 8

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

Parametry:

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

API java.util.regex.Matcher **1.4**

- `boolean matches()`
zwraca `true`, jeśli łańcuch wejściowy pasuje do wzorca.
- `boolean lookingAt()`
zwraca `true`, jeśli początek łańcucha wejściowego pasuje do wzorca.
- `boolean find()`
- `boolean find(int start)`
próbuję odnaleźć następne dopasowanie i zwraca `true`, jeśli próba się powiedzie.
Parametry: `start` indeks, od którego należy rozpocząć poszukiwanie
- `int start()`
- `int end()`
zwraca pozycję początkową dopasowania lub następną pozycję za dopasowaniem.
- `String group()`
zwraca bieżące dopasowanie.
- `int groupCount()`
zwraca liczbę grup we wzorcu wejściowym.
- `int start(int groupIndex)`
- `int end(int groupIndex)`
zwraca pozycję początkową grupy lub następną pozycję za grupą dla danej grupy bieżącego dopasowania.
Parametry: `groupIndex` indeks grupy (wartości indeksu rozpoczynają się od 1) lub 0 dla oznaczenia całego dopasowania
- `String group(int groupIndex)`
zwraca łańcuch dopasowany do podanej grupy.
Parametry: `groupIndex` indeks grupy (wartości indeksu rozpoczynają się od 1) lub 0 dla oznaczenia całego dopasowania
- `String replaceAll(String replacement)`
- `String replaceFirst(String replacement)`
zwracają łańcuch powstały przez zastąpienie podanym łańcuchem wszystkich dopasowań lub tylko pierwszego dopasowania.
Parametry: `replacement` łańcuch zastępujący może zawierać referencje do grup wzorca postaci `$n`. Aby umieścić w łańcuchu symbol `$`, stosujemy sekwencję `\$`.
- `static String quoteReplacement(String str)` **5.0**
cytuje wszystkie znaki `\` i `$` w łańcuchu `str`.

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

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

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

Skorowidz

A

- adnotacja, 436
 - @ActionListenerFor, 438, 441,
 - @BugReport, 453
 - @Deprecated, 450, 451
 - @Documented, 450, 453
 - @Inherited, 450, 453
 - @interface, 438
 - @LogEntry, 459
 - @NotNull, 448
 - @Override, 450, 451
 - @Persistent, 454
 - @PostConstruct, 451
 - @PreDestroy, 451
 - @Resource, 451
 - @Retention, 438, 450, 452
 - @Serializable, 453
 - @SuppressWarnings, 450, 451
 - @Target, 438, 450, 452
 - @TestCase, 437
- cykliczne zależności, 446
- deklaracja elementu, 443
- elementy, 437
- format, 445
- interfejs, 438, 443
- kod bajtowy, 459
- kolejność elementów, 445
- metaadnotacje, 452
- metody, 438
- modyfikacja kodu bajtowego
 - podczas ładowania, 464
- pojedyncze wartości, 445
- przetwarzanie, 442, 455
- regularne, 451
- składnia, 443
- skróty, 445
- standardowe, 450
- typy elementów, 444, 452
- znacznikowe, 445
- adnotacje
 - deklaracji, 446
 - obsługi zdarzeń, 438
 - zarządzania zasobami, 451
 - zastosowań typów, 447
- adres
 - internetowy, 237
 - localhost, 241
 - URI, 252
 - URL, 195, 252, 284, 490, 661
- AES, 535, 536, 542
- agent, 465
- aktualizacje wsadowe, 333
- aktualizowalne zbiory wyników
 - zapytań, 311, 313
- alfa, 745, 746
- algorytm
 - AES, 535, 536, 542
 - CRC, 128
 - DES, 535
 - DSA, 520
 - kryptograficzny, 468
 - MD5, 518
 - RSA, 521, 543
 - SHA, 96
 - SHA1, 517
 - szyfrowania, 535
 - z kluczem symetrycznym, 542
- analiza wyjątków SQL, 294
- animacje GIF, 765
- antialiasing, 753
- Apache, 337
- API dat i czasu, 339
- aplety, 516
 - lokalizacja, 397
- aplikacje
 - interaktywne, 246
 - klient-serwer, 278
 - pulpitu, 849
- appletviewer, 483, 488, 531
- architektura
 - JDBC, 276
 - trójwarstwowa, 278
- archiwum ZIP, 88, 123
- ARGB, 770, 774
- ARRAY, 335, 336
- ASCII, 79, 359
- ASP, 262
- atrybuty, 170
- atrybuty drukowania, 807, 810
 - dokumenty, 807
 - hierarchia, 808
 - klasy, 810
 - usługi drukowania, 807
 - wydruk, 807
 - zbiór, 808, 809
 - żądanie wydruku, 807
- ATTLIST, 169, 176
- AWT, 709
 - aplikacje pulpitu, 849
 - drukowanie, 783
 - figury, 712
 - filtrowanie obrazów, 775
 - Graphics, 709, 712
 - obszar przycięcia, 711
 - operacje na obrazach, 768
 - pliki graficzne, 758
 - pola, 727
 - potokowe tworzenie grafiki, 710
 - przeciągnij i upuść, 828
 - przekształcenia układu
 - współrzędnych, 711, 737
 - przezroczystość, 745
 - prycinanie, 743
 - rysowanie figur, 710
 - schowek, 813
 - składanie obrazów, 745
 - ślad pędzla, 711, 728
 - wskazówki operacji
 - graficznych, 753
 - wypełnianie obszaru, 711, 735
 - zasady składania obrazów, 711

B

baza danych, 275
 adres URL, 284
 aktualizowalne zbiory wyników
 zapytań, 313
 JNDI, 336
 kolumny, 279
 kursory, 312
 łączenie tabel, 281
 metadane, 322
 model dostępu, 276
 modyfikacja danych, 282
 ODBC, 276
 przewijanie zbioru rekordów, 312
 rekordy, 279, 314
 spójność, 331
 SQL, 275
 tabele, 279
 transakcje, 331
 uruchamianie, 285
 wstawianie danych, 283
 wypełnianie, 296
 zapytania, 280
 zarządzanie połączeniami, 336
 zbiory rekordów, 318
 BCEL, 459
 BEA WebLogic, 337
 bezpieczeństwo, 467
 hierarchia klas pozwoleń, 485
 JAAS, 502
 Java 2, 484
 Kerberos, 545
 klasy pozwoleń, 495
 kryptografia klucza
 publicznego, 542
 ładowanie klas, 468
 menedżer bezpieczeństwa,
 468, 483
 piaskownica, 484
 pliki polityki, 484
 podpis cyfrowy, 516
 podpisywanie kodu, 528
 polityka bezpieczeństwa, 485
 pozwolenia, 483, 485
 przydzielanie praw, 484
 skrótów wiadomości, 517
 SSL, 545
 szyfrowanie, 534
 uwierzelnianie użytkowników,
 502
 uwierzelnianie wiadomości,
 524

weryfikacja kodu maszyny
 wirtualnej, 478

źródło kodu, 484
 biblioteka
 AWT, 709
 BCEL, 459
 DLL, 863
 JCE, 541
 strumieni, 17
 BLOB, 283, 306, 335
 blok try/catch, 294
 blokowanie plików, 133
 błąd UnsatisfiedLinkError, 861
 BOM, 391
 BOOLEAN, 283, 335
 bufor danych, 131
 buforowane zbiory rekordów, 319

C

catch, 294
 CDATA, 154, 170
 cel upuszczanych danych, 836
 certyfikaty, 484
 podpisywanie, 526
 żądanie, 527
 certyfikaty X.509
 keytool, 522
 komponenty, 522
 składnica kluczy, 522
 sprawdzanie wiarygodności, 523
 wydawcy certyfikatów, 523
 zarządzanie, 522
 CGI, 262
 CHAR, 283, 335
 CHARACTER, 283
 CLASSPATH, 469
 CLEAR, 746
 CLOB, 283, 306, 335
 CONCUR_READ_ONLY, 312
 CONCUR_UPDATABLE, 312,
 314
 CONTIGUOUS_TREE_
 SELECTION, 619
 COREJAVA, 284
 CREATE TABLE, 283, 290, 296
 czas, 340, 372
 lokalny, 347
 strefowy, 348
 czcionki, 727, 743
 obrysy, 743

D

DA, 810
 dane binarne, 81
 data, 360, 372
 database.properties, 336
 daty lokalne, 343
 DDL, 291
 DEC, 335
 DECIMAL, 283, 335
 definicja typu dokumentu, 167
 deklaracja typu dokumentu, 152
 dekoratory, 678
 DELETE, 290
 DES, 535, 536
 deskryptor pola składowego, 97
 deskryptory wdrożeń, 437
 DISCONTIGUOUS_TREE_
 SELECTION, 619
 DLL, 863
 DOCTYPE, 167, 209
 dokumenty XML, 152
 analiza zawartości, 156
 atrybuty, 153
 CDATA, 154
 deklaracja typu dokumentu, 152
 DOCTYPE, 167
 DTD, 154
 element korzenia, 152
 elementy, 156
 elementy podrzędne, 157, 163
 komentarze, 154
 kontrola poprawności, 166
 nagłówki, 152
 NodeList, 156
 parsowanie, 155
 PCDATA, 168
 pobieranie węzłów, 158
 przeglądanie atrybutów, 158
 tworzenie, 207
 tworzenie drzewa DOM, 207
 wczytywanie, 155
 wyszukiwanie informacji, 189
 XML Schema, 174
 XPath, 189
 DOM, 155, 168
 analiza drzewa, 159
 tworzenie drzewa, 207
 domena ochronna, 486
 domyślne obiekty rysujące, 572
 domyślny edytor komórki, 609
 dostęp
 do kolumn tabeli, 572
 do pliku, 125

do pól instancji, 874, 878
 do pól statycznych, 877
 do rejestru systemu, 900, 902
 do składowych, 873
 DOUBLE, 283, 335
 DROP TABLE, 290
 drukowanie, 567, 783, 813
 algorytm rozmieszczenia
 materiału, 793
 atrybuty, 807
 format strony, 786
 grafika, 783
 JDK, 783
 liczba stron, 785
 marginesy, 786
 podgląd wydruku, 794
 PostScript, 806
 Printable, 784
 przycięcie kontekstu
 graficznego, 786
 rozmieszczenie materiału
 na stronach, 793
 transparent, 793
 usługa, 802
 wiele stron, 792
 wymiary strony, 786
 zadania, 784
 drukowanie grafiki, 784
 drzewa, 598
 domyślny edytor komórki, 609
 dziedziczenie, 615
 edycja węzłów, 609
 ikony liści, 604
 JTree, 598, 599
 kolejność przeglądania węzłów,
 614
 korzeń, 599, 601
 las, 599, 603
 liście, 599
 model, 599
 modyfikacja ścieżek, 606
 modyfikacje, 606
 nasłuchiwanie zdarzeń, 618
 obiekt nasłuchujący wyboru, 618
 obiekt rysujący komórki, 615
 obiekt użytkownika, 600
 powiązania między węzłami, 600
 przeglądanie od końca, 614
 przeglądanie w głąb, 614
 przeglądanie węzłów, 613
 przewijalny panel, 609
 rozwijanie ścieżek, 609
 rysowanie węzłów, 615
 struktura, 630

ścieżki, 607
 tworzenie modeli, 625
 węzły, 599
 węzły nadrzędne, 599
 węzły podrzędne, 599
 wstawianie węzłów, 609
 wygląd, 602, 615
 zdarzenia, 618
 zmiana struktury węzła, 608
 zwinięcie, 603
 drzewo DOM, 157
 DST, 746
 DST_ATOP, 747
 DST_IN, 747
 DST_OUT, 747
 DST_OVER, 746
 DTD, 154, 166, 171
 dynamiczne tworzenie kodu, 432
 dziedziczenie, 615

E

edycja
 komórek, 586, 588
 plików polityki, 494
 rejestru, 900
 ekran powitalny, 844
 ELEMENT, 166–169, 178, 212
 elipsa, 725
 ENTITY, 171
 env, 869, 895

F

figury, 710, 712, 714, 725
 łuk, 714
 odcinki, 718
 prostokąt, 714
 przycinanie, 743
 punkty kontrolne, 725
 rysowanie, 710
 tworzenie, 725
 wielokąty, 718
 wypełnianie, 712, 735
 filtr RowFilter, 576
 filtrowanie
 dokumentów, 639
 obrazów, 775
 interpolacja, 775
 negatyw, 779
 rozmycie, 780
 wykrywanie krawędzi, 781
 plików, 120
 wierszy, 576
 filtry, 639

 strumieni wejścia-wyjścia, 68
 FIXED, 170
 FLOAT, 283, 335
 format
 liczby, 365
 pliku serializacji obiektów, 95
 XML, 150
 ZIP, 70, 88
 formatowanie
 daty i czasu, 352
 komunikatów, 385
 warianty, 387
 liczb, 365
 formularze HTML, 262, 265
 metoda GET, 263
 przetwarzanie danych, 262
 serwlety, 262
 skrypty CGI, 262
 Submit, 262
 wysyłanie informacji
 do serwera, 263
 fraktale, 771
 FROM, 281
 FTP, 257
 funkcja
 CallNonvirtualXxxMethod, 885
 CallStaticXxxMethod, 883
 ExceptionOccured, 891
 fprintf, 880
 GetArrayLength, 887
 GetIntField, 904
 GetMethodID, 884
 GetObjectArrayElement, 887
 GetStaticMethodID, 883
 GetXxxArrayRegion, 889
 Java_HelloNative_greeting, 862
 NewObject, 890
 NewXxxArray, 889
 SetIntField, 904
 SetObjectArrayElement, 887
 SetXxxArrayRegion, 889
 sprintf, 880
 Throw, 890, 891
 ThrowNew, 891
 funkcje
 języka C, 860
 zwracające wartości
 opcjonalne, 32

