



Helion



WYDANIE XIII

JAVA

Techniki zaawansowane

ORACLE

Cay S. Horstmann

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

Tłumaczenie: Piotr Rajca

ISBN: 978-83-289-3004-9

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

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 © 2025.

Pearson uses AI, including to support the creation of content. We claim copyright to the fullest extent permissible by law.

Portions copyright © 1996-2013 Oracle and/or its affiliates. All Rights Reserved.

Microsoft® Windows®, and Microsoft Office® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

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 wydawca 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 wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

helion.pl/user/opinie/jatz13

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 230 98 63

e-mail: helion@helion.pl

WWW: helion.pl (księgarnia internetowa, katalog książek)

Printed in Poland.

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

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

Spis treści

Wstęp	13
Podziękowania	19
1. Strumienie	21
1.1. Od iteracji do operacji na strumieniach	21
1.2. Tworzenie strumieni	24
1.3. Metody filter, map oraz flatMap	30
1.4. Pobieranie podstrumieni i łączenie strumieni	32
1.5. Inne przekształcenia strumieni	34
1.6. Proste operacje redukcji	35
1.7. Typ Optional	37
1.7.1. Pobieranie wartości Optional	37
1.7.2. Korzystanie z wartości Optional	38
1.7.3. Potoki wartości opcjonalnych	39
1.7.4. Jak nie należy używać wartości opcjonalnych	40
1.7.5. Tworzenie obiektów typu Optional	41
1.7.6. Łączenie funkcji zwracających wartości opcjonalne przy użyciu flatMap	42
1.7.7. Przekształcanie wartości opcjonalnej w strumień	43
1.8. Gromadzenie wyników	45
1.9. Gromadzenie wyników w mapach	50
1.10. Grupowanie i podział	53
1.11. Kolektory przetwarzające	55
1.12. Operacje redukcji	60
1.13. Strumienie danych typów prostych	62
1.14. Strumienie równoległe	67
2. Wejście i wyjście	73
2.1. Strumienie wejścia-wyjścia	73
2.1.1. Odczyt i zapis bajtów	74
2.1.2. Zoo pełne strumieni	77
2.1.3. Łączenie filtrów strumieni wejścia-wyjścia	81
2.1.4. Tekstowe operacje wejścia-wyjścia	84
2.1.5. Wczytywanie tekstu	85

2.1.6. Zapisywanie tekstu	86
2.1.7. Zapis obiektów w formacie tekstowym	89
2.1.8. Zbiory znaków	92
2.1.9. Wczytywanie znaków	96
2.2. Odczyt i zapis danych binarnych	97
2.2.1. Interfejsy DataInput i DataOutput	97
2.2.2. Strumienie plików o swobodnym dostępie	100
2.2.3. Archiwa ZIP	104
2.3. Strumienie obiektów i serializacja	107
2.3.1. Zapisywanie i wczytywanie obiektów serializowalnych	107
2.3.2. Format pliku serializacji obiektów	112
2.3.3. Pola pomijane podczas serializacji	119
2.3.4. Metody readObject i writeObject	119
2.3.5. Metody readExternal i writeExternal	121
2.3.6. Metody readResolve i writeReplace	122
2.3.7. Wersje	123
2.3.8. Serializacja w roli klonowania	126
2.3.9. Deserializacja a bezpieczeństwo	128
2.4. Praca z plikami	130
2.4.1. Ścieżki dostępu	130
2.4.2. Odczyt i zapis plików	133
2.4.3. Tworzenie plików i katalogów	135
2.4.4. Kopiowanie, przenoszenie i usuwanie plików	136
2.4.5. Informacje o plikach	138
2.4.6. Przeglądanie zawartości katalogu	140
2.4.7. Stosowanie strumieni katalogów	141
2.4.8. Systemy plików ZIP	145
2.5. Mapowanie plików w pamięci	146
2.5.1. Wydajność plików mapowanych w pamięci	146
2.5.2. Struktura bufora danych	153
2.6. Blokowanie plików	156
2.7. Wyrażenia regularne	160
2.7.1. Składnia wyrażeń regularnych	160
2.7.2. Testowanie dopasowania	165
2.7.3. Znajdowanie wszystkich dopasowań w łańcuchu	166
2.7.4. Grupy	168
2.7.5. Podział w miejscach wystąpienia separatora	170
2.7.6. Zastępowanie dopasowań	171
2.7.7. Flagi	172
3. Język XML	176
3.1. Wprowadzenie do języka XML	176
3.2. Struktura dokumentu XML	179
3.3. Parsowanie dokumentów XML	181
3.4. Kontrola poprawności dokumentów XML	190
3.4.1. Definicje typów dokumentów	191
3.4.2. XML Schema	199
3.4.3. Praktyczny przykład	202
3.5. Wyszukiwanie informacji za pomocą XPath	208

3.6. Przestrzenie nazw	212
3.7. Parsery strumieniowe	215
3.7.1. Wykorzystanie parsera SAX	216
3.7.2. Wykorzystanie parsera StAX	221
3.8. Tworzenie dokumentów XML	224
3.8.1. Dokumenty bez przestrzeni nazw	225
3.8.2. Dokumenty z przestrzenią nazw	225
3.8.3. Zapisywanie dokumentu	226
3.8.4. Zapis dokumentu XML za pomocą parsera StAX	228
3.9. Przekształcenia XSL	234
4. Programowanie aplikacji sieciowych	244
4.1. Połączenia z serwerem	244
4.1.1. Stosowanie programu telnet	244
4.1.2. Nawiązywanie połączenia z serwerem z wykorzystaniem Javy	247
4.1.3. Limity czasu gniazd	248
4.1.4. Adresy internetowe	250
4.2. Implementacja serwerów	251
4.2.1. Gniazda serwera	252
4.2.2. Obsługa wielu klientów	254
4.2.3. Połączenia częściowo zamknięte	257
4.2.4. Przerwywanie działania gniazd sieciowych	258
4.2.5. Stosowanie bezpiecznych gniazd	262
4.3. Połączenia wykorzystujące URL	266
4.3.1. URL i URI	266
4.3.2. Zastosowanie klasy URLConnection do pobierania informacji	268
4.3.3. Wysyłanie danych do formularzy	275
4.4. Klient HTTP	284
4.4.1. Klasa HttpClient	284
4.4.2. Klasa HttpRequest i publikowanie zawartości	285
4.4.3. Interfejs HttpResponse i obsługa zawartości	286
4.4.4. Przetwarzanie asynchroniczne	287
4.5. Prosty serwer HTTP	293
4.5.1. Narzędzie uruchamiane z wiersza poleceń	293
4.5.2. API serwera HTTP	293
4.5.3. Obiekty obsługi	294
4.5.4. Filtry	295
4.6. Wysyłanie poczty elektronicznej	297
5. Programowanie baz danych: JDBC	301
5.1. Architektura JDBC	301
5.1.1. Typy sterowników JDBC	302
5.1.2. Typowe zastosowania JDBC	303
5.2. Język SQL	304
5.3. Instalacja JDBC	310
5.3.1. Adresy URL baz danych	310
5.3.2. Pliki JAR zawierające sterownik	311
5.3.3. Uruchamianie baz danych	311
5.3.4. Nawiązywanie połączenia z bazą danych	312

5.4. Stosowanie poleceń SQL	315
5.4.1. Wykonywanie poleceń SQL	315
5.4.2. Zarządzanie połączeniami, poleceniami i zbiorami wyników	318
5.4.3. Analiza wyjątków SQL	319
5.4.4. Wypełnianie bazy danych	321
5.5. Wykonywanie zapytań	325
5.5.1. Polecenia przygotowane	325
5.5.2. Odczyt i zapis dużych obiektów	331
5.5.3. Sekwencje sterujące	334
5.5.4. Zapytania o wielu zbiorach wyników	335
5.5.5. Pobieranie wartości kluczy wygenerowanych automatycznie	336
5.6. Przewijalne i aktualizowalne zbiory wyników zapytań	337
5.6.1. Przewijalne zbiory wyników	337
5.6.2. Aktualizowalne zbiory rekordów	339
5.7. Zbiory rekordów	344
5.7.1. Tworzenie zbiorów rekordów	344
5.7.2. Buforowane zbiory rekordów	345
5.8. Metadane	348
5.9. Transakcje	356
5.9.1. Programowanie transakcji w JDBC	357
5.9.2. Punkty kontrolne	357
5.9.3. Aktualizacje wsadowe	358
5.9.4. Zaawansowane typy języka SQL	360
5.10. Zaawansowane zarządzanie połączeniami	361
6. API dat i czasu	364
6.1. Oś czasu	364
6.2. Daty lokalne	369
6.3. Modyfikatory dat	374
6.4. Czas lokalny	376
6.5. Czas strefowy	377
6.6. Formatowanie i parsowanie	382
6.7. Współdziałanie ze starym kodem	388
7. Internacjonalizacja	390
7.1. Lokalizatory	391
7.1.1. Dlaczego stosuje się lokalizatory?	391
7.1.2. Określanie lokalizatorów	392
7.1.3. Lokalizator domyślny	395
7.1.4. Nazwa lokalizatora	395
7.2. Formaty liczb	397
7.2.1. Formatowanie wartości liczbowych	398
7.2.2. Klasa DecimalFormat	402
7.2.3. Waluty	404
7.3. Data i czas	405
7.4. Porządek alfabetyczny i normalizacja	409
7.5. Formatowanie komunikatów	415
7.5.1. Formatowanie liczb i dat	415
7.5.2. Formatowanie z wariantami	417

7.6. Granice tekstu	419
7.7. Wczytywanie i wyświetlanie tekstów	420
7.7.1. Pliki tekstowe	420
7.7.2. Znaki końca wiersza	421
7.7.3. Konsola	421
7.7.4. BOM — znacznik kolejności bajtów UTF-8	422
7.7.5. Kodowanie plików źródłowych	423
7.8. Komplet zasobów	423
7.8.1. Wyszukiwanie kompletów zasobów	424
7.8.2. Pliki właściwości	425
7.8.3. Klasy kompletów zasobów	426
7.9. Kompletny przykład	428
8. Skrypty, kompilacja i adnotacje	433
8.1. Interfejs kompilatora	433
8.1.1. Wywoływanie kompilatora	433
8.1.2. Uruchamianie zadania kompilacji	434
8.1.3. Przechwytywanie informacji diagnostycznych	435
8.1.4. Wczytywanie plików źródłowych z pamięci	435
8.1.5. Zapis kodów bajtowych w pamięci	436
8.1.6. Przykład: dynamiczne tworzenie kodu w języku Java	437
8.2. Skrypty na platformie Java	442
8.2.1. Wybór silnika skryptowego	443
8.2.2. Wykonywanie skryptów i wiązania zmiennych	444
8.2.3. Przekierowanie wejścia i wyjścia	446
8.2.4. Wywoływanie funkcji i metod skryptów	447
8.2.5. Kompilacja skryptu	449
8.2.6. Przykład: arkusze skryptowe	450
9. Bezpieczeństwo	452
9.1. Ładowanie klas	453
9.1.1. Proces wczytywania plików klas	453
9.1.2. Hierarchia klas ładowania	454
9.1.3. Zastosowanie procedur ładujących w roli przestrzeni nazw	456
9.1.4. Implementacja własnej procedury ładującej	457
9.1.5. Weryfikacja kodu maszyny wirtualnej	462
9.2. Uwierzytelnianie użytkowników	466
9.2.1. Framework JAAS	466
9.2.2. Moduły JAAS	469
9.3. Podpis cyfrowy	477
9.3.1. Skróty wiadomości	477
9.3.2. Podpisywanie wiadomości	480
9.3.3. Weryfikacja podpisu	483
9.3.4. Problem uwierzytelniania	485
9.3.5. Podpisywanie certyfikatów	487
9.3.6. Żądania certyfikatu	488
9.3.7. Podpisywanie kodu	490
9.3.8. Skróty haseł	491

9.4. Szyfrowanie	492
9.4.1. Szyfrowanie symetryczne	492
9.4.2. Generowanie klucza	494
9.4.3. Strumień szyfrujący	499
9.4.4. Szyfrowanie kluczem publicznym	500

10. Graficzne interfejsy użytkownika 504

10.1. Historia zestawów narzędzi do tworzenia interfejsów użytkownika	504
10.2. Wyświetlanie ramki	506
10.2.1. Tworzenie ramki	506
10.2.2. Właściwości ramki	508
10.3. Wyświetlanie informacji w komponencie	512
10.3.1. Figury 2D	516
10.3.2. Kolory	523
10.3.3. Czcionki	524
10.3.4. Wyświetlanie obrazów	530
10.4. Obsługa zdarzeń	533
10.4.1. Podstawowe koncepcje obsługi zdarzeń	533
10.4.2. Przykład: obsługa kliknięcia przycisku	534
10.4.3. Zwiąże definiowanie procedur nasłuchowych	538
10.4.4. Klasy adaptacyjne	539
10.4.5. Akcje	541
10.4.6. Zdarzenia generowane przez mysz	547
10.4.7. Hierarchia zdarzeń w bibliotece AWT	552
10.5. API Preferences	554

11. Komponenty Swing interfejsu użytkownika 562

11.1. Swing i wzorzec model-widok-kontroler	562
11.2. Wprowadzenie do zarządzania rozkładem	566
11.2.1. Zarządcy układu	566
11.2.2. Rozkład brzegowy	568
11.2.3. Rozkład siatkowy	570
11.3. Wprowadzanie tekstu	571
11.3.1. Pola tekstowe	572
11.3.2. Etykiety komponentów	574
11.3.3. Pola haseł	575
11.3.4. Obszary tekstowe	576
11.3.5. Panele przewijane	577
11.4. Komponenty umożliwiające wybór opcji	579
11.4.1. Pola wyboru	579
11.4.2. Przełączniki	582
11.4.3. Obramowanie	585
11.4.4. Listy rozwijane	588
11.4.5. Suwaki	591
11.5. Menu	597
11.5.1. Tworzenie menu	597
11.5.2. Ikony w elementach menu	600
11.5.3. Pola wyboru i przełączniki jako elementy menu	601
11.5.4. Menu podręczne	602
11.5.5. Mnemoniki i akcelatory	604

11.5.6. Aktywowanie i dezaktywowanie elementów menu	606
11.5.7. Paski narzędzi	610
11.5.8. Dymki	612
11.6. Rozkład siatkowy	613
11.6.1. Rozkład GridBagLayout	614
11.6.2. Parametry gridx, gridy, gridwidth i gridheight	616
11.6.3. Pola weight	616
11.6.4. Parametry fill i anchor	617
11.6.5. Dopełnienie	617
11.6.6. Inny sposób ustawiania wartości parametrów gridx, gridy, gridwidth i gridheight	617
11.6.7. Układ GridBagLayout — receptura	618
11.6.8. Klasa pomocnicza ułatwiająca pracę z ograniczeniami GridBagLayout	618
11.7. Niestandardowi zarządcy rozkładu	623
11.8. Okna dialogowe	627
11.8.1. Okna dialogowe opcji	628
11.8.2. Tworzenie okien dialogowych	633
11.8.3. Wymiana danych	636
11.8.4. Okna dialogowe wyboru plików	642
12. Zaawansowane możliwości pakietu Swing i grafiki	651
12.1. Tabele	651
12.1.1. Najprostsze tabele	652
12.1.2. Modele tabel	655
12.2. Wiersze i kolumny	659
12.2.1. Klasy kolumn	659
12.2.2. Dostęp do kolumn tabeli	660
12.2.3. Zmiana rozmiaru kolumn	660
12.2.4. Zmiana rozmiaru wierszy	662
12.2.5. Wybór wierszy, kolumn i komórek	662
12.2.6. Sortowanie wierszy	663
12.2.7. Filtrowanie wierszy	664
12.2.8. Ukrywanie i wyświetlanie kolumn	666
12.3. Rysowanie i edytowanie komórek	674
12.3.1. Rysowanie komórek	674
12.3.2. Wyświetlanie nagłówka	676
12.3.3. Edytowanie komórek	676
12.3.4. Tworzenie własnych edytorów	681
12.4. Drzewa	685
12.4.1. Proste drzewa	686
12.4.2. Modyfikowanie drzew i ścieżek drzew	694
12.4.3. Przeglądanie węzłów	701
12.4.4. Rysowanie węzłów	703
12.4.5. Nasłuchiwanie zdarzeń w drzewach	706
12.4.6. Własne modele drzew	712
12.5. Zaawansowane możliwości biblioteki AWT	720
12.5.1. Potokowe tworzenie grafiki	720
12.5.2. Figury	723
12.5.3. Tworzenie kształtów	724
12.5.4. Pola	737

12.5.5. Ślad pędzla	738
12.5.6. Wypełnienia	745
12.5.7. Przekształcenia układu współrzędnych	747
12.5.8. Przycinanie	752
12.5.9. Przezroczystość i składanie obrazów	755
12.6. Grafika rastrowa	763
12.6.1. Odczyt i zapis plików graficznych	763
12.6.2. Wykorzystanie obiektów zapisu i odczytu plików graficznych	764
12.6.3. Odczyt i zapis plików zawierających sekwencje obrazów	768
12.6.4. Operacje na obrazach	773
12.6.5. Filtrowanie obrazów	780
12.7. Drukowanie	788
12.7.1. Drukowanie grafiki	788
12.7.2. Drukowanie wielu stron	797
12.7.3. Usługi drukowania	805
12.7.4. Usługi drukowania za pośrednictwem strumieni	808
12.7.5. Atrybuty drukowania	811

13. Metody macierzyste 818

13.1. Wywołania funkcji języka C z programów w języku Java	819
13.2. Numeryczne parametry metod i wartości zwracane	825
13.3. Łańcuchy znaków jako parametry	827
13.4. Dostęp do składowych obiektu	832
13.4.1. Dostęp do pól instancji	833
13.4.2. Dostęp do pól statycznych	836
13.5. Sygnatury	837
13.6. Wywoływanie metod języka Java	839
13.6.1. Wywoływanie metod obiektów	839
13.6.2. Wywoływanie metod statycznych	842
13.6.3. Konstruktory	843
13.6.4. Alternatywne sposoby wywoływania metod	844
13.7. Dostęp do elementów tablic	846
13.8. Obsługa błędów	849
13.9. Interfejs programowy wywołań języka Java	854
13.10. Kompletny przykład: dostęp do rejestru systemu Windows	859
13.10.1. Rejestr systemu Windows	859
13.10.2. Interfejs dostępu do rejestru na platformie Java	860
13.10.3. Implementacja dostępu do rejestru za pomocą metod macierzystych	861
13.11. Funkcje obce: spojrzenie w przyszłość	873

Skorowidz 877

2

Wejście i wyjście

W tym rozdziale omówimy interfejsy programowe związane z obsługą wejścia i wyjścia programów. Przedstawimy sposoby dostępu do plików i katalogów oraz sposoby zapisywania do i wczytywania informacji z plików w formacie tekstowym i binarnym. W rozdziale przedstawiony jest również mechanizm serializacji obiektów, który umożliwia przechowywanie obiektów z taką łatwością, z jaką przechowujesz tekst i dane numeryczne. Rozdział zakończymy przedstawieniem problematyki wyrażeń regularnych, mimo że nie jest ona bezpośrednio związana z zagadnieniami wejścia-wyjścia. Nie potrafiliśmy jednak znaleźć dla niej lepszego miejsca w książce. W naszym wyborze nie byliśmy zresztą osamotnieni, ponieważ zespół Javy dołączył specyfikację interfejsów programowych związanych z przetwarzaniem wyrażeń regularnych do specyfikacji „nowej wersji” obsługi wejścia i wyjścia.

2.1. Strumienie wejścia-wyjścia

W języku Java obiekt, z którego możemy odczytać sekwencję bajtów, nazywamy *strumieniem wejścia*. Obiekt, do którego możemy zapisać sekwencję bajtów, nazywamy *strumieniem wyjścia*. Źródłem bądź celem tych sekwencji bajtów mogą być, i często właśnie są, pliki, ale także i połączenia sieciowe, a nawet bloki pamięci. Klasy abstrakcyjne `InputStream` i `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` ma metodę abstrakcyjną:

```
abstract int read()
```

Metoda ta wczytuje jeden bajt i zwraca jego wartość lub `-1`, jeżeli natrafi na koniec źródła danych. Projektanci konkretnych klas strumieni wejścia przesłaniają tę metodę, dostarczając w ten sposób użytecznej funkcjonalności. Dla przykładu, w klasie `FileInputStream` metoda `read` czyta jeden bajt z pliku. `System.in` to predefiniowany obiekt klasy pochodnej od `InputStream`, pozwalający pobierać informacje ze „standardowego wejścia”, czyli konsoli lub przekierowanego pliku.

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

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

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

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

Analogicznie, klasa `OutputStream` definiuje metodę abstrakcyjną

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

która wysyła jeden bajt do aktualnego wyjścia.

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

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

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

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

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

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

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

Gdy skończymy odczytywać albo zapisywać dane do strumienia wejścia-wyjścia, zamykamy go, wywołując metodę `close`. Metoda ta uwalnia zasoby systemu operacyjnego, do tej pory udostępnione wątkowi. Jeżeli aplikacja otworzy zbyt wiele strumieni wejścia-wyjścia, nie zamykając ich, zasoby systemu mogą zostać naruszone. Co więcej, zamknięcie strumienia wyjścia powoduje *opróżnienie* bufora używanego przez ten strumień — wszystkie bajty, przechowywane tymczasowo w buforze, aby mogły zostać zapisane w jednym większym pakiecie, zostaną natychmiast wysłane. Jeżeli nie zamkniemy strumienia, ostatni pakiet bajtów może nigdy nie dotrzeć do odbiorcy. Bufor możemy również opróżnić własnoręcznie, przy użyciu metody `flush`.

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

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

java.io.InputStream 1.0

- **abstract int read()**
pobiera jeden bajt i zwraca jego wartość. Metoda `read` zwraca `-1`, gdy natrafi na koniec strumienia wejścia.
- **int read(byte[] b)**
wczytuje dane do tablicy i zwraca liczbę wczytanych bajtów, a jeżeli natrafi na koniec strumienia wejścia, zwraca `-1`. Metoda `read` czyta co najwyżej `b.length` bajtów.
- **int read(byte[] b, int off, int len)**
- **int readNBytes(byte[] b, int off, int len) 9**
wczytuje bez blokownia (`read`) do tablicy co najwyżej `len` bajtów, o ile są one dostępne, ewentualnie blokuje operację aż do momentu, kiedy wartości staną się dostępne (`readNBytes`). Wartości są umieszczane w tablicy `b`, zaczynając do jej elementu o indeksie `off`. Zwraca faktyczną liczbę wczytanych bajtów, a jeżeli natrafi na koniec strumienia wejścia, zwraca `-1`.

- `byte[] readAllBytes()` **9**
zwraca tablicę zawierającą wszystkie bajty, które można odczytać z tego strumienia.
- `long transferTo(OutputStream out)` **9**
przekazuje wszystkie bajty z tego strumienia wejściowego do podanego strumienia wyjściowego i zwraca liczbę przekazanych bajtów. Żaden ze strumieni nie jest zamykany.
- `long skip(long n)`
ignoruje `n` bajtów w strumieniu wejściowym, zwraca faktyczną liczbę zignorowanych bajtów (która może być mniejsza niż `n`).
- `long skipNBytes(long n)` **12**
ignoruje `n` bajtów w strumieniu wejściowym, zwraca faktyczną liczbę zignorowanych bajtów (która może być mniejsza od `n`, jeśli przed zadaną liczbą bajtów został osiągnięty koniec strumienia).
- `int available()`
zwraca liczbę bajtów dostępnych bez konieczności zablokowania wątku (pamiętajmy, że zablokowanie oznacza, że wykonanie aktualnego wątku zostaje wstrzymane).
- `void close()`
zamyka strumień wejścia.
- `void mark(int readLimit)`
ustawia znacznik na aktualnej pozycji strumienia wejścia (nie wszystkie strumienie obsługują tę możliwość). Jeżeli ze strumienia zostało pobranych więcej niż `readLimit` bajtów, strumień ma prawo usunąć znacznik.
- `void reset()`
wraca do ostatniego znacznika. Późniejsze wywołania `read` będą powtórnie czytać pobrane już bajty. Jeżeli znacznik nie istnieje, strumień wejścia nie zostanie zresetowany.
- `boolean markSupported()`
zwraca `true`, jeżeli strumień wejścia obsługuje znaczniki.
- `static InputStream nullInputStream()` **11**
zwraca strumień wejściowy, który nie zawiera żadnych bajtów.

API | java.io.OutputStream 1.0

- `abstract void write(int n)`
zapisuje jeden bajt.

- `void write(byte[] b)`
- `void write(byte[] b, int off, int len)`
zapisują wszystkie bajty tablicy `b` lub pewien ich zakres.
- `void close()`
opróżnia i zamyka strumień wyjścia.
- `void flush()`
opróżnia strumień wyjścia, czyli wysyła do odbiorcy wszystkie dane znajdujące się w buforze.
- `static OutputStream nullOutputStream()` 11
zwraca strumień wyjściowy, który nie zawiera żadnych bajtów.

