

JAVA®

Techniki zaawansowane

WYDANIE XI



CAY S. HORSTMANN

Tytuł oryginału: Core Java, Volume II – Advanced Features, 11th Edition

Tłumaczenie: Piotr Rajca

na podstawie „Java. Techniki zaawansowane. Wydanie IX” w tłumaczeniu Jaromira Senczyka

ISBN: 978-83-283-6066-2

Authorized translation from the English language edition, entitled CORE JAVA VOLUME II – ADVANCED FEATURES, 11th Edition, by HORSTMANN, CAY S. published by Pearson Education, Inc, publishing as Prentice Hall, Copyright © 2019 Pearson Education Inc.

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

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 SA, Copyright © 2020.

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

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/jatz11>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/jatz11.zip>

Printed in Poland.

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

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

Spis treści

| | |
|--|-----------|
| Wstęp | 11 |
| Podziękowania | 15 |
| Rozdział 1. Strumienie | 17 |
| 1.1. Od iteracji do operacji na strumieniach | 18 |
| 1.2. Tworzenie strumieni | 20 |
| 1.3. Metody filter, map oraz flatMap | 26 |
| 1.4. Pobieranie podstrumieni i łączenie strumieni | 27 |
| 1.5. Inne przekształcenia strumieni | 29 |
| 1.6. Proste operacje redukcji | 30 |
| 1.7. Typ Optional | 32 |
| 1.7.1. Pobieranie wartości Optional | 32 |
| 1.7.2. Korzystanie z wartości Optional | 33 |
| 1.7.3. Potoki wartości opcjonalnych | 33 |
| 1.7.4. Jak nie należy używać wartości opcjonalnych | 35 |
| 1.7.5. Tworzenie obiektów typu Optional | 36 |
| 1.7.6. Łączenie funkcji zwracających wartości opcjonalne przy użyciu flatMap | 36 |
| 1.7.7. Przekształcanie wartości opcjonalnej w strumień | 37 |
| 1.8. Gromadzenie wyników | 40 |
| 1.9. Gromadzenie wyników w mapach | 44 |
| 1.10. Grupowanie i podział | 48 |
| 1.11. Kolektory przetwarzające | 50 |
| 1.12. Operacje redukcji | 54 |
| 1.13. Strumienie danych typów prostych | 57 |
| 1.14. Strumienie równoległe | 62 |
| Rozdział 2. Wejście i wyjście | 67 |
| 2.1. Strumienie wejścia-wyjścia | 67 |
| 2.1.1. Odczyt i zapis bajtów | 68 |
| 2.1.2. Zoo pełne strumieni | 71 |
| 2.1.3. Łączenie filtrów strumieni wejścia-wyjścia | 74 |
| 2.1.4. Strumienie tekstowe | 78 |

| | |
|--|------------|
| 2.1.5. Zapisywanie tekstu | 79 |
| 2.1.6. Wczytywanie tekstu | 81 |
| 2.1.7. Zapis obiektów w formacie tekstowym | 82 |
| 2.1.8. Zbiory znaków | 85 |
| 2.2. Odczyt i zapis danych binarnych | 88 |
| 2.2.1. Interfejsy DataInput i DataOutput | 88 |
| 2.2.2. Strumienie plików o swobodnym dostępie | 90 |
| 2.2.3. Archiwa ZIP | 94 |
| 2.3. Strumienie obiektów i serializacja | 97 |
| 2.3.1. Zapisywanie i wczytywanie obiektów serializowalnych | 97 |
| 2.3.2. Format pliku serializacji obiektów | 101 |
| 2.3.3. Modyfikowanie domyślnego mechanizmu serializacji | 107 |
| 2.3.4. Serializacja singletonów i wyliczeń | 109 |
| 2.3.5. Wersje | 110 |
| 2.3.6. Serializacja w roli klonowania | 113 |
| 2.4. Zarządzanie plikami | 115 |
| 2.4.1. Ścieżki dostępu | 115 |
| 2.4.2. Odczyt i zapis plików | 118 |
| 2.4.3. Tworzenie plików i katalogów | 119 |
| 2.4.4. Kopiowanie, przenoszenie i usuwanie plików | 120 |
| 2.4.5. Informacje o plikach | 122 |
| 2.4.6. Przeglądanie zawartości katalogu | 124 |
| 2.4.7. Stosowanie strumieni katalogów | 125 |
| 2.4.8. Systemy plików ZIP | 129 |
| 2.5. Mapowanie plików w pamięci | 130 |
| 2.5.1. Wydajność plików mapowanych w pamięci | 130 |
| 2.5.2. Struktura bufora danych | 136 |
| 2.6. Blokowanie plików | 138 |
| 2.7. Wyrażenia regularne | 140 |
| 2.7.1. Składnia wyrażeń regularnych | 141 |
| 2.7.2. Dopasowywanie wyrażeń regularnych do łańcucha | 145 |
| 2.7.3. Znajdowanie wielu dopasowań | 148 |
| 2.7.4. Podział w miejscach wystąpienia separatora | 150 |
| 2.7.5. Zastępowanie dopasowań | 150 |
| Rozdział 3. Język XML | 155 |
| 3.1. Wprowadzenie do języka XML | 156 |
| 3.2. Struktura dokumentu XML | 158 |
| 3.3. Parsowanie dokumentów XML | 160 |
| 3.4. Kontrola poprawności dokumentów XML | 169 |
| 3.4.1. Definicje typów dokumentów | 170 |
| 3.4.2. XML Schema | 178 |
| 3.4.3. Praktyczny przykład | 180 |
| 3.5. Wyszukiwanie informacji i XPath | 186 |
| 3.6. Przestrzenie nazw | 190 |
| 3.7. Parsery strumieniowe | 193 |
| 3.7.1. Wykorzystanie parsera SAX | 193 |
| 3.7.2. Wykorzystanie parsera StAX | 198 |
| 3.8. Tworzenie dokumentów XML | 202 |
| 3.8.1. Dokumenty bez przestrzeni nazw | 202 |
| 3.8.2. Dokumenty z przestrzenią nazw | 203 |

| | |
|---|-----|
| 3.8.3. Zapisywanie dokumentu | 203 |
| 3.8.4. Zapis dokumentu XML za pomocą parsera StAX | 206 |
| 3.8.5. Przykład: tworzenie pliku SVG | 210 |
| 3.9. Przekształcenia XSL | 212 |

Rozdział 4. Programowanie aplikacji sieciowych221

| | |
|--|-----|
| 4.1. Połączenia z serwerem | 221 |
| 4.1.1. Stosowanie programu telnet | 221 |
| 4.1.2. Nawiązywanie połączenia z serwerem z wykorzystaniem Javy | 224 |
| 4.1.3. Limity czasu gniazd | 225 |
| 4.1.4. Adresy internetowe | 227 |
| 4.2. Implementacja serwerów | 228 |
| 4.2.1. Gniazda serwera | 229 |
| 4.2.2. Obsługa wielu klientów | 231 |
| 4.2.3. Połączenia częściowo zamknięte | 235 |
| 4.2.4. Przerwywanie działania gniazd sieciowych | 236 |
| 4.3. Połączenia wykorzystujące URL | 242 |
| 4.3.1. URL i URI | 242 |
| 4.3.2. Zastosowanie klasy URLConnection do pobierania informacji | 244 |
| 4.3.3. Wysyłanie danych do formularzy | 251 |
| 4.4. Klient HTTP | 259 |
| 4.5. Wysyłanie poczty elektronicznej | 266 |