G

generowanie
 klucza, 536
 kodu źródłowego, 456
 liczb losowych, 537

geometria pól, 727
 GET, 263
 GIF, 758, 802
 gniazda, 234
 adresy internetowe, 237
 kanały, 246
 limity czasu, 235
 nawiązanie połączenia, 246
 otwieranie, 234
 połączenia częściowo
 zamknięte, 244
 serwera, 239
 zamykanie, 241
 graficzny interfejs użytkownika,
 420, 813
 grafika, 709
 antialiasing, 753
 drukowanie, 783, 784
 filtrowanie obrazów, 775
 Java 2D, 710
 klasy obiektów graficznych, 714
 operacje na obrazach, 768
 potok rysowania, 711
 potokowe tworzenie, 710
 prostokąt ograniczający, 713
 przekształcenia układu
 współrzędnych, 737
 przezroczystość, 745
 prycinanie, 743
 składanie obrazów, 745
 ślad pędzla, 728
 współrzędne, 713
 wypełnianie obszaru, 735
 zbiór Mandelbrota, 771
 gromadzenie wyników, 34
 w mapach, 39
 grupowanie, 43
 GSS, 545
 gzip, 261

H

hasła, 509, 512
 hasła dostępu, 256
 hierarchia
 dziedziczenia typów, 888
 klas ładowania, 469
 klas pozwoleń, 485
 klas Reader i Writer, 66
 klas Shape, 713
 komponentów tekstowych, 633
 strumieni, 65
 hiperłącza, 662
 host, 232

HTML, 151, 233, 661
 HTTP, 278
 nagłówki żądań, 256
 odpowiedzi, 257

I

ICC, 770
 ID, 171
 identyfikator URI, 809
 IDREF, 171
 image/gif, 261
 IMAP, 545
 implementacja serwerów, 238
 IMPLIED, 170
 import, 472
 informacje o plikach, 117
 INSERT, 283, 290
 INSERT INTO, 296
 instalacja JDBC, 284
 INT, 283, 335
 INTEGER, 283, 335
 interfejs
 ActionListener, 442
 AnnotatedElement, 441, 442
 Annotation, 444
 Appendable, 68
 Attribute, 807, 809, 812
 Attributes, 203
 AttributeSet, 813
 BaseStream, 36
 BasicFileAttributes, 118
 BufferedImageOp, 768, 775, 782
 CachedRowSet, 318–321
 CachedRowSet, 319
 Callback, 513
 CallbackHandler, 515
 CellEditor, 597
 CharacterData, 165
 CharSequence, 50, 54, 68
 ClipboardOwner, 815, 818
 Clob, 307, 335
 Closeable, 67
 Collector, 35
 Comparator, 379
 Compilable, 420
 CompilationTask, 430
 Connection, 293, 306, 308, 316,
 330
 ContentHandler, 198, 200, 202
 DatabaseMetaData, 293, 313,
 317, 322, 335
 DatabaseMetaData, 323
 DataInput, 82

DataOutput, 83
 DataSource, 337
 Diagnostic, 427
 DiagnosticListener, 427
 Doc, 804
 DocAttribute, 807, 810
 DocAttributeSet, 809
 DocPrintJob, 804, 805
 Document, 156, 159, 164, 211,
 636
 DocumentEvent, 637
 DocumentListener, 635
 DoubleStream, 50, 53
 EntityResolver, 155, 173, 200
 Enumeration, 290
 ErrorHandler, 172, 173, 200
 Externalizable, 97, 103
 FileVisitor, 121
 FilteredRowSet, 318
 FlavorListener, 819
 Flushable, 67
 HyperlinkListener, 663, 666
 Icon, 559
 ImageInputStream, 763
 ImageOutputStream, 766
 Instrumentation, 465
 Instrumentation, 465
 InternalFrameListener, 699
 IntStream, 50
 IntStream, 52
 Invocable, 419
 Iterator, 290
 JavaCompiler, 429
 JavaFileManager, 427
 JavaFileObject, 429
 JdbcRowSet, 318
 JNDI, 336
 JNI, 866
 JoinRowSet, 318
 ListCellRenderer, 560, 563
 ListModel, 558
 ListSelectionListener, 553
 ListSelectionModel, 584
 LoginModule, 516
 LongStream, 50
 LongStream, 53
 MDI, 687
 MutableTreeNode, 600, 606
 NamedNodeMap, 158, 165
 Node, 156, 164, 197
 NodeList, 156
 NodeList, 156, 165
 Pageable, 792
 Paint, 735

Path, 111
 Permission, 495, 501
 Policy, 485
 PreparedStatement, 301, 306
 Principal, 507
 Printable, 784
 Printable, 784, 785, 791
 PrintJobAttribute, 807, 810
 PrintRequestAttribute, 807, 810
 PrintRequestAttributeSet, 784
 PrintService, 804–806, 813
 PrintServiceAttribute, 807, 810
 PrivilegedAction, 503, 507
 PrivilegedExceptionAction, 504, 507
 PrivilegedAction, 503
 Processor, 455
 Readable, 67
 ReadableByteChannel, 246
 Result, 224
 ResultSet, 290–293, 307, 311, 318, 331
 ResultSetMetaData, 323
 ResultSetMetaData, 323, 331
 RowSet, 318
 RowSet, 318, 320
 Savepoint, 334
 ScriptEngine, 416
 ScriptEngineFactory, 415
 Serializable, 92, 97, 102, 107, 453
 Shape, 724, 727
 Source, 224
 StandardJavaFileManager, 429
 Statement, 293, 310, 311, 334
 Stream, 20, 23, 27, 50
 Stroke, 728
 SupportedValuesAttribute, 807
 TableCellEditor, 593, 595, 597
 TableCellRenderer, 586, 587, 596
 TableColumnModel, 572, 584
 TableModel, 571, 582
 Text, 158
 Throwable, 890
 Tool, 430
 Transferable, 814, 818
 TreeCellRenderer, 615–617
 TreeModel, 163, 599, 625, 632
 TreeModelListener, 632
 TreeNode, 600, 605, 607, 613
 TreeSelectionListener, 618, 624
 TreeSelectionModel, 619
 VetoableChangeListener, 698, 703
 WebRowSet, 318

WindowListener, 699
 WritableByteChannel, 246
 interfejsy
 adnotacji, 438, 443
 kompilatora, 425
 programowe wywołań języka Java, 895, 899
 użytkownika
 internacjonalizacja
 czas, 372
 data, 360, 372
 komplety zasobów, 392
 komunikaty, 385
 liczby, 365
 Locale, 361
 lokalizatory, 360
 łańcuchy znaków, 394
 porządek alfabetyczny, 378
 waluta, 365, 370
 zbiory znaków, 389
 interpolacja, 775
 inżynieria kodu bajtowego, 459
 BCEL, 459
 modyfikacja kodu podczas ładowania, 464
 IPv6, 237
 iteracje, 18

J

JAAS, 502, 507
 moduł logowania, 508
 moduły, 507
 uwierzytelnianie oparte na rolach, 508
 jaas.config, 506
 JAR, 252, 469
 jarray, 887, 903
 jarsigner, 524, 531
 Java, 11, 545
 Java 2, 484, 828
 Java 2D, 709, 710
 figury, 711, 712
 geometria pól, 727
 klasy obiektów graficznych, 714
 krzywe, 713, 716
 krzywe Beziera, 717
 linie, 718
 łuk, 714
 operacje na polach, 728
 pola, 727
 prostokąt, 714
 przezroczystość, 745
 punkty kontrolne, 725
 składanie obrazów, 745
 ślad pędzla, 728
 wartość alfa, 745
 wskazówki operacji graficznych, 753
 współrzędne, 713
 wypełnianie obszaru, 735
 Java Authentication and Authorization Service, 502
 java.awt.datatransfer, 814
 java.net.Socket, 235
 java.nio, 244, 246
 java.security, 468, 517
 java.text, 365
 javah, 862, 874
 javap, 879
 Javascript, 662
 JavaServer Faces, 262
 javax.imageio, 758
 javax.security.auth.login.
 LoginContext, 506
 javax.security.auth.Subject, 507
 javax.sql.rowset, 318
 JAXP, 155
 JCE, 541
 jclass, 874, 884
 JDBC, 275
 adres URL baz danych, 284
 aktualizacja
 danych, 275
 aktualizacja wsadowa, 333
 aktualizowalne zbiory wyników
 zapytań, 311, 313
 architektura, 276
 instalacja, 284
 JNDI, 336
 klient-serwer, 278
 menedżer sterowników, 287
 metadane, 322
 nawiązywanie połączenia, 285
 polecenia, 293
 polecenia przygotowane, 300
 połączenia krótkotrwale, 293
 przewijalne zbiory wyników
 zapytań, 311
 przewijanie zbioru rekordów, 312
 pula połączeń, 337
 rekordy wstawiania, 314
 serwer, 278
 SQL, 275, 278
 sterowniki, 276
 transakcje, 331
 warstwa pośrednia, 278
 wersje, 275

JDBC

- wykonywanie zapytań, 300
- wypełnianie bazy danych, 296
- zamykanie zbioru wyników, 293
- zapytania, 275
- zarządzanie połączeniami, 293
- zastosowania, 278
- zbiory rekordów, 311, 318
- zbiory wyników, 293
- źródło danych, 284