2.1.2. Zoo pełne strumieni

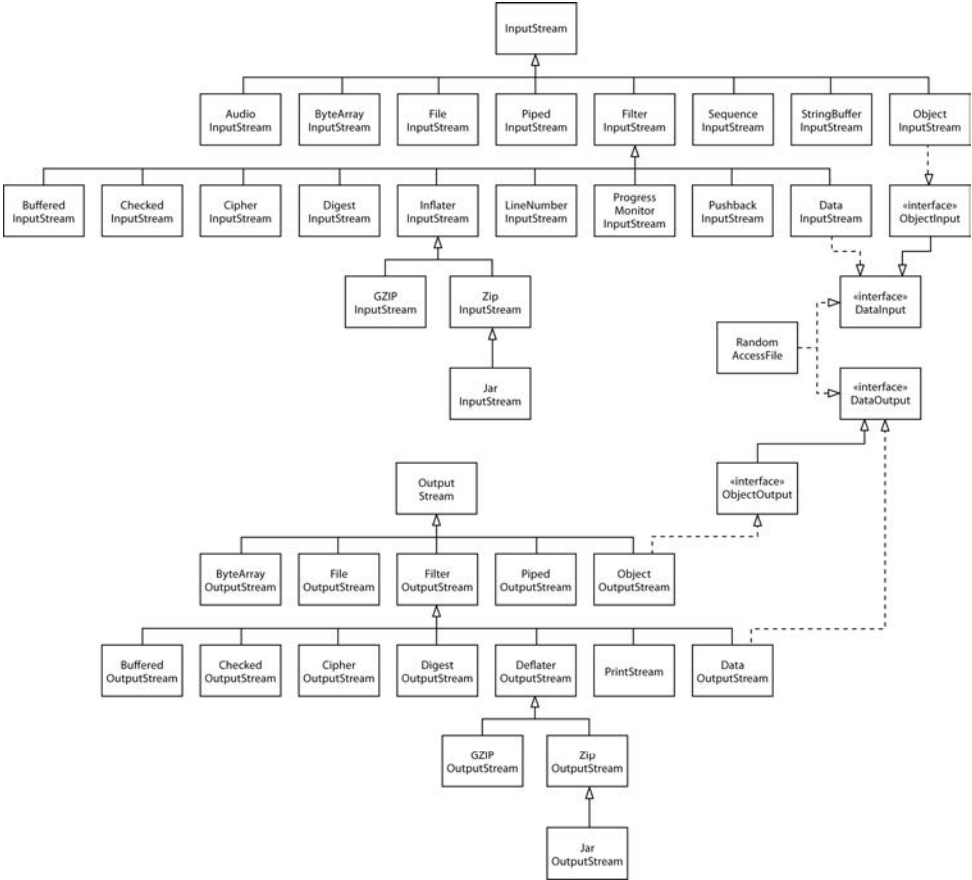
W przeciwieństwie do języka C, który w zupełności zadowala się jednym typem `FILE*`, Java ma prawdziwą menażerię ponad 60 (!) różnych klas i interfejsów związanych ze strumieniami wejścia i wyjścia.

Podzielmy gatunki należące do zoo strumieni zależnie od ich przeznaczenia. Istnieją osobne hierarchie klas przetwarzających bajty i znaki. Jak już o tym wspomnieliśmy, klasy `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 klas pochodnych. Na przykład `DataInputStream` i `DataOutputStream` pozwalają wczytywać i zapisywać wszystkie podstawowe typy Javy w postaci binarnej. I w końcu istnieje także wiele klas strumieni służących do wyspecjalizowanych zadań, na przykład `ZipInputStream` i `ZipOutputStream`, pozwalające odczytywać i zapisywać dane w plikach skompresowanych w formacie ZIP.

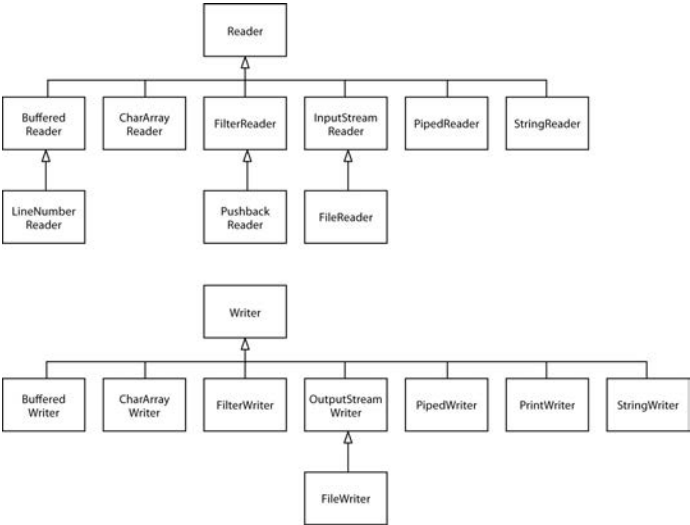
Z drugiej strony, o czym już wspominaliśmy, do obsługi tekstu Unicode używamy klas pochodzących od klas abstrakcyjnych `Reader` i `Writer` (patrz rysunek 2.2) Podstawowe metody klas `Reader` i `Writer` są podobne do tych należących do `InputStream` i `OutputStream`.

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

Metoda `read` zwraca albo kod znaku UTF-16 (jako liczbę z przedziału od 0 do 65535), albo `-1`, jeżeli natrafi na koniec pliku. Metoda `wri` te jest wywoływana dla podanego kodu znaku Unicode (więcej informacji na temat kodów Unicode znajdziesz w rozdziale 3. książki *Java. Podstawy*).



Rysunek 2.1. Hierarchia strumieni wejścia i wyjścia



Rysunek 2.2. Hierarchia klas Reader i Writer

Dostępne są również cztery dodatkowe interfejsy: `Closeable`, `Flushable`, `Readable` i `Appendable` (patrz rysunek 2.3). Pierwsze dwa z nich są wyjątkowo proste i zawierają odpowiednio metody:

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

i

```
void flush()
```

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



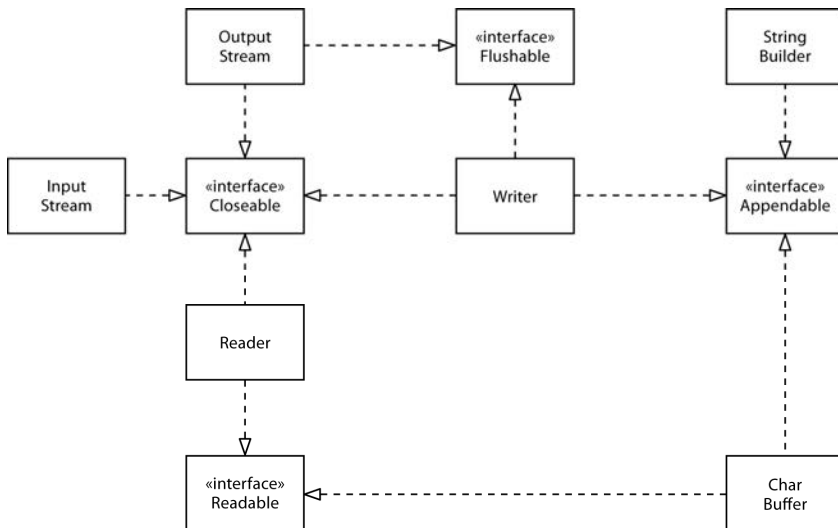
Interfejs `java.io.Closeable` stanowi rozszerzenie interfejsu `java.lang.AutoCloseable`. Dzięki temu dla każdego obiektu implementującego interfejs `Closeable` możemy użyć wersji instrukcji try zarządzającej zasobami. Ale po co nam dwa interfejsy? Metoda `close` interfejsu `Closeable` może zgłaszać jedynie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

Klasy `OutputStream` i `Writer` implementują interfejs `Flushable`.

Interfejs `Readable` ma tylko jedną metodę

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

Klasa `CharBuffer` ma metody do sekwencyjnego oraz swobodnego odczytu i zapisu. Reprezentuje ona bufor w pamięci lub mapę pliku w pamięci. (Patrz punkt 2.5.2).



Rysunek 2.3. Interfejsy `Closeable`, `Flushable`, `Readable` i `Appendable`

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

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

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

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

`java.io.Closeable` 5.0

- `void close()`
zamyka obiekt implementujący interfejs `Closeable`. Może zgłaszać wyjątek `IOException`.

`java.io.Flushable` 5.0

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

`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.

`java.lang.Appendable` 5.0

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

`java.lang.CharSequence` 1.4

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

2.1.3. Łączenie filtrów strumieni wejścia-wyjścia

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

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

spróbuje odszukać plik o nazwie *employee.dat*.



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



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

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

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

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

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

Ale tak jak `FileInputStream` nie posiada metod czytających typy liczbowe, tak `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`.

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

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

Różne funkcjonalności możemy dodawać poprzez zagnieżdżanie filtrów. Na przykład — domyślnie strumień 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, monsturalnej sekwencji konstruktorów:

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

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

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

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

Teraz możemy odczytać wartość następnego bajta:

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

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

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



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

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

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

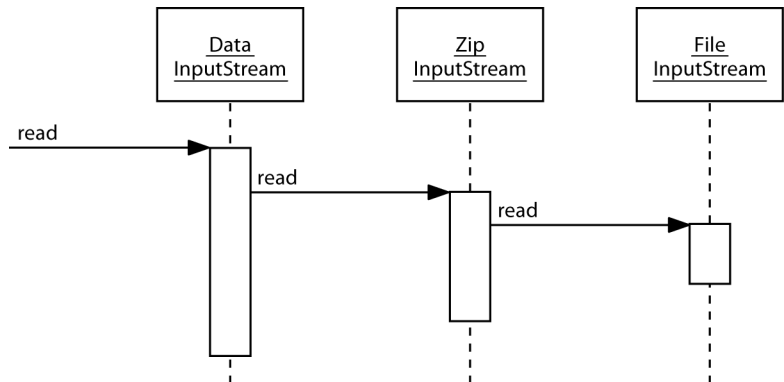
```
new FileInputStream("employee.dat"));
var din = new DataInputStream(pbin);
```

Oczywiście w bibliotekach wejścia-wyjścia innych języków programowania takie udogodnienia jak buforowanie i kontrolowanie kolejnych bajtów są wykonywane automatycznie, więc konieczność tworzenia ich kombinacji w języku Java wydaje się niepotrzebnym zawracaniem głowy. Jednak możliwość łączenia klas filtrów i tworzenia w ten sposób naprawdę użytecznych sekwencji strumieni wejścia-wyjścia daje nam niespotykaną elastyczność. Na przykład, korzystając z poniższej sekwencji strumieni, możemy wczytywać liczby ze skompresowanego pliku ZIP (patrz rysunek 2.4).

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

Rysunek 2.4.

Sekwencja
filtrowanych strumieni



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

API java.io.FileInputStream 1.0

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

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

API java.io.FileOutputStream 1.0

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

tworzy nowy strumień wyjściowy pliku określonego za pomocą łańcucha name lub obiektu file (klasa File zostanie omówiona pod koniec tego rozdziału).

Jeżeli parametr `append` ma wartość `true`, dane dołączane są na końcu pliku, a istniejący plik o tej samej nazwie nie zostanie skasowany. W przeciwnym razie istniejący plik o tej samej nazwie zostanie skasowany.

API **java.io.BufferedInputStream 1.0**■ **BufferedInputStream(InputStream in)**

tworzy nowy buforowany strumień wejściowy typu `BufferedInputStream`. Wejściowy strumień buforowany wczytuje znaki ze strumienia danych, nie wymuszając za każdym razem dostępu do urządzenia. Gdy bufor zostanie opróżniony, system prześle do niego nowy blok danych.

API **java.io.BufferedOutputStream 1.0**■ **BufferedOutputStream(OutputStream out)**

tworzy nowy buforowany strumień wyjściowy typu `BufferedOutputStream`. Strumień umieszcza w buforze znaki, które powinny zostać zapisane, nie wymuszając za każdym razem dostępu do urządzenia. Gdy bufor zapełni się lub gdy strumień zostanie opróżniony, dane są przesyłane odbiorcy.

API **java.io.PushbackInputStream 1.0**■ **PushbackInputStream(InputStream in)**■ **PushbackInputStream(InputStream in, int size)**

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

■ **void unread(int b)**

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

2.1.4. Tekstowe operacje wejścia-wyjścia

Zapisując dane, możemy wybierać pomiędzy formatem binarnym i tekstowym. Dla przykładu: jeżeli liczba całkowita 1234 zostanie zapisana w postaci binarnej, w pliku pojawi się sekwencja bajtów 00 00 04 D2 (w notacji szesnastkowej). W formacie tekstowym liczba ta zostanie zapisana jako łańcuch "1234". Mimo iż zapis danych w postaci binarnej jest szybki i efektywny, to uzyskany wynik jest kompletnie nieczytelny dla ludzi. W poniższym podrozdziale skoncentrujemy się na *tekstowym* wejściu-wyjściu, a odczyt i zapis danych binarnych omówimy w podrozdziale 2.2.

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

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

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

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

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

Więcej informacji na temat kodowania znaków znajdziesz w punkcie 2.1.8.

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

2.1.5. Wczytywanie tekstu

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

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

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

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

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

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

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



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

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

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

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

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

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

Kolejny token można pobrać, wywołując metodę `next`:

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

Można także pobrać strumień zawierający wszystkie tokeny:

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



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

2.1.6. Zapisywanie tekstu

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

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



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



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

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

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

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

Spójrzmy na poniższy kod:

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

Rezultatem jego wykonania będzie wysłanie napisu

```
Harry Hacker 75000.0
```

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

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

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

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

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



Weterani Javy prawdopodobnie zastanawiają się, co się stało z klasą `PrintStream` i obiektem `System.out`. W języku Java 1.0 klasa `PrintStream` obsługiwała znaki Unicode do znaków ASCII, po prostu opuszczając górny bajt (wówczas Unicode był jeszcze kodem 16-bitowym). Takie rozwiązanie nie pozwalało na przenoszenie kodu na inne platformy i w języku Java 1.1 zostało zastąpione przez koncepcję obiektów odczytu i zapisu. Ze względu na konieczność zachowania zgodności z istniejącym kodem `System.in`, `System.out` i `System.err` wciąż są strumieniami wejścia-wyjścia, nie obiektami odczytu i zapisu. Ale obecna klasa `PrintStream` konwertuje znaki Unicode na schemat kodowania lokalnego systemu w ten sam sposób, co klasa `PrintWriter`. Gdy używamy metod `print` i `println`, obiekty `PrintStream` działają tak samo jak obiekty `PrintWriter`, ale w przeciwieństwie do `PrintWriter` pozwalają wysyłać bajty za pomocą metod `write(int)` i `write(byte[])`.

**java.io.PrintWriter 1.1**

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

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

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

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

- `void print(Object obj)`

zapisuje łańcuch zwracany przez metodę `toString` danego obiektu.

- `void print(String s)`

zapisuje łańcuch Unicode.

- `void println(String s)`

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

- `void print(char[] s)`

zapisuje tablicę znaków Unicode.

- `void print(char c)`

zapisuje znak Unicode.

- `void print(int i)`

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

zapisuje podaną wartość w formacie tekstowym.

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

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

- `boolean checkError()`

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

2.1.7. Zapis obiektów w formacie tekstowym

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

Oto przykładowy zbiór danych obiektów:

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

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

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

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

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

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

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

Kompletny program został przedstawiony na listingu 2.1. Metoda statyczna

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

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

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

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

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

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

Listing 2.1. `textfile/TextFileTest.java`

```
package textfile;

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

/**
 * @version 1.16 2023-08-16
 * @author Cay Horstmann
 */
public class TextFileTest
{
    public static void main(String[] args) throws IOException
    {
        var staff = new Employee[3];

        staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
```

```
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);

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

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

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

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

    for (Employee e : employees)
        writeEmployee(out, e);
}

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

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

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

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

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

2.1.8. Zbiory znaków

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

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

Tabela 2.1. Kodowanie UTF-8

Zakres znaku	Sposób kodowania
0 ... 7F	0a ₆ a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
80 ... 7FF	110a ₁₀ a ₉ a ₈ a ₇ a ₆ 10a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
800 ... FFFF	1110a ₁₅ a ₁₄ a ₁₃ a ₁₂ 10a ₁₁ a ₁₀ a ₉ a ₈ a ₇ a ₆ 10a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
10000 ... 10FFFF	11110a ₂₀ a ₁₉ a ₁₈ 10a ₁₇ a ₁₆ a ₁₅ a ₁₄ a ₁₃ a ₁₂ 10a ₁₁ a ₁₀ a ₉ a ₈ a ₇ a ₆ 10a ₅ a ₄ a ₃ a ₂ a ₁ a ₀

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

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

Tabela 2.2. Kodowanie UTF-16

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



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

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

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

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

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

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

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

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

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

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

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

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



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



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

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

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

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

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

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

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



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

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



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

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

```
Chcp 65001
```

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

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



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

```
java -XshowSettings:properties
```

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

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



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

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

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

2.1.9. Wczytywanie znaków

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

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

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

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

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

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

```
isUpperCase  
isLowerCase  
isDigit  
isSpaceChar  
isEmoji
```

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

```
isJavaIdentifierStart  
isJavaIdentifierPart  
isWhiteSpace
```

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

2.2. Odczyt i zapis danych binarnych

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

2.2.1. Interfejsy `DataInput` i `DataOutput`

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

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

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

Metoda `writeUTF` zapisuje łańcuchy, używając zmodyfikowanej wersji 8-bitowego kodu UTF (ang. *Unicode Text Format*). Zamiast po prostu zastosować od razu standardowe kodowanie UTF-8 (przedstawione w tabeli 2.1), znaki łańcucha są najpierw reprezentowane w kodzie UTF-16 (patrz tabela 2.2), a dopiero potem przekodowywane na UTF-8. Wynik takiego kodowania różni się dla znaków o kodach większych od `0xFFFF`. Kodowanie takie stosuje się dla zachowania zgodności z maszynami wirtualnymi powstałymi, gdy Unicode zadowalał się tylko 16 bitami.



Zależnie od platformy użytkownika, liczby całkowite i zmiennoprzecinkowe mogą być przechowywane w pamięci na dwa różne sposoby. Załóżmy, że pracujesz z czterobajtową wartością, taką jak `int`, na przykład 1234, czyli 4D2 w zapisie szesnastkowym ($1234 = 4 \times 256 + 13 \times 16 + 2$). 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 firmy Intel. Może to powodować problemy z przenoszeniem nawet najprostszych plików danych pomiędzy różnymi platformami, gdyż programy w C lub C++ zapisują dane w sposób typowy dla danego procesora. W języku Java zawsze stosowany jest pierwszy sposób, niezależnie od procesora. Dzięki temu pliki danych programów w języku Java są niezależne od platformy.

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

Aby odczytać dane, korzystamy z poniższych metod interfejsu `DataInput`:

<code>readInt</code>	<code>readShort</code>
<code>readLong</code>	<code>readFloat</code>
<code>readDouble</code>	<code>readChar</code>
<code>readBoolean</code>	<code>readUTF</code>

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

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

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

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

API | `java.io.DataInput` 1.0

- `boolean readBoolean()`
- `byte readByte()`
- `char readChar()`

- `double readDouble()`
- `float readFloat()`
- `int readInt()`
- `long readLong()`
- `short readShort()`
wczytuje wartość określonego typu.
- `void readFully(byte[] b)`
wczytuje bajty do tablicy `b`, blokując wątek, dopóki wszystkie bajty nie zostaną wczytane.
- `void readFully(byte[] b, int off, int len)`
wczytuje bajty do tablicy `b`, blokując wątek, dopóki wszystkie bajty nie zostaną wczytane.
- `String readUTF()`
wczytuje łańcuch znaków zapisanych w zmodyfikowanym formacie UTF-8.
- `int skipBytes(int n)`
ignoruje `n` bajtów, blokując wątek, dopóki wszystkie bajty nie zostaną zignorowane.

java.io.DataOutput 1.0

- `void writeBoolean(boolean b)`
- `void writeByte(int b)`
- `void writeChar(char c)`
- `void writeDouble(double d)`
- `void writeFloat(float f)`
- `void writeInt(int i)`
- `void writeLong(long l)`
- `void writeShort(short s)`
zapisują wartość określonego typu.
- `void writeChars(String s)`
zapisuje wszystkie znaki podanego łańcucha.
- `void writeUTF(String s)`
zapisuje łańcuch znaków w zmodyfikowanym formacie UTF-8.

2.2.2. Strumienie plików o swobodnym dostępie

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Odczyt rekordu jest równie prosty.

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

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

```
package randomAccess;

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

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

    public static void main(String[] args) throws IOException
    {
        var staff = new Employee[3];

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

        try (var out = new DataOutputStream(new FileOutputStream("employee.dat")))
        {
            // Zapisuje rekordy wszystkich pracowników w pliku employee.dat
            for (Employee e : staff)
                writeData(out, e);
        }

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

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

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

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

```

        System.out.println(e);
    }
}

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

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

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

```

java.io.RandomAccessFile 1.0

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

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

- long getFilePointer()

zwraca aktualne położenie wskaźnika pliku.

- void seek(long pos)

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

■ `long length()`

zwraca długość pliku w bajtach.

2.2.3. Archiwa ZIP

Pliki ZIP to archiwa, w których można przechowywać jeden lub więcej plików w postaci (zazwyczaj) skompresowanej. Każdy plik ZIP posiada nagłówek zawierający informacje, takie jak nazwa pliku i użyta metoda kompresji. W języku Java, aby czytać z pliku ZIP, korzystamy z klasy `ZipInputStream`. Odczyt dotyczy określonej *pozycji* w archiwum. Metoda `getNextEntry` zwraca obiekt typu `ZipEntry` opisujący pozycję archiwum. Strumień należy odczytać do końca, który jest jednocześnie końcem bieżącej pozycji archiwum. Następnie należy wywołać metodę `closeEntry`, by przejść do następnej pozycji. Nie należy zamykać strumienia `ZipInputStream`, zanim zostanie odczytana ostatnia pozycja. Oto typowa sekwencja wywołań służąca do odczytu zawartości pliku ZIP:

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

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

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



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

Strumienie wejściowe ZIP są dobrym przykładem potęgi abstrakcji strumieni. Odczytując dane przechowywane w skompresowanej postaci, nie musimy zajmować się ich dekompresją. Źródło bajtów formatu ZIP nie musi być plikiem — dane ZIP mogą być ściągane przez połączenie sieciowe.



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

API `java.util.zip.ZipInputStream` 1.1

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

API `java.util.zip.ZipOutputStream` 1.1

- `ZipOutputStream(OutputStream out)`
tworzy obiekt typu `ZipOutputStream`, który umożliwia kompresję i zapis danych w podanym strumieniu `OutputStream`.
- `void putNextEntry(ZipEntry ze)`
zapisuje informacje podanej pozycji `ZipEntry` do strumienia i przygotowuje strumień do odbioru danych. Dane mogą zostać zapisane w strumieniu przy użyciu metody `write()`.
- `void closeEntry()`
zamyka aktualnie otwartą pozycję archiwum ZIP. Aby otworzyć następną pozycję, wywołujemy metodę `putNextEntry`.
- `void setLevel(int level)`
określa domyślny stopień kompresji następnych pozycji archiwum o trybie DEFLATED. Domyślną wartością jest `Deflater.DEFAULT_COMPRESSION`. Zgłasza wyjątek `IllegalArgumentException`, jeżeli podany stopień jest nieprawidłowy.

- `void setMethod(int method)`

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

API | java.util.zip.ZipEntry 1.1

- `ZipEntry(String name)`

tworzy pozycję archiwum o podanej nazwie.

- `long getCrc()`

zwraca wartość sumy kontrolnej CRC32 danego elementu.

- `String getName()`

zwraca nazwę elementu.

- `long getSize()`

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

- `boolean isDirectory()`

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

- `void setMethod(int method)`

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

- `void setSize(long rozmiar)`

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

- `void setCrc(long crc)`

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

API | java.util.zip.ZipFile 1.1

- `ZipFile(String name)`

- `ZipFile(File file)`

- `ZipFile(String name, Charset charset) 1.7`

- `ZipFile(File file, Charset charset) 1.7`

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

- `Enumeration entries()`
zwraca obiekt typu `Enumeration`, wyliczający obiekty `ZipEntry` opisujące elementy archiwum `ZipFile`.
- `ZipEntry getEntry(String name)`
zwraca element archiwum o podanej nazwie lub `null`, jeżeli taki element nie istnieje.
- `InputStream getInputStream(ZipEntry ze)`
zwraca obiekt `InputStream` dla podanego elementu.
- `String getName()`
zwraca ścieżkę dostępu do pliku ZIP.