Rozdział 5. Programowanie baz danych: JDBC271

| | |
|---|-----|
| 5.1. Architektura JDBC | 272 |
| 5.1.1. Typy sterowników JDBC | 272 |
| 5.1.2. Typowe zastosowania JDBC | 274 |
| 5.2. Język SQL | 274 |
| 5.3. Instalacja JDBC | 280 |
| 5.3.1. Adresy URL baz danych | 280 |
| 5.3.2. Pliki JAR zawierające sterownik | 281 |
| 5.3.3. Uruchamianie baz danych | 281 |
| 5.3.4. Rejestracja klasy sterownika | 282 |
| 5.3.5. Nawiązywanie połączenia z bazą danych | 283 |
| 5.4. Stosowanie poleceń SQL | 285 |
| 5.4.1. Wykonywanie poleceń SQL | 285 |
| 5.4.2. Zarządzanie połączeniami, poleceniami i zbiorami wyników | 289 |
| 5.4.3. Analiza wyjątków SQL | 289 |
| 5.4.4. Wypełnianie bazy danych | 292 |
| 5.5. Wykonywanie zapytań | 295 |
| 5.5.1. Polecenia przygotowane | 296 |
| 5.5.2. Odczyt i zapis dużych obiektów | 301 |
| 5.5.3. Sekwencje sterujące | 303 |
| 5.5.4. Zapytania o wielu zbiorach wyników | 305 |
| 5.5.5. Pobieranie wartości kluczy wygenerowanych automatycznie | 306 |
| 5.6. Przewijalne i aktualizowalne zbiory wyników zapytań | 306 |
| 5.6.1. Przewijalne zbiory wyników | 307 |
| 5.6.2. Aktualizowalne zbiory rekordów | 309 |
| 5.7. Zbiory rekordów | 313 |
| 5.7.1. Tworzenie zbiorów rekordów | 313 |
| 5.7.2. Buforowane zbiory rekordów | 314 |

| | |
|--|------------|
| 5.8. Metadane | 317 |
| 5.9. Transakcje | 326 |
| 5.9.1. Programowanie transakcji w JDBC | 326 |
| 5.9.2. Punkty kontrolne | 327 |
| 5.9.3. Aktualizacje wsadowe | 327 |
| 5.9.4. Zaawansowane typy języka SQL | 330 |
| 5.10. Zaawansowane zarządzanie połączeniami | 331 |
| Rozdział 6. API dat i czasu | 333 |
| 6.1. Oś czasu | 334 |
| 6.2. Daty lokalne | 338 |
| 6.3. Modyfikatory dat | 343 |
| 6.4. Czas lokalny | 344 |
| 6.5. Czas strefowy | 346 |
| 6.6. Formatowanie i parsowanie | 351 |
| 6.7. Współdziałanie ze starym kodem | 355 |
| Rozdział 7. Internacjonalizacja | 357 |
| 7.1. Lokalizatory | 358 |
| 7.1.1. Dlaczego stosuje się lokalizatory? | 358 |
| 7.1.2. Określanie lokalizatorów | 359 |
| 7.1.3. Lokalizator domyślny | 361 |
| 7.1.4. Nazwa lokalizatora | 362 |
| 7.2. Formaty liczb | 364 |
| 7.2.1. Formatowanie wartości liczbowych | 364 |
| 7.2.2. Waluty | 369 |
| 7.3. Data i czas | 371 |
| 7.4. Porządek alfabetyczny i normalizacja | 377 |
| 7.5. Formatowanie komunikatów | 384 |
| 7.5.1. Formatowanie liczb i dat | 384 |
| 7.5.2. Formatowanie z wariantami | 386 |
| 7.6. Wczytywanie i wyświetlanie tekstów | 388 |
| 7.6.1. Pliki tekstowe | 388 |
| 7.6.2. Znaki końca wiersza | 388 |
| 7.6.3. Konsola | 389 |
| 7.6.4. Pliki dzienników | 390 |
| 7.6.5. BOM — znacznik kolejności bajtów UTF-8 | 390 |
| 7.6.6. Kodowanie plików źródłowych | 391 |
| 7.7. Kompletzy zasobów | 391 |
| 7.7.1. Wyszukiwanie kompletów zasobów | 392 |
| 7.7.2. Pliki właściwości | 393 |
| 7.7.3. Klasy kompletów zasobów | 393 |
| 7.8. Kompletny przykład | 396 |
| Rozdział 8. Skrypty, kompilacja i adnotacje | 411 |
| 8.1. Skrypty na platformie Java | 411 |
| 8.1.1. Wybór silnika skryptów | 412 |
| 8.1.2. Wykonywanie skryptów i wiązania zmiennych | 413 |
| 8.1.3. Przekierowanie wejścia i wyjścia | 415 |
| 8.1.4. Wywoływanie funkcji i metod skryptów | 416 |
| 8.1.5. Kompilacja skryptu | 418 |
| 8.1.6. Przykład: skrypty i graficzny interfejs użytkownika | 418 |

| | |
|--|------------|
| 8.2. Interfejs kompilatora | 423 |
| 8.2.1. Wywoływanie kompilatora | 423 |
| 8.2.2. Uruchamianie zadania kompilacji | 424 |
| 8.2.3. Przechwytywanie informacji diagnostycznych | 425 |
| 8.2.4. Wczytywanie plików źródłowych z pamięci | 425 |
| 8.2.5. Zapis kodów bajtowych w pamięci | 426 |
| 8.2.6. Przykład: dynamiczne tworzenie kodu w języku Java | 427 |
| 8.3. Stosowanie adnotacji | 433 |
| 8.3.1. Wprowadzenie do stosowania adnotacji | 434 |
| 8.3.2. Przykład: adnotacje obsługi zdarzeń | 435 |
| 8.4. Składnia adnotacji | 440 |
| 8.4.1. Interfejsy adnotacji | 440 |
| 8.4.2. Adnotacje | 441 |
| 8.4.3. Adnotacje deklaracji | 443 |
| 8.4.4. Adnotacje zastosowań typów | 444 |
| 8.4.5. Adnotacje i this | 445 |
| 8.5. Adnotacje standardowe | 447 |
| 8.5.1. Adnotacje kompilacji | 448 |
| 8.5.2. Adnotacje zarządzania zasobami | 448 |
| 8.5.3. Metaadnotacje | 449 |
| 8.6. Przetwarzanie adnotacji w kodzie źródłowym | 452 |
| 8.6.1. Procesory adnotacji | 452 |
| 8.6.2. Interfejs programowy modelu języka | 452 |
| 8.6.3. Stosowanie adnotacji do generacji kodu źródłowego | 453 |
| 8.7. Inżynieria kodu bajtowego | 456 |
| 8.7.1. Modyfikowanie plików klasowych | 456 |
| 8.7.2. Modyfikacja kodu bajtowego podczas ładowania | 461 |
| Rozdział 9. System modułów platformy Javy | 465 |
| 9.1. Pojęcie modułu | 466 |
| 9.2. Nadawanie nazw modułom | 467 |
| 9.3. Modularny program „Witaj, świecie!” | 468 |
| 9.4. Żądanie użycia modułów | 470 |
| 9.5. Eksportowanie pakietów | 471 |
| 9.6. Modularne pliki JAR | 475 |
| 9.7. Moduły a technika refleksji | 476 |
| 9.8. Moduły automatyczne | 479 |
| 9.9. Moduł nienazwany | 481 |
| 9.10. Flagi wiersza poleceń stosowane podczas migracji | 482 |
| 9.11. Wymagania przechodnie i statyczne | 484 |
| 9.12. Eksport kwalifikowany i otwieranie | 485 |
| 9.13. Wczytywanie usług | 486 |
| 9.14. Narzędzia do pracy z modułami | 489 |
| Rozdział 10. Bezpieczeństwo | 493 |
| 10.1. Ładowanie klas | 494 |
| 10.1.1. Proces wczytywania plików klas | 494 |
| 10.1.2. Hierarchia klas ładowania | 495 |
| 10.1.3. Zastosowanie procedur ładujących w roli przestrzeni nazw | 497 |
| 10.1.4. Implementacja własnej procedury ładującej | 498 |
| 10.1.5. Weryfikacja kodu maszyny wirtualnej | 504 |