- JDBC 3, 336

- JDBC API, 276

- jdbc.password, 297

- jdbc.property, 286

- jdbc.url, 297

- jdbc.username, 297

- język

- C, 860

- łańcuchy znakowe platformy
 - Java, 870

- obsługa błędów, 890, 894

- tablice, 886

- wywoływanie metod Java,
 - 880, 885

- wywoływanie metod
 - statycznych, 883

- C++, 863

- DDL, 291

- HTML, 151, 233

- Java, 11

- Javascript, 662

- SGML, 151

- SQL, 275, 278, 290

- XML, 149, 151

- XML Schema, 174

- XPath, 189

- Scheme, 425

- jfieldID, 874

- JNDI, 336

- nawiązywanie połączenia, 337

- pula połączeń, 337

- zarządzanie nazwami

- użytkowników, 337

- źródła danych, 337

- JNI, 866, 868

- konwencja wywołań, 868

- wywołania funkcji, 869

- JNICALL, 872

- JNIEXPORT, 872

- javax, 874, 903

- JPEG, 758, 759

- jstring, 868, 884, 903

- jvalue, 886

- jvm, 896

K

- kalkulator emerytalny, 397

- kanal dostępu do pliku, 125

- kanały, 246

- SocketChannel, 246

- KEY_ALPHA_INTERPOLATION,
- 754

- KEY_ANTIALIASING, 754

- KEY_COLOR_RENDERING, 754

- KEY_DITHERING, 754

- KEY_FRACTIONAL_METRICS,
- 754

- KEY_INTERPOLATION, 754

- KEY_RENDERING, 754

- KEY_STROKE_CONTROL, 754

- KEY_TEXT_ANTIALIASING,
- 754

- keytool, 522, 527, 531

- klasa, 472

- AbstractCellEditor, 593–595

- AbstractListModel, 554

- AbstractSpinnerModel, 661

- AbstractTableModel, 568, 588

- ActionListenerInstaller, 441

- AffineTransform, 740

- AffineTransformOp, 775, 782

- AllPermission, 492, 495

- AllPermissions, 486

- AlphaComposite, 748

- Annotation, 444

- Arc2D, 713, 716

- Arc2D.Double, 726

- ArcMaker, 725

- Area, 727, 728

- ArrayIndexOutOfBoundsException

- Exception, 891

- ArrayList, 631

- ArrayStoreException, 891

- Attribute, 807, 809, 812

- Attributes, 203

- AttributesImpl, 228

- AudioPermission, 492

- AuthPermission, 492

- Banner, 793

- BasicPermission, 492

- BasicPermission, 492

- BasicStroke, 728, 729

- BasicStroke, 728, 729, 734, 735

- BigDecimal, 335

- Book, 801

- Book, 801, 802

- Buffer, 129, 133, 246

- Buffer, 131

- BufferedImage, 736, 748, 768,
 - 773

- BufferedImage, 736, 768, 769

- BufferedInputStream, 71

- BufferedOutputStream, 71

- ByteBuffer, 125, 130

- ByteLookupTable, 779, 782, 783

- Channels, 251

- CharBuffer, 66, 131

- Charset, 80

- ChoiceFormat, 388

- Chromaticity, 810

- Cipher, 534

- Cipher, 534, 539

- CipherInputStream, 541

- Class, 441, 477, 487, 614, 809

- ClassLoader, 469, 473, 477

- ClassLoader, 473

- ClassNameTreeCellRenderer,
 - 617

- ClassTreeFrame, 617

- Clipboard, 814

- Clipboard, 814, 817, 819, 828

- CodeSource, 487

- CollationKey, 384

- Collator, 379, 384

- Collectors, 38, 42, 43, 47

- Color, 735, 745, 771, 824

- ColorConvertOp, 780

- ColorModel, 772, 774

- ColorSupported, 810

- CompilationTask, 428

- Compression, 810

- Constructor, 441

- ContentHandler, 198, 200, 202

- ConvolveOp, 781, 783

- Copies, 807, 809, 810

- CopiesSupported, 807

- CubicCurve2D, 713, 716, 717,
 - 726

- Currency, 371

- DatabaseMetaData, 313, 323

- DataFlavor, 814, 818

- DataFlavor, 814, 818, 819

- DataInputStream, 82

- Date, 335, 356

- DateFormat, 363, 373

- DateTimeAtCompleted, 810

- DateTimeAtCreation, 810

- DateTimeAtProcessing, 810

- DateTimeFormatter, 352

- DefaultCellEditor, 593, 596,
 - 597, 609

- DefaultFormatter, 651

- DefaultHandler, 200
- DefaultListModel, 558, 559
- DefaultModelList, 558
- DefaultMutableTreeNode, 600, 604, 613, 617
- DefaultTreeCellRenderer, 615–618
- DefaultTreeModel, 599, 606, 608, 613, 625
- Desktop, 849, 853
- Destination, 810
- DiagnosticCollector, 427, 431
- DialogCallbackHandler, 512
- DocFlavor, 802
- DocPrintJob, 804
- Document, 156
- DocumentBuilder, 155, 159, 164, 172, 207, 211
- DocumentBuilderFactory, 164, 171, 173, 197
- DocumentFilter, 651
- DocumentName, 810
- domena ochronna, 486
- DOMResult, 225, 228
- DOMSource, 212, 224
- DOMTreeModel, 163
- DriverManager, 289, 337
- Duration, 341
- Ellipse2D, 713, 714
- EntryLogger, 465
- EnumSyntax, 810
- EnumSyntax, 810
- EventHandler, 442
- EventListenerList, 630
- Fidelity, 810
- Field, 441, 883
- File, 112, 600
- FileChannel, 129, 135
- FileInputStream, 70, 129, 486, 673
- FileLock, 135
- FileOutputStream, 68, 71, 129, 884
- FilePermission, 485, 490
- FileReader, 486
- Files, 113, 117, 122
- FileSystems, 124
- Finishings, 810
- Format, 387
- ForwardingJavaFileManager, 429, 431
- GeneralPath, 713, 718, 724–727, 734, 744
- GradientPaint, 735–737
- Graphics, 709, 710, 712, 744
- Graphics, 709, 712
- Graphics2D, 710–713, 735–738, 742–745, 758
- GregorianCalendar, 356
- GridBagLayout, 176–178
- HashPrintRequestAttributeSet, 784, 808
- URLConnection, 269
- HyperlinkEvent, 663–666
- Icon, 559
- IllegalArgumentException, 891
- IllegalStateException, 39, 764
- ImageInputStream, 763
- ImageIO, 736, 759, 765
- ImageIO, 759
- ImageReader, 766
- ImageReaderWriterSpi, 767
- ImageWriteParam, 765
- ImageWriter, 764, 768
- IndexOutOfBoundsException, 764
- InetAddress, 237, 238
- InetSocketAddress, 251
- informacyjne ziarnka, 437
- InitialContext, 337
- InputSource, 168, 173
- InputStream, 63, 239, 251, 252
- InputStreamReader, 673, 674
- Instant, 340, 356
- IntegerSyntax, 809, 810
- InterruptedIOException, 236
- InvalidPathException, 110
- JavaFileObject, 426, 427
- JComboBox, 725
- JDesktopPane, 688, 690
- JEditorPane, 661
- JFormattedTextField, 642
- JFrame, 688
- JInternalFrame, 688, 690
- JLabel, 562, 615
- JLayer, 703
- JList, 553
- JobAttributes, 812
- JobHoldUntil, 810
- JobImpressions, 810
- JobImpressionsCompleted, 810
- JobKOctets, 810
- JobKOctetsProcessed, 810
- JobMediaSheets, 810
- JobMediaSheetsCompleted, 810
- JobMessageFromOperator, 810
- JobName, 810
- JobOriginatingUserName, 811
- JobPriority, 811
- JobSheets, 811
- JobState, 811
- JobStateReason, 811
- JobStateReasons, 811
- JPanel, 788
- JProgressBar, 667
- JSplitPane, 700
- JTabbedPane, 681
- JTable, 572
- JTextArea, 496
- JTree, 598, 607
- Kernel, 781, 783
- KeyGenerator, 536, 540
- KeyPairGenerator, 543
- komplety zasobów, 395
- LayerUI, 703
- Line2D, 712–714
- ListResourceBundle, 395
- LocalDate, 344
- Locale, 361–365
- LocalTime, 347
- LoggingPermission, 492
- LoginContext, 502
- LoginContext, 502, 503, 506, 512
- LookupOp, 779–782
- LookupTable, 779
- ładowanie, 468
- Manager, 92
- MaskFormatter, 643, 652
- Matcher, 146
- MediaName, 811
- MediaSize, 811
- MediaSizeName, 811
- MediaTray, 811
- MessageDigest, 518
- MessageDigest, 518, 519
- MessageFormat, 385, 386, 388
- Method, 441, 883
- MissingResourceException, 393
- MonthDay, 345
- MultipleDocumentHandling, 811
- NameCallback, 513, 515
- NetPermission, 491
- NullPointerException, 891
- Number, 366
- NumberFormat, 365, 366, 369
- NumberOfDocuments, 811
- NumberOfInterveningJobs, 811
- NumberUp, 811
- ObjectInputStream, 95
- ObjectOutputStream, 95
- Optional, 29
- Optional, 30, 31, 34, 55

- ul>
- klasa
 - OrientationRequested, 811
 - OutOfMemoryError, 891
 - OutputDeviceAssigned, 811
 - OutputStream, 64, 239, 252
 - OutputStreamWriter, 72
 - PageFormat, 786
 - Package, 441
 - PageAttributes, 812
 - PageFormat, 786, 791, 792
 - PageRanges, 811
 - PagesPerMinute, 811
 - PagesPerMinuteColor, 811
 - ParseException, 366
 - PasswordCallback, 513, 515
 - Path, 110
 - Paths, 111, 123
 - Pattern, 23, 145
 - Permission, 495, 501
 - Point2D, 725, 744
 - Policy, 485
 - PopupMenu, 854
 - pozwolen, 495
 - PreparedStatement, 301
 - PresentationDirection, 811
 - Principal, 507
 - PrinterException, 784
 - PrinterInfo, 811
 - PrinterIsAcceptingJobs, 811
 - PrinterJob, 784–787, 791, 792, 802
 - PrinterLocation, 812
 - PrinterMakeAndModel, 812
 - PrinterMessageFromOperator, 812
 - PrinterMoreInfo, 812
 - PrinterMoreInfoManufacturer, 812
 - PrinterName, 812
 - PrinterResolution, 812
 - PrinterState, 812
 - PrinterStateReason, 812
 - PrinterStateReasons, 812
 - PrinterURI, 812
 - PrintJob, 784
 - PrintPreviewDialog, 794, 801
 - PrintQuality, 809, 812
 - PrintService, 804, 806
 - PrintServiceLookup, 802, 805
 - PrintStream, 73
 - PrintWriter, 72, 239, 264, 880
 - PrivilegedAction, 503
 - ProgressMonitor, 667, 670, 677
 - ProgressMonitorInputStream, 667, 673, 678
 - Properties, 150
 - PropertyChangeEvent, 698, 703
 - PropertyPermission, 490
 - PropertyVetoException, 695–699, 703
 - ProtectionDomain, 487
 - PushbackInputStream, 71
 - QuadCurve2D, 713, 716
 - QuadCurve2D.Double, 726
 - QueuedJobCount, 812
 - Random, 54, 537
 - RandomAccessFile, 84
 - RandomAccessFile, 87, 129, 766
 - Raster, 770, 773
 - Reader, 66, 224
 - Rectangle2D, 712–714
 - Rectangle2D, 714
 - ReferenceUriSchemesSupported, 812
 - ReflectPermission, 491
 - RenderingHints, 755, 758
 - RequestingUserName, 812
 - RescaleOp, 779, 782
 - ResourceBundle, 393, 396
 - ResourceBundles, 395
 - ResultSetMetaData, 323
 - RetentionPolicy, 452
 - RetinaScanCallback, 513
 - RoundRectangle2D, 713, 714
 - RoundRectangle2D.Double, 726
 - rozwiązywanie, 468
 - Runtime, 483
 - RuntimePermission, 491
 - SAXSource, 225
 - Scanner, 76, 239, 246
 - ScriptEngineManager, 414
 - SecretKeySpec, 540
 - SecureRandom, 537
 - SecurityException, 484, 486
 - SecurityManager, 486
 - SecurityManager, 486, 487
 - SecurityPermission, 492
 - Serializable, 107
 - SerializablePermission, 491
 - ServerSocket, 239, 241
 - Severity, 812
 - ShapeMaker, 725
 - ShapePanel, 725
 - SheetCollate, 812
 - ShortLookupTable, 779
 - Sides, 812
 - SimpleDateFormat, 386
 - SimpleDateFormat, 386, 660
 - SimpleDoc, 804, 806
 - SimpleFileVisitor, 121
 - SimpleJavaFileObject, 431
 - SimpleLoginModule, 511
 - SimplePrincipal, 509
 - SimulatedActivity, 670
 - Socket, 235, 236, 245
 - SocketChannel, 246
 - SocketChannel, 246, 251
 - SocketPermission, 490
 - SocketTimeoutException, 260
 - SpinnerListModel, 660
 - SpinnerNumberModel, 654, 659
 - SplashScreen, 848
 - Statement, 293, 310, 311, 334
 - StreamPrintService, 806
 - StreamPrintServiceFactory, 806
 - StreamResult, 212
 - StreamSource, 224, 225, 228
 - String, 335, 880
 - StringBuffer, 131, 676
 - StringBuilder, 427
 - StringSelection, 815, 820
 - Subject, 507
 - SyncProviderException, 320
 - System, 865
 - SystemTray, 854, 856
 - szyfrowanie plików, 477
 - TableColumn, 572, 584, 596
 - TableColumnModel, 572
 - TemporalAdjusters, 346
 - TextField, 365
 - TextLayout, 743, 744
 - TexturePaint, 735–737
 - Thread, 478
 - ThreadedEchoHandler, 242
 - Time, 335
 - Timestamp, 335
 - TimeZone, 378
 - Toolkit, 817
 - TransferHandler, 833, 835, 841
 - Transformer, 212
 - TransformerFactory, 212, 228
 - TrayIcon, 857
 - TreeModelEvent, 633
 - TreeNode, 607
 - TreePath, 607, 612, 619
 - TreeSelectionEvent, 619, 624
 - TreeSelectionModel, 619
 - UnixNumericGroupPrincipal, 503
 - UnixPrincipal, 502
 - UnknownHostException, 234
 - URI, 252

- URL, 252
 - URLClassLoader, 469
 - URLConnection, 252
 - WordCheckPermission, 496
 - WritableRaster, 769, 771, 774
 - Writer, 66
 - XPath, 191
 - Year, 345
 - YearMonth, 345
 - ZipEntry, 88
 - ZipFile, 90
 - ZipInputStream, 64, 88
 - ZipOutputStream, 89
 - ZonedDateTime, 349
 - klasy
 - abstrakcyjne, 616
 - kolumn, 571
 - klient, 233
 - klonowanie, 107
 - klucz, 535
 - prywatny, 520
 - publiczny, 520, 542
 - klucze wygenerowane automatycznie, 310
 - kod
 - ASCII, 359
 - bajtowy, 459
 - języków, 361
 - macierzysty, 859
 - Unicode, 359
 - kodowanie
 - plików źródłowych, 392
 - UTF-16, 79
 - UTF-8, 79, 391
 - znaków, 389
 - kolekcja typu NodeList, 156
 - kolektory przetwarzające, 44
 - kolory, 745
 - RGB, 745, 770
 - kolumny, 279, 571
 - ukrywanie, 578
 - wyświetlanie, 578
 - komórki
 - edycja, 588
 - rysowanie, 586
 - kompilacja, 426
 - skryptu, 420
 - kompilator, 468
 - GNU, 863
 - komplety zasobów, 392
 - implementacja klas, 395
 - klasy, 395
 - ładowanie, 393
 - pliki właściwości, 394
 - komponent
 - JEditorPane, 661
 - JList, 548
 - JProgressBar, 667, 846
 - JSpinner, 653
 - JTable, 563
 - JTextArea, 661
 - JTextField, 661
 - JTree, 159, 598, 843
 - Metal, 688
 - organizatory, 678
 - panele dzielone, 678
 - rozmieszczenie, 690
 - komponenty tekstowe, 633
 - formatowanie tekstu, 633
 - sformatowane pola wejściowe, 637
 - zmiana zawartości, 634
 - komunikaty, 385
 - formatowanie z wariantami, 387
 - indeks znacznika, 385
 - konsola, 390
 - konstruktory, 884
 - kontekst
 - graficzny, 739, 786, 787
 - tworzenia czcionki, 743
 - kontrola
 - dostępu, 467
 - poprawności dokumentów
 - XML, 166
 - atributy, 169
 - ATTLIST, 169
 - byty, 171
 - CDATA, 170
 - definicja typu dokumentu, 167
 - DOCTYPE, 167
 - DTD, 166, 167
 - ELEMENT, 166, 168
 - parser XML, 169
 - reguły zawartości elementów, 168
 - skrót, 171
 - typy atrybutów, 170
 - wartości domyślne
 - atributów, 170
 - warunki kontroli, 166
 - XML Schema, 166, 174
 - pozwoleń, 487
 - typów, 914
 - kończenie pracy maszyny
 - wirtualnej, 484
 - kopiowanie plików, 115
 - korzeń, 599
 - kryptografia klucza publicznego, 520, 542
 - krzywe, 713, 716, 725
 - Beziera, 717
 - drugiego stopnia, 717
 - punkty kontrolne, 716
 - trzeciego stopnia, 717
 - kursory, 312
- ## L
- las, 599, 603, 604
 - LD_LIBRARY_PATH, 899
 - LDAP, 545
 - liczby, 365
 - formatowanie, 365
 - formaty, 366
 - lokalizatory, 365
 - losowe, 537
 - LIKE, 282
 - limit czasu gniazda, 236
 - linie, 718
 - Linux, 282, 899
 - listy, 547
 - JList, 548
 - kolor tła komórki, 562
 - modele, 553
 - obiekt odrysowujący zawartość
 - komórek, 559
 - odrysowywanie zawartości, 559
 - powiadomienia, 550
 - prezentacja elementów, 549
 - przewijanie zawartości, 548
 - rozwijalne, 548
 - tworzenie, 548
 - usuwanie elementów, 558
 - wizualizacja danych, 553
 - wstawianie elementów, 558
 - zaznaczanie elementów, 549
 - liście, 599
 - localFile, 490
 - logika biznesowa, 504
 - logowanie, 503, 515
 - lokalizacja, 397
 - kod, 484
 - zasoby, 393
 - lokalizatory, 360
 - czas, 372
 - data, 372
 - domyślne, 363
 - języki, 363
 - liczby, 365
 - lokalna data, 343