2.3. Strumienie obiektów i serializacja

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

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

2.3.1. Zapisywanie i wczytywanie obiektów serializowalnych

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

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

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

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

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

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

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

```
var e1 = (Employee)in.readObject();
var e2 = (Employee)in.readObject();
```

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

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

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



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

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

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

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

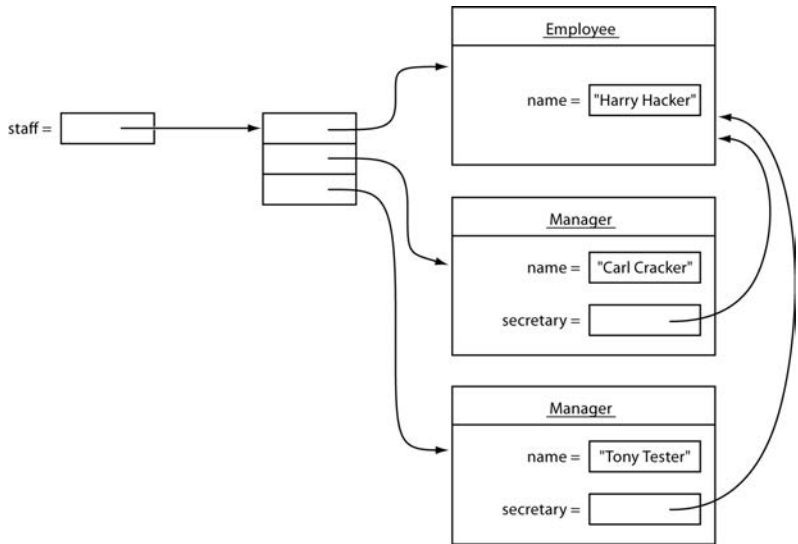
```
class Manager extends Employee
{
    private Employee secretary;
    . . .
}
```

Każdy obiekt typu `Manager` przechowuje teraz referencję do obiektu klasy `Employee` opisującego asystenta, a nie osobną kopię tego obiektu. Oznacza to, że dwóch menedżerów może mieć tego samego asystenta, tak jak zostało to przedstawione na rysunku 2.5 i w poniższym kodzie:

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

Rysunek 2.5.

Dwóch menedżerów
może mieć
wspólnego asystenta



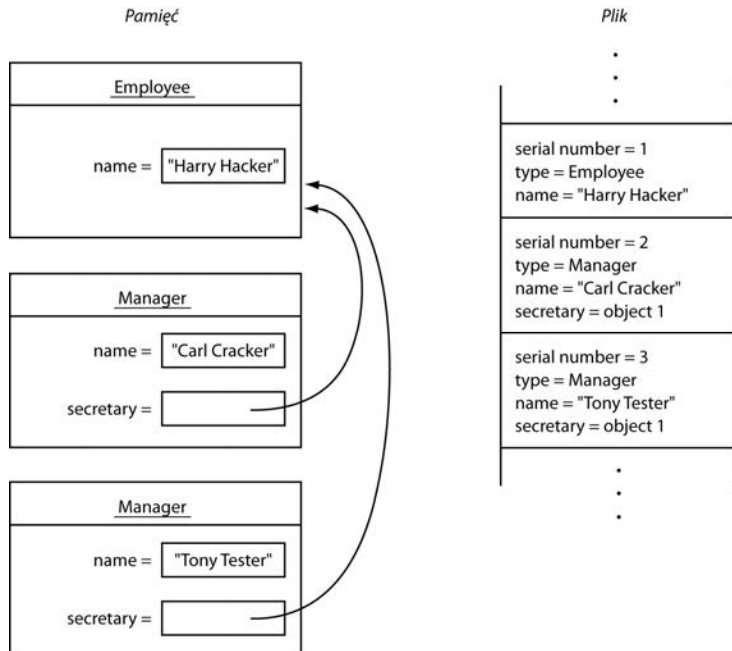
Teraz założymy, że zapisujemy dane pracowników na dysk. Oczywiście nie możemy zapisać i przywrócić adresów obiektów asystentów w pamięci, ponieważ po ponownym załadowaniu obiekt asystenta najprawdopodobniej znajdzie się w zupełnie innym miejscu pamięci.

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

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

Rysunek 2.6.

Przykład serializacji
obiektów



2. Jeśli referencja do obiektu została napotkana po raz pierwszy, obiekt zostaje zapisany w strumieniu wyjściowym.
3. Jeżeli obiekt został już zapisany, Java zapisuje, że w danym miejscu znajduje się „ten sam obiekt, co pod numerem seryjnym x”.

Wczytując obiekty z powrotem, Java odwraca całą procedurę.

1. Gdy obiekt pojawia się w strumieniu wejściowym po raz pierwszy, Java tworzy go, inicjuje danymi ze strumienia i zapamiętuje związek pomiędzy numerem i referencją do obiektu.
2. Gdy natrafi na znacznik „ten sam obiekt, co pod numerem seryjnym x”, sprawdza, gdzie znajduje się obiekt o danym numerze, i nadaje referencji do obiektu adres tego miejsca.



W tym rozdziale korzystamy z serializacji, aby zapisać zbiór obiektów na dysk, a później z powrotem je wczytać. Innym bardzo ważnym zastosowaniem serializacji jest przesyłanie obiektów przez sieć na inny komputer. Podobnie jak adresy pamięci są bezużyteczne dla pliku, tak samo są bezużyteczne dla innego rodzaju procesora. Ponieważ serializacja zastępuje adresy pamięci numerami seryjnymi, możemy transportować zbiory danych z jednego komputera do drugiego.



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

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

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

Listing 2.3 zawiera program zapisujący i wczytujący sieć powiązanych obiektów klas `Employee` i `Manager` (niektóre z nich mają referencję do tego samego asystenta). Zwróć uwagę, że po wczytaniu istnieje tylko jeden obiekt każdego asystenta — gdy pracownik `newStaff[1]` dostaje podwyżkę, znajduje to odzwierciedlenie za pomocą pól sekretary obiektów klasy `Manager`.

Listing 2.3. `serial/ObjectStreamTest.java`

```
package serial;

import java.io.*;

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

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

        var staff = new Employee[3];

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

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

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

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

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

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

API | java.io.ObjectOutputStream 1.1**■ ObjectOutputStream(OutputStream wy)**

tworzy obiekt `ObjectOutputStream`, dzięki któremu możesz zapisywać obiekty do podanego strumienia wyjścia.

■ void writeObject(Object ob)

zapisuje podany obiekt do `ObjectOutputStream`. Metoda ta zachowuje klasę obiektu, sygnaturę klasy oraz wartości wszystkich niestatycznych, nieprzechodnych pól składowych tej klasy, a także jej klas nadrzędnych.

API | java.io.ObjectInputStream 1.1**■ ObjectInputStream(InputStream we)**

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

- `Object readObject()`

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

2.3.2. Format pliku serializacji obiektów

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

Każdy plik zaczyna się dwubajtową „magiczną liczbą”:

```
AC ED
```

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

```
00 05
```

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

Łańcuchy zapisywane są jako:

- 74,
- długość zapisana w dwóch bajtach),
- znaki w formacie „zmodyfikowanego UTF-8”.

Dla przykładu, łańcuch "Harry" będzie wyglądał tak:

```
74 00 05 Harry
```

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

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

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

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

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

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

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

Tabela 2.3. Maski bitowe

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

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

Każdy deskryptor pola składowego składa się z następujących elementów:

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

Kod typu może mieć jedną z następujących wartości:

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

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

Dla przykładu, pole salary klasy Employee zostanie zapisane jako:

```
D 00 06 salary
```

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

```
72 00 08 Employee
  74 1E C8 E6 C3 F8 B8 77 02  „Odcisk” i flagi
  00 03                        Liczba pól składowych
  D 00 06 salary              Typ i nazwa pola składowego
  L 00 07 hireDay             Typ i nazwa pola składowego
  74 00 10 Ljava/util/Date;   Nazwa klasy pola składowego — java.util.Date
  L 00 04 name                Typ i nazwa pola składowego
  74 00 12 Ljava/lang/String; Nazwa klasy pola składowego — java.lang.String
  78                          Znacznik końca
  70                          Brak klasy bazowej
```

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

- 71
- numer seryjny (4 bajty)

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

Obiekt jest przechowywany w następującej postaci:

- 73
- opis klasy
- dane obiektu

Dla przykładu, oto zapis obiektu klasy `Employee`:

40 E8 6A 00 00 00 00	Wartość pola <code>salary</code> — <code>double</code>
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> — <code>String</code>

Jak widzimy, plik danych zawiera informacje wystarczające do odtworzenia obiektu klasy `Employee`. (W tym przykładzie datę zatrudnienia zapisujemy jako obiekt typu `Date`, a nie `LocalDate`, gdyż serializacja obiektów klasy `LocalDate` jest nieco trudniejsza).



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

Tablice są zapisywane w następujący sposób:

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

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

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

75	Tablica
72 00 0B [LEmployee;	Nowa klasa, długość łańcucha, nazwa klasy <code>Employee[]</code>
40 05 D6 7E 3B BB F2 50 02	„Odcisk” i flagi
00 00	Liczba pól składowych

78	Znacznik końca
70	Brak klas nadrzędnych
00 00 00 03	Liczba komórek tablicy

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

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

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

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

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

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

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

Referencja `null` jest zapisywana jako

```
70
```

Oto plik zapisany przez program `ObjectRefTest` z poprzedniego podrozdziału, wraz z komentarzami. Jeśli chcesz, uruchom program, spójrz na zapis pliku `employee.ser` w notacji szesnastkowej i porównaj z poniższymi komentarzami. Zwróć uwagę na najważniejsze wiersze, które są zamieszczone pod koniec pliku — zawierają one referencje zapisanego wcześniej obiektu.

AC ED 00 05	Nagłówek pliku
75	Tablica <code>staff</code> (nr 1)
72 00 0B [Lserial.Employee;	Nowa klasa, długość łańcucha, nazwa klasy <code>Employee[]</code> (nr 0)
40 05 D6 7E 3B BB F2 50 02	Odcisk i flagi
00 00	Liczba pól składowych
78	Znacznik końca

70		Brak klasy bazowej
00 00 00 03		Liczba elementów tablicy
73		staff[0] — nowy obiekt (nr 7)
72 00 0E serial.Manager		Nowa klasa, długość łańcucha, nazwa klasy (nr 2)
2F FF 1F 1D E8 7A A9 74 02		Odcisk i flagi
00 01		Liczba pól składowych
L 00 09 secretary		Typ i nazwa pola składowego
74 00 11 Lserial/Employee;		Nazwa klasy pola składowego — String (nr 3)
78		Znacznik końca
72 00 0F serial.Employee		Klasa nadrzędna — nowa klasa, długość łańcucha, nazwa klasy (nr 4)
74 1E C8 E6 C3 F8 B8 77 02		Odcisk i flagi
00 03		Liczba pól składowych
D 00 06 salary		Typ i nazwa pola składowego
L 00 07 hireDay		Typ i nazwa pola składowego
74 00 10 Ljava/util/Date;		Nazwa klasy pola składowego — String (nr 5)
L 00 04 name		Typ i nazwa pola składowego
74 00 12 Ljava/lang/String;		Nazwa klasy pola składowego — String (nr 6)
78		Znacznik końca
70		Brak klasy bazowej
40 F3 88 00 00 00 00 00		Wartość pola salary — double
73		Wartość pola hireDay — nowy obiekt (nr 9)
72 00 0E java.util.Date		Nowa klasa, długość łańcucha, nazwa klasy (nr 8)
68 6A 81 01 4B 59 74 19 03		Odcisk i flagi
00 00		Brak zmiennych składowych
78		Znacznik końca
70		Brak klasy bazowej
77 08		Zawartość zewnętrzna, liczba bajtów
00 00 00 83 E7 4B 7D 80		Nieprzezroczone bajty
78		Znacznik końca
74 00 0C Carl Cracker		Wartość pola name — String (nr 10)

73		Wartość pola secretary — nowy obiekt (nr 11)
71 00 7E 00 04		Istniejąca klasa (użyj nr 4)
40 E8 6A 00 00 00 00 00		Wartość pola pensja — double
73		Wartość pola dzienZatrudnienia — nowy obiekt (nr 12)
71 00 7E 00 08		Istniejąca klasa (użyj nr 8)
77 08		Zawartość zewnętrzna, liczba bajtów
00 00 00 91 1B 4E B1 80		Nieprzezroczyste bajty
78		Znacznik końca
74 00 0C Harry Hacker		Wartość pola name — String (nr 13)
71 00 7E 00 0B		staff[1] — istniejący obiekt (użyj nr 11)
73		obsługa[2] — nowy obiekt (nr 14)
71 00 7E 00 02		Istniejąca klasa (użyj nr 2)
40 E3 88 00 00 00 00 00		Wartość pola salary — double
73		Wartość pola hireDay — nowy obiekt (nr 15)
71 00 7E 00 08		Istniejąca klasa (użyj nr 8)
77 08		Zawartość zewnętrzna, liczba bajtów
00 00 00 94 6D 3E EC 00 00		Nieprzezroczyste bajty
78		Znacznik końca
74 00 0D Tony Tester		Wartość pola name — String (nr 16)
71 00 7E 00 0B		Wartość pola secretary — istniejący obiekt (użyj nr 11)

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

Powinniśmy zapamiętać, że:

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

2.3.3. Pola pomijane podczas serializacji

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

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

2.3.4. Metody `readObject` i `writeObject`

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

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

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

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

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

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

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

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

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

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

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

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

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

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



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



Jeśli klasa, którą można serializować, definiuje pole:

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

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

2.3.5. Metody `readExternal` i `writeExternal`

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

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

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

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

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

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

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

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

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



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



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

2.3.6. Metody `readResolve` i `writeReplace`

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

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

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

```
public enum PersonDatabase {
    INSTANCE;

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

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

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

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

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

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

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

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

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

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



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



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

2.3.7. Wersje

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

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

```
serialver Employee
```

otrzymujemy:

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

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

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

```

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

```

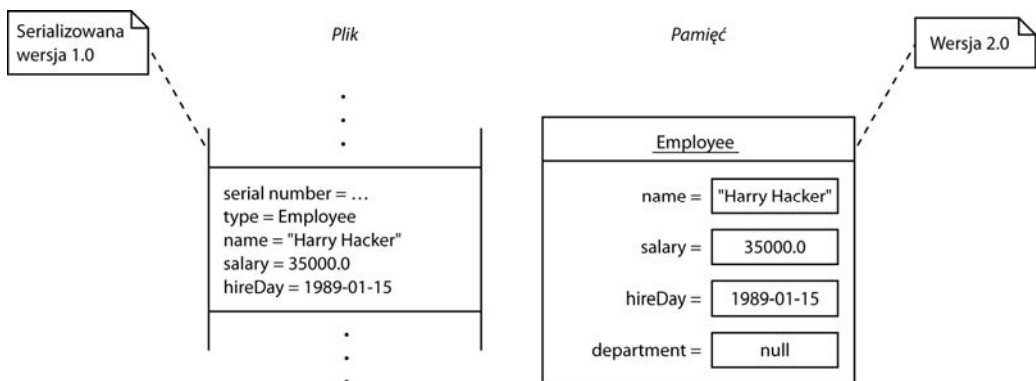
Klasa posiadająca statyczne pole składowe o nazwie `serialVersionUID` nie obliczy własnego „odcisku palca”, ale skorzysta z już istniejącej wartości.

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

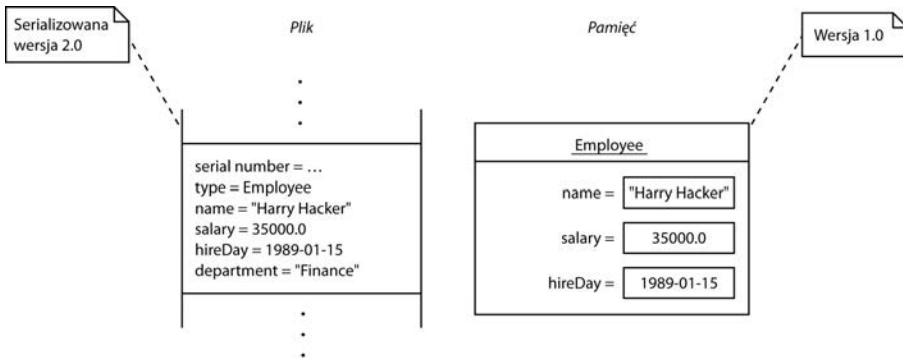
Jeśli zmienia się tylko metody danej klasy, sposób odczytu danych nie ulegnie zmianie. Jednakże jeżeli zmieni się pole składowe, możemy mieć pewne problemy. Na przykładu stary obiekt może mieć więcej lub mniej pól składowych niż aktualny albo typy pól mogą się różnić. W takim wypadku wejściowy strumień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

Wejściowy strumień obiektów porównuje pola składowe aktualnej wersji klasy z polami składowymi serializowanej wersji klasy znajdującej się w strumieniu. Oczywiście strumień bierze pod uwagę wyłącznie 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 ma pola składowe nieobecne w aktualnej wersji, strumień ignoruje te dodatkowe dane. Jeżeli aktualna wersja ma pola składowe nieobecne w serializowanym obiekcie, dodatkowe zmienne otrzymują swoje domyślne wartości (`null` dla obiektów, `0` dla liczb i `false` dla wartości logicznych).

Oto przykład. Załóżmy, że zapisaliśmy na dysku pewną liczbę obiektów klasy `Employee`, używając przy tym oryginalnej (1.0) wersji klasy. Teraz wprowadzamy nową wersję 2.0 klasy `Employee`, dodając do niej pole składowe `department`. Rysunek 2.7 pokazuje, co się dzieje, gdy obiekt wersji 1.0 jest wczytywany przez program korzystający z obiektów 2.0. Pole `department` otrzymuje wartość `null`. Rysunek 2.8 ilustruje odwrotną sytuację — program korzystający z obiektów 1.0 wczytuje obiekt 2.0. Dodatkowe pole `department` jest ignorowane.



Rysunek 2.7. Odczytywanie obiektu o mniejszej liczbie pól



Rysunek 2.8. Odczytywanie obiektu o większej liczbie pól

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



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



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



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

```
@Serial private void readObjectNoData() throws ObjectStreamException
```

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

2.3.8. Serializacja w roli klonowania

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

Kod z listingu 2.4 udowadnia, że aby otrzymać metodę `clone` „za darmo”, wystarczy rozszerzyć klasę `Serializable`.

Listing 2.4. `serialClone/SerialCloneTest.java`

```
package serialClone;

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

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

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

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

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

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

```

```

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

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

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

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

    public String getName()
    {
        return name;
    }

    public double getSalary()
    {
        return salary;
    }

    public LocalDate getHireDay()
    {
        return hireDay;
    }

    /**
     * Metoda zwiększająca wynagrodzenie tego pracownika
     */

```

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

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

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

2.3.9. Deserializacja a bezpieczeństwo

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



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

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

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

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

```

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

```

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

```

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

```

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

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

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

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

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

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

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

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

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

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

2.4. Praca z plikami

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

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

2.4.1. Ścieżki dostępu

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

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

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

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

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



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

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

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

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

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

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

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



Metoda `resolve` nie upraszcza wyniku. Na przykład wywołanie:

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

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

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

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

utworzy ścieżkę `/opt/myapp/temp`.

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

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

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

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

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



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

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

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



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

API `java.nio.file.Path` 7

- `static Path of(String first, String... more)` 11
tworzy ścieżkę poprzez połączenie przekazanych łańcuchów znaków.
- `Path resolve(Path other)`
- `Path resolve(String other)`
jeśli `other` jest ścieżką bezwzględną, zwraca `other`; w przeciwnym razie zwraca ścieżkę powstałą z połączenia `this` i `other`.
- `Path resolveSibling(Path other)`
- `Path resolveSibling(String other)`
jeśli `other` jest ścieżką bezwzględną, zwraca `other`; w przeciwnym razie zwraca ścieżkę powstałą z połączenia ścieżki nadrzędnej dla `this` i `other`.
- `Path relativize(Path other)`
zwraca ścieżkę względną, która rozwiązana względem `this` daje w wyniku `other`.
- `Path normalize()`
usuwa nadmiarowe elementy ścieżki, takie jak `.` i `..`.
- `Path toAbsolutePath()`
zwraca daną ścieżkę, jeśli jest ona ścieżką bezwzględną, a w przeciwnym razie ścieżkę względną wyznaczoną względem katalogu roboczego.

- `Path getParent()`
zwraca ścieżkę nadrzędną lub `null`, gdy ścieżka nadrzędna nie istnieje.
- `Path getFileName()`
zwraca ostatni komponent ścieżki lub `null`, gdy ścieżka nie ma komponentów.
- `Path getRoot()`
zwraca komponent korzenia danej ścieżki lub `null`, gdy ścieżka nie ma takiego komponentu.
- `toFile()`
tworzy obiekt `File` dla danej ścieżki.

java.io.File 1.0

- `Path toPath()` 7
tworzy obiekt `Path` dla danego pliku.

2.4.2. Odczyt i zapis plików

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

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

A oto analogiczne operacje zapisu:

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

Aby dopisać łańcuch na końcu pliku, zastosujemy następujące wywołanie:

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

To wywołanie:

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

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

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

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

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

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

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

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

java.nio.file.Files 7

- `static byte[] readAllBytes(Path path)`
- `static String readString(Path path, Charset charset)`
- `static List<String> readAllLines(Path path, Charset charset)`
wczytują zawartość pliku.
- `static Path write(Path path, byte[] contents, OpenOption... options)`
- `static Path write(Path path, String contents, Charset charset, OpenOption... options)`
- `static Path writeString(Path path, CharSequence contents, Charset cs, OpenOption... options)` 11
- `static Path write(Path path, Iterable<? extends CharSequence> contents, Charset cs, OpenOption options)`
zapisują podaną zawartość w pliku i zwracają ścieżkę.
- `static long mismatch(Path path, Path path2)` 12
- zwraca położenie pierwszego bajtu, na którym zawartości dwóch plików różnią się od siebie.
- `static String probeContentType(Path path)` 12
zwraca wynik próby odgadnięcia typu MIME zawartości pliku wykonanej przy użyciu zainstalowanych detektorów typów, a jeśli tej próby nie udało się pomyślnie wykonać, wartość `null`.
- `static InputStream newInputStream(Path path, OpenOption... options)`
- `static OutputStream newOutputStream(Path path, OpenOption... options)`
- `static BufferedReader newBufferedReader(Path path, Charset charset)`

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

otwierają plik do odczytu lub zapisu.

2.4.3. Tworzenie plików i katalogów

Aby utworzyć nowy katalog, wywołamy:

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

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

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

Pusty plik tworzymy, wywołując:

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

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

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

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

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

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

java.nio.file.Files 7

- `static Path createFile(Path path, FileAttribute<?>... attrs)`
 - `static Path createDirectory(Path path, FileAttribute<?>... attrs)`
 - `static Path createDirectories(Path path, FileAttribute<?>... attrs)`
- tworzą plik lub katalog. Metoda `createDirectories` tworzy również katalogi pośrednie.
- `static Path createTempFile(String prefix, String suffix, FileAttribute<?>... attrs)`

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

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

2.4.4. Kopiowanie, przenoszenie i usuwanie plików

Aby skopiować plik z jednej lokalizacji do innej, wywołamy:

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

Aby przenieść plik, użyjemy następującego wywołania:

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

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

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

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

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

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

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

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

Aby usunąć plik, wywołujemy po prostu:

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

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

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

Metody te możemy również wykorzystać do usunięcia pustego katalogu.

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

Tabela 2.4. Standardowe opcje stosowane w operacjach na plikach

Opcja	Opis
StandardOpenOption — do stosowania w metodach <code>newBufferedWriter</code> , <code>newInputStream</code> , <code>newOutputStream</code> , <code>write</code>	
READ	Otwiera plik do odczytu.
WRITE	Otwiera plik do zapisu.
APPEND	Jeśli plik otworzono do zapisu, dane będą dopisywane na jego końcu.
TRUNCATE_EXISTING	Jeśli plik otworzono do zapisu, jego dotychczasowa zawartość zostanie usunięta.
CREATE_NEW	Tworzy nowy plik lub zgłasza błąd, jeśli plik istnieje.
CREATE	Automatycznie tworzy nowy plik, jeśli jeszcze nie istnieje.
DELETE_ON_CLOSE	Środowisko postara się usunąć plik po jego zamknięciu.
SPARSE	Podpowiedź dla systemu operacyjnego, że plik będzie rzadki.
DSYNC SYNC	Wymaga, by wszelkie zmiany w danych pliku, jego zawartości oraz metadanych były synchronicznie zapisywane na nośniku.
StandardCopyOption — do stosowania w metodach <code>copy</code> , <code>move</code>	
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 <code>exists</code> , <code>isDirectory</code> , <code>isRegularFile</code>	
NOFOLLOW_LINKS	Nie należy używać łączy symbolicznych.
FileVisitOption — do stosowania w metodach <code>find</code> , <code>walk</code> , <code>walkFileTree</code>	
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)`
kopiuje lub przenosi `from` do lokalizacji `to`. Zwraca `to`.
- `static long copy(InputStream from, Path to, CopyOption... options)`
- `static long copy(Path from, OutputStream to, CopyOption... options)`
kopiuje dane ze strumienia wejściowego do pliku lub z pliku do strumienia wyjściowego, zwracając przy tym liczbę skopiowanych bajtów.

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

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

2.4.5. Informacje o plikach

Poniższe metody statyczne zwracają wartość boolean pozwalającą sprawdzić właściwość ścieżki:

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

Metoda `size` zwraca liczbę bajtów w pliku.