| | |
|---|-----|
| 10.2. Menedżery bezpieczeństwa i pozwolenia | 508 |
| 10.2.1. Sprawdzanie uprawnień | 508 |
| 10.2.2. Bezpieczeństwo na platformie Java | 509 |
| 10.2.3. Pliki polityki bezpieczeństwa | 512 |
| 10.2.4. Tworzenie własnych klas pozwoleń | 519 |
| 10.2.5. Implementacja klasy pozwoleń | 520 |
| 10.3. Uwierzelnianie użytkowników | 526 |
| 10.3.1. Framework JAAS | 526 |
| 10.3.2. Moduły JAAS | 531 |
| 10.4. Podpis cyfrowy | 540 |
| 10.4.1. Skróty wiadomości | 541 |
| 10.4.2. Podpisywanie wiadomości | 544 |
| 10.4.3. Weryfikacja podpisu | 546 |
| 10.4.4. Problem uwierzelniania | 548 |
| 10.4.5. Podpisywanie certyfikatów | 550 |
| 10.4.6. Żądania certyfikatu | 551 |
| 10.4.7. Podpisywanie kodu | 552 |
| 10.5. Szyfrowanie | 555 |
| 10.5.1. Szyfrowanie symetryczne | 555 |
| 10.5.2. Generowanie klucza | 557 |
| 10.5.3. Strumienie szyfrujące | 561 |
| 10.5.4. Szyfrowanie kluczem publicznym | 562 |

Rozdział 11. Zaawansowane możliwości pakietu Swing i grafiki567

| | |
|--|-----|
| 11.1. Tabele | 567 |
| 11.1.1. Najprostsze tabele | 568 |
| 11.1.2. Modele tabel | 571 |
| 11.1.3. Wiersze i kolumny | 575 |
| 11.1.4. Rysowanie i edycja komórek | 590 |
| 11.2. Drzewa | 601 |
| 11.2.1. Proste drzewa | 602 |
| 11.2.2. Przeglądanie węzłów | 616 |
| 11.2.3. Rysowanie węzłów | 618 |
| 11.2.4. Nasłuchiwanie zdarzeń w drzewach | 621 |
| 11.2.5. Własne modele drzew | 627 |
| 11.3. Zaawansowane możliwości biblioteki AWT | 635 |
| 11.3.1. Potokowe tworzenie grafiki | 635 |
| 11.3.2. Figury | 638 |
| 11.3.3. Pola | 652 |
| 11.3.4. Ślad pędzla | 653 |
| 11.3.5. Wypełnienia | 661 |
| 11.3.6. Przekształcenia układu współrzędnych | 663 |
| 11.3.7. Przycinanie | 668 |
| 11.3.8. Przezroczystość i składanie obrazów | 670 |
| 11.4. Grafika rastrowa | 678 |
| 11.4.1. Odczyt i zapis plików graficznych | 678 |
| 11.4.2. Operacje na obrazach | 688 |

| | |
|--|-----|
| 11.5. Drukowanie | 703 |
| 11.5.1. Drukowanie grafiki | 703 |
| 11.5.2. Drukowanie wielu stron | 711 |
| 11.5.3. Usługi drukowania | 720 |
| 11.5.4. Usługi drukowania za pośrednictwem strumieni | 722 |
| 11.5.5. Atrybuty drukowania | 725 |

Rozdział 12. Metody macierzyste 733

| | |
|---|-----|
| 12.1. Wywołania funkcji języka C z programów w języku Java | 734 |
| 12.2. Numeryczne parametry metod i wartości zwracane | 740 |
| 12.3. Łańcuchy znaków jako parametry | 742 |
| 12.4. Dostęp do składowych obiektu | 747 |
| 12.4.1. Dostęp do pól instancji | 747 |
| 12.4.2. Dostęp do pól statycznych | 751 |
| 12.5. Sygnatury | 752 |
| 12.6. Wywoływanie metod języka Java | 754 |
| 12.6.1. Wywoływanie metod obiektów | 754 |
| 12.6.2. Wywoływanie metod statycznych | 757 |
| 12.6.3. Konstruktory | 758 |
| 12.6.4. Alternatywne sposoby wywoływania metod | 758 |
| 12.7. Dostęp do elementów tablic | 760 |
| 12.8. Obsługa błędów | 764 |
| 12.9. Interfejs programowy wywołań języka Java | 768 |
| 12.10. Kompletny przykład: dostęp do rejestru systemu Windows | 773 |
| 12.10.1. Rejestr systemu Windows | 773 |
| 12.10.2. Interfejs dostępu do rejestru na platformie Java | 775 |
| 12.10.3. Implementacja dostępu do rejestru za pomocą metod macierzystych | 776 |

Skorowidz 789

2

Wejście i wyjście

W tym rozdziale:

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

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

2.1. 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` posiada metodę abstrakcyjną:

```
abstract int read()
```

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

Klasa `InputStream` posiada również nieabstrakcyjne metody pozwalające pobrać lub zignorować tablicę bajtów. Od Javy 9 dostępna jest także bardzo wygodna metoda pozwalająca na wczytanie wszystkich bajtów 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`, tak więc podklasy muszą przesłaniać tylko tę jedną metodę.

Analogicznie, klasa `OutputStream` definiuje metodę abstrakcyjną

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

która wysyła jeden bajt do aktualnego wyjścia.