L

- ładowanie klas, 468
 - ClassLoader, 473
 - implementacja procedury
 - ładującej, 473
 - klasy systemowe, 469
 - maszyna wirtualna, 468
 - procedura rozszerzona, 469
 - procedura systemowa, 469
 - procedury, 469
 - przestrzeń nazw, 471
 - rozszerzenia maszyny wirtualnej, 469
 - URLClassLoader, 469
- łańcuch zaufania, 525
- łańcuchy znaków, 388, 868
- łączenie
 - filtrów strumieni, 68
 - funkcji zwracających wartości
 - opcjonalne, 32
 - strumieni, 25
 - tabel, 281
- łuk, 714, 725

M

- macierz przekształceń, 740
- mailto, 252
- mapa, 39
- mapowanie plików w pamięci, 124
- maska layerEventMask, 706
- maszyna wirtualna, 478
- MD5, 518
- MDI, 687
- mechanizm przeciągnij i upuść, 828
- menedżer bezpieczeństwa, 468, 483, 489
 - domyślny, 484
 - pliki polityki, 489
 - pozwolenia, 483
 - SecurityManager, 486
- menedżer
 - ForwardingJavaFileManager, 435
- metaadnotacje, 452
- metadane, 322
 - DatabaseMetaData, 323
 - pozyskiwanie, 322
 - ResultSetMetaData, 323
- metoda
 - abort(), 516
 - absolute(), 316
 - accept(), 239, 241

- acceptChanges(), 320, 321
- actionPerformed(), 442
- add(), 606, 813
- addActionListener(), 440, 442, 857
- addAttribute(), 228
- addBatch(), 334
- addCellEditorListener(), 597
- addChangeListener(), 683, 686
- addClass(), 624
- addColumn(), 578, 583
- addDocumentListener(), 635, 636
- addElement(), 558, 559
- addFlavorListener(), 820
- addHyperlinkListener(), 666
- addListSelectionListener(), 553
- addSelectionListener(), 619
- addTab(), 682, 685
- addTreeModelListener(), 629, 632
- addVetoableChangeListener(), 698, 702
- afterLast(), 317
- allMatch(), 28
- allocate(), 131
- andFilter(), 577
- annotationType(), 444
- anyMatch(), 28
- append(), 724, 727, 802
- appendChild(), 208, 211
- applyPattern(), 386
- asCharBuffer(), 131
- available(), 62, 63
- average(), 51
- beforeFirst(), 316
- breadthFirstEnumeration(), 613, 617
- call(), 430
- CallNonVirtualXxxMethod(), 886
- CallNonVirtualXxxMethodA(), 886
- CallNonVirtualXxxMethodV(), 886
- CallStaticObjectMethod(), 884
- CallStaticXxxMethod(), 886
- CallStaticXxxMethodA(), 886
- CallStaticXxxMethodV(), 886
- CallXxxMethod(), 885
- CallXxxMethodA(), 885
- CallXxxMethodV(), 885
- cancelCellEditing(), 594–597
- cancelRowUpdates(), 317
- canImport(), 836, 841
- canInsertImage(), 765, 768
- capacity(), 133
- changedUpdate(), 637
- characters(), 198, 203
- charAt(), 68
- checkError(), 73
- checkExit(), 483–486
- checkLogin(), 509
- checkPermission(), 487, 495, 496
- children(), 613
- clear(), 133
- clearParameters(), 306
- clip(), 711, 743, 744
- close(), 67, 135, 239, 241, 291–293, 670, 849
- closePath(), 718, 727
- codePoint(), 54
- collect(), 39
- Collection.parallelStream(), 55
- Collectors.groupingBy
 - Concurrent(), 57
- Collectors.toMap(), 39
- column(), 293
- commit(), 332, 334, 516
- commitEdit(), 651
- compare(), 384
- compareTo(), 378, 384
- compile(), 145, 420
- connect(), 236, 254, 261
- containsAll(), 499
- copy(), 115
- count(), 20
- counting(), 47
- createBindings(), 417
- createBlob(), 308
- createClob(), 308
- createElement(), 208, 211
- createGraphics(), 848
- createImageInputStream(), 763, 766
- createImageOutputStream(), 764, 766
- createPrintJob(), 805
- createStatement(), 290, 311, 316, 332, 333
- createTextNode(), 208, 211
- createTransferable(), 835
- creationTime(), 118
- curveTo(), 718, 726
- dateFilter(), 577
- dayOfWeekInMonth(), 346
- decode(), 270
- defaultPage(), 791
- defineClass(), 473, 478

- deleteRow(), 315, 317
- depthFirstEnumeration(), 613, 617
- depthFirstTraversal(), 614
- DestroyJavaVM(), 896, 900
- DiagnosticCollector(), 431
- digest(), 519
- displayMessage(), 857
- distinct(), 26, 27
- doAs(), 503, 504, 507
- doAsPrivileged(), 503, 504, 507
- doFinal(), 535, 536, 540
- doubles(), 54
- draw(), 711, 712
- draw3DRect(), 712, 713
- drawArc(), 712
- drawLine(), 712
- drawOval(), 712
- drawPolygon(), 712, 713
- drawPolyline(), 712
- drawRect(), 710
- drawRectangle(), 712
- drawRoundRect(), 712
- edit(), 853
- encode(), 270
- end(), 146
- endDocument(), 198, 202
- endElement(), 198, 202
- equals(), 384, 444, 451, 499
- error(), 172, 173
- eval(), 420
- evaluate(), 191, 195
- ExceptionCheck(), 895
- ExceptionClear(), 895
- ExceptionOccured(), 892, 895
- execute(), 291, 311, 319, 321
- executeBatch(), 333, 334
- executeQuery(), 290, 291, 301, 306
- executeUpdate(), 290, 301, 306, 311, 332
- exit(), 483
- exitInternal(), 484
- exportAsDrag(), 835
- exportDone(), 835
- fatalError(), 172, 173
- fileKey(), 118
- Files.list(), 118
- Files.newDirectoryStream(), 120
- Files.walk(), 120
- FileVisitResult visitFileFailed(), 123
- fill(), 711, 712
- fillOval(), 710
- filter(), 19, 24, 782
- find(), 146
- findAny(), 28
- findClass(), 473, 477, 877
- findFirst(), 28
- first(), 316
- firstDayOfMonth(), 346
- firstDayOfNextMonth(), 346
- firstDayOfNextYear(), 346
- flatMap(), 25, 32
- flavorsChanged(), 820
- flip(), 132, 133
- flush(), 67, 542
- forEach(), 34
- forEachOrdered(), 34
- forLanguageTag(), 364
- format(), 369, 385
- fprint(), 891
- generateKey(), 536, 540
- get(), 31, 130, 813
- getAddress(), 237, 238
- getAdvance(), 745
- getAllByName(), 237, 238
- getAllFrames(), 695, 700
- getAllowsChildren(), 606
- getAllowUserInteraction(), 260
- getAnnotation(), 441, 443
- getAnnotations(), 443
- GetArrayLength(), 887, 889
- getAscent(), 745
- getAttribute(), 164, 178
- getAttributes(), 158, 165, 813
- getAutoCommit(), 334
- getAvailableDataFlavors(), 819
- getAvailableIDs(), 378
- getAvailableLocales(), 363, 366, 384
- getAverage(), 38
- getBackground(), 560, 562
- getBinaryStream(), 307
- getBlob(), 307
- getBlockSize(), 539
- GetBooleanArrayElements(), 889
- getBounds(), 848
- getBundle(), 393–396
- getByName(), 237, 238
- GetByteArrayElements(), 904
- getBytes(), 307
- getCategory(), 809, 812
- getCellEditorValue(), 593–597
- getCellSelectionEnabled(), 583
- getCertificates(), 487
- getChannel(), 125, 129
- getCharacterStream(), 308
- getCharContent(), 427, 431
- getChild(), 163, 625, 631, 632
- getChildAt(), 613
- getChildCount(), 608, 613, 632
- getChildIndex(), 843
- getChildNodes(), 164
- getClassLoader(), 469, 477
- getClip(), 744
- getClob(), 307
- getCodeSource(), 487
- getCollationKey(), 380, 384
- getColorModel(), 770, 773
- getColumn(), 584, 843
- getColumnClass(), 571, 582
- getColumnCount(), 331, 568, 571
- getColumnDisplaySize(), 331
- getColumnLabel(), 331
- getColumnName(), 331, 571
- getColumnNumber(), 173, 431
- getColumnSelectionAllowed(), 583
- getCommand(), 321
- getComponent(), 842
- getComponentAt(), 686
- getConcurrency(), 316
- getConnection(), 287, 289, 297, 337
- getConnectTimeout(), 260
- getContent(), 261
- getContentEncoding(), 257, 261
- getContentLength(), 257, 261
- getContentPane(), 688, 702
- getContents(), 817
- getContentType(), 257, 261
- getContext(), 417
- getContextClassLoader(), 478
- getCount(), 38
- getCountry(), 364
- getCrc(), 90
- getCurrencyCode(), 371
- getCurrencyInstance(), 365, 370, 642
- getData(), 158, 165, 818
- getDataElements(), 770–775
- getDataFlavors(), 842
- getDate(), 257, 261
- getDateInstance(), 378, 642
- getDateTimeInstance(), 642
- getDayOfMonth(), 344, 350
- getDayOfWeek(), 344, 345, 350
- getDayOfYear(), 344, 350
- getDeclaredAnnotations(), 443

metoda

- getDecomposition(), 384
- getDefault(), 363, 364
- getDefaultEditor(), 596
- getDefaultFractionsDigits(), 371
- getDefaultName(), 515
- getDefaultRenderer(), 596
- getDefaultToolkit(), 815
- getDescent(), 745
- getDesktop(), 853
- getDesktopPane(), 702
- getDisplayCountry(), 364
- getDisplayLanguage(), 364
- getDisplayName(), 363–366
- getDocument(), 637, 663
- getDocumentElement(), 156, 164
- getDocumentFilter(), 651
- getDoInput(), 260
- getDoOutput(), 260
- getDouble(), 291
- GetDoubleField(), 874
- getDropAction(), 842
- getDropPoint(), 842
- getElementAt(), 554, 558
- getEngineByExtension(), 415
- getEngineByMimeType(), 415
- getEngineByName(), 415
- getEngineFactories(), 414
- getErrorCode(), 295
- getErrorStream(), 269
- getEventType(), 663
- getExpiration(), 257, 261
- getExtensions(), 415
- getFieldDescription(), 624
- GetFieldID(), 874, 878
- getFields(), 631
- getFileName(), 112
- getFilePointer(), 84
- getFileSuffixes(), 767
- getFirstChild(), 164
- getFocusLostBehavior(), 651
- getFontRenderContext(), 743
- getFontRendererContext(), 744
- getForeground(), 560, 562
- getFormatNames(), 767
- getFrameIcon(), 702
- getHeaderField(), 255, 257, 261
- getHeaderFieldKey(), 255, 257, 261
- getHeaderFields(), 255, 257, 261
- getHeight(), 767, 786, 792
- getHostAddress(), 238
- getHostName(), 238
- getHour(), 348, 350
- getHumanPresentableName(), 819
- getIconAt(), 686
- getIdentifier(), 586
- getIfModifiedSince(), 260
- getImage(), 857
- getImageableHeight(), 786, 792
- getImageableWidth(), 786, 792
- getImageableX(), 792
- getImageableY(), 792
- getImageReadersByFormatName(), 766
- getImageReadersByMimeType(), 759, 766
- getImageReadersBySuffix(), 759, 766
- getImageURL(), 848
- getImageWritersByFormatName(), 766
- getImageWritersByMimeType(), 766
- getImageWritersBySuffix(), 766
- getIndex(), 843
- getIndexOfChild(), 625, 632
- getInputStream(), 235, 239, 255, 264
- getInstance(), 371, 384, 518, 534, 539, 748, 753
- getIntegerInstance(), 640
- GetIntField(), 874
- getInvalidCharacters(), 652
- getJDBCMinorVersion(), 330
- getJDBCMinorVersion(), 330
- getKeys(), 397
- getKind(), 431
- getLanguage(), 364
- getLastChild(), 158, 164
- getLastModified(), 257, 261
- getLastPathComponent(), 607, 612
- getLastSelectedPathComponent(), 607, 612
- getLayoutOrientation(), 552
- getLayoutPolicy(), 686
- getLeading(), 745
- getLength(), 156, 165, 203, 636
- getLineNumber(), 173, 431
- getListCellRendererComponent(), 562, 563
- getLocale(), 386
- getLocalHost(), 237, 238
- getLocalName(), 197, 203
- getLocation(), 487
- getLogger(), 460
- getMax(), 39
- getMaxConnections(), 330
- getMaximum(), 676
- getMaximumFractionDigits(), 370
- getMaximumIntegerDigits(), 370
- getMaxStatement(), 293
- getMaxStatements(), 330
- getMessage(), 431
- getMetaData(), 330, 331
- GetMethodID(), 885, 886
- getMimeType(), 819
- getMimeTypes(), 415, 767
- getMin(), 39
- getMinimum(), 676
- getMinimumFractionDigits(), 370
- getMinimumIntegerDigits(), 370
- getMinute(), 348, 350
- getModel(), 558, 559, 586
- getMonth(), 344, 350
- getMonthValue(), 344, 350
- getMoreResults(), 310
- getName(), 90, 498, 507, 515, 804, 812
- getNames(), 415
- getNamespaceURI(), 197, 198
- getNano(), 348, 350
- getNewValue(), 703
- getNextException(), 295
- getNextSibling(), 158, 165
- getNextValue(), 655
- getNodeName(), 158, 165, 197
- getNodeValue(), 158, 165
- getNumberInstance(), 365, 642
- getNumImages(), 767
- getNumThumbnails(), 764, 767
- getObject(), 396
- GetObjectArrayElement(), 889
- GetObjectClass(), 874, 877
- GetObjectField(), 874
- getOffset(), 350
- getOrientation(), 792
- getOriginatingProvider(), 759, 767, 768
- getOutline(), 743
- getOutputSize(), 540
- getOutputStream(), 235, 239, 255, 264
- getOverwriteMode(), 651
- getPageCount(), 793
- getParent(), 112, 477, 613
- getParentNode(), 165
- getPassword(), 321, 515