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

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

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

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

Atrybuty te pobieramy za pomocą następującego wywołania:

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

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

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

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

API `java.nio.file.Files` 7

- `static boolean exists(Path path)`
- `static boolean isHidden(Path path)`
- `static boolean isReadable(Path path)`
- `static boolean isWritable(Path path)`
- `static boolean isExecutable(Path path)`
- `static boolean isRegularFile(Path path)`
- `static boolean isDirectory(Path path)`
- `static boolean isSymbolicLink(Path path)`
sprawdzają wybraną właściwość pliku określonego przez ścieżkę `path`.
- `static long size(Path path)`
zwraca rozmiar pliku w bajtach.
- `A readAttributes(Path path, Class<A> type, LinkOption... options)`
wczytuje atrybuty pliku typu `A`.

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

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

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

- `Instant toInstant()` 8
zwraca instancję typu `Instant` reprezentującą ten sam czas co instancja `FileTime`.

2.4.6. Przeglądanie zawartości katalogu

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

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

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

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

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

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

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

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

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



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

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

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

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



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

2.4.7. Stosowanie strumieni katalogów

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

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

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

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

Przeglądane pliki możemy filtrować za pomocą wzorca:

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

Dostępne wzorce zostały przedstawione w tabeli 2.5.

Tabela 2.5. Wzorce filtrowania plików

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

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

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

W każdym z tych przypadków możemy określić, czy chcemy:

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

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



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

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

Oto fragment kodu wyświetlający wszystkie podkatalogi danego katalogu:

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

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

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

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

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

API `java.nio.file.Files` 7

- `static DirectoryStream<Path> newDirectoryStream(Path path)`
- `static DirectoryStream<Path> newDirectoryStream(Path path, String glob)`
pobierają iterator umożliwiający przeglądanie plików i katalogów w danym katalogu. Druga z metod akceptuje jedynie elementy zgodne z podanym wzorcem `glob`.
- `static Path walkFileTree(Path start, FileVisitor<? super Path> visitor)`
przełąca wszystkie elementy podrzędne danej ścieżki, stosując do nich obiekt `visitor`.

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

- `static FileVisitResult visitFile(T path, BasicFileAttributes attrs)`
wywoływana, gdy przetwarzany jest plik lub katalog, zwraca `CONTINUE`, `SKIP_SUBTREE`, `SKIP_SIBLINGS` lub `TERMINATE`. Domyślna implementacja nie podejmuje żadnych działań, tym samym powodując kontynuację przeglądania.

- `static FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)`
- `static FileVisitResult postVisitDirectory(T dir, IOException exc)`
 wywoływane przed i po wizycie w katalogu. Domyślna implementacja nie podejmuje żadnych działań, tym samym powodując kontynuację przeglądania. Jeśli jednak wyjątek `exc` będzie różny od `null`, to zostanie on zgłoszony, co spowoduje przerwanie działania metody.
- `static FileVisitResult visitFileFailed(T path, IOException exc)`
 wywołwana, gdy podczas próby uzyskania informacji o danym pliku został zgłoszony wyjątek. Domyślna implementacja ponownie zgłasza ten wyjątek, powodując zakończenie przeglądania. Jeśli przeglądanie ma być kontynuowane, należy zastąpić metodę własną implementacją.

2.4.8. Systemy plików ZIP

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

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

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

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

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

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

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

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

API `java.nio.file.FileSystem` 7

- `static FileSystem newFileSystem(Path path)` 13

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

API `java.nio.file.FileSystem` 7

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

tworzy ścieżkę, łącząc podane łańcuchy.

2.5. Mapowanie plików w pamięci

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

2.5.1. Wydajność plików mapowanych w pamięci

Na końcu tego podrozdziału zamieściliśmy program, który oblicza sumę kontrolną CRC32 dla pliku, używając standardowych operacji wejścia i wyjścia, a także pliku mapowanego w pamięci. Na jednej i tej samej maszynie otrzymaliśmy wyniki jego działania przedstawione w tabeli 2.6 dla pliku `src.jar` (51 MB) znajdującego się w katalogu `lib` pakietu JDK.

Tabela 2.6. Czasy wykonywania operacji na pliku

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

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

Oczywiście dokładne wartości pomiarów będą się znacznie różnić dla innych komputerów, ale łatwo domyślić się, że w przypadku swobodnego dostępu do pliku zastosowanie mapowania da zawsze poprawę efektywności działania programu.

Korzystanie z plików mapowanych w pamięci jest nieco mniej intuicyjne niż API plików o dostępie swobodnym. Poniżej podajemy przepis na jego zastosowanie.

Najpierw musimy uzyskać *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.

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

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

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

Na przykład:

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

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

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

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

Alternatywnie możemy również wykorzystać dostęp swobodny:

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

Możemy także czytać tablice bajtów, stosując metody:

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

Dostępne są również poniższe metody:

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

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

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

Aby poznać bieżący sposób uporządkowania bajtów w buforze, wywołujemy:

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

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

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

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

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

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

Listing 2.5. `memoryMap/MemoryMapTest.java`

```
package memoryMap;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Program uruchamiamy w następujący sposób:

```
java memoryMap.MemoryMapTest nazwapliku
```

API `java.io.FileInputStream` 1.0

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

API `java.io.FileOutputStream` 1.0

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

API `java.io.RandomAccessFile` 1.0

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

API `java.nio.channels.FileChannel` 1.4

- `static FileChannel open(Path path, OpenOption... options)` 7
otwiera kanał dla pliku o podanej ścieżce dostępu. Domyślnie kanał zostaje otwarty dla odczytu. Parametr `options` może przyjmować jedną z następujących wartości typu wyliczeniowego `StandardOpenOption`: `WRITE`, `APPEND`, `TRUNCATE_EXISTING`, `CREATE`.
- `MappedByteBuffer map(FileChannel.MapMode mode, long position, long size)`
tworzy w pamięci mapę fragmentu pliku. Parametr `mode` to jedna ze stałych `READ_ONLY`, `READ_WRITE` lub `PRIVATE` zdefiniowanych w klasie `FileChannel.MapMode`.

API `java.nio.Buffer` 1.4

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

API `java.nio.ByteBuffer` 1.4

- `byte get()`
pobiera bajt z bieżącej pozycji bufora i przesuwa pozycję do kolejnego bajta.

- `byte get(int index)`
pobiera bajt o podanym indeksie.
- `ByteBuffer put(byte b)`
umieszcza bajt na bieżącej pozycji bufora i przesuwa pozycję do kolejnego bajta.
- `ByteBuffer put(int index, byte b)`
umieszcza bajt na podanej pozycji bufora. Zwraca referencję do bufora.
- `ByteBuffer get(byte[] destination)`
- `ByteBuffer get(byte[] destination, int offset, int length)`
wypełnia tablicę bajtów lub jej zakres bajtami z bufora i przesuwa pozycję bufora o liczbę wczytanych bajtów. Jeśli bufor nie zawiera wystarczającej liczby bajtów, to nie są one w ogóle wczytywane i zostaje zgłoszony wyjątek `BufferUnderflowException`. Zwracają referencję do bufora.
- `ByteBuffer put(byte[] source)`
- `ByteBuffer put(byte[] source, int offset, int length)`
umieszcza w buforze wszystkie bajty z tablicy lub jej zakresu i przesuwa pozycję bufora o liczbę umieszczonych bajtów. Jeśli w buforze nie ma wystarczająco miejsca, to nie są zapisywane żadne bajty i zostaje zgłoszony wyjątek `BufferOverflowException`. Zwraca referencję do bufora.
- `Xxx 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`.

2.5.2. Struktura bufora danych

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

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



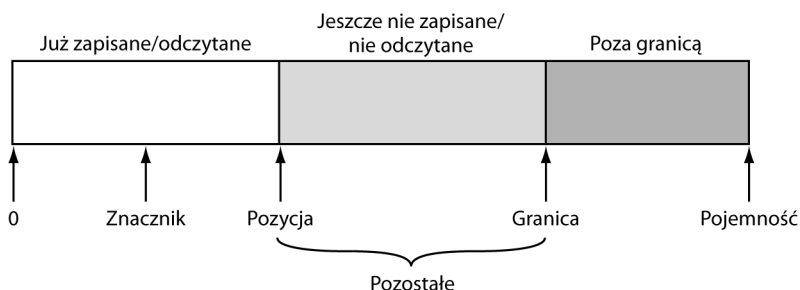
Klasa `StringBuffer` nie jest związana z omawianą tutaj hierarchią klas.

W praktyce najczęściej używane są klasy `ByteBuffer` i `CharBuffer`. Na rysunku 2.9 pokazaliśmy, że bufor jest scharakteryzowany przez:

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

Rysunek 2.9.

Bufor



Wymienione wartości spełniają następujący warunek:

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

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

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

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

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

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

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

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

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

API `java.nio.Buffer` 1.4

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

API `java.nio.ByteBuffer` 1.4

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

API `java.nio.ReadableByteChannel` 1.4

- `int read(ByteBuffer dst)`
wypełnia pozostały obszar bufora dostępnymi bajtami i zwraca liczbę bajtów przeniesionych danych lub -1, jeśli zawartość kanału została w całości odczytana.

API `java.nio.WritableByteChannel` 1.4

- `int write(ByteBuffer src)`
próbuję pobrać z bufora następne bajty i zwraca liczbę przeniesionych bajtów.

2.6. Blokowanie plików

Rozważmy sytuację, w której wiele równocześnie wykonywanych programów musi zmodyfikować ten sam plik. Jeśli pomiędzy programami nie będzie mieć miejsca pewien rodzaj komunikacji, to bardzo prawdopodobne jest, że plik zostanie uszkodzony. Blokady plików pozwalają kontrolować dostęp do plików lub pewnego zakresu bajtów w pliku.

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

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

```
FileChannel channel = FileChannel.open(path);
FileLock lock = channel.lock();
```

lub

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

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

Można również zablokować dostęp do fragmentu pliku za pomocą wywołania

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

lub

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

Parametrowi `shared` nadajemy wartość `false`, aby zablokować dostęp do pliku zarówno dla operacji odczytu, jak i zapisu. W przypadku blokady *współdzielonej* parametr `shared` otrzymuje wartość `true`, co umożliwia wielu procesom odczyt pliku, zapobiegając jednak uzyskaniu przez którykolwiek z nich wyłącznej blokady pliku. Nie wszystkie systemy operacyjne obsługują jednak blokady współdzielone. W takim przypadku możemy uzyskać blokadę wyłączną, nawet jeśli żądaliśmy jedynie blokady współdzielonej. Metoda `isShared` klasy `FileLock` pozwala nam dowiedzieć się, którą z blokad otrzymaliśmy.



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

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

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

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

- W niektórych systemach, w tym w Linuksie, blokady plików mają jedynie charakter *pomocniczy*. Nawet jeśli aplikacji nie uda się zdobyć blokady, to może zapisywać dane w pliku „zablokowanym” wcześniej przez inną aplikację.
- W niektórych systemach nie jest możliwe zablokowanie dostępu do mapy pliku w pamięci.
- Blokad plików są obsługiwane globalnie na poziomie maszyny wirtualnej Javy. Dwa działające współbieżnie wątki nie mogą uzyskać blokady tego samego pliku. Metody `lock` i `tryLock` zgłaszają wyjątek `OverlappingFileLockException` w sytuacji, gdy inny wątek ma już blokadę danego pliku lub pokrywającego się zakresu.
- W niektórych systemach zamknięcie kanału zwalnia wszystkie blokady pliku. Dlatego też należy unikać tworzenia wielu kanałów dostępu do tego samego, zablokowanego pliku.
- Działanie blokad plików w sieciowych systemach plików zależy od konkretnego systemu i dlatego należy unikać stosowania blokad w takich systemach.
- Poprawność działania blokad jest gwarantowana wyłącznie dla programów napisanych w Javie. Programy napisane w innych językach mogą korzystać z innych, niezgodnych mechanizmów blokowania.

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

Listing 2.6. fileLock/FileLockDemo.java

```
package fileLock;

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

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

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

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

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

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

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

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

```

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

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

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

```

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

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

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

`java.nio.channels.FileChannel` 1.4

■ `FileLock lock()`

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

■ `FileLock tryLock()`

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

■ `FileLock lock(long position, long size, boolean shared)`

■ `FileLock tryLock(long position, long size, boolean shared)`

uzyskuje blokadę dostępu do fragmentu pliku. Pierwsza wersja blokuje działanie programu do momentu uzyskania blokady, a druga zwraca natychmiast wartość `null`, jeśli nie może uzyskać od razu blokady. Parametr `shared` ma wartość `true` dla blokady współdzielonej, `false` dla wyłączonej.

■ `FileChannel truncate(long size)`

przycina plik do zadanej wielkości, o ile jest większy.

API `java.nio.channels.FileLock` **1.4**■ `void close()` **1.7**

zwalnia blokadę.

API `java.io.RandomAccessFile` **1.0**■ `FileChannel getChannel()` **1.4**

zwraca kanał do tego pliku o dostępie swobodnym.

2.7. Wyrażenia regularne

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

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

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

2.7.1. Składnia wyrażeń regularnych

Zacznijmy od czegoś prostego. Z wyrażeniem regularnym

`[Jj]ava.+`

może zostać uzgodniony dowolny łańcuch znaków następującej postaci:

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

Na przykład łańcuch `"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.

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

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

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

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

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

Tabela 2.7. Składnia wyrażeń regularnych

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

Tabela 2.7. Składnia wyrażeń regularnych — ciąg dalszy

Składnia	Objaśnienie	Przykład
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
\d, \D	Cyfry ([0-9] lub \p{Digit} w przypadku użycia flagi UNICODE_CHARACTER_CLASS); dopełnienie, czyli znak, który nie jest cyfrą.	\d+ — sekwencja cyfr
\w, \W	Znak słowa ([a-zA-Z0-9_] lub znak słowa w Unicode, jeśli została użyta flaga UNICODE_CHARACTER_CLASS).	
\s, \S	Znak odstępu ([\t\n\r\f\x0B] lub \p{IsWhiteSpace}, jeśli została użyta flaga UNICODE_CHARACTER_CLASS); znak, który nie jest odstępem.	\s*, \s* — przecinek opcjonalnie otoczony znakami odstępu
\h, \v, \H, \V	Odstęp w poziomie, odstęp w pionie; dopełnienia tych znaków.	
Sekwencje i alternatywy		
XY	Dowolny łańcuch z X, po którym następuje dowolny łańcuch z Y.	[1-9][0-9]* – liczba dodatnia bez początkowego zera
X Y	Dowolny łańcuch z X lub Y.	http ftp
Grupowanie		
(X)	Przechwycenie dopasowania X.	'([']*)' – tekst w apostrofach
\n	Dopasowanie n-tej grupy.	(["']).*\1 – pasuje do "Fred" lub 'Fred', lecz nie do "Fred'
(?<nazwa>X)	Przechwytuje dopasowanie X, nadając mu podaną nazwę.	'(?<id>[A-Za-z0-9]+)' przechwytuje dopasowanie, nadając mu nazwę id

Tabela 2.7. Składnia wyrażeń regularnych — ciąg dalszy

Składnia	Objaśnienie	Przykład
Grupowanie		
<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 X .	W wyrażeniu regularnym <code>(?:http ftp):/(.*)</code> dopasowanie po <code>://</code> jest grupą <code>\1</code>
<code>(?fi f2 ... :X), (?fi ... -fk ... :X),</code> gdzie f_i należy do <code>[dimSUx]</code>	Dopasowuje X , lecz go nie przechwytuje, używając lub nie używając (jeśli zastosowano <code>-</code>) 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 X .	<code>\x?</code> to opcjonalny znak <code>+</code>
<code>X*, X+</code>	X , 0 lub więcej razy oraz X 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>	X n razy, co najmniej n razy, pomiędzy n i m razy.	<code>[0-7]{1,3}</code> od jednej do trzech cyfr ósemkowych.
<code>Q?</code> , gdzie Q jest wyrażeniem z kwantyfikatorem	Kwantyfikator oporny; próbuje znaleźć jak najkrótsze dopasowanie, zanim spróbuje dobrać dłuższe.	<code>.*(<.+?>).*</code> — przechwytuje najkrótszą sekwencję, umieszczoną pomiędzy nawiasami kątowymi
<code>Q+</code> , 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		
<code>^, \$</code>	Początek, koniec wejścia (lub początek, koniec wiersza w trybie wielowierszowym).	<code>^Java\$</code> — pasuje do wpisanego słowa <code>Java</code> lub wiersza o tej zawartości
<code>\A, \Z, \z</code>	Początek wejścia, koniec wejścia, bezwzględny koniec wejścia (niezmienane w trybie wielowierszowym)	
<code>\b, \B</code>	Granica słowa, granica inna niż słowa.	<code>\bJava\b</code> — pasuje do słowa <code>Java</code>

Tabela 2.7. Składnia wyrażeń regularnych — ciąg dalszy

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

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

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

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

Tabela 2.8. Wstępnie zdefiniowane nazwy klas znaków

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

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

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



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

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

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



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

Niestety, składnia wyrażeń regularnych nie jest całkowicie ustandaryzowana. Istnieje zgodność w zakresie podstawowych konstrukcji, ale diabeł tkwi w szczegółach. Klasy języka Java związane z przetwarzaniem wyrażeń regularnych używają składni podobnej do zastosowanej w języku Perl. Wszystkie konstrukcje tej składni zostały przedstawione w tabeli 2.7. Więcej informacji na temat składni wyrażeń regularnych znajdziesz w dokumentacji klasy `Pattern` lub książce *Wyrażenia regularne. Wprowadzenie* autorstwa Michaela Fitzgeralda (Wydawnictwo Helion, 2013).

2.7.2. Testowanie dopasowania

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

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

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

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

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

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

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

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

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

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

Ten wzorzec można przekształcić w predykcję:

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

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

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

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

2.7.3. Znajdowanie wszystkich dopasowań w łańcuchu

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

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

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

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

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

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

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

Program przedstawiony na listingu 2.7 wykorzystuje powyższy mechanizm. Odnajduje on wszystkie hiperłącza na stronie internetowej i wyświetla je. Uruchamiając program, podajemy adres URL jako parametr w wierszu poleceń, na przykład:

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

Listing 2.7. match/HrefMatch.java

```
package match;

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

/**
 * Program wyświetlający wszystkie adresy URL na stronie WWW poprzez dopasowanie
 * wyrażenia regularnego opisującego znacznik <a href=...> języka HTML.
 * Uruchamianie: java match.HrefMatch adresURL
 *
 * @version 1.05 2023-08-16
 * @author Cay Horstmann
 */
public class HrefMatch
{
    public static void main(String[] args)
    {
        try
        {
            // Pobiera URL z wiersza poleceń lub używa domyślnego
            String urlString;
            if (args.length > 0)
                urlString = args[0];
            else
                urlString = "https://openjdk.org/";
        }
    }
}
```

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

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

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

2.7.4. Grupy

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

Blackwell Toaster USD29.95

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

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

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

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

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

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

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

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

Jeszcze lepszym rozwiązaniem może być zastosowanie grup nazwanych:

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

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

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

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

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

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

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



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



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

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

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

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

Listing 2.8. `regex/RegexTest.java`

```
package regex;

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

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

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

        Pattern pattern = Pattern.compile(patternString);

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

2.7.5. Podział w miejscach wystąpienia separatora

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

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

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

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

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

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

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

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

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



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

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

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

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

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

Ewentualnie można skorzystać z metody `Pattern.quote`:

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

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

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

2.7.6. Zastępowanie dopasowań

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

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

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

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

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

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

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

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

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

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

Metoda `replaceFirst` zastępuje tylko pierwsze wystąpienie wzorca.

2.7.7. Flagi

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

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

Ewentualnie można je także podawać w wyrażeniu:

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

Obsługiwane są poniższe znaczniki:

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

- `Pattern.MULTILINE` lub `m` — `^` i `$` oznaczają początek i koniec wiersza, a nie całego wejścia.
- `Pattern.UNIX_LINES` lub `d` — tylko `'\n'` jest rozpoznawany jako zakończenie wiersza podczas dopasowywania do `^` i `$` w trybie wielowierszowym.
- `Pattern.DOTALL` lub `s` — symbol `.` oznacza wszystkie znaki, w tym końca wiersza.
- `Pattern.COMMENTS` lub `x` — 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.
- `Pattern.CANON_EQ` — bierze pod uwagę kanoniczny odpowiednik znaków Unicode. Na przykład znak `u`, po którym następuje znak `¨` (diareza), zostanie dopasowany do znaku `ü`.

Ostatnich dwóch znaczników nie można używać wewnątrz wyrażeń regularnych.

API `java.util.regex.Pattern` 1.4

- `static Pattern compile(String expression)`
- `static Pattern compile(String expression, int flags)`
 kompiluje łańcuch wyrażenia regularnego, tworząc obiekt wzorca przyspieszający przetwarzanie. Parametr `flags` może mieć ustawiony jeden lub kilka bitów: `CASE_INSENSITIVE`, `UNICODE_CASE`, `MULTILINE`, `UNIX_LINES`, `DOTALL` i `CANON_EQ`.
- `Matcher matcher(CharSequence input)`
 tworzy obiekt pozwalający odnajdywać dopasowania do wzorca w łańcuchu wejściowym.
- `String[] split(CharSequence input)`
- `String[] split(CharSequence input, int limit)`
- `String[] splitWithDelimiters(CharSequence input, int limit)`
- `Stream<String> splitAsStream(CharSequence input)` 8
 rozбивa łańcuch wejściowy na tokeny, stosując wzorec do określenia granic podziału. Zwraca tablicę tokenów, które nie zawierają separatorów. Druga wersja metody ma parametr `limit`, określający maksymalną liczbę łańcuchów, które mogą zostać zwrócone. Jeśli dopasowanych zostało `limit - 1` separatorów, to ostatni element zwracanej tablicy zawiera niepodzielną resztę łańcucha wejściowego. Jeśli `limit` jest równy lub mniejszy od 0, to zostanie podzielony cały łańcuch wejściowy. Jeśli `limit` jest równy 0, to puste łańcuchy kończące dane wejściowe nie są umieszczane w tablicy.

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

zwraca niezmienną mapę odwzorowującą nazwy grup na ich indeksy.

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

- `boolean matches()`

zwraca true, jeśli łańcuch wejściowy pasuje do wzorca.

- `boolean lookingAt()`

zwraca true, jeśli początek łańcucha wejściowego pasuje do wzorca.

- `boolean find()`

- `boolean find(int start)`

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

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

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

- `String replaceAll(String replacement)`

- `String replaceFirst(String replacement)`

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

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

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

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

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

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

zwraca strumień zawierający wszystkie dopasowania.

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

- `string group()`

- `string group(int group)`

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

zwraca dopasowany łańcuch lub łańcuch dopasowany przez określoną grupę.

- `int start()`

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

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

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

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

API `java.util.Scanner` **5**

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

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

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

Skorowidz

A

adnotacja
 @Override, 119
 @Serial, 119
 @SuppressWarnings, 125
adres
 internetowy, 250
 localhost, 254
 URI, 212, 266, 285, 813
 URL, 192, 212, 266, 285, 310
akceleratory, accelerators, 604
akcesorium, accessory
 component, 647
akcje, 541
aktualizacje wsadowe, 358
aktywność komponentu,
 keyboard focus, 544
algorytm
 AES, 493, 494, 500
 DES, 493
 DSA, 481
 MD5, 478
 RSA, 481, 501
 SHA-1 (SHA1), 113, 477, 478
algorytmy
 szyfrowania, 493
 z kluczem symetrycznym, 500
analiza wyjątków SQL, 319
animacje GIF, 770
Apache, 362
API, 134, 159, 284, 874, 875
 dat i czasu, 364
 JOptionPane, 630
 Preferences, 554–557
 serwera HTTP, 293, 296
aplety, 477
 lokalizacja, 428
aplikacje
 interaktywne, 258
 klient-serwer, 304
architektura
 JDBC, 301
 trójwarstwowa, 304
archiwa ZIP, 104
Area, 738
arena ograniczona, confined
 arena, 875
arkusz roboczy, worksheet, 437
ASCII, 390
atributy, 180
 drukowania, 811
 hierarchia, 812
 hierarchia zbiorów, 812
 klasy, 814
 zbiór, 813
 zestawienie, 814–816
typy, 195
wartości domyślne, 195
automatyczne opakowywanie, 653
AWT, Abstract Window Toolkit,
 504, 720
 drukowanie, 788
 figury, 723
 filtrowanie obrazów, 780
 kształty, 724
 obszar przycięcia, 721
 operacje na obrazach, 773
 pliki graficzne, 763
 podstawowe przekształcenia,
 747
 pola, 737
 potokowe tworzenie grafiki,
 720

przekształcenia układu
 współrzędnych, 721
przezroczystość, 755
prycinanie, 752
rysowanie figur, 721
składanie obrazów, 721, 756,
 757
składanie przekształceń, 749
ślad pędzla, 721, 738
wypełnianie obszaru, 721, 745

B

backlog, 294
bajt
 najbardziej znaczący, MSB, 98
 najmłodszy, LSB, 98
bajty
 odczyt i zapis, 74
baza danych, 301
 adres URL, 310
 interfejs JNDI, 361
 kolumny, 304
 łączenie tabel, 307
 metadane, 348
 model dostępu, 302
 rekordy, 304, 338
 spójność, 356
 tabele, 304
 transakcje, 356
 uruchamianie, 311
 wiersz wstawiania, 341
 wypełnianie, 321
 zapytania, 305
 zarządzanie połączeniami, 361
 zbiory rekordów, 337, 344

bezpieczeństwo, 452
 JAAS, 466
 kryptografia klucza
 publicznego, 500
 ładowanie klas, 453
 podpis cyfrowy, 477–481, 487
 podpisywanie kodu, 490
 skróty haseł, 491
 skróty wiadomości, 477
 szyfrowanie, 492
 uwierzytelnianie
 użytkowników, 466
 uwierzytelnianie wiadomości,
 485
 weryfikacja kodu maszyny
 wirtualnej, 462
 biblioteka
 AWT, 504, 624, 720
 IFC, 505
 Java 2D (Java2D), 516, 720
 JCE, 499
 Swing, 505
 blokowanie plików, 156, 157
 błąd
 OutOfMemoryError, 851
 UnsatisfiedLinkError, 820
 BOM, byte order mark, 93, 422
 bufor danych, 147, 153
 buforowane zbiory rekordów, 345

C

certyfikaty
 podpisywanie, 487
 X.509
 keytool, 483, 487
 komponenty, 483
 składnica kluczy, 262, 483
 sprawdzanie
 wiarygodności, 484
 wydawcy, 484
 zarządzanie, 483
 żądanie, 488
 czas, 364, 405
 letni, 379
 lokalny, 376
 strefowy, 377

czcionki, 524–526, 737
 nazwa, 524
 obrys, 753
 rodzina, 524
 styl, 525

D

data, 405
 lokalna, 369
 modyfikatory, 374
 DDL, Data Definition Language,
 317
 deklaracja typu dokumentu,
 DTD, 179, 181, 191, 196
 delegacja zdarzeń, 534
 deserializacja, 122, 128
 obiektu, 110
 rekordów, 128
 dokumenty XML, 176, 179
 analiza zawartości, 182
 atrybuty, 178, 180
 bez przestrzeni nazw, 225
 CDATA, 181, 195
 deklaracja
 ATTLIST, 194
 DOCTYPE, 192
 ELEMENT, 191
 SYSTEM, 192
 deklaracja typu dokumentu,
 DTD, 179
 181, 191, 196
 element korzenia, 179
 elementy podrzędne, 183, 188
 instrukcje przetwarzania, 181
 JAXP, 182
 komentarze, 181
 kontrola poprawności, 190
 mieszana zawartość, 179
 nagłówek, 179
 parser DOM, 182
 parser SAX, 182
 parsowanie, 181
 PCDATA, 193
 pliki, 177
 pobieranie węzłów, 185
 przeglądanie atrybutów, 185
 przestrzeń nazw, 212

przetwarzanie dokumentów,
 182
 referencje bytów, 181
 referencje znaków, 180
 SAX, 215
 struktura dokumentu, 179
 tworzenie drzewa DOM, 225
 tworzenie, 224, 229
 wartości atrybutów, 178, 195
 wczytywanie, 182
 XML Schema, 199
 z przestrzenią nazw, 225
 zapisywanie, 226, 228
 znaczniki, 178
 DOM, Document Object Model,
 181
 drzewo, 184
 tworzenie drzewa, 224
 DOMSource, 239
 dostęp do składowych, 832, 836,
 837
 drukowanie, 788
 algorytm rozmieszczenia
 materiału, 797
 atrybuty, 811–816
 format strony, 791
 grafiki, 788
 liczba stron, 790
 PostScript, 808
 przycięcie kontekstu
 graficznego, 790
 rozmiar strony, 791
 strony transparentu, 798
 usługa, 805
 wielu stron, 797
 zadania, 789
 drzewa, 685
 dziedziczenie, 702
 ikony liści, 692
 las, 691
 model, 687
 modyfikowanie, 694
 nasłuchiwanie zdarzeń, 706
 obiekt użytkownika, 687
 proste, 686
 rozwijanie ścieżek, 696
 struktura, 716
 ścieżki, 694

terminologia, 686
 tworzenie modeli, 712
 węzły
 wybieranie węzłów, 707
 zmiana struktury węzła, 695
 rysowanie węzłów, 703
 przeglądanie węzłów, 701
 powiązania między węzłami, 688
 edycja węzłów, 697
 wstawianie, 697
 wygląd, 689, 690, 703
 DTD, 179, 181, 191, 196
 dymki, tooltips, 612
 dynamiczna kompilacja, 440
 dynamiczne tworzenie kodu, 437

E

edycja komórek, 676
 etykieta węzła, 703
 etykiety, 574

F

figury, 723
 2D, 516
 krzywa drugiego stopnia, 727
 krzywa trzeciego stopnia, 727
 łuk, 725, 726, 735
 odcinek, 728
 prostokąt, 518, 725
 prycinanie, 752
 punkty kontrolne, 735
 rysowanie, 721, 723
 tworzenie, 735
 wielokąt, 728
 wypełnianie, 723, 745
 filtrowanie
 obrazów, 780
 plików, 142, 644
 filtry, 295
 serializacji, 129
 flagi dla wyrażeń regularnych, 172
 format
 JSON, 188, 286, 288
 liczby, 397
 PostScript, 808, 809
 SVG, 232

formatowanie
 daty i czasu, 382–384, 406, 415
 komunikatów, 415, 417
 liczb, 397, 415
 formularze, 275
 HTML, 276, 279
 metoda GET, 277
 przetwarzanie danych, 275
 wysyłanie informacji do serwera, 277
 framework JAAS, 466
 FTP, 270
 funkcja
 CallNonVirtualXxxMethod, 844, 845
 CallStaticObjectMethod, 843
 CallStaticXxxMethod, 842, 845
 CallXxxMethod, 844
 DestroyJavaVM, 858
 ExceptionCheck, 854
 ExceptionClear, 854
 ExceptionOccured, 850, 851, 854
 FindClass, 836
 fprintf, 839
 GetArrayLength, 847, 848
 GetByteArrayElements, 863
 GetDoubleField, 833
 GetFieldID, 833, 837
 GetIntField, 833
 GetMethodID, 843, 845
 GetObjectArrayElement, 847, 848
 GetObjectField, 833
 GetStaticFieldID, 836
 GetStaticMethodID, 842
 GetStaticXxxField, 836, 837
 GetStringChars, 830
 GetStringLength, 830
 GetStringRegion, 829
 GetStringUTFChars, 829, 831
 GetStringUTFLength, 829
 GetStringUTFRegion, 829
 GetSuperClass, 873
 GetXxxArrayElements, 848
 GetXxxArrayRegion, 849
 GetXxxField, 837
 IsAssignableFrom, 862, 873
 JNI_CreateJavaVM, 855, 858

NewGlobalRef, 834
 NewObject, 845, 849
 NewString, 829
 NewStringUTF, 829
 ReleaseStringChars, 830
 ReleaseStringUTFChars, 829, 831
 ReleaseXxxArrayElements, 849
 SetDoubleField, 833
 SetIntField, 833, 863
 SetObjectArrayElement, 847–850
 SetObjectField, 833
 SetStaticXxxField, 836, 837
 SetXxxArrayRegion, 849
 SetXxxField, 837
 sprintf, 839
 Throw, 849, 853
 ThrowNew, 853
 funkcje
 interfejsu JNI, 819, 827–830
 obce, 874

G

generator
 klucza, 494
 liczb losowych, 495
 geometria pól, 737
 gniazda, 247
 adresy internetowe, 250
 kanały, 258
 limity czasu, 248
 nawiązanie połączenia, 258
 otwieranie, 247
 połączenia częściowo zamknięte, 257, 262
 przerywanie działania, 258
 serwera, 252
 zamykanie, 254
 graficzny interfejs użytkownika, GUI, 504
 grafika
 czcionki, 524
 drukowanie, 788
 kolory, 523
 obrazy, 530
 potokowe tworzenie, 720
 rastrowa, 763

GridBagLayout, 617
 dopełnienie wewnętrzne,
 internal padding, 617
 dopełnienie zewnętrzne,
 external padding, 617
 klasa pomocnicza GBC, 618
 ograniczenia, 616–618
 ograniczenie fill, 617
 poła weight, 616
 pole anchor, 617
 receptura, 618
 grupowanie lokalizatorów, 53
 grupy przycisków radiowych, 582

H

hasła, 470, 491
 hierarchia klas ładowania, 454
 host, 244
 HTML, 178, 246
 HTTP, 304
 nagłówki żądań, 270
 odpowiedzi, 271

I

identyfikator URI, 813
 identyfikatory walut, 404
 IETF, Internet Foundation Classes,
 505
 ikony w elementach menu, 600
 image/gif, 274
 import, 456
 informacje diagnostyczne, 435
 instalacja JDBC, 310
 instrukcja try, 25
 interfejs, *Patrz także* metody
 interfejsu
 Action, 541, 598, 604, 611, 612
 ActionListener, 534, 542, 555,
 580
 AdjustmentListener, 555
 Appendable, 79
 Attribute, 811, 813
 AttributeSet, 812, 817
 AutoCloseable, 285
 BasicFileAttributes, 141
 Bindings, 446
 BufferedImage, 773
 BufferedImageOp, 780

 ButtonModel, 564, 565
 CachedRowSet, 344, 345
 Callback, 473
 CallbackHandler, 475
 ChangeListener, 591
 CharacterData, 190
 CharSequence, 25, 63, 66
 Closeable, 79
 Collector, 46
 Comparator, 34, 410
 CompilationTask, 441
 Connection, 316
 Consumer, 31
 ContentHandler, 216, 217
 DatabaseMetaData, 318, 339,
 348, 349
 DataOutput, 97, 98
 DataSource, 362
 DiagnosticListener, 435
 DirectoryStream, 141
 Doc, 806
 DocAttribute, 811
 DocAttributeSet, 813
 DocPrintJob, 808
 Document, 227
 DoubleStream, 62
 DTDHandler, 217
 Element, 189
 EntityResolver, 182, 217
 ErrorHandler, 197, 198, 217
 Externalizable, 113, 121
 FileVisitor, 143, 144
 FilteredRowSet, 344
 Flushable, 79
 FocusListener, 555
 HttpClient, 284
 HttpResponse, 286
 ImageOutputStream, 771
 IntStream, 62
 ItemSelectable, 583
 Iterable, 26
 Iterator, 26
 ItemListener, 555
 Jakarta Mail, 298
 JdbcRowSet, 344
 JoinRowSet, 344
 JNDI, 361
 JNI, 825, 827, 829
 KeyListener, 555
 LayoutManager, 623, 627
 LayoutManager2, 624
 ListSelectionModel, 672
 LoginModule, 476
 LongStream, 62
 Map, 51, 295
 MatchResult, 167
 MenuListener, 606
 Metal, 689
 MouseListener, 548–550, 555
 MouseMotionListener,
 548–550, 555
 MouseWheelListener, 555
 MutableTreeNode, 687, 693
 NamedNodeMap, 185
 nasłuchu, listener interface, 533
 Node, 182, 183, 189, 215
 NodeList, 183
 ObjectInputValidation, 128
 ObjectStreamConstants, 113
 Pageable, 797
 Paint, 745
 Path, 130
 PreparedStatement, 326
 Printable, 788, 789, 792, 795
 PrintJobAttribute, 811
 PrintRequestAttribute, 811
 PrintRequestAttributeSet, 789
 PrintServiceAttribute, 811
 PropertyChangeListener, 648
 RandomGenerator, 63
 Readable, 79
 ReadableByteChannel, 155, 258
 Result, 238
 ResultSet, 315, 337, 344
 ResultSetMetaData, 349
 RowSet, 344
 Savepoint, 359
 ScriptEngineFactory, 448
 Serializable, 108, 113
 Shape, 516, 723, 734, 738
 Source, 238
 StandardJavaFileManager, 441
 Statement, 315, 318, 336, 337
 Stream, 24
 Stream.Builder, 26
 Stroke, 738
 SupportedValuesAttribute, 811
 TableCellEditor, 682, 683

TableCellRenderer, 674, 684
 TableColumnModel, 672
 TableModel, 659, 670
 Text, 184
 TreeCellRenderer, 703–705
 TreeModel, 687, 693, 713
 TreeModelListener, 716, 720
 TreeNode, 687, 700
 TreeSelectionListener, 706
 UnaryOperator<T>, 24
 WebRowSet, 344
 WindowFocusListener, 555
 WindowListener, 539, 555
 WindowStateListener, 541, 555
 WritableByteChannel, 155, 258
 XMLReader, 243
 XPath, 208, 211, 212

interfejsy

kompilatora, 433
 programowy JNI, 819
 programowy wywołań języka
 Java, 854, 858
 użytkownika, 562

internacjonalizacja

czas, 405
 data, 405
 komplety zasobów, 423
 komunikaty, 415
 liczby, 397
 lokalizatory, 391
 łańcuchy znaków, 425
 porządek alfabetyczny, 409
 waluta, 397, 404
 zbiory znaków, 419, 420

interpolacja, 780

IPv6, 250

iterator, 316

J

JAAS, Java Authentication and
 Authorization Service, 466, 469
 moduł logowania, 467
 moduły, 467

Java 2D

figury geometryczne, 516
 geometria pól, 737
 operacje na polach, 737
 rysowanie figur, 520
 współrzędne, 518

JavaServer Faces, 275

JDBC, Java Database

Connectivity, 301–303, 361
 adres URL baz danych, 310
 aktualizacja danych, 301
 aktualizacja wsadowa, 358
 API, 302
 architektura, 301
 instalacja, 310
 klient-serwer, 304
 limit poleceń, 318
 menedżer sterowników, 312
 nawiązywanie połączenia, 311
 połączenia krótkotrwałe, 319
 serwer, 303
 sterowniki, 302
 transakcje, 356
 wypełnianie bazy danych, 321
 zarządzanie połączeniami, 318
 zastosowania, 303
 zbiory rekordów, 338, 344
 zbiory wyników, 318, 319, 337
 źródło danych, 310

język

C, 819
 wywoływanie metod języka
 Java, 839, 844
 wywoływanie metod
 statycznych, 842

C++, 821

DDL, 317

HTML, 178, 246

SGML, 178

SQL, 301, 304, 315

XML, 176, 178

XML Schema, 199

XPath, 208

JNDI, Java Naming and Directory Interface, 361

nawiązywanie połączenia, 362
 pula połączeń, 362
 zarządzanie nazwami
 użytkowników, 362
 źródła danych, 362

JNI, Java Native Interface, 819, 875

funkcje, 827, 829, 830
 identyfikatory składowych, 842
 wywołania funkcji, 828

K

kanały, 258

katalog roboczy, 81

katalogi

przeglądanie zawartości, 140
 stosowanie strumieni, 141
 tworzenie, 135

klasa, *Patrz także* metody klasy

AboutDialog, 634
 AbstractAction, 543, 546, 601
 AbstractButton, 600
 AbstractCellEditor, 682, 683
 AbstractTableModel, 656, 676
 abstrakcyjne, 704
 ActionEvent, 533, 554
 AffineTransform, 750
 AffineTransformOp, 780, 787
 AlphaComposite, 758
 Arc2D, 723, 726
 Arc2D.Double, 736
 ArcMaker, 735
 Area, 738
 ArrayList, 718
 Attribute, 817
 Attributes, 220
 AttributesImpl, 243
 AWTEvent, 552
 Banner, 798
 BasicStroke, 738, 739, 744, 745
 BorderFactory, 585
 BorderLayout, 569, 570
 BreakIterator, 419
 Buffer, 147, 153, 258
 BufferedImage, 746, 758, 773,
 774, 778
 BufferedInputStream, 84
 BufferedOutputStream, 84
 BufferedReader, 86, 96
 ButtonGroup, 582, 583, 601
 ButtonModel, 583
 ByteBuffer, 147, 155
 ByteLookupTable, 784, 787
 CatalogFeatures, 199
 CatalogManager, 199
 ChangeEvent, 591
 Channels, 261
 Character, 97
 CharBuffer, 79

klasa, *Patrz także* metody klasy

Charset, 94
 ChoiceFormat, 418
 Cipher, 492
 CipherInputStream, 499
 CipherOutputStream, 500
 Class, 461, 584, 631, 702
 ClassLoader, 454, 457, 461
 ClassTreeFrame, 705
 CollationKey, 414
 Collator, 410, 414
 Collectors, 46, 55
 Color, 523, 745
 ColorAction, 543
 ColorConvertOp, 785
 ColorModel, 776, 779
 Component, 567
 ConcurrentHashMap, 53
 Container, 538, 567
 ContentHandler, 220
 ConvolveOp, 785–788
 Copies, 811, 813
 CopiesSupported, 811
 CubicCurve2D, 723, 736, 727
 Currency, 404, 405
 Cursor, 548
 DefaultTreeModel, 695
 DataInputStream, 77, 81, 82, 98
 DataOutputStream, 77, 98
 Date, 113, 388