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

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

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

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

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

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

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

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

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

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

API | java.io.InputStream 1.0

■ abstract int read()

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

■ int read(byte[] b)

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

■ int read(byte[] b, int off, int len)

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

wczytuje bez blokownia (`read`) do tablicy co najwyżej `len` bajtów, o ile są one dostępne, ewentualnie blokuje operację aż do momentu, kiedy wartości staną się dostępne (`readNBytes`). Wartości są umieszczane w tablicy `b`, zaczynając do jej elementu o indeksie `off`. Zwraca 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 wejścia. Zwraca faktyczną liczbę zignorowanych bajtów (która może być mniejsza niż `n`, jeżeli natrafimy na koniec strumienia wejścia).
- `int available()`
zwraca liczbę bajtów dostępnych bez konieczności zablokowania wątku (pamiętajmy, że zablokowanie oznacza, że wykonanie aktualnego wątku zostaje wstrzymane).
- `void close()`
zamyka strumień wejścia.
- `void mark(int readlimit)`
ustawia znacznik na aktualnej pozycji strumienia wejścia (nie wszystkie strumienie obsługują tę możliwość). Jeżeli ze strumienia zostało pobranych więcej niż `readlimit` bajtów, strumień ma prawo usunąć znacznik.
- `void reset()`
wraca do ostatniego znacznika. Późniejsze wywołania `read` będą powtórnie czytać pobrane już bajty. Jeżeli znacznik nie istnieje, strumień wejścia nie zostanie zresetowany.
- `boolean markSupported()`
zwraca `true`, jeżeli strumień wejścia obsługuje znaczniki.

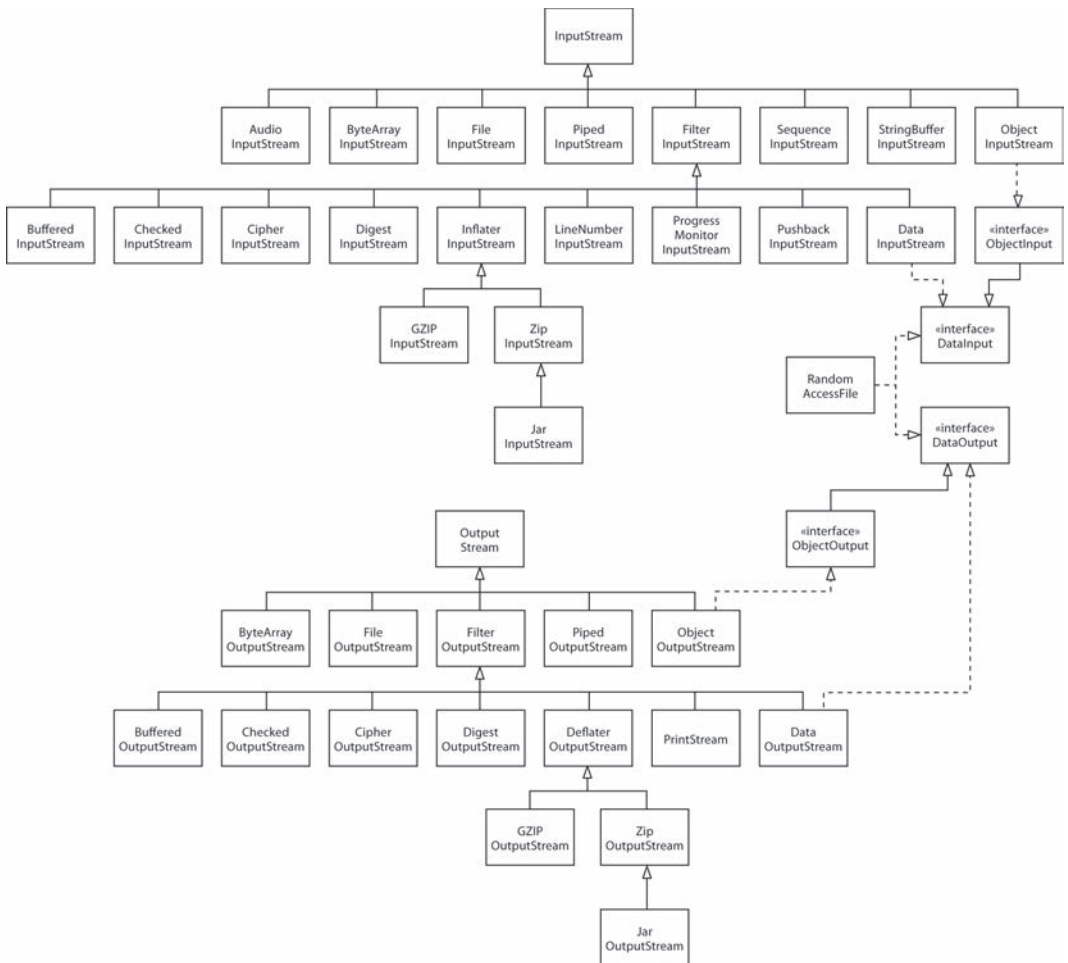
API `java.io.OutputStream` **1.0**

- `abstract void write(int n)`
zapisuje jeden bajt.
- `void write(byte[] b)`
- `void write(byte[] b, int off, int len)`
zapisują wszystkie bajty tablicy `b` lub pewien ich zakres.
- `void close()`
opróżnia i zamyka strumień wyjścia.
- `void flush()`
opróżnia strumień wyjścia, czyli wysyła do odbiorcy wszystkie dane znajdujące się w buforze.