- getPath(), 624, 843
- getPaths(), 624
- getPathToRoot(), 609
- getPercentInstance(), 365, 642
- getPixel(), 770, 773, 775
- getPixels(), 774
- getPlaceholder(), 653
- getPlaceholderCharacter(), 653
- getPointCount(), 725
- getPopupMenu(), 857
- getPreferredSize(), 560, 562, 636
- getPreviousSibling(), 165
- getPreviousValue(), 655
- getPrincipals(), 507
- getPrintable(), 802
- getPrinterJob(), 784, 791
- getPrintService(), 807
- getPrompt(), 515
- getProperty(), 883, 884
- getPropertyNames(), 703
- getProtectionDomain(), 487
- getPrototypeCell(), 557
- getQName(), 203
- getRaster(), 769, 773
- getReaderFormatNames(), 766
- getReaderMIMETypes(), 766
- getRepresentationClass(), 819
- getRequestProperties(), 261
- getResultSet(), 292
- getRGB(), 771, 774, 775
- getRoot(), 112, 163, 632
- getRotateInstance(), 740, 741
- getRow(), 316, 843
- getRowCount(), 568, 569, 571
- getRowHeight(), 583
- getRowMargin(), 583
- getRowSelectionAllowed(), 583
- getSavepointId(), 334
- getSavepointName(), 334
- getScaleInstance(), 740, 741
- getSecond(), 348, 350
- getSelectedColumns(), 578
- getSelectedComponent(), 686
- getSelectedIndex(), 683, 686
- getSelectedNode(), 608
- getSelectedValue(), 553
- getSelectedValues(), 550, 553
- getSelectionBackground(), 560, 562
- getSelectionForeground(), 560, 563
- getSelectionMode(), 552
- getSelectionModel(), 574, 583
- getSelectionPath(), 612, 619, 624
- getShearInstance(), 740, 741
- getSize(), 90, 554, 558
- getSource(), 431
- getSourceActions(), 835
- getSourceDropActions(), 842
- getSplashScreen(), 848
- getSQLState(), 295
- getStandardFileManager(), 430
- GetStaticFieldID(), 877, 878
- GetStaticMethodID(), 886
- GetStaticXxxField(), 877, 878
- getStrength(), 384
- getString(), 291, 322, 397, 677
- getStringArray(), 397
- GetStringChars(), 871
- GetStringLength(), 871
- GetStringRegion(), 870
- GetStringUTFChars(), 869, 870, 872, 904
- GetStringUTFLength(), 870
- GetStringUTFRegion(), 870
- getStringValue(), 586
- getSubject(), 507
- getSum(), 38
- GetSuperClass(), 914
- getSymbol(), 371
- getSystemClassLoader(), 477
- getSystemClipboard(), 815, 817
- getSystemJavaCompiler(), 426
- getSystemTray(), 856
- getTabCount(), 686
- getTableCellEditorComponent(), 593, 595, 597
- getTableCellRendererComponent(), 593, 596
- getTableName(), 321
- getTables(), 330
- getTagName(), 156, 164
- getText(), 636
- getTimeInstance(), 378, 642
- getTitleAt(), 686
- getTooltip(), 857
- getTransferData(), 818
- getTransferDataFlavors(), 820
- getTranslateInstance(), 740, 742
- getTrayIconSize(), 856
- getTreeCellRendererComponent(), 616, 617
- getType(), 316
- getUpdateCount(), 292
- getURI(), 203
- getURL(), 320, 664, 666
- getUseCaches(), 260
- getUserDropAction(), 842
- getUsername(), 320
- getValidCharacters(), 652
- getValue(), 203, 586, 677, 809, 902
- getValueAt(), 568, 571, 593
- getValueContainsLiteralCharacters(), 653
- getValueCount(), 586
- getVendorName(), 767
- getVersion(), 760, 767
- getVisibleRowCount(), 552
- getWidth(), 767, 786, 792
- getWriterFormatNames(), 763, 766
- getWriterMIMETypes(), 766
- getXxx(), 130, 292
- GetXxxArrayElements(), 890
- GetXxxArrayRegion(), 890
- GetXxxField(), 878
- getYear(), 344, 350
- group(), 146
- groupCount(), 146
- groupingBy(), 43, 44
- groupingByConcurrent(), 43
- handle(), 515
- handleGetObject(), 397
- hashCode(), 444, 460
- hasMoreElements(), 902, 903, 912
- hasRemaining(), 129
- hyperlinkUpdate(), 663, 666
- ifPresent(), 30, 55
- implies(), 487, 495, 501
- importData(), 838, 841
- indexOfTab(), 686
- init(), 486, 540
- initialize(), 511, 516
- insertNodeInto(), 608, 613
- insertRow(), 315, 317
- insertTab(), 682, 685
- insertUpdate(), 637
- installUI(), 707
- Instant.now(), 340
- interrupt(), 246
- ints(), 54
- intValue(), 638, 904
- isAdjusting(), 550
- isAfter(), 344, 348, 350
- isAfterLast(), 317
- isAnnotationPresent(), 442
- IsAssignableFrom(), 903, 914
- isBefore(), 344, 348, 350
- isBeforeFirst(), 317
- isCanceled(), 678

- metoda
 - isCellEditable(), 571, 588, 594, 597
 - isClosable(), 701
 - isClosed(), 236, 702
 - isConnected(), 236
 - isContinuousLayout(), 681
 - isDataFlavorAvailable(), 817
 - isDataFlavorSupported(), 818
 - isDesktopSupported(), 853
 - isDirectory(), 90, 118
 - isEchoOn(), 515
 - isEditValid(), 651
 - isFirst(), 317
 - isGroupingUsed(), 370
 - isIcon(), 701
 - isIconifiable(), 701
 - isIgnoringElementContentWhitespace(), 173
 - isImageAutoSize(), 857
 - isIndeterminate(), 677
 - isInputShutdown(), 245
 - isInsert(), 843
 - isInsertColumn(), 843
 - isInsertRow(), 843
 - isLast(), 317
 - isLeaf(), 604–606, 629, 632
 - isLeapYear(), 344
 - isMaximizable(), 701
 - isMaximum(), 701
 - isMimeTypeEqual(), 819
 - isNameSpaceAware(), 198, 202
 - isOneTouchExpandable(), 681
 - isOutputShutdown(), 245
 - isParseIntegerOnly(), 370
 - isPresent(), 31
 - isRegularFile(), 118
 - isResizable(), 701
 - isSelected(), 702
 - isStringPainted(), 677
 - isSupported(), 853, 856
 - isSymbolicLink(), 118
 - isUnresolved(), 251
 - isValidating(), 173, 202
 - isVisible(), 702
 - item(), 165
 - iterator(), 36, 294
 - JNI_CreateJavaVM(), 895, 899
 - last(), 316
 - lastAccessTime(), 118
 - lastDayOfMonth(), 346
 - lastDayOfYear(), 346
 - lastInMonth(), 346
 - lastModifiedTime(), 118
 - layoutPages(), 793
 - length(), 68, 307
 - letters(), 25
 - limit(), 26, 129
 - lineTo(), 718, 726
 - loadClass(), 473
 - loadLibrary(), 864, 865
 - lock(), 133, 135
 - login(), 506, 516
 - logout(), 506, 516
 - longs(), 54
 - lookingAt(), 146
 - lookup(), 337
 - lookupPrintServices(), 802–804
 - lostOwnership(), 818
 - mail(), 853
 - main(), 468
 - makeShape(), 725
 - makeVisible(), 609, 612
 - map(), 24, 31
 - mapping(), 45, 48
 - mark(), 64, 133
 - matches(), 146
 - max(), 28, 51
 - maxBy(), 47
 - min(), 28, 51
 - minBy(), 47
 - minus(), 344, 348, 350
 - minusDays(), 344, 350
 - minusHours(), 348, 350
 - minusMinutes(), 348, 350
 - minusMonths(), 344, 350
 - minusNanos(), 348, 350
 - minusSeconds(), 348, 350
 - minusWeeks(), 344, 350
 - minusYears(), 344, 350
 - moveColumn(), 583
 - moveTo(), 718, 726
 - moveToBack(), 702
 - moveToCurrentRow(), 315, 317
 - moveToFront(), 702
 - moveToInsertRow(), 315, 317
 - NewByteArray(), 903
 - newDocument(), 207
 - newDocumentBuilder(), 164
 - newInputStream(), 251
 - newInstance(), 164, 194, 202, 212
 - NewObject(), 886
 - NewObjectA(), 886
 - NewObjectV(), 886
 - newOutputStream(), 246, 252
 - newSAXParser(), 202
 - NewString(), 871
 - NewStringUTF(), 870, 903
 - newXPath(), 194
 - next(), 292, 346
 - nextElement(), 902, 903, 913
 - nextInt(), 77
 - nextOrSame(), 346
 - nodeChanged(), 608, 613
 - noneMatch(), 28
 - normalize(), 111
 - notFilter(), 577
 - now(), 348, 349
 - of(), 32, 348
 - ofInstant(), 349
 - ofLocalizedDate(), 354
 - ofLocalizedDateTime(), 354
 - ofLocalizedTime(), 354
 - ofNullable(), 32
 - open(), 129, 853
 - openConnection(), 254, 259
 - openInputStream(), 261
 - openOutputStream(), 261, 431
 - openStream(), 252, 255, 259
 - order(), 130
 - orElse(), 30, 55
 - orElseGet(), 30, 55
 - orElseThrow(), 30
 - orFilter(), 577
 - pageDialog(), 787, 791
 - paint(), 707, 710
 - paintComponent(), 562, 710, 734, 788, 801
 - parallel(), 59
 - parallelStream(), 18, 20, 59
 - parse(), 164, 202, 228, 365, 369
 - partitioningBy(), 43, 45
 - pathFromAncestorEnumeration(), 614
 - Paths.get(), 110
 - plus(), 344, 348, 350
 - plusDays(), 344, 350
 - plusHours(), 348, 350
 - plusMinutes(), 348, 350
 - plusMonths(), 344, 350
 - plusNanos(), 350
 - plusSeconds(), 348, 350
 - plusWeeks(), 344, 350
 - plusYears(), 344, 350
 - populate(), 321
 - position(), 133
 - postOrderEnumeration(), 617
 - postOrderTraversal(), 614
 - premain(), 465
 - preOrderEnumeration(), 617
 - preOrderTraversal(), 614