 DateFormat, 384, 407
 DateTimeFormatter, 382, 384, 388, 406
 DecimalFormat, 402
 DecimalFormatSymbol, 403
 DefaultButtonModel, 564
 DefaultCellEditor, 680, 683, 684, 697
 DefaultHandler, 217
 DefaultMutableTreeNode, 687, 692, 705
 DefaultTableCellRenderer, 675
 DefaultTreeCellRenderer, 703–706
 DefaultTreeModel, 687, 696, 713, 716
 DiagnosticCollector, 435, 441
 Dimension, 510
 DocFlavor, 805

DocPrintJob, 806
 Document, 182
 DocumentBuilder, 182, 188, 225
 DocumentBuilderFactory, 196, 199, 214
 DOMResult, 240, 243
 DOMSource, 228, 239
 Double, 522
 DoubleSummaryStatistics, 63
 DriverManager, 314, 362
 Duration, 366
 Ellipse2D, 516, 518, 519, 723, 724
 EnumSyntax, 814
 EventListenerList, 716
 EventObject, 533, 552
 Field, 842
 File, 132, 688
 FileChannel, 147, 156, 159
 FileFilter, 645
 FileInputStream, 74, 81, 83
 FileLock, 156
 FileNameExtensionFilter, 649
 FileOutputStream, 81, 83
 Files, 130, 133
 FileSystem, 146
 FileSystems, 146
 FileView, 646
 Filter, 296
 FilterInputStream, 82
 FilterOutputStream, 82
 FocusEvent, 554
 Font, 525
 FontRenderContext, 526
 FontType, 201
 Format, 417
 Frame, 511
 GeneralPath, 723, 728, 734, 736, 744, 754
 GradientPaint, 745–747
 Graphics, 512, 514, 530, 720, 723
 Graphics2D, 516, 517, 526, 527, 721, 747
 GraphicsEnvironment, 524
 GregorianCalendar, 388
 GridBagConstraints, 615–618, 622, 623
 GridLayout, 571

HashMap, 53
 HashPrintRequestAttributeSet, 789, 813
 Headers, 295
 HttpClient, 284, 285
 HttpExchange, 294
 HttpHeaders, 287
 HttpRequest, 285
 HttpRequest.Builder, 285
 HttpURLConnection
 IIOServiceProvider, 772
 ImageInputStream, 768
 ImageIO, 746, 763, 764
 ImageReader, 772
 ImageReaderWriterSpi, 772
 ImageWriteParam, 769
 ImageWriter, 769, 773
 InetAddress, 250
 InetSocketAddress, 261
 InputMap, 545
 InputSource, 198
 InputStream, 73, 74, 252, 261, 266
 InputStreamReader, 85
 Instant, 365, 366, 388
 IntegerSyntax, 813
 IntSummaryStatistics, 63
 ItemEvent, 554
 JarInputStream, 105
 JarOutputStream, 105
 JButton, 534, 538, 543, 565
 JCheckBoxMenuItem, 601
 JComboBox, 735
 JComponent, 512, 527, 544, 566
 JDialog, 633, 636
 JFileChooser, 642, 644
 JFrame, 506, 508
 JLabel, 703
 JMenuItem, 599, 600
 JOptionPane, 628
 JPanel, 570
 JRadioButtonMenuItem, 601, 602
 JScrollPane, 654
 JTable, 652, 655, 660, 671, 680
 JTextComponent, 573
 JTree, 685, 694
 Kernel, 786, 788
 KeyEvent, 554

- KeyGenerator, 494, 498
- KeyPairGenerator, 501
- KeyStroke, 544, 605
- kompletów zasobów, 426
- LabeledPoint, 119
- Line2D, 516, 723, 724
- LineBorder, 586
- LineMetrics, 527
- ListResourceBundle, 426
- LocalDate, 115, 369, 388
- LocalDateTime, 378, 388
- Locale, 50, 392–395, 398
- LocalTime, 376
- LoginContext, 466, 468
- LongSummaryStatistics, 63
- LookupOp, 784–787
- LookupTable, 784
- ładowanie, 453
- Matcher, 166–168, 171
- MessageDigest, 478, 480
- MessageFormat, 415, 418
- Method, 842
- MouseEvent, 554
- MouseHandler, 549
- MouseMotionHandler, 549
- MouseMotionListener, 548
- MouseWheelEvent, 554
- NameCallback, 475
- Normalizer, 415
- Number, 398
- NumberFormat, 398, 399
- Object, 108
- ObjectInputStream, 107, 111
- ObjectOutputStream, 107
- OffsetDateTime, 379
- Optional, 36, 37
- OutputStream, 73, 74, 252, 261
- OutputStreamWriter, 85
- PageFormat, 791, 795, 796
- PasswordCallback, 476
- PasswordChooser, 637
- Pattern, 25, 165
- Period, 370, 379
- Point, 519
- Point2D, 519, 735
- Point2D.Double, 518
- Point2D.Float, 518
- Preferences, 556
- PreparedStatement, 327
- Principal, 469
- PrinterJob, 789–792, 797, 796
- PrintJob, 789
- PrintService, 806, 808
- PrintServiceLookup, 805, 807
- PrintStream, 88, 95
- PrintWriter, 86–89, 252, 278, 839
- Properties, 177
- PushbackInputStream, 82, 84
- QuadCurve2D, 723
- QuadCurve2D.Double, 736
- Random, 495
- RandomAccessFile, 100, 159, 771
- Raster, 775, 778
- Reader, 77, 78, 239
- Rectangle, 519
- Rectangle2D, 516–519, 723, 724
- Rectangle2D.Double, 517, 518
- Rectangle2D.Float, 518
- RectangularShape, 519, 723
- RescaleOp, 784, 787
- ResourceBundle, 427
- ResourceBundles, 426
- ResultSetMetaData, 349
- RetinaScanCallback, 473
- RoundRectangle2D, 723, 724
- RoundRectangle2D.Double, 736
- RowFilter, 664
- rozwiązywanie, 453
- SAXParser, 217, 219
- SAXParserFactory, 217, 219
- SAXSource, 239, 242
- Scanner, 85, 89, 252, 258
- ScriptEngineManager, 446
- SecretKeySpec, 498
- SecureRandom, 495
- ServerSocket, 252, 254
- ShapeMaker, 735
- ShapePanel, 735
- ShortLookupTable, 784
- SimpleDateFormat, 416
- SimpleDoc, 806, 808
- SimpleFileVisitor, 143
- SimpleLoginModule, 470
- Socket, 247, 248
- SocketChannel, 258
- SoftBevelBorder, 586
- Splitter, 26
- SSLServerSocket, 262
- SSLSocket, 262
- StandardCharsets, 93, 94
- StAXSource, 239
- StreamPrintService, 808
- StreamPrintServiceFactory, 808, 809
- StreamResult, 228
- StreamSource, 239, 242
- String, 172
- StringBuffer, 153
- StringBuilder, 97, 101
- Subject, 469
- szyfrowanie plików, 460
- TableColumn, 660, 672
- TableColumnModel, 660
- TemporalAdjusters, 374
- TextLayout, 753, 754
- TexturePaint, 746, 747
- Thread, 461
- Timestamp, 388
- TimeZone, 388
- Toolkit, 510
- Transformer, 227
- TransformerFactory, 242
- TreeMap, 51
- TreeModelEvent, 720
- TreeNode, 694
- TreePath, 695, 700, 707
- TreeSelectionEvent, 707
- TreeSelectionModel, 706
- UnixNumericGroupPrincipal, 467
- URI, 266
- URLConnection, 268, 269, 277, 284
- URLDecoder, 284
- URLEncoder, 284
- Variable, 719
- WindowEvent, 533, 554
- WritableRaster, 774, 776, 779
- Writer, 77, 78
- XPathConstants, 209
- XPathFactory, 211
- XPathNodes, 209
- ZipInputStream, 77, 104
- ZipOutputStream, 77, 104
- ZonedDateTime, 378, 388

- klasy, 456
 - abstrakcyjne, 73, 77, 153
 - adaptacyjne, adapter class, 539, 540
 - drzew, 687
 - Java 2D, 723
 - kolumn, 659
 - modelowe, 564
 - nadrzędne, 121
 - reprezentujące figury, 725
 - strumieni, 78
 - tabel, 661
 - zdarzeniowe AWT, 553, 554
 - znaków, 162, 164
 - klawiatura, 544
 - klient HTTP, 245, 284
 - rejestracja w dzienniku, 287
 - klonowanie obiektów, 126
 - klucz
 - prywatny, 481
 - publiczny, 481, 500
 - klucze
 - generowanie, 494
 - szyfrowanie, 493
 - wygenerowane automatycznie, 336
 - kod macierzysty, 818
 - plików źródłowych, 423
 - kodowanie znaków, 420
 - ASCII, 390
 - ISO 8859-1, 93
 - rodzime, native encoding, 94
 - Shift-JIS, 93
 - Unicode, 77, 98, 390, 411
 - US-ASCII, 95
 - UTF-16, 77, 84, 92, 828
 - UTF-32, 828
 - UTF-8, 85, 92, 98, 827
 - kody
 - języków, 392
 - krajów ISO-3166-1, 393
 - kolekcje, 22
 - polimorficzne, 107
 - kolektor przetwarzający,
 - downstream collector, 55–57
 - kolory, 523
 - definiowanie, 523
 - tło, 523
 - kolumny, 304, 659
 - ukrywanie, 666
 - wyświetlanie, 666
 - wyświetlanie nagłówka, 676
 - zmienianie rozmiaru, 660
 - komórki
 - edytowanie, 676
 - domyślny edytor, 697
 - rysowanie, 674
 - wybieranie koloru, 681
 - kompilacja
 - dynamiczna, 437
 - skryptu, 449
 - kompilator, 433, 453
 - Gnu C, 822
 - javac, 119
 - pakietu Cygwin, 822, 858, 872
 - przechwytywanie informacji diagnostycznych, 435
 - uruchamianie, 434
 - wczytywanie plików źródłowych, 435
 - wywoływanie, 433
 - zapis kodów bajtowych, 436
 - komplety zasobów, 423
 - implementacja klas, 426
 - klasy, 426
 - ładowanie, 424
 - pliki właściwości, 425
 - komponenty, *Patrz także* Swing
 - treść, 562
 - wygląd, 562
 - zachowanie, 562
 - komunikaty, 415
 - formatowanie z wariantami, 417
 - indeks znacznika, 416
 - konsola, 421
 - konstruktory, 843
 - kontekst graficzny, 791–794, 798
 - kontrola poprawności
 - dokumentów XML, 190
 - atrybuty, 194
 - byty, 196
 - parser XML, 194
 - PCDATA, 193
 - reguły zawartości elementów, 193
 - skrót, 196
 - typy atrybutów, 195
 - wartości domyślne atrybutów, 195
 - XML Schema, 191, 199
 - kontroler, controller, 563, 564
 - konwersje daty i czasu, 388
 - kryptografia klucza publicznego, 500
 - krzywa Beziera, 727
 - krzywe
 - drugiego stopnia, 727
 - punkty kontrolne, 727
 - trzeciego stopnia, 727
 - kształty
 - tworzenie, 724
 - kursory, 338, 549
- L**
- las, 686, 691
 - liczby, 397
 - formatowanie, 397
 - formaty, 399
 - lokalizatory, 398
 - losowe, 495
 - Linux, 308
 - lista, 46
 - rozwijalna, combo box, 588
 - logiczne nazwy czcionek, 525
 - logowanie, 475
 - lokalizacja
 - kalkulatora emerytalnego, 428
 - zasoby, 424
 - lokalizatory, 54, 391
 - czas, 405
 - data, 405
 - domyślne, 395
 - języki, 394
 - kody języków, 392
 - kody krajów, 393
 - liczby, 398
 - nazwa, 395
 - predefiniowane, 394
- Ł**
- ładowanie klas, 453
 - implementacja procedury ładującej, 457
 - klasy systemowe, 454

maszyna wirtualna, 453
 procedury, 454
 przestrzeń nazw, 456
 łańcuch zaufania, 487
 łańcuchy znakowe platformy
 Java, 829
 łączenie tabel, 307

M

macierz
 przekształcenia afinicznego,
 751
 przekształceń, 750
 mailto, 266
 mapa, 50, 51, 286
 akcji, action map, 545
 mapowanie plików, 146
 maski bitowe, 113
 maszyna wirtualna, 854
 modyfikacja kodu, 465
 weryfikacja kodu, 462
 menedżer bezpieczeństwa, 452
 menu, 597
 akceleratorzy, 604
 akcje, 598
 aktywowanie elementów, 606
 dezaktywowanie elementów,
 606
 elementy, 597, 598
 ikony w elementach, 600
 mnemoniki, 604
 pasek, 597
 podmenu, 597
 podręczne, pop-up menu, 602
 obsługa, 603
 tworzenie, 602
 pola wyboru, 601
 przełączniki, 601
 separatory, 598
 skrótów klawiszowe, 604
 tworzenie, 597
 metadane, 348, 349
 metoda
 abort, 476
 absolute, 342
 accept, 252, 254, 649
 acceptChanges, 346, 347

actionPerformed, 534, 536,
 542, 543, 555, 580, 598
 adaptRequest, 296
 add, 515, 538, 568, 585, 599,
 613, 693, 817
 addActionListener, 534, 580,
 598, 634
 addAttribute, 243
 addBatch, 359
 addCellEditorListener, 685
 addChoosableFileFilter, 645,
 649
 addClass, 711
 addColumn, 666, 671
 addItem, 589, 591
 addLayoutComponent, 624, 627
 addPropertyChangeListener,
 542
 addSelectionListener, 706
 addSeparator, 598, 599, 611, 613
 addTreeModelListener, 714, 720
 adjustmentValueChanged, 555
 afterHandler, 296
 afterLast, 343
 allMatch, 36
 andFilter, 665, 673
 anyMatch, 36
 append, 79, 80, 579, 734, 737
 appendChild, 225, 227
 appendCodePoint, 97
 applyPattern, 416
 Arrays.stream, 24, 62
 asPredicate, 166
 available, 75, 76
 availableCharsets, 94
 availableLocales, 25
 average, 63, 65, 66
 beforeFirst, 342
 beforeHandler, 296
 between, 368
 body, 292
 BodyHandlers.discarding, 286
 BodyHandlers.ofFile, 286
 BodyHandlers.ofFileDownload,
 286
 BorderFactory.createCompo
 ↳undBorder, 586
 BorderFactory.createEtched
 ↳Border, 586

BorderFactory.createTitled
 ↳Border, 586
 boxed, 63, 65, 66
 breadthFirstEnumeration,
 701, 705
 build, 284, 285, 291, 292
 buildSource, 438
 call, 441
 cancelCellEditing, 682, 683,
 685
 cancelRowUpdates, 343
 canInsertImage, 770, 773
 characters, 216, 220
 charAt, 80
 chars, 63
 Charset.defaultCharset, 95
 checkError, 87, 89
 checkLogin, 470
 children, 700
 clear, 155, 243
 clearParameters, 331
 clip, 721, 753, 754
 clone, 108, 128
 close, 75–80, 234, 252,
 316–319
 closeEntry, 104, 105
 closeOnCompletion, 317
 closePath, 728, 737
 codePoint, 66
 codePoints, 63
 Collator.getInstance, 410
 collect, 46, 48, 51
 collectingAndThen, 56, 59
 column, 318
 commit, 357, 359, 476
 compact, 155
 compare, 414
 compareTo, 409, 414
 compile, 173, 449
 concat, 34
 connect, 249, 269, 274
 convertColumnIndexToModel,
 671
 convertRowIndexToModel, 671
 copy, 137
 copyArea, 531, 533
 count, 23, 35
 counting, 59
 createBevelBorder, 587

metoda

createBindings, 446
 createBlob, 333
 createCachedRowSet, 348
 createClob, 333
 createCompoundBorder, 587
 createDirectory, 135
 createElement, 225, 227
 createElementNS, 225
 createEmptyBorder, 586
 createEtchedBorder, 587
 createFile, 135
 createFilteredRowSet, 348
 createImageInputStream, 768, 771
 createImageOutputStream, 769, 771
 createJdbcRowSet, 348
 createJoinRowSet, 348
 createLineBorder, 586
 createLoweredBevelBorder, 587
 createMatteBorder, 586
 createPrintJob, 807
 createServerSocket, 265
 createSocket, 265
 createStatement, 316, 337, 342, 357, 358
 createTempDirectory, 136
 createTempFile, 135, 136
 createTextNode, 225, 227
 createTitledBorder, 587
 createWebRowSet, 348
 createXMLStreamReader, 223
 createXMLStreamWriter, 233
 creationTime, 139
 curveTo, 728, 736
 dateFilter, 665, 673
 datesUntil, 371
 dayOfWeekInMonth, 375
 decode, 284
 defaultPage, 796
 defaultWriteObject, 119
 defineClass, 458, 461
 delete, 138
 DELETE, 292
 deleteIfExists, 138
 deleteRow, 341, 343
 depthFirstEnumeration, 701, 705

depthFirstTraversal, 702
 digest, 480
 distinct, 34, 35, 69
 dividedBy, 369
 doFinal, 493, 494, 498
 doPost, 279
 doubles, 63, 66
 draw, 517, 523, 722
 draw3DRect, 723
 drawArc, 723
 drawImage, 530, 532
 drawLine, 723
 drawOval, 723
 drawPolygon, 723
 drawPolyline, 723
 drawRectangle, 723
 drawRoundRect, 723
 drawString, 523, 525, 530
 dropWhile, 33, 34
 Duration.between, 365
 empty, 28, 41
 encode, 284
 end, 167, 169, 175
 endDocument, 216, 220
 endElement, 216, 220
 entries, 107
 equals, 414
 error, 197, 198
 eval, 446, 450
 evaluate, 209, 211
 execute, 317, 317, 337, 345, 347
 executeBatch, 358, 359
 executeQuery, 315, 316, 326, 331
 executeUpdate, 315, 317, 326, 331, 337, 357
 executor, 291
 exists, 139
 exportNode, 561
 exportSubtree, 561
 fatalError, 197, 198
 fileKey, 139
 Files.lines, 25, 70
 Files.walk, 141
 fill, 523, 524, 722, 723
 filter, 22, 23, 30, 32, 39, 786
 filtering, 60
 find, 141, 166, 174
 findAll, 175

findAny, 36
 findClass, 458, 461
 findFirst, 36
 first, 342
 firstDayOfMonth, 375
 firstDayOfNextMonth, 375
 firstDayOfNextYear, 375
 firstDayOfYear, 375
 firstValue, 286, 292
 flatMap, 30–32, 42–45
 flatMapping, 56, 59
 flip, 155
 flush, 75, 77, 80, 500
 focusGained, 555
 focusLost, 555
 followRedirects, 291
 forEach, 45, 48, 68
 forEachOrdered, 45
 ForkJoinPool.commonPool, 70
 forLanguageTag, 396
 format, 383, 386, 401, 409, 415, 417
 forName, 94
 generate, 24, 28, 62
 generateKey, 494, 498
 GET, 292
 get, 30, 40–43, 130, 151, 373, 446, 556, 560, 817
 getActionCommand, 585
 getActionCommands, 553
 getActionMap, 547
 getAddress, 250, 251
 getAdvance, 754
 getAllByName, 250, 251
 getAllowsChildren, 693
 getAncestorOfClass, 642
 GetArrayLength, 847
 getAs, 67
 getAscent, 529, 754
 getAsDouble, 63
 getAsInt, 63
 getAsLong, 63
 getAttribute, 189
 getAttributeCount, 224
 getAttributeLocalName, 224
 getAttributeName, 224
 getAttributes, 185, 189, 817
 getAttributeValue, 224
 getAutoCommit, 359

- getAvailableFontFamilyNames, 524
- getAvailableLocales, 399, 414
- getAvailableZoneIds, 378
- getAverage, 49
- getBackground, 524
- getBinaryStream, 333
- getBlob, 332
- getBlockSize, 498
- getBoolean, 557, 560
- getBundle, 424, 427
- getByName, 250, 251
- getByteArray, 557, 560
- getBytes, 333
- getCategory, 813, 817
- getCellEditorValue, 681, 683, 685
- getCellSelectionEnabled, 671
- getCenter, 519
- getCenterX, 519, 522
- getCenterY, 519, 522
- getChannel, 151, 160
- getCharacterInstance, 419
- getCharacterStream, 333
- getCharContent, 442
- getChild, 713, 719
- getChildAt, 700
- getChildCount, 695, 700, 719
- getChildNodes, 189
- getClassLoader, 454, 461
- getClickCount, 552
- getClip, 754
- getClob, 332
- getCollationKey, 412, 414
- getColorModel, 775, 778
- getColumn, 672
- getColumnClass, 659, 670
- getColumnCount, 355, 356, 656–658
- getColumnDisplaySize, 355
- getColumnLabel, 355
- getColumnModel, 670
- getColumnName, 355, 659
- getColumnNumber, 198, 199, 442
- getColumns, 573
- getColumnSelectionAllowed, 671
- getCommand, 347
- getComponentPopupMenu, 603
- getConcurrency, 342
- getConnection, 312, 314, 322
- getConnectTimeout, 274
- getContent, 275
- getContentEncoding, 271, 275, 280
- getContentLength, 271, 274
- getContentType, 271, 274
- getContext, 447
- getContextClassLoader, 461
- getCount, 49, 67
- getCountry, 54, 397
- getCrc, 106
- getCurrencyCode, 405
- getCurrencyInstance, 398, 400, 404
- getData, 185, 190
- getDataElements, 775, 778–780
- getDataSize, 321
- getDate, 271, 275
- getDayOfMonth, 372, 381
- getDayOfWeek, 370, 373, 381
- getDayOfYear, 373, 381
- getDecomposition, 414
- getDefault, 265, 395, 396
- getDefaultEditor, 684
- getDefaultFractionsDigits, 405
- getDefaultName, 476
- getDefaultRenderer, 684
- getDefaultToolkit, 510, 511
- getDescent, 530, 755
- getDescription, 646, 649, 650
- getDisplayCountry, 397
- getDisplayLanguage, 397
- getDisplayName, 395–397
- getDocumentElement, 182, 183, 189
- getDoInput, 273
- getDoOutput, 273
- getDouble, 316, 557, 560
- getEngineByExtension, 444
- getEngineByMimeType, 444
- getEngineByName, 444
- getEngineFactories, 444
- getEntry, 107
- getErrorCode, 321
- getErrorMessage, 280, 283
- getErrorWriter, 447
- getExpiration, 271, 275
- getExtensions, 444
- getFamily, 529
- getFieldDescription, 711
- getFields, 718
- getFileName, 133
- getFilePointer, 100, 103
- getFileSuffixes, 772
- getFirst, 295
- getFirstChild, 185, 189
- getFloat, 557, 560
- getFont, 574
- getFontMetrics, 527, 530
- getFontName, 529
- getFontRenderContext, 526, 527, 530, 753
- getFontRendererContext, 754
- getForeground, 524
- getFormatNames, 772
- getHeaderField, 269, 271, 274
- getHeaderFieldKey, 269, 270, 274
- getHeaderFields, 269, 271, 274
- getHeight, 519, 522, 527, 530, 772, 791, 796
- getHostAddress, 251
- getHostName, 251
- getHour, 377, 381
- getIcon, 575, 646, 650
- getIconImage, 511
- getIdentifier, 673
- getIfModifiedSince, 274
- getImage, 511
- getImageableHeight, 791, 796
- getImageableWidth, 791, 796
- getImageableX, 796
- getImageableY, 796
- getImageReadersByFormat
 - ↳ Name, 770
- getImageReadersByMimeType, 764, 771
- getImageReadersBySuffix, 764, 770
- getImageWritersByFormat
 - ↳ Name, 771
- getImageWritersByMimeType, 771
- getImageWritersBySuffix, 771

metoda

getIndex, 321
 getIndexOfChild, 713, 719
 getInheritsPopupMenu, 603
 getInputMap, 545, 547
 getInputStream, 107, 247, 248, 252, 269, 278
 getInstance, 405, 414, 478, 492, 497, 498, 758, 763
 getInt, 556, 560
 getInterface, 449
 getISOCountries, 397
 getJavaFileForOutput, 442
 getJDBCMinorVersion, 354
 getJDBCMinorVersion, 354
 getKeys, 428
 getKeyStroke, 544–547
 getKind, 441
 getLanguage, 397
 getLargeUpdateCount, 317
 getLastChild, 185, 189
 getLastModified, 271, 275
 getLastPathComponent, 695, 700
 getLastSelectedPathComponent
 → ent, 695, 700
 getLeading, 530, 755
 getLength, 183, 190, 220
 getLineInstance, 420
 getLineMetrics, 527, 529
 getLineNumber, 198, 442
 getLocale, 417
 getLocalHost, 250, 251
 getLocalName, 214, 215, 220, 224
 getLong, 557, 560
 getMax, 50, 67
 getMaxConnections, 354
 getMaximumFractionDigits, 401
 getMaximumIntegerDigits, 401
 getMaxStatement, 318
 getMaxStatements, 355
 getMaxX, 522
 getMaxY, 522
 getMessage, 441
 getMetaData, 354, 355
 getMethodCallSyntax, 448
 getMimeType, 444

getMIMETypes, 772
 getMin, 50, 67
 getMinimumFractionDigits, 401
 getMinimumIntegerDigits, 401
 getMinute, 377, 381
 getMinX, 522
 getMinY, 522
 getModel, 674
 getMonth, 373, 381
 getMonthValue, 373, 381
 getMoreResults, 335, 336
 getName, 106, 107, 224, 469, 475, 529, 646, 649, 806, 817
 getNames, 444
 getNameSpaceURI, 214, 215
 getNano, 377, 381
 getNextEntry, 104, 105
 getNextException, 320
 getNextSibling, 185, 189
 getNextWarning, 321
 getNodeName, 185, 189, 214
 getNodeValue, 185, 190
 getNumberInstance, 398, 400
 getNumericCode, 405
 getNumericCodeAsString, 405
 getNumImages, 772
 getNumThumbnails, 769, 772
 getObject, 318, 427
 getOffset, 381
 getOrientation, 797
 getOriginatingProvider, 764, 772, 773
 getOutline, 753
 getOutputSize, 498
 getOutputStream, 248, 252, 269, 278
 getOwner, 138
 getPageCount, 798
 getPageSize, 347
 getPaint, 524
 getParameter, 321
 getParent, 133, 461, 700
 getParentNode, 189
 getPassword, 347, 476, 576
 getPath, 146, 644, 712
 getPaths, 712
 getPathToRoot, 696
 getPercentInstance, 398, 401

getPixel, 774, 778, 780
 getPixels, 778
 getPoint, 552
 getPointCount, 735
 getPredefinedCursor, 548
 getPreferredSize, 516
 getPreviousSibling, 189
 getPrincipals, 469
 getPrinterJob, 789, 795
 getPrintService, 808, 809
 getPrompt, 476
 getQName, 220
 getRaster, 774, 778
 getReader, 447
 getReaderFileSuffixes, 771
 getReaderFormatNames, 771
 getReaderMIMETypes, 771
 getRequestProperties, 274
 getResultSet, 317
 getRGB, 775, 779
 getRoot, 133, 719
 getRootPane, 638, 642
 getRotateInstance, 750, 751
 getRow, 342
 getRowCount, 656–658
 getRowHeight, 671
 getRowMargin, 670
 getRowSelectionAllowed, 671
 getSavepointId, 359
 getSavepointName, 359
 getScaleInstance, 750, 751
 getScreenSize, 510, 511
 getSecond, 377, 381
 getSeconds, 366
 getSelectedColumns, 666
 getSelectedFile, 644, 648
 getSelectedFiles, 644, 649
 getSelectedItem, 589, 591
 getSelectedNode, 695
 getSelectedObjects, 583
 getSelection, 583, 585
 getSelectionModel, 662, 671
 getSelectionPath, 700, 707, 711
 getSelectionPaths, 707, 712