2.1.2. Zoo pełne strumieni

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

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

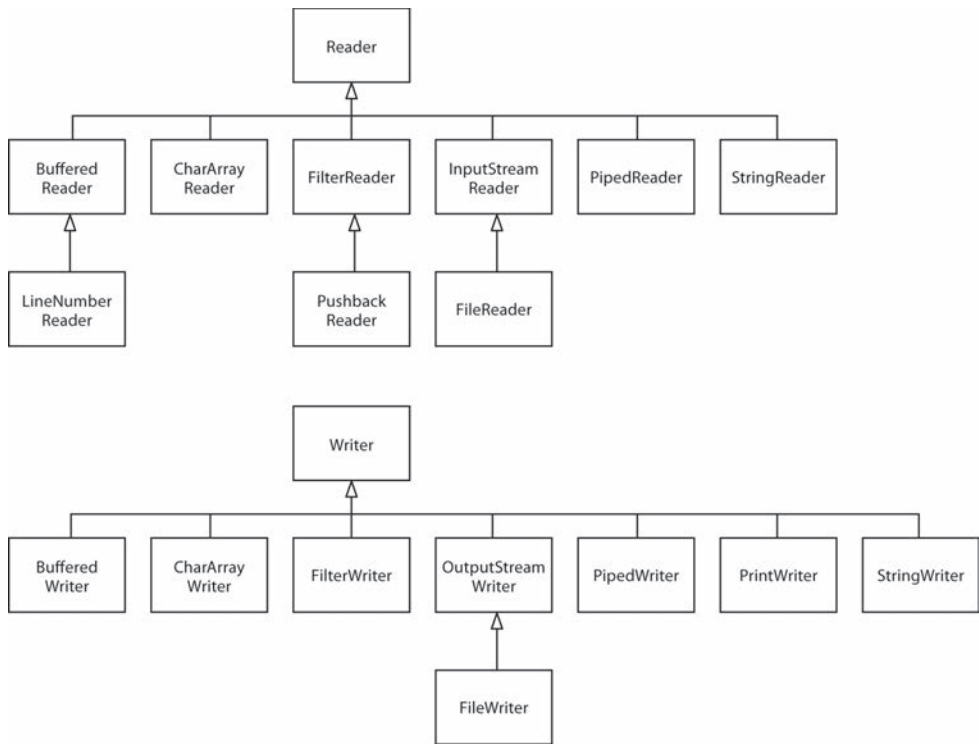


Rysunek 2.1. Hierarchia strumieni wejścia i wyjścia

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

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

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



Rysunek 2.2. Hierarchia klas `Reader` i `Writer`

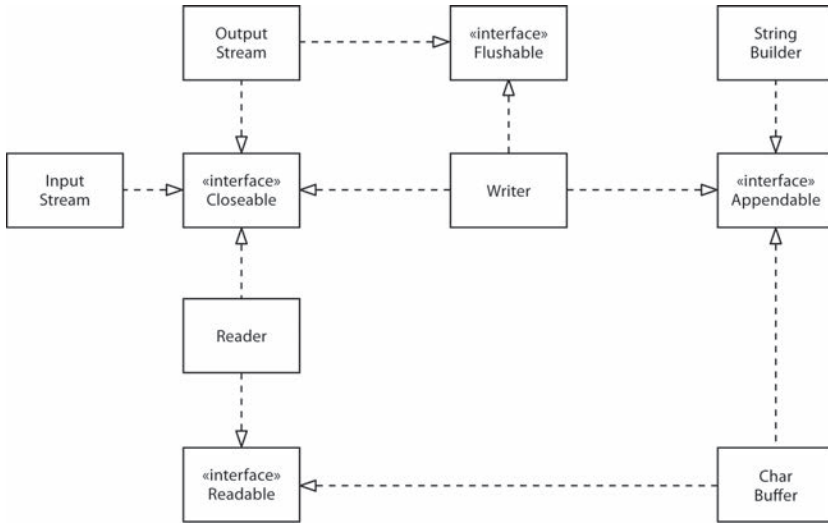
Dostępne są również cztery dodatkowe interfejsy: `Closeable`, `Flushable`, `Readable` i `Appendable` (patrz rysunek 2.3). Pierwsze dwa z nich są wyjątkowo proste i zawierają odpowiednio metody:

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

i

```
void flush()
```

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



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



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

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

Interfejs `Readable` ma tylko jedną metodę

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

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

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`.

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

API `java.io.Closeable` 5.0

■ `void close()`

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

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

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

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

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

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

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

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

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

2.1.3. Łączenie filtrów strumieni wejścia-wyjścia

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

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

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



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



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

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

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

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

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

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

Java korzysta ze sprytnego mechanizmu rozdzielającego te dwa rodzaje funkcjonalności. Niektóre strumienie wejścia (takie jak `FileInputStream` i strumień wejścia zwracany przez metodę `openStream` klasy `URL`) mogą udostępniać bajty z plików i innych, bardziej egzotycznych lokalizacji. Inne strumienie wejścia (takie jak `DataInputStream`) potrafią tworzyć z bajtów reprezentację bardziej użytecznych typów danych. Programista Javy musi połączyć te dwa mechanizmy w jeden. Dla przykładu, aby wczytywać liczby z pliku, powinien utworzyć obiekt typu `FileInputStream`, a następnie przekazać go konstruktorowi `DataInputStream`.

```
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 podklasy możemy wykorzystać do zwiększania możliwości strumieni wejścia-wyjścia służących do operowania na zwykłych bajtach.

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

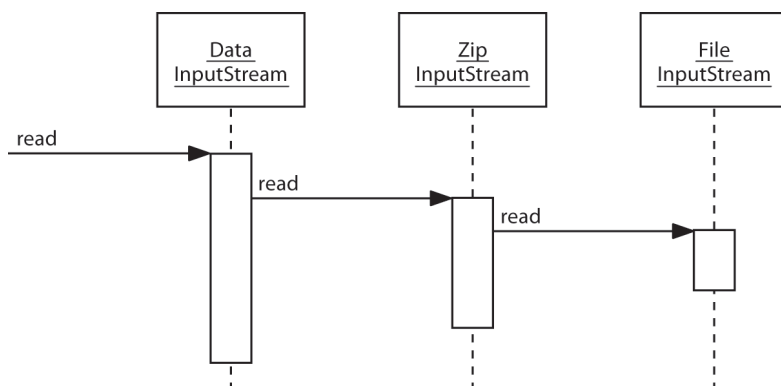
Ale wczytywanie i powtórne wstawianie bajtów to *jedyn*e metody obsługiwane przez klasę `Pushback-InputStream`. Jeżeli chcemy podejrzeć bajty, a także wczytywać liczby, potrzebujemy referencji zarówno do `PushbackInputStream`, jak i do `DataInputStream`.

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

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

```
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, „Archiwa ZIP”, poświęconego strumieniom plików ZIP.)

API java.io.FileInputStream 1.0

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

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

API java.io.FileOutputStream 1.0

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

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

API java.io.BufferedInputStream 1.0

- BufferedInputStream(InputStream in)

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

API java.io.BufferedOutputStream 1.0

- BufferedOutputStream(OutputStream out)

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

API java.io.PushbackInputStream 1.0

- PushbackInputStream(InputStream in)
- PushbackInputStream(InputStream in, int size)

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

- void unread(int b)

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

2.1.4. Strumienie tekstowe

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

Zapisując łańcuchy znakowe, musimy uwzględnić sposób *kodowania znaków*. W przypadku kodowania UTF-16, którego Java używa wewnętrznie, łańcuch "José" zostanie zakodowany jako 00 4A 00 6F 00 73 00 E9 (w notacji szesnastkowej). Jednakże wiele programów oczekuje, że pliki tekstowe będą zapisane przy wykorzystaniu innych sposobów kodowania. W przypadku kodowania UTF-8, obecnie najczęściej stosowanego w internecie, ten sam łańcuch znaków został zapisany jako następująca sekwencja bajtów: 4A 6F 73 C3 A9, czyli bez bajtów 00 dla pierwszych trzech znaków oraz bez używania dwóch bajtów do zapisu znaku é.

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

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

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

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

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

Więcej informacji na temat kodowania znaków znajdziesz w punkcie 2.1.8, „Zbiory znaków”.

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. Zapisywanie tekstu

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

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

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

Spójrzmy na poniższy kod:

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

Rezultatem jego wykonania będzie wysłanie napisu

```
Harry Hacker 75000.0
```

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

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

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

```
var out = new PrintWriter(
    new OutputStreamWriter(
        new FileOutputStream("employee.txt"), StandardCharsets.UTF_8),
    true); //Automatyczne opróżnianie
```

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



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

API java.io.PrintWriter 1.1

- `PrintWriter(Writer out)`
- `PrintWriter(Writer writer)`
tworzy nowy obiekt klasy `PrintWriter` zapisujący dane w przekazanym obiekcie zapisu.
- `PrintWriter(String filename, String encoding)`
- `PrintWriter(File file, String encoding)`
tworzy nowy obiekt klasy `PrintWriter` zapisujący dane do podanego pliku z wykorzystaniem określonego sposobu kodowania.
- `void print(Object obj)`
zapisuje łańcuch zwracany przez metodę `toString` danego obiektu.
- `void print(String s)`
zapisuje łańcuch Unicode.
- `void println(String s)`
zapisuje łańcuch zakończony znakiem końca wiersza. Jeżeli automatyczne opróżnianie jest włączone, opróżnia bufor strumienia.
- `void print(char[] s)`
zapisuje tablicę znaków Unicode.
- `void print(char c)`
zapisuje znak Unicode.
- `void print(int i)`
- `void print(long l)`
- `void print(float f)`
- `void print(double d)`
- `void print(boolean b)`
zapisuje podaną wartość w formacie tekstowym.