- prepareStatement(), 306, 311, 316
- previous(), 316, 346
- previousOrSame(), 346
- preVisitDirectory(), 123
- print(), 784, 786, 791, 805, 853, 880
- printDialog(), 784, 791
- printf(), 860, 879
- println(), 73
- process(), 457
- processAnnotations(), 441, 442
- processComponentEvent(), 707
- processKeyEvent(), 707
- processMouseEvent(), 707
- processMouseMotionEvent(), 707
- processMouseWheelEvent(), 707
- put(), 130
- putClientProperty(), 603, 606
- putXxx(), 130
- quadTo(), 718, 726
- quoteReplacement(), 146
- range(), 52
- rangeClosed(), 52, 53
- read(), 62, 67, 246, 541, 759, 763, 767
- readBoolean(), 82
- readByte(), 82
- readChar(), 82
- readDouble(), 82
- readFixedString(), 85
- readFloat(), 82
- readInt(), 82
- readLong(), 82
- readObject(), 103
- readResolve(), 105
- readShort(), 83
- readThumbnail(), 764, 767
- reduce(), 48, 50
- regexFilter(), 577
- relative(), 312, 316
- relativize(), 112, 254
- release(), 134
- releaseSavepoint(), 332, 334
- ReleaseStringChars(), 871
- ReleaseStringUTFChars(), 870, 872
- ReleaseXxxArrayElements(), 890
- reload(), 608, 613
- remaining(), 133
- remove(), 813
- removeActionListener(), 857
- removeCellEditorListener(), 597
- removeColumn(), 578, 583
- removeElement(), 558, 559
- removeNodeFromParent(), 608, 613
- removeTabAt(), 682, 685
- removeTreeModelListener(), 629, 632
- removeUpdate(), 637
- replace(), 640
- replaceAll(), 146
- replaceFirst(), 146
- reset(), 133, 147, 519
- reshape(), 689, 702
- resolve(), 111, 254
- resolveEntity(), 173
- resolveSibling(), 111
- rewind(), 133
- rollback(), 332, 334
- rotate(), 738, 742
- run(), 430, 507
- scale(), 738, 742
- scrollPathToVisible(), 609, 612
- set(), 711
- setAllowsChildren(), 605, 606
- setAllowUserInteraction(), 256, 260
- setAsksAllowsChildren(), 605, 606
- setAsynchronousLoadPriority(), 663
- setAttribute(), 208, 212
- setAttributeNS(), 212
- setAutoCommit(), 332, 334
- setAutoResizeMode(), 582
- setBackground(), 425
- setBottomComponent(), 681
- SetByteArrayRegion(), 903
- setCellEditor(), 596
- setCellRenderer(), 561, 563, 596
- setCellSelectionEnabled(), 574
- setClip(), 743, 744, 786
- setClosable(), 701
- setClosed(), 698, 702
- setClosedIcon(), 618
- setColumnSelectionAllowed(), 574, 583
- setCommand(), 319, 321
- setComponentAt(), 686
- setComposite(), 711, 748, 753
- setConnectTimeout(), 260
- setContentHandler(), 228
- setContentPane(), 702
- setContents(), 815, 817
- setContextClassLoader(), 478
- setContinuousLayout(), 679, 681
- setCrc(), 90
- setDataElements(), 771, 774
- setDecomposition(), 384
- setDefault(), 364
- setDefaultRenderer(), 587
- setDoInput(), 255, 260
- setDoOutput(), 255, 260, 264
- SetDoubleField(), 874
- setDragEnabled(), 833
- setDragMode(), 700, 701
- setDropAction(), 842
- setDropMode(), 842
- setEditable(), 609, 662
- setEditor(), 659
- setEntityResolver(), 172
- setErrorHandler(), 172, 173
- setFixedCellHeight(), 557
- setFixedCellWidth(), 557
- setFocusLostBehavior(), 639, 651
- setFrameIcon(), 702
- setGroupingUsed(), 370
- setHeaderRenderer(), 588, 596
- setHeaderValue(), 588, 596
- setIcon(), 701
- setIconAt(), 686
- setIconifiable(), 701
- setIfModifiedSince(), 260
- setIgnoringElementContent
 Whitespace(), 172, 174
- setImage(), 857
- setImageAutoSize(), 857
- setImageURL(), 848
- setIndeterminate(), 670, 677
- setInput(), 763
- SetIntField(), 874
- setInvalidCharacters(), 653
- setLayerEventMask(), 706
- setLayoutOrientation(), 552
- setLeafIcon(), 618
- setLeftComponent(), 681
- setLocale(), 363, 386
- setLogWriter(), 289
- setMaximizable(), 701
- setMaximum(), 667, 676, 695, 702
- setMaximumFractionDigits(), 370
- setMaximumIntegerDigits(), 370
- setMaxWidth(), 573, 584
- setMillisToDecidePopup(), 673
- setMillisToPopup(), 673
- setMinimum(), 667, 676
- setMinimumFractionDigits(), 370
- setMinimumIntegerDigits(), 370
- setMinWidth(), 573, 584
- setModifiedSince(), 256
- setName(), 515

metoda

- setNamespaceAware(), 176, 197–202
- setNote(), 677
- SetObjectArrayElement(), 891
- SetObjectField(), 874
- setOneTouchExpandable(), 679, 681
- setOpenIcon(), 618
- setOutput(), 768
- setOutputProperty(), 212
- setOverwriteMode(), 651
- setPage(), 662, 664, 666
- setPageable(), 792, 802
- setPaint(), 711, 735–737
- setParseIntegerOnly(), 370
- setPassword(), 319, 321, 515
- setPixel(), 769, 774
- setPixels(), 769, 774
- setPlaceholder(), 653
- setPlaceholderCharacter(), 653
- setPopupMenu(), 857
- setPreferredSize(), 636
- setPreferredSize(), 573, 584
- setPrintable(), 791
- setProgress(), 670, 677
- setPrototypeCell(), 557
- setPrototypeCellValue(), 557
- setReadTimeout(), 260
- setRenderHints(), 755
- setRenderingHint(), 753, 758
- setRenderingHints(), 710, 758
- setRequestProperty(), 256, 260
- setResizable(), 573, 584, 701
- setRightComponent(), 681
- setRootVisible(), 603, 605
- setRowFilter(), 584
- setRowHeight(), 573, 574, 583
- setRowMargin(), 574, 583
- setRowSelectionAllowed(), 574, 583
- setSavepoint(), 332, 334
- setSecurityManager(), 489
- setSeed(), 537
- setSelected(), 702
- setSelectedIndex(), 682, 685
- setSelectionMode(), 549, 574, 584
- setShowsRootHandles(), 603, 605
- setSize(), 90
- setSortable(), 584
- setSoTimeout(), 235, 236
- SetStaticXxxField(), 877
- setStrength(), 384
- setString(), 301, 667, 677
- setStringPainted(), 667, 677
- setStroke(), 711, 728, 734
- setTabLayoutPolicy(), 682, 686
- setTableName(), 321
- setText(), 663
- setTitleAt(), 686
- setTooltip(), 857
- setTopComponent(), 681
- setToRotation(), 740, 742
- setToScale(), 740, 742
- setToShear(), 740, 742
- setToTranslation(), 740, 742
- setTransferHandle(), 832, 833
- setTransferHandler(), 841
- setTransform(), 740, 742
- setURL(), 319, 320
- setUseCaches(), 256, 260
- setUsername(), 319, 320
- setUserObject(), 600, 606
- setValidating(), 173, 202
- setValidCharacters(), 652
- setValue(), 638, 655, 677, 902–904
- setValueAt(), 571, 595
- setValueContainsLiteralCharacters(), 653
- setVisible(), 689, 702
- setVisibleRowCount(), 549, 552
- setWidth(), 573, 584
- setWriter(), 417
- setXxx(), 306
- SetXxxArrayRegion(), 890
- SetXxxField(), 878
- shear(), 738, 742
- shouldSelectCell(), 594–597
- show(), 702
- shutdownInput(), 245
- shutdownOutput(), 245
- size(), 118
- skip(), 26, 63
- sort(), 379
- sorted(), 27
- split(), 76
- splitAsStream(), 23
- start(), 146
- startDocument(), 198, 202
- startElement(), 198, 200, 202
- stateChanged(), 683
- stopCellEditing(), 594–597
- stream(), 18
- Stream.sorted(), 56
- stringToValue(), 645
- sum(), 51
- summarizingDouble(), 38
- summarizingInt(), 38
- summarizingLong(), 38
- summaryStatistics(), 51
- summingDouble(), 47
- summingInt(), 47
- summingLong(), 47
- supportsBatchUpdates(), 335
- supportsResultSetConcurrency(), 313, 318
- supportsResultSetType(), 313, 317
- Throw(), 894
- ThrowNew(), 894
- toAbsolutePath(), 112
- toArray(), 35, 51, 53, 813
- toConcurrentMap(), 42
- toFile(), 112
- toInstant(), 350
- toList(), 38
- toLocalDate(), 350
- toLocalTime(), 350
- toMap(), 40, 42
- toNanoOfDay(), 348
- toPath(), 112
- toPattern(), 660
- toSecondOfDay(), 348
- toSet(), 38
- toString(), 365, 371, 444, 457, 600, 725
- transform(), 212, 224, 711, 740, 742
- translate(), 738, 742, 793
- treeNodesChanged(), 632
- treeNodesInserted(), 632
- treeNodesRemoved(), 632
- treeStructureChanged(), 632
- trim(), 158, 366
- tryLock(), 135
- uninstallUI(), 707
- unordered(), 59
- until(), 344
- update(), 519, 535, 540, 849
- updateDouble(), 314
- updateRow(), 314–317
- updateXxx(), 317
- valueChanged(), 550, 618, 624
- valueForPathChanged(), 630, 632
- valueToString(), 644, 651
- vetoableChange(), 698, 703
- visitFile(), 123
- warning(), 172, 173
- with(), 346

withDayOfMonth(), 344, 350
 withDayOfYear(), 344, 350
 withHour(), 348, 350
 withMinute(), 348, 350
 withMonth(), 344, 350
 withNano(), 348, 350
 withSecond(), 348, 350
 withYear(), 344, 350
 withZoneSameInstant(), 350
 withZoneSameLocal(), 350
 write(), 62, 246, 542, 759, 768
 writeBoolean(), 83
 writeByte(), 83
 writeChar(), 83
 writeChars(), 82
 writeDouble(), 83
 writeFixedString(), 84
 writeFloat(), 83
 writeInsert(), 765, 768
 writeInt(), 83
 writeLong(), 83
 writeObject(), 102
 writeShort(), 83
 writeUTF(), 81

metody
 LocalDate, 344
 klasy LocalTime, 348
 klasy TemporalAdjusters, 346
 klasy ZonedDateTime, 349
 macierzyste, 859
 alternatywne wywoływanie
 metod, 885
 dostęp do pól instancji, 874,
 878
 dostęp do pól statycznych,
 877
 funkcje języka C, 860
 implementacja, 863
 interfejs programowy
 wywołań, 895
 jarray, 887
 język C++, 863
 JNI, 866
 konstruktory, 884
 łańcuchy znaków, 868, 879
 obsługa błędów, 890, 894
 parametry numeryczne, 866
 przeciążanie identyfikatorów,
 861
 rejestr systemu Windows, 900
 składowe obiektu, 873
 sygnatury, 878
 tablice, 886, 889
 wartości zwracane, 866

wyrzucanie wyjątków, 891
 wywoływanie metod języka
 Java, 885
 wywoływanie metod
 obiektów, 880
 wywoływanie metod
 statycznych, 883
 statyczne, 883
 Microsoft ASP, 262
 MIME, 759, 818
 model
 drzewa, 599
 języka, 455
 kolorów, 770
 tabeli, 563, 568
 model-widok-nadzorca, 553
 moduł
 JAAS, 507
 logowania, 503, 508
 hasła, 509, 512
 nazwa użytkownika, 512
 options, 511
 SimpleLoginModule, 509
 uwierzytelniania, 503
 modyfikacja
 klas podczas ładowania, 466
 kodu bajtowego podczas
 ładowania, 464
 kodu maszyny wirtualnej, 481
 modyfikatory, 437
 dat, 346
 modyfikowanie
 mechanizmu serializacji, 102
 plików klasowych, 459
 monitory postępu, 670
 strumieni wejścia, 673
 mostek JDBC/ODBC, 276

N

nagłówek, 588
 nagłówki żądań HTTP, 256
 native2ascii, 392
 negatyw obrazu, 779
 NMTOKEN, 170
 normalizacja łańcuchów
 znakowych, 380
 NOT LIKE, 282
 NUMERIC, 283, 335
 numeryczne parametry metod, 866

O

obiekt
 ByteArrayJavaClass, 435
 Charset, 80
 CompilationTask, 427
 Diagnostic, 427
 DiagnosticListener, 427
 JavaCompiler, 429
 JavaFileManager, 427
 LayerUI, 703
 Optional<T>, 29
 Path, 111
 SQLException, 294
 typu Buffer, 131
 typu Document, 159
 typu JPanel, 425
 typu NamedNodeMap, 158
 ZipEntry, 88

obiekty
 binarne, 306
 dostęp do składowych, 873
 formatujące, 642
 nietypowe, 644
 obsługi zdarzeń, 424
 odrysowujące zawartość
 komórek listy, 559
 rysujące, 587
 serializowalne, 91
 użytkownika, 600
 w formacie tekstowym, 75
 znakowe, 306

obrazy
 dostęp do danych, 769
 filtrowanie, 775
 model kolorów, 770
 modyfikacja pikseli, 769
 negatyw, 779
 obrót, 776
 operacje, 768
 próbki piksela, 769
 przejrzystość, 745
 przyrostowe tworzenie, 769
 rozmycie, 780
 sekwencje, 763
 wczytywanie, 759
 wykrywanie krawędzi, 781
 zapisywanie, 759
 zbiór Mandelbrota, 771