 getSentenceIterator, 420
 getShearInstance, 750, 751
 getSize, 106, 511
 getSource, 441, 553
 getSQLState, 321

- getStandardFileManager, 440
- getStrength, 414
- getString, 316, 349, 427
- getStringArray, 427
- getStringBounds, 526, 529
- getStringValue, 674
- getSubject, 469
- getSubString, 333
- getSum, 49, 67
- getSymbol, 405
- getSystemClassLoader, 461
- getTableCellEditorComponent, 682–684
- getTableCellRendererCompo-
 ↳ nent, 674, 682, 684
- getTableName, 347
- getTables, 354
- getTagName, 183, 189
- getTask, 434, 441
- getText, 224, 572–575
- getTitle, 509, 511
- getTransferSize, 321
- getTranslateInstance, 750, 752
- getTreeCellRendererCompo-
 ↳ nent, 704, 705
- getType, 342
- getTypeDescription, 646, 650
- getUpdateCount, 317
- getURI, 220
- getURL, 346
- getUsername, 347
- getValue, 220, 542, 546, 674, 813
- getValueAt, 656, 659, 681
- getValueCount, 674
- getVendorName, 772
- getVersion, 764, 772
- getWarnings, 321
- getWidth, 517–519, 522, 527,
 772, 791, 796
- getWordInstance, 420
- getWriter, 447
- getWriterFileSuffixes, 771
- getWriterFormatNames, 767,
 771
- getWriterMIMETypes, 771
- getX, 522, 552
- getXxx, 317
- getY, 522, 527, 552
- getYear, 373, 381
- graphemeClusters, 31
- group, 167, 174
- groupingBy, 53–59
- groupingByConcurrent, 54, 69
- handle, 475
- handleGetObject, 428
- hasMatch, 174
- hasNext, 223
- hasRemaining, 151
- headers, 292
- identity, 50
- ifPresent, 38, 67
- ifPresentOrElse, 38
- importPreferences, 561
- in.readAllBytes, 74
- include, 673
- init, 498
- initialize, 472, 476
- insert, 599
- insertItemAt, 589, 591
- insertNodeInto, 695, 701
- insertRow, 341, 343
- insertSeparator, 599
- interrupt, 258
- ints, 63, 66
- IntStream.of, 62
- inverse, 42
- invokeFunction, 449
- invokeMethod, 449
- isAfter, 373, 377, 382
- isAfterLast, 343
- isBefore, 373, 377, 382
- isBeforeFirst, 343
- isCellEditable, 659, 676, 682,
 685
- isCharacters, 224
- isClosed, 249, 317, 318
- isConnected, 249
- isDefaultButton, 642
- isDigit, 97
- isDirectory, 106, 139
- isEchoOn, 476
- isEditable, 572, 591
- isEmoji, 97
- isEmpty, 40, 41
- isEnabled, 542, 546
- isEndElement, 224
- isExecutable, 139
- isFirst, 343
- isGroupingUsed, 401
- isHidden, 139
- isIgnoringElementContent
 ↳ Whitespace, 199
- isInputShutdown, 258
- isJavaIdentifierPart, 97
- isJavaIdentifierStart, 97
- isLast, 343
- isLeaf, 692, 693, 714, 719
- isLeapYear, 373
- isLowerCase, 97
- isNamespaceAware, 215, 219
- isNegative, 369
- isOutputShutdown, 258
- isParseIntegerOnly, 401
- isPopupTrigger, 603
- isPresent, 40–43
- isReadable, 139
- isRegularFile, 139
- isResizable, 511
- isSelected, 582, 583, 602
- isShared, 156
- isSpaceChar, 97
- isStartElement, 223
- isSymbolicLink, 139
- isTraversable, 646, 650
- isUnresolved, 261
- isUpperCase, 97
- isValidating, 199, 219
- isVisible, 510
- isWhiteSpace, 97, 224
- isWritable, 139
- isZero, 369
- item, 190
- itemStateChanged, 555
- iterate, 24, 28, 34, 62
- iterator, 45, 48, 320
- JMenu.remove, 606
- JNI_CreateJavaVM, 854
- JNI_OnLoad, 824
- joining, 46, 49
- keyPressed, 555
- keyReleased, 555
- keys, 560
- keyTyped, 555
- last, 342
- lastAccessTime, 139
- lastDayOfMonth, 375
- lastDayOfYear, 375

- metoda
 - lastInMonth, 375
 - lastModifiedTime, 139
 - layoutContainer, 624, 627
 - layoutPages, 798
 - length, 80, 104, 333
 - limit, 26, 33, 69, 151
 - lines, 29
 - lineTo, 728, 736
 - list, 140
 - loadClass, 457
 - loadLibrary, 823, 824
 - lock, 156, 157, 159
 - login, 469, 476
 - logout, 469, 477
 - longs, 63, 66
 - lookingAt, 174
 - lookupPrintServices, 805, 806
 - main, 453
 - makeShape, 735
 - makeVisible, 696, 700
 - map, 30, 32, 39, 68, 292
 - mapMulti, 31, 32
 - mapping, 56, 59
 - mapToDouble, 63
 - mapToInt, 63
 - mapToLong, 63
 - mark, 76, 155
 - markSupported, 76
 - match, 166
 - matcher, 173
 - matches, 165, 174
 - max, 35, 36, 63–66
 - maxBy, 59
 - menuCanceled, 606, 607
 - menuDeselected, 606, 607
 - menuSelected, 606, 607
 - min, 35, 36, 63–66
 - minBy, 59
 - minimumLayoutSize, 624, 627
 - minus, 368, 372, 374, 376, 381
 - mismatch, 134
 - mouseClicked, 548, 555
 - mouseDragged, 549, 555
 - mouseEntered, 549, 555
 - mouseExited, 549, 555
 - mouseMoved, 548, 549, 555
 - mousePressed, 548, 555
 - mouseReleased, 548, 555
 - mouseWheelMoved, 555
 - move, 137
 - moveColumn, 671
 - moveTo, 727, 728, 736
 - moveToCurrentRow, 341, 343
 - moveToInsertRow, 341, 343
 - multipliedBy, 369
 - namedGroups, 169, 174, 175
 - negated, 369
 - newBufferedReader, 134
 - newBufferedWriter, 135
 - newBuilder, 291
 - newDirectoryStream, 144
 - newDocument, 225
 - newDocumentBuilder, 188
 - newFactory, 348
 - newFileSystem, 146
 - newInputStream, 134, 261
 - newInstance, 188, 211, 219, 222, 233, 227
 - newOutputStream, 134, 258, 261
 - newSAXParser, 219
 - newTransformer, 227
 - newXPath, 211
 - next, 86, 223, 316, 317, 375
 - nextPage, 347
 - node, 560
 - nodeChanged, 696, 701
 - noneMatch, 36, 37
 - normalize, 132, 415
 - notFilter, 665, 673
 - now, 367, 369, 372, 376, 380
 - nullInputStream, 76
 - nullOutputStream, 77
 - numberFilter, 673
 - of, 28, 41, 65–67, 132, 368, 369, 372, 373, 376, 380, 396
 - ofDateAdjuster, 375
 - ofLocalizedDate, 387, 409
 - ofLocalizedDateTime, 387, 409
 - ofLocalizedPattern, 385, 387
 - ofLocalizedTime, 387, 409
 - ofNullable, 29, 41, 44
 - ofPattern, 387
 - open, 151
 - openConnection, 268, 273
 - openInputStream, 275
 - openOutputStream, 275, 442
 - openStream, 81, 266, 269, 273
 - or, 39
 - orElse, 37, 67
 - orElseGet, 37, 67
 - orElseThrow, 38, 40, 41
 - orFilter, 665, 673
 - pack, 516
 - pageDialog, 792, 796
 - paint, 721
 - paintComponent, 512–516, 527, 629, 721, 744, 792
 - parallel, 68, 71
 - parallelStream, 22–24, 67, 72
 - parse, 188, 219, 243, 387, 398, 401, 409
 - partitioningBy, 54, 57
 - pathFromAncestorEnumeration
 - ↪tion, 702
 - peek, 34, 35
 - plus, 368, 372, 374, 376, 380
 - populate, 347
 - POST, 292
 - postOrderEnumeration, 705
 - postOrderTraversal, 702
 - postVisitDirectory, 142–145
 - preferredLayoutSize, 624, 627
 - preOrderEnumeration, 705
 - preOrderTraversal, 702
 - prepareStatement, 331, 337, 342
 - previous, 342, 375
 - previousOrSame, 375
 - previousPage, 347
 - preVisitDirectory, 145
 - print, 87, 88, 788, 791, 795, 796, 808
 - printDialog, 789, 796
 - printf, 88, 89
 - println, 87, 88
 - probeContentType, 134
 - PUT, 292
 - put, 152, 153, 446, 557, 560
 - putBoolean, 561
 - putByteArray, 561
 - putClientProperty, 690, 694
 - putDouble, 561
 - putFloat, 561
 - putInt, 557, 560
 - putLong, 561
 - putNextEntry, 105

- putValue, 542, 546, 600, 612
- quadTo, 728, 736
- quoteReplacement, 172, 174
- range, 64, 65
- rangeClosed, 64, 65
- read, 74, 80, 155, 258, 499, 764, 768–771
- readAllBytes, 76, 134
- readAllLines, 134
- readAttributes, 139
- readBoolean, 98
- readByte, 98
- readChar, 98
- readDouble, 99
- readExternal, 121, 123
- readFixedString, 101
- readFloat, 99
- readFully, 99
- readInt, 99, 108
- readLine, 86
- readLong, 99
- readNBytes, 75
- readObject, 108, 112, 119, 123
- readResolve, 122, 123
- readShort, 99
- readString, 134
- readThumbnail, 769, 772
- readUTF, 99
- reduce, 60–62
- regexFilter, 665, 673
- relative, 338, 342
- relativize, 132, 268
- release, 156
- releaseSavepoint, 357, 359
- reload, 696, 701
- remaining, 155
- remove, 599, 817
- removeAllItems, 591
- removeCellEditorListener, 685
- removeColumn, 666, 671
- removeItem, 589, 591
- removeItemAt, 589, 591
- removeLayoutComponent, 624, 627
- removeNodeFromParent, 696, 701
- removePropertyChange
 - ↳ Listener, 542
- removeTreeModelListener, 714, 720
- repaint, 513, 516
- replaceAll, 171–174
- replaceFirst, 174
- reset, 76, 155, 480
- resetChoosableFileFilters, 649
- resetChoosableFilters, 645
- resolve, 131, 132, 268
- resolveEntity, 198
- resolveSibling, 131, 132
- results, 167, 174
- revalidate, 573, 574
- rewind, 155
- rollback, 357, 359
- rotate, 748, 752
- run, 440
- scale, 747, 748, 752
- Scanner.findAll, 167
- Scanner.tokens, 25
- scrollPathToVisible, 696, 700
- seek, 100, 103
- sendAsync, 287
- set, 722
- setAccelerator, 605
- setAcceptAllFileFilterUsed, 645, 649
- setAccessory, 649
- setAction, 600
- setActionCommand, 585
- setAllowsChildren, 692, 694
- setAnchor, 619
- setAsksAllowsChildren, 692, 693
- setAttribute, 225, 227
- setAttributeNS, 226, 227
- setAutoCommit, 357, 359
- setAutoCreateRowSorter, 663
- setAutoResizeMode, 670
- setBackground, 523, 524
- setBinaryStream, 333
- setBorder, 586, 588
- setBounds, 508–511
- setCellEditor, 684
- setCellRenderer, 684
- setCellSelectionEnabled, 662, 671
- setCharacterStream, 333
- setClip, 752, 754, 791
- setClosedIcon, 706
- setColumns, 573, 577, 578
- setColumnSelectionAllowed, 662, 671
- setCommand, 345, 347
- setComparator, 672
- setComponentPopupMenu, 603
- setComposite, 721, 758, 762
- setConnectTimeout, 274
- setContentHandler, 243
- setContextClassLoader, 462
- setCrc, 106
- setCurrentDirectory, 643, 648
- setCursor, 552
- setDataElements, 776, 779
- setDecomposition, 414
- setDefault, 396
- setDefaultButton, 638, 642
- setDefaultCloseOperation, 508
- setDefaultNamespace, 233
- setDefaultRenderer, 675
- setDisplayedMnemonicIndex, 604, 606
- setDoInput, 269, 273
- setDoOutput, 269, 273, 277
- setEchoChar, 576
- setEditable, 572, 591, 697
- setEnabled, 542, 546, 606, 607
- setEntityResolver, 198
- setErrorHandler, 197, 198
- setErrorWriter, 447
- setFileFilter, 649
- setFileSelectionMode, 644, 648
- setFileView, 647, 649
- setFill, 619
- setFont, 574
- setForeground, 524
- setFrameFromCenter, 519
- setGroupingUsed, 401
- setHeaderRenderer, 676, 684
- setHeaderValue, 676, 684
- setHorizontalTextPosition, 600, 601
- setIcon, 574, 575
- setIconImage, 508, 511
- setIfModifiedSince, 274
- setIgnoringElementContent
 - ↳ Whitespace, 197, 199
- setInheritsPopupMenu, 603

metoda

- setInput, 768
- setInverted, 596
- setJMenuBar, 597, 600
- setLabelTable, 593, 597
- setLayout, 567, 568, 571
- setLeafIcon, 705
- setLevel, 105
- setLineWrap, 577, 579
- setLocale, 395, 417
- setLocation, 508–510
- setLogWriter, 314
- setMajorTickSpacing, 592, 596
- setMaximumFractionDigits, 401
- setMaximumIntegerDigits, 401
- setMaxWidth, 660, 672
- setMethod, 106
- setMinimumFractionDigits, 401
- setMinimumIntegerDigits, 401
- setMinorTickSpacing, 592, 596
- setMinWidth, 660, 672
- setMnemonic, 604, 605
- setModifiedSince, 270
- setMultiSelectionEnabled, 644, 648
- setName, 476
- setNamespaceAware, 202, 215, 218, 219
- setOpenIcon, 705
- setOutput, 772
- setOutputProperty, 227
- setPageable, 797
- setPageSize, 347
- setPaint, 523, 524, 721, 745, 746
- setPaintLabels, 593, 596
- setPaintTicks, 592, 596
- setPaintTrack, 596, 597
- setParseIntegerOnly, 401
- setPassword, 345, 347, 476
- setPixel, 774, 779
- setPixels, 774, 779
- setPreferredWidth, 660, 672
- setPrefix, 233
- setPrintable, 796
- setProperty, 223
- setReader, 446, 447
- setReadTimeout, 274
- setRenderingHints, 721
- setRequestProperty, 270, 274
- setResizable, 508, 511, 661, 672
- setRootVisible, 691, 693
- setRowFilter, 672
- setRowHeight, 662, 671
- setRowMargin, 662, 670
- setRows, 576, 579
- setRowSelectionAllowed, 662, 671
- setRowSorter, 672
- setSavepoint, 357, 359
- setSeed, 495
- setSelected, 580, 582, 601, 602
- setSelectedFile, 644, 648
- setSelectedFiles, 648
- setSelectionMode, 662, 672
- setShowsRootHandles, 691, 693
- setSize, 106, 510, 511
- setSnapToTicks, 592, 597
- setSortable, 672
- setSoTimeout, 248, 249
- setStrength, 414
- setString, 326
- setStringConverter, 672
- setStroke, 721, 738, 739, 744, 745
- setTableName, 347
- setTabSize, 579
- setText, 572–575, 637
- setTitle, 508–511
- setToolTip, 612
- setToolTipText, 612, 613
- setToRotation, 750, 752
- setToScale, 750, 752
- setToShear, 750, 752
- setToTranslation, 750, 752
- setTransform, 750, 752
- setURL, 345, 346
- setUsername, 345, 347
- setUserObject, 688, 693
- setValidating, 199, 219
- setValue, 863
- setValueAt, 659, 683
- setVisible, 508, 510, 634–638
- setWidth, 661, 672
- setWrapStyleWord, 579
- setWriter, 446, 447
- setXxx, 331
- shear, 748, 752
- shouldSelectCell, 682–685
- show, 603
- showConfirmDialog, 630, 631
- showDialog, 648
- showInputDialog, 629, 632
- showInternalConfirmDialog, 631
- showInternalInputDialog, 632
- showInternalMessageDialog, 631
- showInternalOptionDialog, 632
- showMessageDialog, 629, 631
- showOpenDialog, 643, 644, 648
- showOptionDialog, 629–632
- showSaveDialog, 644, 648
- shutdownInput, 257
- shutdownOutput, 257
- size, 138, 139
- skip, 33, 76
- skipBytes, 99
- skipNBytes, 76
- sort, 410
- sorted, 34, 35, 68
- split, 33, 89, 170–173
- splitAsStream, 25, 29, 173
- splititerator, 30
- splititeratorUnknownSize, 29
- SplittableGenerator.of, 70
- splitWithDelimiters, 171, 173
- sprint, 838
- squareRoot, 42
- start, 167, 169, 174
- startDocument, 216, 220
- startElement, 216, 217, 220
- statusCode, 292
- stopCellEditing, 682–685
- stream, 22–24, 29, 43
- Stream.empty, 24
- subSequence, 80
- sum, 63, 65, 66
- summarizing, 46
- summarizingDouble, 49
- summarizingInt, 49
- summarizingLong, 49
- summaryStatistics, 63, 65, 66
- summingDouble, 59

- summingInt, 59
- summingLong, 59
- supplier, 62
- supportsBatchUpdates, 360
- supportsResultSetConcurrency, 339, 344
- supportsResultSetType, 339, 343
- System.exit, 508
- System.setProperty, 95
- systemNodeForPackage, 556, 560
- systemRoot, 556, 560
- takeWhile, 33
- toAbsolutePath, 132
- toArray, 45, 48, 65, 66, 817
- toCollection, 49
- toConcurrentMap, 51, 53, 54
- toDays, 368
- toDaysPart, 368
- toFile, 132, 133
- toHours, 368
- toInstant, 139, 382
- tokens, 30
- toLanguageTag, 397
- toList, 26, 45, 48, 49
- toLocalDate, 382
- toLocalDateTime, 382
- toLocalTime, 382
- toMap, 50–52
- toMillis, 368
- toMinutes, 368
- toNanoOfDay, 377
- toNanos, 368
- toPath, 132, 133
- toSecondOfDay, 377
- toSeconds, 366, 368
- toSet, 46, 49, 55
- toString, 80, 397, 405, 673, 687, 735
- toUnmodifiableList, 49
- toUnmodifiableMap, 53
- toUnmodifiableSet, 49
- transferTo, 74, 76
- transform, 228, 238, 721, 750, 752
- translate, 748, 752, 798
- treeNodesChanged, 720
- treeNodesInserted, 720
- treeNodesRemoved, 720
- treeStructureChanged, 720
- trim, 185, 399, 573
- truncate, 159
- tryLock, 156–159
- type, 212
- unordered, 69, 71
- unread, 82, 84
- until, 373
- update, 480, 493, 498
- updateDouble, 340
- updateObject, 318
- updateRow, 340–343
- updateXxx, 343
- uri, 292
- userNodeForPackage, 556, 560
- userRoot, 556, 560
- validate, 574
- validateObject, 128
- value, 212
- valueChanged, 706, 712
- valueForPathChanged, 716, 720
- visitFile, 142, 144
- visitFileFailed, 143, 145
- walk, 140
- walkFileTree, 141–144
- warning, 197, 198
- windowActivated, 540, 541, 555
- windowClosed, 540, 541, 555
- windowClosing, 539, 540, 555
- windowDeactivated, 540, 541, 555
- windowDeiconified, 540, 541, 555
- windowGainedFocus, 555
- windowIconified, 540, 541, 555
- windowLostFocus, 555
- windowOpened, 539, 540, 555
- windowStateChanged, 541, 555
- with, 374, 375, 377
- withDayOfMonth, 372
- withDayOfYear, 372
- withLocale, 387, 406, 409
- withMonth, 372
- withYear, 372
- withZoneSameInstant, 381
- withZoneSameLocal, 381
- write, 74–76, 88, 134, 155, 258, 500, 764, 770, 772
- writeAttribute, 234
- writeBoolean, 99
- writeByte, 99
- writeCData, 234
- writeChar, 99
- writeCharacters, 229, 234
- writeChars, 98, 99
- writeComment, 234
- writeDouble, 97, 99
- writeDTD, 234
- writeEmptyElement, 234
- writeEndDocument, 229, 234
- writeEndElement, 229, 233
- writeExternal, 121, 123
- writeFixedString, 101
- writeFloat, 99
- writeInsert, 769, 773
- writeInt, 97, 99, 108
- writeLong, 99
- writeObject, 107, 113, 119, 120, 123
- writeReplace, 122, 123
- writeShort, 99
- writeStartDocument, 233
- writeStartElement, 233
- writeString, 134
- writeUTF, 98, 99
- metody
 - abstrakcyjne, 74
 - interfejsu
 - Action, 546
 - Appendable, 80
 - Attribute, 817
 - Attributes, 220
 - AttributeSet, 817
 - BaseStream, 48, 71
 - BasicFileAttributes, 139
 - Blob, 333
 - BufferedImageOp, 786
 - ButtonModel, 585
 - CachedRowSet, 347
 - CellEditor, 685
 - CharacterData, 190
 - CharSequence, 80
 - Clob, 333
 - Closeable, 80
 - Collection, 24, 72
 - CompiledScript, 450

- metody
 - interfejsu
 - Connection, 316, 331, 333, 342, 354, 359
 - ContentHandler, 220
 - DatabaseMetaData, 343, 354, 355, 360
 - DataInput, 98
 - DataOutput, 97, 99
 - Diagnostic, 441
 - DocPrintJob, 808, 817
 - Document, 189, 227
 - DoubleStream, 66
 - Element, 189, 227
 - EntityResolver, 198
 - ErrorHandler, 198
 - Flushable, 80
 - HttpClient.Builder, 291
 - HttpRequest.Builder, 292
 - HttpResponse, 292
 - IntStream, 64
 - Invocable, 449
 - Iterable, 30
 - JavaCompiler, 440
 - LayoutManager, 627
 - ListSelectionModel, 672
 - LoginModule, 476
 - LongStream, 65
 - MatchResult, 174
 - MouseListener, 606, 607
 - MutableTreeNode, 693
 - NamedNodeMap, 190
 - Node, 189, 227
 - NodeList, 190
 - Path, 132
 - PreparedStatement, 331
 - Principal, 469
 - PrintService, 807, 817
 - RandomGenerator, 63, 66
 - Readable, 80
 - ResultSet, 317, 332, 342, 355
 - ResultSetMetaData, 355
 - RowSet, 346
 - RowSetFactory, 348
 - Savepoint, 359
 - ScriptContext, 447
 - ScriptEngine, 446, 447
 - ScriptEngineFactory, 444
 - Statement, 316, 336, 337, 359
 - Stream, 28, 32–36, 48
 - Stream<T>, 23
 - Supplier<T>, 30
 - TableCellEditor, 684
 - TableCellRenderer, 684
 - TableColumnModel, 672
 - TableModel, 658, 670
 - Tool, 440
 - TreeCellRenderer, 705
 - TreeModel, 693, 719
 - TreeModelListener, 716, 720
 - TreeNode, 693, 700
 - TreeSelectionListener, 712
 - WindowListener, 540
 - XMLReader, 243
 - XMLStreamReader, 223
 - XMLStreamWriter, 233
 - XPath, 211
- klasy
 - AbstractButton, 585, 600–602, 605
 - AbstractTableModel, 656
 - AffineTransform, 750–752
 - AlphaComposite, 763
 - Arrays, 29
 - AttributesImpl, 243
 - BorderFactory, 586
 - Buffer, 151, 155
 - BufferedImage, 778
 - ButtonGroup, 585
 - ByteBuffer, 151
 - Channels, 261
 - CharBuffer, 153
 - Cipher, 497
 - CipherOutputStream, 500
 - Collator, 414
 - Collectors, 49, 52, 54, 59
 - Color, 524, 779
 - ColorModel, 779
 - Compilable, 449
 - Component, 510, 516, 524, 552, 574
 - Container, 568
 - Currency, 405
 - DataTruncation, 321
 - DateTimeFormatter, 386, 409
 - DefaultMutableTreeNode, 693, 701, 705
 - DefaultRowSorter<M, I>, 672
 - DefaultTreeCellRenderer, 705
 - DefaultTreeModel, 693, 701
 - DocumentBuilder, 188, 198, 227
 - DocumentBuilderFactory, 188, 199, 215
 - Double, 522
 - DriverManager, 314
 - Duration, 368
 - FileChannel, 151, 159
 - FileFilter, 649
 - FileInputStream, 151
 - FileLock, 160
 - FileOutputStream, 151
 - Files, 29, 134, 135, 137, 139, 144
 - FileTime, 139
 - FileView, 646, 649
 - FlowLayout, 568
 - Font, 529
 - FontMetrics, 530
 - ForwardingJavaFileManager, 442
 - Graphics, 532, 723, 754
 - Graphics2D, 524, 530, 722, 745, 746, 752, 754, 762
 - HttpClient, 291
 - HttpHeaders, 292
 - HttpRequest, 291
 - URLConnection, 283
 - IIOImage, 773
 - IIOServiceProvider, 772
 - ImageIcon, 511
 - ImageIO, 770
 - ImageReader, 771
 - ImageReaderWriterSpi, 772
 - ImageWriter, 772
 - InetAddress, 251
 - InetSocketAddress, 261
 - InputStream, 75
 - Instant, 367
 - JButton, 642
 - JCheckBox, 581
 - JComboBox, 591

- JComponent, 516, 530,
 547, 574, 588, 603, 613,
 642, 694
 JFileChooser, 648
 JFrame, 515, 600
 JLabel, 575
 JMenu, 599
 JMenuItem, 605, 607
 JOptionPane, 631–633
 JPasswordField, 576
 JPopupMenu, 603
 JRootPane, 642
 JScrollPane, 579
 JSlider, 596
 JTable, 670, 684
 JTextArea, 578
 JTextComponent, 572
 JTextField, 573
 JToolBar, 611, 612
 JTree, 692, 700, 711
 KeyGenerator, 498
 KeyStroke, 547
 LineBorder, 588
 LineMetrics, 529
 LocalDate, 372, 375, 387
 Locale, 396
 LocalTime, 376
 LoginContext, 468
 Matcher, 174
 MessageFormat, 416
 MouseEvent, 552, 603
 NameCallback, 475
 NumberFormat, 400
 Optional, 37, 39, 41, 45, 67
 OutputStream, 76
 PageFormat, 796
 PasswordCallback, 476
 Path2D, 737
 Path2D.Float, 736
 Pattern, 29, 173
 Period, 373
 Preferences, 560
 PrinterJob, 795
 PrintWriter, 88
 RandomAccessFile, 103,
 151, 160
 Raster, 778
 RectangularShape, 522
 ResourceBundle, 427
 RowFilter.Entry<M, I>, 673
 RowFilter<M, I>, 673
 RowSetProvider, 348
 SAXParseException, 198
 SAXParser, 219
 SAXParserFactory, 219
 Scanner, 30
 ScriptEngineManager, 444
 ServerSocket, 254
 SimpleFileVisitor, 144
 SimpleJavaFileObject, 442
 Socket, 248, 249, 257
 SocketChannel, 261, 265
 SoftBevelBorder, 587
 Spliterators, 29
 SQLException, 320
 StreamPrintServiceFactory,
 809
 StreamSupport, 29
 String, 29
 SummaryStatistics, 67
 SwingUtilities, 642
 System, 824
 TableColumn, 672, 684
 TableRowSorter, 672
 TableStringConverter, 673
 TemporalAdjusters, 375
 TextLayout, 754
 Toolkit, 511
 Transformer, 227