- `void printf(String format, Object... args)`
zapisuje podane wartości według łańcucha formatującego. Specyfikację łańcucha formatującego znajdziesz w rozdziale 3. książki *Java. Podstawy*.
- `boolean checkError()`
zwraca `true`, jeżeli wystąpił błąd formatowania lub zapisu. Jeżeli w strumieniu danych wystąpi błąd, strumień zostanie uznany za niepewny (ang. *tainted*) i wszystkie następane wywołania metody `checkError` będą zwracać `true`.

2.1.6. 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:

```
var content = new String(Files.readAllBytes(path), charset);
```

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

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

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

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

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:

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

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

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

2.17. Zapis obiektów w formacie tekstowym

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

Oto przykładowy zbiór danych obiektów:

```
Harry Hacker|35500|1989-10-01
Carl Cracker|75000|1987-12-15
Tony Tester|38000|1990-03-15
```

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

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

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

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

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

Kompletny program został przedstawiony na listingu 2.1. Metoda statyczna

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

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

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

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

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

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

Listing 2.1. `textfile/TextFileTest.java`

```
1 package textfile;
2
3 import java.io.*;
4 import java.nio.charset.*;
5 import java.time.*;
6 import java.util.*;
7
8 /**
9  * @version 1.15 2018-03-17
```

```
10 * @author Cay Horstmann
11 */
12 public class TextFileTest
13 {
14     public static void main(String[] args) throws IOException
15     {
16         var staff = new Employee[3];
17
18         staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
19         staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
20         staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
21
22         // Zapisuje wszystkie rekordy pracowników w pliku employee.dat
23         try (var out = new PrintWriter("employee.dat", StandardCharsets.UTF_8))
24         {
25             writeData(staff, out);
26         }
27
28         // Wczytuje wszystkie rekordy do nowej tablicy
29         try (var in = new Scanner(
30             new FileInputStream("employee.dat"), "UTF-8"))
31         {
32             Employee[] newStaff = readData(in);
33
34             // Wyświetla wszystkie wczytane rekordy
35             for (Employee e : newStaff)
36                 System.out.println(e);
37         }
38     }
39
40     /**
41     * Zapisuje dane wszystkich obiektów klasy Employee umieszczonych
42     * @param employees tablica obiektów klasy Employee
43     * @param out obiekt klasy PrintWriter
44     */
45     private static void writeData(Employee[] employees, PrintWriter out)
46         throws IOException
47     {
48         // Zapisuje liczbę obiektów
49         out.println(employees.length);
50
51         for (Employee e : employees)
52             writeEmployee(out, e);
53     }
54
55     /**
56     * Wczytuje tablicę obiektów klasy Employee
57     * @param in obiekt klasy Scanner
58     * @return tablica obiektów klasy Employee
59     */
60     private static Employee[] readData(Scanner in)
61     {
62         // Pobiera rozmiar tablicy
63         int n = in.nextInt();
64         in.nextLine(); // Pobiera znak nowego wiersza
65     }
66 }
```

```
66     var employees = new Employee[n];
67     for (int i = 0; i < n; i++)
68     {
69         employees[i] = readEmployee(in);
70     }
71     return employees;
72 }
73
74 /**
75  * Zapisuje dane obiektu klasy Employee do obiektu klasy PrintWriter
76  * @param out obiekt klasy PrintWriter
77  */
78 public static void writeEmployee(PrintWriter out, Employee e)
79 {
80     out.println(e.getName() + "|" + e.getSalary() + "|" + e.getHireDay());
81 }
82
83 /**
84  * Wczytuje dane obiektu klasy Employee
85  * @param in obiekt klasy Scanner
86  */
87 public static Employee readEmployee(Scanner in)
88 {
89     String line = in.nextLine();
90     String[] tokens = line.split("\\|");
91     String name = tokens[0];
92     double salary = Double.parseDouble(tokens[1]);
93     LocalDate hireDate = LocalDate.parse(tokens[2]);
94     int year = hireDate.getYear();
95     int month = hireDate.getMonthValue();
96     int day = hireDate.getDayOfMonth();
97     return new Employee(name, salary, year, month, day);
98 }
99 }
```

2.1.8. Zbiory znaków

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

W Javie znaki są kodowane przy wykorzystaniu standardu Unicode. Każdy znak, nazywany także punktem kodowym, jest skojarzony z 21-bitową liczbą całkowitą. Istnieje kilka różnych *sposobów kodowania znaków*, czyli metod zapisu tych 21-bitowych wartości w formie bajtów.

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

Tabela 2.1. Kodowanie UTF-8

| Zakres znaku | Sposób kodowania |
|------------------|--|
| 0 ... 7F | $0a_6a_5a_4a_3a_2a_1a_0$ |
| 80 ... 7FF | $110a_{10}a_9a_8a_7a_6$ $10a_5a_4a_3a_2a_1a_0$ |
| 800 ... FFFF | $1110a_{15}a_{14}a_{13}a_{12}$ $10a_{11}a_{10}a_9a_8a_7a_6$ $10a_5a_4a_3a_2a_1a_0$ |
| 10000 ... 10FFFF | $11110a_{20}a_{19}a_{18}$ $10a_{17}a_{16}a_{15}a_{14}a_{13}a_{12}$ $10a_{11}a_{10}a_9a_8a_7a_6$ $10a_5a_4a_3a_2a_1a_0$ |

Kolejnym popularnym sposobem kodowania jest UTF-16, w którym punkty kodowe Unicode są zapisywane przy wykorzystaniu jednej lub dwóch wartości 16-bitowych (patrz tabela 2.2). To właśnie ten sposób kodowania jest używany do zapisu łańcuchów znaków w Javie. W rzeczywistości istnieją dwa rodzaje kodowania UTF-16, określane — odpowiednio — jako *big-endian* oraz *little-endian*. W ramach przykładu przeanalizujemy 16-bitową wartość $0x2122$. W przypadku wykorzystania formatu *big-endian* jako pierwszy jest zapisywany bardziej znaczący bajt tej wartości: $0x21$, a za nim $0x22$. Z kolei w przypadku formatu *little-endian* wartość ta zostanie zapisana jako: $0x22$ $0x21$. Aby możliwe było określenie, który z tych dwóch rodzajów kodowania UTF-16 jest używany, na początku pliku umieszcza się tak zwany znak BOM (ang. *byte order mark*¹) — 16-bitową wartość $0xFEFF$. Obiekt odczytujący może użyć tej wartości do określenia kolejności zapisu bajtów w pliku bądź też ją zignorować.

Tabela 2.2. Kodowanie UTF-16

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

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.



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.

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



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



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

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

```
StandardCharsets.UTF_8
StandardCharsets.UTF_16
StandardCharsets.UTF_16BE
StandardCharsets.UTF_16LE
StandardCharsets.ISO_8859_1
StandardCharsets.US_ASCII
```

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

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

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

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



W języku Java 10 wszystkie metody w pakiecie `java.io` pozwalają określać sposoby kodowania przy użyciu obiektu `Charset` lub łańcucha znaków. Warto jednak wybierać stałe klasy `StandardCharsets`, aby nie trzeba było zwracać uwagi na ich poprawny zapis.