obróć, 738, 776

obrys
 czcionek, 743
 figury, 710
 tekstu, 744

- obsługa
 - błędów, 890
 - zdarzeń, 424
 - adnotacje, 438
 - ODBC, 276
 - odcinki, 718
 - odcisk
 - klasy, 96
 - palca, 517
 - odczyt
 - bajtów, 62
 - danych binarnych, 81
 - dużych obiektów, 306
 - plików, 112
 - odszyfrowywanie danych, 541
 - odwołanie transakcji, 331
 - okna dialogowe, 681
 - drukowanie, 791
 - ramki wewnętrzne, 699
 - opcja
 - APPEND, 116
 - ATOMIC_MOVE, 115, 116
 - COPY_ATTRIBUTES, 115, 116
 - CREATE, 116
 - CREATE_NEW, 116
 - DELETE_ON_CLOSE, 116
 - FOLLOW_LINKS, 116
 - NOFOLLOW_LINKS, 116
 - READ, 116
 - REPLACE_EXISTING, 115, 116
 - SPARSE, 116
 - TRUNCATE_EXISTING, 116
 - WRITE, 116
 - operacje
 - końcowe, 28
 - na obrazach, 768
 - na plikach, 116
 - na strumieniach, 18
 - redukcji, 27, 48
 - ORDER BY, 290
 - organizatory komponentów, 678
 - oś czasu, 340
- P**
- pakiet OpenSSL, 527
 - pakiet Swing, 547
 - pakiety, 471, 472
 - panele dzielone, 678
 - JSplitPane, 678
 - panele
 - pulpitu, 687
 - z zakładkami, 681
 - JTabbedPane, 681
 - parametr odbiorczy, 449
 - parser XML, 155, 159
 - DOM, 155, 157
 - DOMTreeModel, 163
 - implementacja, 169
 - SAX, 155, 198
 - TreeModel, 163
 - XML Schema, 176
 - parsery
 - drzewiaste, 155
 - strumieniowe, 155, 198
 - parsowanie daty i czasu, 352
 - pasek postępu, 667
 - nieokreślony, 670
 - ustawianie wartości, 667
 - PCDATA, 168–171, 179
 - piaskownica, 484
 - piksel, 745
 - docelowy, 745
 - PJA, 810
 - PKCS#5, 536
 - plik
 - applet.policy, 534
 - applets.policy, 532
 - client.certs, 534
 - java.policy, 488
 - pliki, 493
 - blokowanie, 133
 - class, 468
 - dzienników, 391
 - informacje, 117
 - JAR, 252, 285, 469
 - graficzne, 758
 - animacje GIF, 765
 - formaty, 758
 - ImageIO, 759
 - interfejs dostawcy, 759
 - JPEG, 759
 - MIME, 759
 - obiekt odczytu, 759
 - obiekty, 759
 - sekwencje obrazów, 763
 - wczytywanie, 758
 - zapisywanie, 758, 759, 763
 - kanał dostępu, 125
 - kopiowanie, 115
 - mapowanie, 124
 - odczyt, 112
 - PNG, 758
 - polityki bezpieczeństwa, 484, 487
 - baza kodu, 490
 - edycja, 494
 - grant, 489
 - implementacja klasy
 - pozwoleń, 496
 - menedżer bezpieczeństwa, 489
 - niezależne od platformy
 - systemowej, 494
 - odczyt, 494
 - operacje sieciowe, 493
 - pozwolenia, 490, 498
 - składnica kluczy, 531
 - system plików, 492
 - testowanie aplikacji, 489
 - właściwości systemowe, 493
 - zapis, 494
 - pomocnicze, 437
 - pozwoleń, 485
 - przenoszenie, 115
 - serializacji obiektów, 95
 - ścieżka dostępu, 110
 - tekstowe, 389
 - tworzenie, 114
 - usuwanie, 115
 - właściwości, 150
 - wzorce filtrowania, 120
 - XML, 150
 - XML Schema, 166
 - właściwości, 394
 - zapis, 112
 - ZIP, 88, 469
 - pobieranie
 - informacji, 254
 - podstrumieni, 25
 - wartości kluczy, 310
 - pochylenie, 738
 - poczta elektroniczna, 231, 270
 - SMTP, 270
 - wysyłanie, 270
 - podgląd wydruku, 794
 - podpis cyfrowy, 516, 526
 - aplety, 516
 - generator liczb losowych, 537
 - MD5, 518
 - MessageDigest, 518
 - SHA1, 517, 518
 - skrót wiadomości, 517
 - weryfikacja, 521
 - podpisywanie
 - certyfiatów, 526
 - kodu, 467, 528
 - jarsigner, 531
 - wiadomości, 520
 - podział, 43
 - pola, 727
 - add, 727
 - exclusiveOr, 727

- intersect, 727
 - operacje, 728
 - subtract, 727
 - ślad pędzla, 728
 - tworzenie, 727
 - poła wejściowe
 - filtry, 639
 - formatowanie, 637
 - kontrola poprawności danych, 638
 - wartości całkowite, 638
 - weryfikatory, 641
 - polecenia
 - przygotowane, 300
 - SQL, 290
 - polecenie telnet, 231
 - policytool, 494
 - polityka bezpieczeństwa, 484, 485
 - pliki, 487
 - połączenia sieciowe
 - bazodanowe, 287
 - JNDI, 337
 - częściowo zamknięte, 244
 - gniazda, 234
 - klient, 233
 - porty, 232
 - serwer, 231, 234
 - strumienie, 235
 - TCP, 235
 - UDP, 235
 - URL, 252
 - wysyłanie danych do formularzy, 262
 - porty, 232
 - porządek alfabetyczny, 378
 - POST, 263
 - potok rysowania, 711
 - potokowe tworzenie grafiki, 710
 - pozwolenia, 483, 485, 498
 - implementacja klasy, 496
 - klasy, 495
 - operacje sieciowe, 493
 - parametry, 490–492
 - pliki polityki, 490
 - stos wywołań metod, 486
 - system plików, 492
 - PRA, 810
 - problem uwierzytelniania, 524
 - procedury ładowania klas
 - implementacja, 473
 - procesor
 - adnotacji, 455
 - XSLT, 222
 - profil ICC, 770
 - program
 - FileReadApplet, 533
 - FormatTest, 645
 - jawah, 862
 - telnet, 231
 - programowanie
 - baz danych, 275
 - transakcji, 332
 - prostokąt, 714, 725
 - ograniczający, 713
 - protokół
 - HTTP, 256
 - Kerberos, 545
 - LDAP, 545
 - SMTP, 270
 - TCP, 235
 - UDP, 235
 - prywatne metody macierzyste, 484
 - przeciąganie zarysu ramki, 700
 - przeciągnij i upuść, 828
 - gesty, 828
 - inicjacja operacji przeciągnięcia, 828
 - kopiowanie, 828
 - przesunięcie, 828
 - tworzenie łącza, 829
 - upuszczenie obiektu, 829
 - przeglądanie zawartości katalogu, 118
 - przeglądarka klas, 618
 - przekazywanie
 - danych, 829
 - referencji obiektów, 827
 - przekierowanie wejścia i wyjścia, 417
 - przekształcenia
 - afiniczne, 740, 775
 - figur, 710
 - układu współrzędnych, 711, 737
 - macierz przekształceń, 740
 - obrót, 738
 - pochylenie, 738
 - przesunięcie, 738
 - skalowanie, 738
 - składanie przekształceń, 738
 - XSL
 - Java, 224
 - style, 221
 - szablon przekształcenia, 221
 - XSLT, 222
 - przenoszenie plików, 115
 - przestrzeń nazw, 195, 471
 - alias, 196
 - definiowanie, 196
 - identyfikatory, 195
 - URI, 195
 - XML, 195
 - przesunięcie, 738
 - przetwarzanie
 - adnotacji, 442, 455
 - dokumentów XML, 155
 - przewijalne zbiory wyników zapytań, 311
 - przezroczystość, 745
 - przycinanie, 710, 743
 - określanie obszaru, 744
 - przyrostowe tworzenie obrazów, 769
 - PSA, 810
 - PUBLIC, 168, 437
 - pula połączeń, 337
 - punkty kontrolne transakcji, 332
- ## R
- ramki wewnętrzne, 687
 - obiekt nasłuchujący weta zmiany, 698
 - odrysowywanie zawartości, 700
 - okna dialogowe, 699
 - przeciąganie zarysu ramki, 700
 - rozmieszczenie, 690
 - zamknięcie, 698
 - zgłaszanie weta zmiany właściwości, 697
 - REAL, 283, 335
 - redukcja, 27, 48
 - referencja
 - bytów, 154
 - obiektu, 827
 - this, 449
 - znaków, 154
 - refleksja, 452
 - REG_BINARY, 903
 - REG_DWORD, 903
 - REG_SZ, 903
 - rejestr systemu Windows, 900
 - edytor rejestru, 900
 - funkcje kontroli typów, 914
 - interfejs dostępu, 902
 - klucze, 901
 - metody macierzyste, 902
 - pobieranie wartości, 903
 - przeglądanie nazw kluczy, 904
 - uchwyt klucza, 904
 - węzły, 901
 - Win32RegKey, 902
 - Win32RegKeyNameEnumeration, 904
 - zapisywanie wartości, 904

- rejestracja klasy sterownika, 286
 - rekordy, 279
 - wstawiania, 314
 - RELATIVE, 178
 - relatywizacja ścieżki, 111
 - REMAINDER, 178
 - REMARKS, 330
 - REQUIRED, 170
 - RGB, 745, 770
 - role, 508
 - rozmieszczenie
 - kaskadowe, 690
 - sąsiadujące, 690
 - rozmycie obrazu, 780
 - rozszerzenia maszyny wirtualnej, 469
 - rozwiązywanie klasy, 468
 - RSA, 521, 543
 - klucze, 543
 - rt.jar, 469, 470
 - RTF, 661
 - rysowanie
 - antialiasing, 753
 - figur, 710
 - komórek, 586
 - potok, 712
 - węzłów, 615
 - rysunki, 709
- S**
- SASL, 545
 - SAX, 155, 198
 - wyszukiwanie elementów, 199
 - schowek, 813, 827
 - Clipboard, 814
 - Copy, 815
 - formaty danych, 818
 - interfejsy, 814
 - klasy, 814
 - przekazywanie danych, 814
 - przekazywanie obiektów Java, 824
 - przekazywanie obrazów, 820
 - przekazywanie tekstu, 815
 - rodzaje danych, 814
 - Transferable, 814, 818
 - sekwencje
 - filtrowanych strumieni, 70
 - obrazów, 763
 - sterujące, 308
 - SELECT, 281
 - FROM, 281
 - LIKE, 282
 - NOT LIKE, 282
 - WHERE, 281
 - serializacja, 91, 102, 107
 - obiektów, 91, 93, 95
 - singletonów, 104
 - wyliczeń, 104
 - serwer, 231, 278
 - aplikacji, 337
 - FTP, 257
 - gniazda, 239
 - HTTP, 239
 - implementacja, 238
 - obsługa klientów, 241
 - połączenia, 231
 - wątki, 242
 - Web, 231, 263
 - wysyłanie danych do formularzy, 262
 - serwlety, 262, 337
 - SGML, 151
 - SHA1, 517
 - short, 335
 - sieć, 231
 - hasła dostępu, 256
 - poczta elektroniczna, 270
 - przesyłanie danych, 239
 - silnik skryptów, 414
 - SINGLE_TREE_SELECTION, 619
 - singleton, 104
 - skalowanie, 738
 - składanie
 - obrazów, 745
 - CLEAR, 746
 - DST, 746
 - DST_ATOP, 747
 - DST_IN, 747
 - DST_OUT, 747
 - DST_OVER, 746
 - piksel docelowy, 745
 - projektowanie reguły, 746
 - reguły Portera-Duffa, 747
 - reguły składania, 745
 - SRC, 746
 - SRC_ATOP, 747
 - SRC_IN, 747
 - SRC_OUT, 747
 - SRC_OVER, 746
 - XOR, 747
 - przekształceń, 738
 - składnia
 - adnotacji, 443
 - wyrażeń regularnych, 137
 - składnica kluczy, 522–524, 531
 - składowe obiektu, 873
 - skrót wiadomości, 517
 - skrypty, 413
 - CGI, 262
 - kompilacja, 420
 - słowo kluczowe transient, 102
 - SMALLINT, 283, 335
 - SMTP, 270
 - sortowanie wierszy, 575
 - splot, 780
 - spójność bazy danych, 331
 - sprawdzanie uprawnień, 483
 - SQL, 275, 278, 289
 - aktualizowalne zbiory wyników zapytań, 313
 - analiza wyjątków, 294
 - ARRAY, 336
 - CREATE TABLE, 283, 290
 - DDL, 291
 - DROP TABLE, 290
 - FROM, 281
 - funkcje, 283
 - INSERT, 283
 - LIKE, 282
 - łączenie tabel, 281
 - metadane, 322
 - modyfikacja danych, 282
 - NOT LIKE, 282
 - polecenia, 293
 - polecenia przygotowane, 300
 - sekwencje sterujące, 308
 - SELECT, 281
 - słowa kluczowe, 281
 - tworzenie tabel, 283
 - typy danych, 283, 335
 - typy wyjątków, 295
 - WHERE, 281, 282
 - wstawianie danych, 283
 - wykonywanie zapytań, 300
 - wynik zapytania, 280
 - wypełnianie bazy danych, 296
 - zapytania, 281
 - zarządzanie połączeniami, 293
 - zbiór wyników, 293, 309
 - SRC, 746
 - SRC_ATOP, 747
 - SRC_IN, 747
 - SRC_OUT, 747
 - SRC_OVER, 746
 - sRGB, 770
 - SSL, 545
 - stała serialVersionUID, 105
 - static, 437
 - stdarg.h, 886
 - sterowniki JDBC, 276, 277
 - stosowanie adnotacji, 437
 - strona WWW, 262