 TransformerFactory, 227
 TreePath, 700
 TreeSelectionEvent, 712
 URL, 273
 URLConnection, 273
 URLDecoder, 284
 URLEncoder, 284
 Warning, 321
 Window, 516
 WritableRaster, 779
 XMLInputFactory, 222
 XMLOutputFactory, 233
 XPathFactory, 211
 ZipEntry, 106
 ZipFile, 106
 ZipInputStream, 105
 ZipOutputStream, 105
 ZonedDateTime, 380, 387
 macierzyste, 818
 dostęp do pól instancji,
 833, 837
 dostęp do pól statycznych,
 836
 funkcje języka C, 819
 funkcje języka C++, 821
 konstruktory, 843
 łańcuchy znaków, 827
 obsługa błędów, 849, 853
 parametry numeryczne, 825
 przeciążanie
 identyfikatorów, 820
 rejestr systemu Windows,
 859
 składowe obiektu, 832
 sygnatury, 837
 tablice, 846, 848
 wartości zwracane, 825
 wywoływania alternatywne,
 844
 wywoływanie metod języka
 Java, 839, 844
 wywoływanie metod
 obiektów, 839
 wywoływanie metod
 statycznych, 842
 sygnatury, 837
 typu get-set, 510
 Microsoft ASP, 275
 MIME, 133, 764, 771, 806
 mnemoniki, 604
 model, 563, 566
 drzewa, 687, 712
 kolorów RGB, 755, 775
 kolorów sRGB, 775
 tabeli, 655
 moduł
 JAAS, 469
 logowania, 467, 469, 470
 hasła, 470
 logowanie, 475
 uwierzytelniania, 467
 modyfikator transient, 119
 modyfikatory dat, 374
 mostek JDBC/ODBC, 302
 MVC, Model-View-Controller, 563
 mysz, 547
 kursory, 549

N

nagłówki żądań HTTP, 270
 nasłuchiwać zdarzeń, listener,
 534, 554
 nazwy rodziny czcionek, 524
 negatyw obrazu, 784
 normalizacja, 411
 numeryczne parametry metod,
 825

O

obiekt
 CompilationTask, 434
 obsługi, handler, 294
 System.in, 74
 zdarzeń, event object, 533
 obiekty
 dostęp do składowych, 832
 klonowanie, 126
 odczytywanie, 108
 pośredniczące, proxy, 122
 serializowalne, 107
 strumień wejścia, 73
 strumień wyjścia, 73
 zapisu i odczytu, 764
 zapisywanie, 108
 obramowanie, 585
 obrazy, 530
 efekt rozmycia, 785
 filtrowanie, 780
 model kolorów, 775
 negatyw, 784
 obrót, 781
 operacje, 773
 tworzenie, 774, 776
 typu TYPE_BYTE_GRAY, 777
 typu TYPE_INT_ARGB, 774
 wykrywanie krawędzi, 786
 wyświetlanie, 530
 obrót, 748, 781
 obrys tekstu, 754
 obsługa
 błędów, 849, 853
 zdarzeń, 555
 akcje, 541
 hierarchia zdarzeń, 552
 interfejs nasłuchu, 533
 klasy, 533, 554

 klasy adaptacyjne, 540
 klawiatury, 534
 myszy, 547
 obiekt zdarzeń, 533
 obszar tekstowy, text area, 571,
 576
 ODBC, 302
 odcisk palca, fingerprint, 113,
 123, 477
 odczyt
 bajtów, 74
 danych binarnych, 97
 dużych obiektów, 331
 plików, 133
 plików graficznych, 763
 plików z sekwencjami
 obrazów, 768
 rekordu, 101
 tekstu, 85
 znaków, 96
 odczytywanie obiektów
 serializowalnych, 107, 124
 odszyfrowywanie danych, 499
 okna dialogowe
 akcesorium podglądu, 647
 dane wejściowe, input dialog,
 628
 klawisz wyzwolenia, trigger
 key, 638
 komunikaty, 629
 modalne, modal dialog box,
 627, 634
 niemodalne, modeless dialog
 box, 627
 opcje, 628
 panel główny, root pane, 638
 potwierdzenia, confirmation
 dialog, 630
 przycisk domyślny, default
 button, 638
 przyciski, 629
 ramka nadrzędna, 633
 tworzenie, 633
 ustawień strony, 792, 795
 wyboru pliku, file chooser,
 642, 645
 wydruku, 789
 wymiana danych, 636
 wyświetlanie, 638

operacje
 leniwe, 22
 na obrazach, 773
 redukcji, 35, 60
 opróżnienie bufora, 75
 automatyczne, 87

P

pakiet
 OpenSSL, 489
 Swing, 651
 pakiety, 456
 panel przewijany, scroll pane, 577
 parser, *Patrz także* kontrola
 poprawności
 SAX, 181, 215
 StAX, 221
 zapis dokumentu xml, 228
 XML
 DOM, 181, 184
 implementacja, 194
 XML Schema, 202
 parsery, 96
 drzewiaste, 181
 strumieniowe, 181, 215
 pasek
 menu, 597
 narzędzi, toolbar, 610
 piaskownica, 490
 pliki, 130
 blokowanie dostępu, 156, 157
 class, 453
 graficzne
 animacje GIF, 770
 formaty, 763
 obiekty zapisu i odczytu, 764
 odczyt i zapis, 763, 767
 sekwencje obrazów, 768
 informacje o pliku, 138
 JAR, 105, 266, 311
 kanał dostępu, 147
 kopiowanie, 136
 mapowanie w pamięci, 146
 MIME, 133
 o swobodnym dostępie, 100
 odczyt i zapis, 133
 okno wyboru, 642
 przenoszenie, 136

- serializacji obiektów, 112, 114, 116, 119
- standardowe opcje, 137
- suma kontrolna, 148
- tekstowe, 420
- tryb mapowania, 147
- tworzenie, 135
- usuwanie, 136
- właściwości, 176, 425
- wzorce filtrowania, 142
- XML, 177, 180
- XML Schema, 191
- zasoby, 423
- ZIP, 104, 105, 145
- źródłowe, 435
- pobieranie wartości kluczy, 336
- pochylenie, 748
- poczta elektroniczna, 244, 293, 297
 - SMTP, 297
 - uwierzytelnianie, 298
 - wysyłanie, 297
- podpis cyfrowy, 477–481, 487
 - aplety, 477
 - generator liczb losowych, 495
 - podpisywanie wiadomości, 480
 - skrót wiadomości, 477
 - weryfikacja, 483
- podpisywanie
 - certyfikatów, 487
 - kodu, 452, 490
 - wiadomości, 480
- pola, 737
 - add, 737
 - exclusiveOr, 737
 - haseł, 575
 - intersect, 737
 - kombi, 588
 - operacje, 737
 - subtract, 737
 - tekstowe, text field, 571, 572
 - tworzenie, 738
 - wyboru, checkbox, 579
- połączenia
 - przygotowane, 325
 - SQL, 315
- połączenia sieciowe
 - częściowo zamknięte, 257, 262
 - gniazda, 247
 - klient, 245
 - porty, 245
 - serwer, 244
 - strumienie, 247
 - TCP, 248
 - UDP, 248
 - URL, 266
 - wysyłanie danych do formularzy, 275
- połączenie
 - z bazą danych, 312
 - z serwerem, 247
- porty, 245
- potok rysowania, 722
- preferencje użytkownika
 - API Preferences, 554
- problem uwierzytelniania, 485
- procedury
 - ładujące klasy, 455, 457, 462
 - nasłuchowe, 538
 - składowane, 335
- procesor XSLT, 236
- program
 - drzewo dziedziczenia, 702, 707
 - drzewo inspekcji obiektów, 712
 - ExecSQL, 322
 - figury Javy 2D, 728, 735
 - filtrowanie nazw plików, 765
 - filtrowanie wierszy, 667
 - formatowanie daty, 407
 - jarsigner, 485
 - javap, 838
 - keytool, 483, 487
 - ObjectRefTest, 116
 - obracanie obrazu, 781
 - podgląd wydruku, 798
 - proces szyfrowania, 460
 - przeglądarka klas, 706
 - przekształcanie wartości kolorów, 777
 - reguły składania obrazów, 759
 - serialver, 123
 - szyfr Cezara, 459
 - tworzenie tabeli, 657
 - usługi drukowania, 809
 - uwierzytelnianie użytkowników, 471
 - ViewDB, 353
 - wybieranie śladu pędzla, 740, 744
 - wyświetlanie i drukowanie, 793
 - wyświetlanie tabeli, 652
- programowanie
 - baz danych, 301
 - transakcji, 357
- prostokąt, 518, 725
- protokoły
 - HTTP, 270, 284, 286
 - HTTP/2, 284
 - SMTP, 297
 - SSL, 262
 - TCP, 248
 - TLS, 262
 - UDP, 248
- przekształcenia
 - afiniczne, 750, 780
 - układu współrzędnych, 721
 - macierz przekształceń, 750
 - obrót, 748
 - pochylenie, 748
 - przesunięcie, 748
 - skalowanie, 748
 - składanie przekształceń, 748
- XSL, 234
 - Java, 238
 - style, 235
 - szablon przekształcenia, 235
 - XSLT, 236
- przełącznik, radio button, 582
- powiadamianie o zdarzeniach, 583
- przestrzeń nazw, 212, 456
 - definiowanie, 214
 - prefiks xsd:, 200, 213
 - URI, 212
 - XML, 212
- przesunięcie, 748
- przetwarzanie
 - kodu macierzystego, 823
 - asynchroniczne, 287
 - dokumentów XML, 182
- przewijalne zbiory wyników, 337
- przezroczystość, 755
- wartość alfa, 755

przycinanie
 określanie obszaru, 754
 przyciski, 534
 przyrostowe tworzenie obrazów,
 773
 pula połączeń, 362
 punkt kodowy, 92
 punkty kontrolne transakcji, 357

R

ramka, frame, 506
 pozycjonowanie, 508
 warstwy, 512
 wewnętrzna struktura, 513
 wyświetlanie, 508
 redukcje strumieni, 35, 60
 referencje, 834, 863
 refleksja, 718
 reguły Portera-Duffa, 756, 757,
 761
 rejestr systemu Windows, 859
 implementacja dostępu, 861
 interfejs dostępu, 861
 klucze, 859
 pobieranie wartości, 862
 przeglądanie nazw kluczy, 863
 uchwyt klucza, 863
 węzły, 859
 zapisywanie wartości, 863
 rekordy, 304
 o stałej długości, 107
 repozytorium Preferences, 556
 RGB, 755, 775
 rodziny czcionek, 524
 rozkład
 BoxLayout, 613
 brzegowy, border layout, 568,
 613
 ciągły, flow layout, 613
 CircleLayout, 624
 GridBagLayout, 613, 614
 kołowy, 624
 siatkowy, grid layout, 570, 613
 sprężynowy, spring layout, 613
 SpringLayout, 613
 rozmiar ekranu, 510
 rozmycie obrazu, 785
 rozwiązywanie klasy, 453

rysowanie
 figur, 721, 723
 figur 2D, 516
 komórek, 674
 potok, 722
 węzłów, 703
 rzutowanie, 517

S

SAX, Simple API for XML, 181,
 215
 wyszukiwanie elementów, 217
 sekwencje
 obrazów, 768
 sterujące, 334
 separatory, 86, 89
 serializacja obiektów, 107, 109
 format pliku, 112
 klonowanie, 126
 metody, 119, 121, 122
 pola pomijane, 119
 wersje kompatybilne, 123
 zapisywanie i wczytywanie, 107
 serwer, 244, 303
 aplikacji, 362
 FTP, 270
 gniazda, 252
 HTTP, 252, 293
 API serwera, 293
 filtry, 295
 obiekty obsługi, 294
 wiersz poleceń, 293
 implementacja, 251
 obsługa wielu klientów, 254
 plików
 uruchamianie, 293
 pocztowy, 298
 połączenia, 244
 wątki, 255
 WWW, 244, 276
 wysyłanie danych do
 formularzy, 275
 serwlety, 275, 362, 437
 SGML, Standard Generalized
 Markup Language, 178
 SHA, Secure Hash Algorithm, 113
 sieć, 244
 przesyłanie danych, 252

silnik
 Rhino, 448
 skryptowy, 443
 skalowanie, 748
 skład tekstów, 526
 składanie
 obrazów
 projektowanie reguły, 756
 reguły Portera-Duffa, 756,
 757
 przekształceń, 748, 749
 składnica kluczy, key store, 262
 składowe obiektu, 832
 skrót
 hasła, 491
 wiadomości, 477
 skróty klawiaturowe, 545
 skrypty, 433, 442
 arkusze skryptowe, 450
 CGI, 275
 kompilacja, 449
 przekierowanie wejścia
 i wyjścia, 446
 wybór silnika skryptowego,
 443
 wykonywane współbieżnie, 445
 wykonywanie, 444
 wywoływanie metod
 skryptów, 447
 słowniki
 klawiaturowe, 544
 wejściowe, 544
 słowo kluczowe
 extern, 821
 native, 820
 SMTP, Simple Mail Transport
 Protocol, 297
 sortowanie
 porządek alfabetyczny, 409
 splot, 785, 786
 spójność bazy danych, 356
 SQL, 301, 304, 315
 analiza wyjątków, 319
 DDL, 317
 funkcje, 309
 funkcje skalarne, 334
 łączenie tabel, 307
 metadane, 348
 modyfikacja danych, 308

- obiekty binarne BLOB, 331
- obiekty znakowe CLOB, 331
- polecenia, 307–309, 315–318
- polecenia przygotowane, 325
- połączenia zewnętrzne, 334
- procedury składowane, 335
- sekwencje sterujące, 334
- tabele, 361
- tworzenie tabel, 309
- typy danych, 309, 360
- typy wyjątków, 320
- wypełnianie bazy danych, 321
- zapytania, 301, 305, 307, 325
- zbiory wyników, 318, 335
 - aktualizowalne, 339
 - przewijalne, 337
- SSL, Secure Socket Layer, 262
- stała serialVersionUID, 123
- stałe interfejsu Action, 542
- standard MIME, 764
- sterowniki JDBC, 302
 - typ 1, 302
 - typ 2, 302
 - typ 3, 302
 - typ 4, 302
- strona
 - kodowa, code page, 95
 - WWW, 275
- struktura dokumentu XML, 179
- strumienie, 21, 247, 278
 - a kolekcje, 22
 - danych typów prostych, 62
 - iteracje, 21
 - katalogów, 141
 - łączenie, 32
 - niepewne, tainted, 89
 - obiektów, 107
 - operacje, 21
 - pobieranie, 32
 - redukcje, 35, 60
 - równoległe, 67
 - sortowanie, 34
 - szyfrujące, 500
 - tworzenie, 24
 - usługi drukowania, 808
 - wejścia-wyjścia, 73–160
 - hierarchia, 78
 - łączenie filtrów, 81
 - odczyt i zapis bajtów, 74
 - zapis obiektów, 89
 - zapis w formacie binarnym, 97
 - zapis w formacie tekstowym, 84, 89
 - zwracanie zawartości w mapach, 50
 - zwracanie zawartości, 45
- suma kontrolna pliku, 148
- suwak, slider, 591
- podziałka, ticks, 592
- Swing, 505
 - akceleratory, 605
 - czcionki, 524
 - dodawanie komponentów, 569
 - figury 2D, 516
 - ikony, 508
 - Java2D, 516
 - JButton, 565
 - JCheckBox, 580
 - JCheckBoxMenuItem, 601
 - JComboBox, 588
 - JFileChooser, 642
 - JFrame, 506, 508
 - JLabel, 574
 - JMenuBar, 597
 - JMenuItem, 604
 - JOptionPane, 628
 - JPanel, 570
 - JPasswordField, 575
 - JPopupMenu, 603
 - JRadioButton, 582, 585
 - JScrollPane, 577
 - JSlider, 591
 - JTable, 651
 - JTextArea, 571, 576
 - JTextField, 571, 572
 - JToolBar, 610, 612
 - JTree, 685
 - kolory, 523
 - komponenty, 562
 - konfiguracja komponentów, 507
 - kontener, 567
 - nazwy klas, 506
 - niestandardowi zarządcy rozkładu, 623
 - obsługa zdarzeń, 513
 - pozycjonowanie ramki, 508
 - rozkład brzegowy, 568
 - rozkład GridBagLayout, 614
 - rozkład siatkowy, 570
 - rozmiar ekranu, 510
 - rysowanie, 512
 - tekst w pasku tytułu, 508
 - warstwy ramki, 512
 - wątek dystrybucji zdarzeń, 507
 - wprowadzanie tekstu, 571
 - wymuszanie ponownego rysowania ekranu, 513
 - wysyłanie zdarzeń, 507
 - wyświetlanie informacji w komponencie, 512
 - zamykanie ramki aplikacji, 508
 - zarządca rozkładu grupowego, 614
 - zarządzanie rozkładem, 566, 613
- sygnatury metody, 837
- system plików ZIP, 145
- szyfrowanie, 459, 492
 - AES, 493, 494, 500
 - algorytmy, 492, 493
 - DES, 493, 494
 - dopełnienie ostatniego bloku, 494
 - klasa Cipher, 492
 - klucze, 493, 494, 500
 - pliki klas, 460
 - RSA, 481, 501
 - strumienie, 499
 - symetryczne, 492

Ś

- ścieżka drzewa, 694
- ścieżki
 - bezwzględne, 130
 - relatywizacja, 131
 - rozwiązywanie, 131
 - względne, 130
- ślad pędzla, 721, 738
 - połączenia, 739
 - szerokość, 738
 - wartość graniczna, 739
 - wzór przerywany, 740
 - zakończenia, 738

T

tabele, 304, 651
 dostęp do kolumn, 660
 filtrowanie wierszy, 664
 kolumny, 659
 model tabeli, 652, 655
 obiekty rysujące, 659, 675
 przesuwanie kolumny, 654
 rysowanie komórek, 674
 sortowanie wierszy, 663
 szerokość kolumn, 654
 tworzenie, 309
 tworzenie edytora komórek, 681
 ukrywanie kolumn, 666
 widoki, 654
 wiersze, 659
 wybór kolumn, 662
 wybór komórek, 662
 wybór wierszy, 662
 wysokość komórek, 662
 wyświetlanie kolumn, 666
 zmiana rozmiaru kolumn, 660
 zmiana rozmiaru wierszy, 662
 tablica skrótów, hash table, 593
 tablice, 846
 C++, 846
 TCP, Transmission Control Protocol, 248
 tekst, 571
 operacje wejścia-wyjścia, 84
 paska tytułu, 508
 wczytywanie, 85
 zapisywanie, 86
 text/plain, 274
 TLS, Transport Layer Security, 262
 tokeny, 86, 89
 Tomcat, 362
 transakcje, 356
 aktualizacje wsadowe, 358
 automatyczne zatwierdzanie, 357
 odwołanie, 356
 punkty kontrolne, 357
 tworzenie, 356
 tworzenie
 akceleratorów, 605
 certyfikatów, 483

 czcionek, 753
 dokumentów XML, 224
 drzewa DOM, 225
 grafiki, 720
 interfejsów użytkownika, 504
 katalogów, 135
 kształtów, 724
 menu, 597
 obiektów typu Optional, 41
 obrazów, 774, 776
 okien dialogowych, 633
 plików, 135
 ramek, 506
 rysunków, 720
 strumieni, 24
 transakcji, 356
 zbiorów rekordów, 344

typ

 jarray, 846
 jclass, 843
 jfieldID, 833
 jobject, 846
 jstring, 843
 jvalue, 845
 MIME, 443
 Optional, 37
 pobieranie wartości, 37
 potoki wartości, 39
 przekształcanie wartości w strumień, 43
 tworzenie obiektów, 41
 użycie flatMap, 42
 używanie wartości, 38, 40
 xsd:boolean, 200
 xsd:int, 200
 xsd:string, 200

typy

 danych
 C, 825
 Java, 360, 825
 proste, 62
 SQL, 309, 360
 tablic
 hierarchia dziedziczenia, 846
 w języku C, 846
 w języku Java, 846
 wyjątków SQL, 320

U

UDP, User Datagram Protocol, 248
 układ współrzędnych, 747
 Unicode, 77, 98, 390, 411
 uporządkowanie
 łańcuchów, 411
 słownikowe, 410
 URI, Uniform Resource Identifier, 212, 266, 285, 813
 absolutny, 267
 hierarchiczny, 267
 identyfikatory, 268
 nieprzenikalny, 267
 relatywizacja, 268
 rozwiązywanie, 268
 specyfikacja, 267
 względny, 267
 URL, Uniform Resource Locator, 192, 212, 266, 285, 310
 nagłówki żądań, 270
 pobieranie informacji, 268
 URN, Uniform Resource Name, 266
 uruchamianie bazy danych, 311
 usługi
 drukowania, 805
 rodzaje dokumentów, 806, 807
 strumienie, 808
 JNDI, 362
 sieciowe, 244
 UTF, Unicode Text Format, 98
 UTF-8, 85, 92, 98, 827
 UTF-16, 77, 84, 92, 828
 UTF-32, 828
 uwierzytelnianie, 298
 użytkowników, 466
 JAAS, 466
 moduły, 468
 nadzorczy, 467
 podmiot, 467
 wiadomości, 485
 klucze, 486
 łańcuch zaufania, 487
 podpis zaufanego pośrednika, 486
 zaufany pośrednik, 486

W

- waluta, 397, 398, 404
 - formatowanie, 404
 - identyfikatory, 404
- warstwa treści, content pane, 512
- wartości Optional, 35
 - korzystanie, 38
 - pobieranie, 37
- wartość null, 35, 40
- wątek dystrybucji zdarzeń, 507
- wątki, 255
- wczytywanie tekstu, 85
- Web, 244
- wejście-wyjście, 73–160, 446
- weryfikacja
 - podpisu cyfrowego, 482
 - podpisu plików JAR, 485
- weryfikator kodu, 462, 465
- wiązania zmiennych, 445
- widok, view, 563
- wielokąty, 723
- wiersz poleceń
 - uruchamianie serwera, 293
- wiersze, 659
 - filtrowanie, 664
 - sortowanie, 663
 - wstawiania, 341
 - wybijanie, 662
 - zmienianie rozmiaru, 662
- Windows
 - rejestr, 859
- własność, property, 510
- właściwości
 - interfejsu ButtonModel, 565
 - klasy DecimalFormat, 402
 - klasy DecimalFormatSymbol, 403
- właściwość systemowa
 - file.encoding, 95
 - native.encoding, 95
 - stderr.encoding, 95
 - stdout.encoding, 95
- wprowadzanie tekstu, 571
- wskaźnik
 - env, 828
 - jvm, 855
 - pliku, 100
- współrzędne
 - ekranowe, 747
 - użytkownika, 747
- wydawca zawartości, body
 - publisher, 285
- wyjątek
 - ArrayIndexOutOfBoundsException
 - ↳ Exception, 850
 - ArrayStoreException, 850
 - EOFException, 849
 - FileNotFoundException, 280
 - IllegalArgumentException, 850
 - IllegalStateException, 50, 53, 768
 - IndexOutOfBoundsException, 768
 - InterruptedException, 249
 - InvalidObjectException, 126
 - IOException, 80, 86
 - MissingResourceException, 424
 - NoSuchElementException, 40
 - NullPointerException, 850
 - OverlappingFileLockException, 157
 - ParseException, 399
 - PrinterException, 789
 - ReadOnlyBufferException, 147
 - SAXParseException, 198
 - SocketTimeoutException, 274
 - SQLException, 319, 357
 - SyncProviderException, 346, 348
 - UncheckedIOException, 86
 - UnknownHostException, 247
- wyjątki sprawdzane, checked exceptions, 86
- wykonywanie zapytań SQL, 325
- wykrywanie krawędzi, 786
- wymiana danych, 636
- wypełnianie
 - figur, 523
 - obszaru, 721, 745
 - gradient, 745
 - kolor, 746
 - obrazek wzorca, 746
 - prostokąt zakotwiczenia, 746
 - tekstura, 746
- wyrażenia regularne, 160
 - flagi, 172
 - grupy, 168
 - nazwy klas znaków, 164
 - składnia, 160–164
 - testowanie dopasowania, 165, 169
 - zastępowanie dopasowań, 171
 - znajdowanie wszystkich dopasowań, 166
- wyrażenie lambda, 374
- wysyłanie
 - danych do formularzy, 275
 - poczty elektronicznej, 297
- wyświetlanie
 - informacji w komponencie, 512
 - obrazów, 530
 - ramki, 508
- wywołania zwrotne, callback, 875
- wywoływanie
 - funkcji języka C, 819
 - parametry, 822
 - metod języka Java, 839, 844
 - metod obiektów, 839
 - metod statycznych, 842
- wzorzec
 - Budowniczy, 285
 - model-widok-kontroler, 562
 - wzajemne relacje, 565
 - model-widok-nadzorca, 686

X

- XML, *Patrz* dokumenty XML
- XML Schema, 191, 199, 213
 - atributy, 201
 - definicje elementów, 201
 - parsowanie dokumentu XML, 202
 - powtórzenia elementów, 201
 - przestrzeń nazw, 200
 - typ elementu, 200
 - typy proste, 200
 - typy złożone, 200
 - funkcje, 209
 - wyrażenia, 208, 209
 - zbiór węzłów, 209

XSL
 reguły przetwarzania, 234
 Schema Definition, 200
 XSLT, XML Style Sheet
 Transformation, 226, 236

Z

zadania drukowania, 789
 zapis
 bajtów, 74
 danych binarnych, 97
 dokumentu XML, 228
 dużych obiektów, 331
 kodów bajtowych, 436
 obiektów, 89
 obiektów serializowalnych, 107
 plików, 133
 plików graficznych, 763, 767
 plików z sekwencjami
 obrazów, 768
 rekordu, 101
 tekstu, 86
 w formacie binarnym, 148
 zapytania SQL, 301, 305, 307, 325
 zarządca rozkładu
 brzegowego, border layout
 manager, 568
 ciągłego, flow layout manager,
 566
 grupowego, group layout
 manager, 614

niestandardowy, 623
 obsługa ograniczeń, 611, 616
 siatkowego, grid layout
 manager, 570, 613
 zarządzanie połączeniami, 361
 zasada
 Gödla, 462
 składania obrazów, 721
 zasoby, 423
 lokalizacja, 424
 zbiory, 46
 atrybutów drukowania, 813
 rekordów, 344
 aktualizowalne, 339
 buforowane, 345
 interfejsy, 344, 345
 kolumny, 354
 modyfikacja, 346
 sprawdzanie, 346
 wyników
 aktualizowalne, 337
 przewijalne, 337
 wartości parametru type,
 338
 znaków, 92, 419, 420
 zbiór Mandelbrota, 776
 zdarzenia, 507, 513, 533
 AWT, 552
 dotyczące drzew, 706
 hierarchia klas, 552
 interfejs nasłuchu, 533

klasy, 539, 552, 591
 myszy, 547
 okna, 539
 zestawy znaków, 420
 ZIP, 104
 znacznik BOM, 93, 422
 znaczniki XML, 178
 znak
 backspace, 165
 echa, 575
 lewego ukośnika, 81, 142
 prawego ukośnika, 81
 znaki
 kodowanie, 92
 końca wiersza, 421
 wczytywanie, 96

Ż

źródło danych, 362
 JDBC, 310
 JNDI, 362

Z

żądania certyfikatu, 488
 żądanie
 DELETE, 285
 GET, 277, 285
 HEAD, 285
 POST, 280, 285
 PUT, 285

PROGRAM PARTNERSKI

— GRUPY 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
Helion 

JAVA:

sztuka programowania na najwyższym poziomie!

To jest najlepsze kompendium wiedzy o języku Java i o całym ekosystemie Javy!

— **Andrew Binstock**, „Java Magazine”

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

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

W książce ponadto:

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

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

Helion 

 helion.pl

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

KOD KORZYŚCI
Sięgnij po więcej! ▶



ISBN 978-83-289-3004-9



9 788328 930049

Cena: 169,00 zł