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

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

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 zadowalała się tylko 16 bitami.



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

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

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

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

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

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

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

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

API `java.io.DataInput` 1.0

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

API `java.io.DataOutput` **1.0**

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

2.2.2. Strumienie plików o swobodnym dostępie

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

```
var in = new 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*

```
1 package randomAccess;
2
3 import java.io.*;
4 import java.time.*;
5
6 /**
7  * @version 1.14 2018-05-01
8  * @author Cay Horstmann
9  */
10 public class RandomAccessTest
11 {
12     public static void main(String[] args) throws IOException
13     {
14         var staff = new Employee[3];
15
16         staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
17         staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
18         staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
19
20         try (var out = new DataOutputStream(new FileOutputStream("employee.dat")))
21         {
22             // Zapisuje rekordy wszystkich pracowników w pliku employee.dat
23             for (Employee e : staff)
```

```
24         writeData(out, e);
25     }
26
27     try (var in = new RandomAccessFile("employee.dat", "r"))
28     {
29         // Wczytuje wszystkie rekordy do nowej tablicy
30
31         // Oblicza rozmiar tablicy
32         int n = (int)(in.length() / Employee.RECORD_SIZE);
33         var newStaff = new Employee[n];
34
35         // Wczytuje rekordy pracowników w odwrotnej kolejności
36         for (int i = n - 1; i >= 0; i--)
37         {
38             newStaff[i] = new Employee();
39             in.seek(i * Employee.RECORD_SIZE);
40             newStaff[i] = readData(in);
41         }
42
43         // Wyświetla wczytane rekordy
44         for (Employee e : newStaff)
45             System.out.println(e);
46     }
47 }
48
49 /**
50  * Zapisuje dane pracownika
51  * @param out obiekt typu DataOutput
52  * @param e   pracownik
53  */
54 public static void writeData(DataOutput out, Employee e) throws IOException
55 {
56     DataIO.writeFixedString(e.getName(), Employee.NAME_SIZE, out);
57     out.writeDouble(e.getSalary());
58
59     LocalDate hireDay = e.getHireDay();
60     out.writeInt(hireDay.getYear());
61     out.writeInt(hireDay.getMonthValue());
62     out.writeInt(hireDay.getDayOfMonth());
63 }
64
65 /**
66  * Wczytuje dane pracownika
67  * @param in obiekt typu DataInput
68  * @return  pracownik
69  */
70 public static Employee readData(DataInput in) throws IOException
71 {
72     String name = DataIO.readFixedString(Employee.NAME_SIZE, in);
73     double salary = in.readDouble();
74     int y = in.readInt();
75     int m = in.readInt();
76     int d = in.readInt();
77     return new Employee(name, salary, y, m - 1, d);
78 }
79 }
```

API java.io.RandomAccessFile 1.0

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

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

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

zmienia położenie wskaźnika pliku, przesuując 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, zanim zostanie odczytana ostatnia pozycja. Oto typowa sekwencja wywołań służąca do odczytu zawartości pliku ZIP:

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

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

```

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

```



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

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



W punkcie 2.4.8, „Systemy plików ZIP”, zobaczymy, w jaki sposób można korzystać z archiwów ZIP bez korzystania ze specjalnego interfejsu programowego, używając w tym celu klasy `FileSystem` wprowadzonej w Java SE 7.

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

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

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

- `ZipOutputStream(OutputStream out)`
tworzy obiekt typu `ZipOutputStream`, który umożliwia kompresję i zapis danych w podanym strumieniu `OutputStream`.
- `void putNextEntry(ZipEntry ze)`
zapisuje informacje podanej pozycji `ZipEntry` do strumienia i przygotowuje strumień do odbioru danych. Dane mogą zostać zapisane w strumieniu przy użyciu metody `write()`.

- `void closeEntry()`

zamyka aktualnie otwartą pozycję archiwum ZIP. Aby otworzyć następną pozycję, wywołujemy metodę `putNextEntry`.
- `void setLevel(int level)`

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

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

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

2.3. Strumienie obiektów i serializacja

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

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

2.3.1. Zapisywanie i wczytywanie obiektów serializowalnych

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

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

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

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

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

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

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

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

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

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

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



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

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

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

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

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

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

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

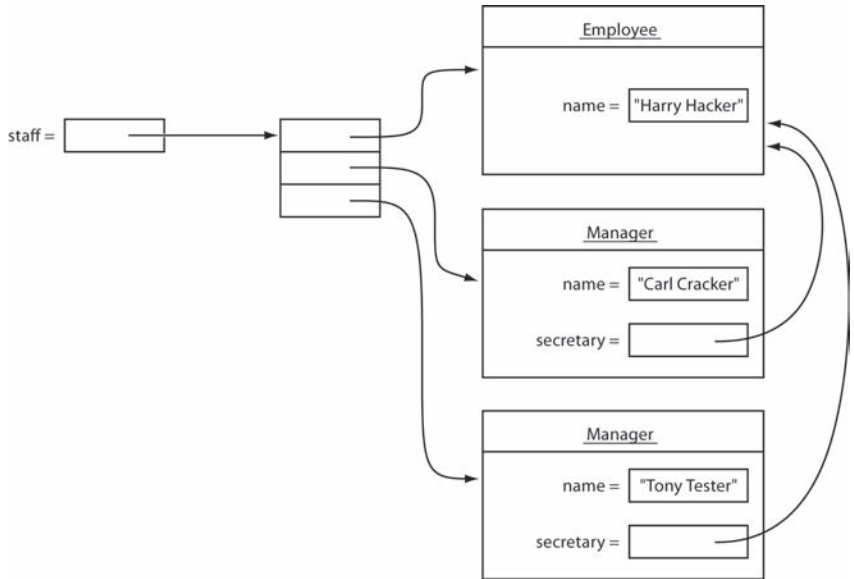
Teraz założmy, że zapisujemy dane pracowników na dysk. Oczywiście nie możemy zapisać i przywrócić adresów obiektów asystentów w pamięci, ponieważ po ponownym załadowaniu obiekt asystenta najprawdopodobniej znajdzie się w zupełnie innym miejscu pamięci.

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

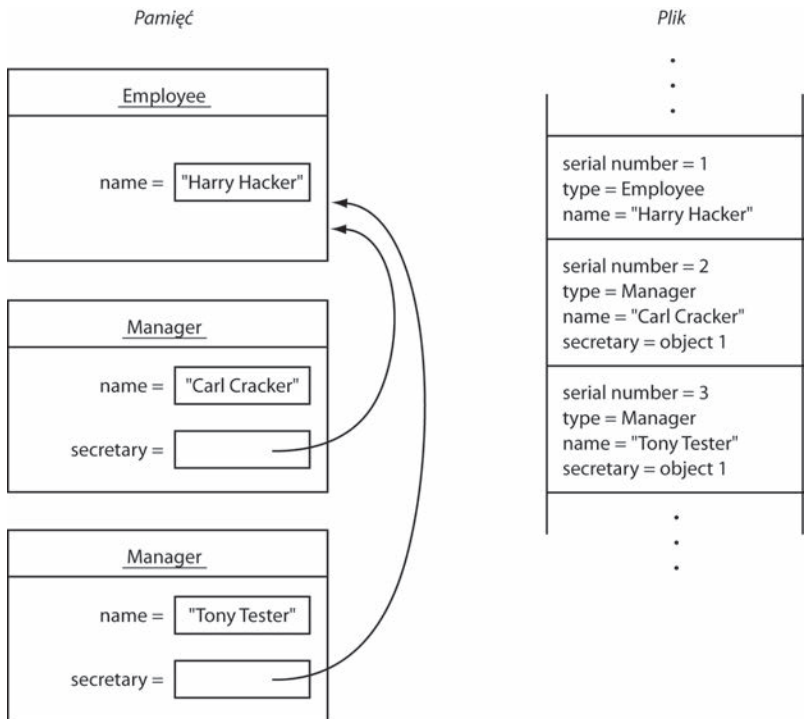
1. Wszystkim napotkanym referencjom do obiektów nadawane są numery seryjne (patrz rysunek 2.6).

Rysunek 2.5.

Dwóch menedżerów może mieć wspólnego asystenta

**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 seryjnymi, możemy transportować zbiory danych z jednego komputera do drugiego.

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

Listing 2.3. `objectStream/ObjectStreamTest.java`