- struktura
 - bufora danych, 131
 - dokumentu XML, 152
 - struktury danych
 - nieskończone, 632
 - strumienie, 17, 235, 264
 - danych typów prostych, 50
 - katalogów, 120
 - łączenie, 25
 - monitorowanie postępu, 673
 - obiektów, 91
 - plików o swobodnym dostępie, 84
 - pobieranie, 25
 - przekształcenia, 26
 - równoległe, 55
 - szyfrujące, 541
 - CipherInputStream, 541
 - tekstowe, 72
 - tworzenie, 20, 673
 - uporządkowane, 56
 - usługi drukowania, 806
 - wejścia i wyjścia, 61, 65
 - wejściowe ZIP, 88
 - strumień
 - DoubleStream, 50
 - IntStream, 50
 - LongStream, 50
 - RandomAccessFile, 84
 - style formatowania dat i czasu, 372
 - SVG, 209, 210
 - Swing
 - drzewa, 598
 - filtr strumieni, 673
 - formatowanie tekstu, 633
 - JDesktopPane, 688
 - JEditorPane, 661
 - JFrame, 688
 - JInternalFrame, 688
 - JList, 548
 - JProgressBar, 667
 - JScrollPane, 548
 - JSplitPane, 678
 - JTabbedPane, 681
 - JTextArea, 661
 - JTextField, 661
 - JTree, 598
 - listy, 547
 - monitory postępu, 670
 - organizatory komponentów, 678
 - panele dzielone, 678
 - panele pulpitu, 687
 - panele z zakładkami, 681
 - przekazywanie danych, 829, 830
 - ramki wewnętrzne, 687
 - rozmieszczenie komponentów, 690
 - tabele, 563
 - tekst, 661
 - wskaźnik postępu, 667
 - swobodny dostęp, 84
 - sygnatury metody, 878
 - symbole formatujące dat i godzin, 355
 - system plików ZIP, 123
 - szyfr Cezara, 476
 - szyfrowanie, 476, 534
 - AES, 535, 536, 542
 - algorytmy, 534, 535
 - Cipher, 534
 - DES, 535, 536
 - dopełnienie ostatniego bloku, 536
 - klucze, 535, 536
 - kluczem publicznym, 542
 - pliki klas, 477
 - RSA, 521, 543
 - strumienie, 541
 - symetryczne, 534
 - tryb pracy algorytmu, 535
- ## Ś
- ścieżka
 - dostępu, 110
 - drzewa, 607
 - śląd pędzla, 711, 728
 - połączenia, 729
 - przerywany wzór, 730
 - szerokość, 728
 - wartość graniczna, 729
 - zakończenia, 728
 - środowisko Cygwin, 863
- ## T
- tabele, 279, 563
 - drukowanie, 567
 - implementacja edytora komórek, 593
 - JTable, 563
 - kolumny, 571
 - model, 563, 568
 - model wyboru, 574
 - obiekt rysujący, 571
 - prezentacja danych, 571
 - przesuwanie kolumny, 566
 - szerokość kolumn, 566
 - tworzenie, 283
 - tworzenie edytorów, 593
 - ukrywanie kolumn, 578
 - widoki, 566
 - wiersze, 571
 - wybór kolumn, 574
 - wybór komórek, 574
 - wybór wierszy, 574
 - wysokość komórek, 573
 - wyświetlanie kolumn, 578
 - zmiana rozmiaru kolumn, 572
 - zmiana rozmiaru wierszy, 573
 - TABLE_CAT, 330
 - TABLE_NAME, 330
 - TABLE_SCHEM, 330
 - TABLE_TYPE, 330
 - tablice, 884, 886
 - C++, 888
 - jarray, 887
 - język C, 886
 - rozmiar, 887
 - TCP, 235
 - tekst, 72, 661
 - hiperłącza, 663
 - wczytywanie, 75
 - zapisywanie, 72
 - testowanie, 437
 - text/plain, 261
 - transakcje, 331
 - aktualizacje wsadowe, 333
 - automatyczne zatwierdzanie, 332
 - odwołanie, 331
 - punkty kontrolne, 332
 - tworzenie, 331
 - try, 294
 - tryb mapowania
 - PRIVATE, 125
 - READ_ONLY, 125
 - READ_WRITE, 125
 - tryb upuszczenia, 837, 843
 - tworzenie
 - certyfikatów, 522
 - dokumentów XML, 207
 - drzewa DOM, 207
 - grafiki, 710
 - katalogów, 114
 - klas pozwoleń, 495
 - obiektów typu Optional, 31
 - plików, 114
 - strumieni, 20
 - zbiorów rekordów, 318
 - typ
 - ENTITY, 171
 - FILE*, 64
 - ID, 171

- typ
 - IDREF, 171
 - IDREFS, 171
 - Optional, 29
 - TYPE_BILINEAR, 776
 - TYPE_BYTE_GRAY, 772
 - TYPE_FORWARD_ONLY, 311, 312
 - TYPE_INT_ARGB, 769
 - TYPE_SCROLL_INSENSITIVE, 312, 314
 - TYPE_SCROLL_SENSITIVE, 312
- typy
 - danych C, 866
 - danych Java, 335, 866
 - danych SQL, 283, 335
 - MIME, 414, 818
 - proste, 50
 - tablic, 887
 - wyjątków SQL, 295
- U**
- UDP, 235
- układ współrzędnych, 737
- Unicode, 79, 359, 392
- uniform resource
 - identifiers, 252
 - locator, 252
 - name, 252
- UPDATE, 290, 301
- uporządkowanie
 - łańcuchów, 379
 - słownikowe, 379
- uprawnienia, 483
- URI, 195, 252, 253, 809
 - absolutny, 253
 - hierarchiczny, 253
 - identyfikatory, 253
 - nieprzenikalny, 253
 - relatywizacja, 254
 - rozwiązywanie, 254
 - specyfikacja, 253
 - względny, 253
- URL, 195, 252, 259, 489
 - nagłówki żądań, 255
 - pobieranie informacji, 254
- URN, 252
- uruchamianie
 - aplikacji pulpitu, 850
 - bazy danych, 285
- usługi
 - drukowania, 802
 - GIF, 802
 - rodzaje dokumentów, 803
 - strumienie, 806
 - źródło danych, 803
- sieciowe, 231
- uwierzytelniania, 502
- usuwanie plików, 115
- UTC, 349
- UTF-16, 79, 392, 868, 869
- UTF-32, 869
- UTF-8, 79, 392, 868
- uwierzytelnianie
 - użytkowników, 502
 - grant, 503
 - JAAS, 502
 - moduł logowania, 503
 - moduły, 503
 - nadzorcy, 503
 - oparte na rolach, 508
 - podmiot, 503
 - pozwolenia, 504
- wiadomości, 524
 - klucze, 525
 - łańcuch zaufania, 525
 - podpis zaufanego pośrednika, 525
 - zaufany pośrednik, 525
- V**
- va_list, 886
- VARCHAR, 283, 335
- vm_args, 895
- W**
- W3C, 199
- waluta, 365, 370
 - formatowanie, 370
 - identyfikatory, 371
- warstwa pośrednia, 278
- warstwy, 703
- wartości opcjonalne, 31
- wartość alfa, 745
- wątki, 242
- wczytywanie
 - obiektów serializowalnych, 91
 - tekstu, 75
- wejsce, 61
- wektor obiektów, 558
- wersje programu, 105
- weryfikacja
 - dokumentu XML, 171
 - kodu maszyny wirtualnej, 478
 - podpisu cyfrowego, 521
 - podpisu plików JAR, 524
- weryfikator, 641
 - kodu, 478, 482
- węzły, 599
 - nadrzędne, 599
 - podrzędne, 599
 - przeglądanie, 613
 - rysowanie, 615
- WHERE, 281, 282
- wiązanie zmiennych, 415
- wielokąty, 713, 718, 725
- wiersze, 571
 - filtrowanie, 576
 - sortowanie, 575
 - wybieranie, 574
- Windows, 899
- właściwości fabryk silników
 - skryptów, 414
- wskazówki operacji graficznych, 710, 753
 - antialiasing, 753
 - KEY_ALPHA_INTERPOLATION, 754
 - KEY_ANTIALIASING, 754
 - KEY_COLOR_RENDERING, 754
 - KEY_DITHERING, 754
 - KEY_FRACTIONAL_METRICS, 754
 - KEY_INTERPOLATION, 754
 - KEY_RENDERING, 754
 - KEY_STROKE_CONTROL, 754
 - KEY_TEXT_ANTIALIASING, 754
- wskaźnik postępu, 667
 - JProgressBar, 667
 - monitory postępu, 670
 - pasek postępu, 667
 - ProgressMonitor, 670
 - strumień wejścia, 673
- współrzędne, 713
 - ekranowe, 738
 - użytkownika, 738
- wybór silnika skryptów, 414
- wygląd drzewa, 602
- wyjątek
 - ArrayIndexOutOfBoundsException, 891
 - ArrayStoreException, 891
 - IllegalArgumentException, 891
 - IllegalStateException, 39, 764
 - InterruptedException, 236
 - InvalidPathException, 110
 - IndexOutOfBoundsException, 764

MissingResourceException, 393
 NullPointerException, 891
 OutOfMemoryError, 891
 ParseException, 366
 PrinterException, 784
 PropertyVetoException,
 695–699, 703
 SAXParseException, 173
 SecurityException, 484, 486
 SocketTimeoutException, 260
 SQLException, 294, 332
 SyncProviderException, 320
 UnknownHostException, 234
 wyjście, 61
 wykonywanie zapytań SQL, 300
 wykrywanie krawędzi, 781
 wynik zapytania, 280
 wypełnianie obszaru, 710–712, 735
 gradient, 736
 kolory, 736
 obrazek wzorca, 736
 prostokąt zakotwiczenia, 736
 wyrażenia regularne, 135
 składnia, 137
 wyrażenia XPath, 190
 wyrzucanie wyjątków, 891
 wysyłanie
 danych do formularzy, 262
 poczty elektronicznej, 270
 wyświetlanie
 nagłówka, 588
 wewnętrznych ramek, 687
 wytnij i wklej, 813
 wywołanie
 s.f().g(), 32
 stream.toArray(), 35
 wywoływania alternatywne, 885
 wywoływanie
 funkcji, 418
 funkcji języka C, 860
 parametry, 863
 metod języka Java, 880, 885
 metod obiektów, 880
 metod skryptów, 418
 metod statycznych, 883
 wzorce
 filtrowania plików, 120
 model-widok-nadzorca, 599

X

X Window, 813
 XML, 150
 atributy, 152, 153
 ATTLIST, 169

CDATA, 154, 170
 deklaracja typu dokumentu, 152
 DOM, 155
 element korzenia, 152
 instrukcje przetwarzania, 154
 JAXP, 155
 komentarze, 154
 kontrola poprawności
 dokumentów, 166
 mieszana zawartość, 153
 parser, 155
 parsowanie dokumentów, 155
 PCDATA, 168
 przestrzeń nazw, 195
 przetwarzanie dokumentów, 155
 referencje znaków, 154
 SAX, 155, 198
 struktura dokumentu, 152
 wartości atrybutów, 152
 wartości domyślne atrybutów,
 170
 wielkość znaków, 151
 XPath, 189
 znaczniki, 151
 XML Schema, 166, 174, 195
 atributy, 176
 definicje elementów, 176
 parsowanie dokumentu XML,
 176
 powtórzenia elementów, 176
 przestrzeń nazw, 174
 typ elementu, 174
 typy proste, 174
 typy złożone, 175
 xmlns, 196
 xmlns:alias, 196
 xsd:attribute, 176
 xsd:boolean, 174
 xsd:choice, 175
 xsd:int, 174
 xsd:schema, 176
 xsd:sequence, 175
 xsd:string, 174
 XSL Schema Definition, 174
 xsl:apply-templates, 221
 xsl:output, 221
 xsl:template, 221
 xsl:value-of, 222
 XSLT, 209, 222

Z

zadania drukowania, 784
 zakładki, 682

zapis
 bajtów, 62
 danych binarnych, 81
 dużych obiektów, 306
 obiektów serializowalnych, 91
 plików, 112
 zapytania SQL, 281
 przygotowane, 300
 zarządzanie
 plikami, 109
 połączeniami, 336
 zasada
 Gödla, 479
 składania obrazów, 710, 711
 zasobnik systemowy, 853
 zasoby, 392
 lokalizacja, 393
 zbiory
 atributów drukowania, 808
 pozwoleń, 484
 rekordów, 311, 314, 318
 aktualizowalne, 313
 buforowane, 319
 CachedRowSet, 319
 modyfikacja, 320
 przewijalne, 311
 sprawdzanie, 320
 znaków, 78, 389
 zbiór Mandelbrota, 771
 zestawy znaków, 389
 ZIP, 469
 zmiana
 rozmiaru kolumn, 572
 rozmiaru wierszy, 573
 zawartości pola tekstowego, 634
 zmienne
 LD_LIBRARY_PATH, 899
 znacznik kolejności bajtów, 391
 znaczniki XML, 151
 znaki, 389
 końca wiersza, 389

Ż

źródło
 danych, 337
 JDBC, 284
 JNDI, 337
 kodu, 484
 certyfikaty, 484
 lokalizacja kodu, 484

Ż

żądania certyfikatu, 527

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

JAVA. DOJRZAŁE ROZWIĄZANIA, PROFESJONALNA JAKOŚĆ APLIKACJI!

Java jest dojrzałym i bezpiecznym językiem programowania, który pozwala na tworzenie kodu działającego niezależnie od platformy. Jest przy tym cały czas konsekwentnie rozwijany przez firmę Oracle. Java w wersji SE 8 to innowacyjne, nowoczesne narzędzie o imponujących możliwościach. Aby je w pełni wykorzystać i tworzyć kod o wysokiej jakości, niezbędne są pogłębione zrozumienie tego języka i gruntowna znajomość jego bibliotek.

Książka ta jest kolejnym, zaktualizowanym i przeorganizowanym wydaniem czołowego podręcznika dla poważnych programistów Javy, którzy chcą skorzystać z nowych możliwości języka. W tym drugim z dwóch tomów książki opisano zagadnienia zaawansowane, takie jak API strumieni, biblioteki do obsługi daty, czasu i kalendarzy, zaawansowane zastosowania biblioteki Swing czy zagadnienia związane z bezpieczeństwem. Przedstawiono również najlepsze praktyki programowania aplikacji. Co ważne, prezentacja zagadnień umożliwia ich łatwe zrozumienie i praktyczne zastosowanie.

NAJWAŻNIEJSZE ZAGADNIENIA UJĘTE W KSIĄŻCE:

- biblioteka strumieni Javy SE 8 oraz strumienie wejścia-wyjścia
- tworzenie aplikacji sieciowych pracujących z użyciem protokołu HTTP
- interfejs JDBC i programowa obsługa baz danych
- interfejs programowy bezpieczeństwa i wykorzystanie algorytmów szyfrowania
- interfejs programowy Java 2D

CAY S. HORSTMANN — jest profesorem informatyki. Wykłada na Uniwersytecie Stanowym w San Jose i współpracuje z uniwersytetami w Szwajcarii i Wietnamie. Otrzymał tytuł Java Champion. Często przemawia podczas konferencji związanych z technikami informatycznymi. W wolnych chwilach dzieli się swoją wiedzą, pisząc książki i artykuły o różnych językach programowania.

Helion	
księgarnia internetowa	
http://helion.pl	
zamówienia telefoniczne	
☎	0 801 339900
☎	0 601 339900
Informatyka w najlepszym wydaniu	

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

Sprawdź najnowsze promocje:
● <http://helion.pl/promocje>
Książki najchętniej czytane:
● <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
● <http://helion.pl/nowosci>

**PRENTICE HALL
PEARSON EDUCATION**

ISBN 978-83-283-3479-3



cena: 149,00 zł

sięgnij po **WIĘCEJ**

KOD KORZYŚCI