```

1 package objectStream;
2
3 import java.io.*;
4
5 /**
6  * @version 1.11 2018-05-01
7  * @author Cay Horstmann
8  */
9 class ObjectStreamTest
10 {
11     public static void main(String[] args) throws IOException, ClassNotFoundException
12     {
13         var harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
14         var carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
15         carl.setSecretary(harry);
16         var tony = new Manager("Tony Tester", 40000, 1990, 3, 15);
17         tony.setSecretary(harry);
18
19         var staff = new Employee[3];
20
21         staff[0] = carl;
22         staff[1] = harry;
23         staff[2] = tony;
24
25         // Zapisuje rekordy wszystkich pracowników w pliku employee.dat
26         try (var out = new ObjectOutputStream(new FileOutputStream("employee.dat")))
27         {
28             out.writeObject(staff);
29         }

```

```

30
31     try (var in = new ObjectInputStream(new FileInputStream("employee.dat")))
32     {
33         // Wczytuje wszystkie rekordy do nowej tablicy
34
35         var newStaff = (Employee[]) in.readObject();
36
37         // Podnosi wynagrodzenie asystenta
38         newStaff[1].raiseSalary(10);
39
40         // Wyświetla wszystkie wczytane rekordy
41         for (Employee e : newStaff)
42             System.out.println(e);
43     }
44 }
45 }

```

API java.io.ObjectOutputStream 1.1

- ObjectOutputStream(OutputStream wy)

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

- void writeObject(Object ob)

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

API java.io.ObjectInputStream 1.1

- ObjectInputStream(InputStream we)

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

- Object readObject()

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

2.3.2. 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 (2 bajty)   znaki
```

Dla przykładu, łańcuch "Harry" będzie wyglądał tak:

```
74 00 05 Harry
```

Znaki Unicode zapisywane są w zmodyfikowanym formacie UTF-8.

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

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

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

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

W chwili odczytu obiektu jego odcisk zostaje porównany z aktualnym odciskiem klasy. Jeśli różni się, oznacza to, że definicja klasy uległa zmianie po zapisaniu obiektu. Oczywiście w praktyce klasy ulegają zmianie i może się okazać, że program będzie musiał wczytać starsze wersje obiektów. Zagadnienie to omówimy w punkcie 2.3.5, „Wersje”.

Oto w jaki sposób przechowywany jest identyfikator klasy:

- 72
- długość nazwy klasy (2 bajty)
- nazwa klasy

- „odcisk” klasy (8 bajtów)
- zbiór flag (1 bajt)
- liczba deskryptorów pól składowych (2 bajty)
- deskryptory pól składowych
- 78 (znacznik końca)
- typ klasy bazowej (70, jeśli nie istnieje)

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

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

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

Każdy deskryptor pola składowego składa się z następujących elementów:

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

Kod typu może mieć jedną z następujących wartości:

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

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

Dla przykładu, pole salary klasy Employee zostanie zapisane jako:

```
D 00 06 salary
```

A oto kompletny opis klasy Employee:

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

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

```
71 numer seryjny (4 bajty)
```

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

Obiekt jest przechowywany w następującej postaci:

```
73 opis klasy dane obiektu
```

Dla przykładu, oto zapis obiektu klasy Employee:

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

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

Tablice są zapisywane w następujący sposób:

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

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

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

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

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

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

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

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

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

Referencja `null` jest zapisywana jako

```
70
```

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

| | |
|----------------------------|--|
| AC ED 00 05 | Nagłówek pliku |
| 75 | Tablica <code>staff</code> (nr #1) |
| 72 00 0B [LEmployee; | Nowa klasa, długość łańcucha, nazwa klasy <code>Employee[]</code> (nr #0) |
| FC BF 36 11 C5 91 11 C7 02 | „Odcisk” oraz flagi |
| 00 00 | Liczba pól składowych |
| 78 | Znacznik końca |
| 70 | Brak klasy bazowej |
| 00 00 00 03 | Liczba elementów tablicy |

| | | |
|----|-----------------------------|--|
| 73 | | staff[0] — nowy obiekt (nr #7) |
| 72 | 00 07 Manager | Nowa klasa, długość łańcucha, nazwa klasy (nr #2) |
| | 36 06 AE 13 63 8F 59 B7 02 | „Odcisk” oraz flagi |
| | 00 01 | Liczba pól składowych |
| | L 00 09 secretary | Typ i nazwa pola składowego |
| | 74 00 0A LEmployee; | Nazwa klasy pola składowego — String (nr #3) |
| | 78 | Znacznik końca |
| | 72 00 08 Employee | Nadklasa — nowa klasa, długość łańcucha, nazwa klasy (nr #4) |
| | E6 D2 86 7D AE AC 18 1B 02 | „Odcisk” oraz flagi |
| | 00 03 | Liczba pól składowych |
| | D 00 06 salary | Typ i nazwa pola składowego |
| | L 00 07 hireDay | Typ i nazwa pola składowego |
| | 74 00 10 Ljava/util/Date; | Nazwa klasy pola składowego — String (nr #5) |
| | L 00 04 name | Typ i nazwa pola składowego |
| | 74 00 12 Ljava/lang/String; | Nazwa klasy pola składowego — String (nr #6) |
| | 78 | Znacznik końca |
| | 70 | Brak klasy bazowej |
| 40 | F3 88 00 00 00 00 00 | Wartość pola salary — double |
| 73 | | Wartość pola hireDay — nowy obiekt (nr #9) |
| | 72 00 0E java.util.Date | Nowa klasa, długość łańcucha, nazwa klasy (nr #8) |
| | 68 6A 81 01 4B 59 74 19 03 | „Odcisk” oraz flagi |
| | 00 00 | Brak zmiennych składowych |
| | 78 | Znacznik końca |
| | 70 | Brak klasy bazowej |
| | 77 08 | Zawartość zewnętrzna, liczba bajtów |
| | 00 00 00 83 E9 39 E0 00 | Data |
| | 78 | Znacznik końca |
| | 74 00 0C Car1 Cracker | Wartość pola name — String (nr #10) |
| 73 | | Wartość pola secretary — nowy obiekt (nr #11) |
| | 71 00 7E 00 04 | Istniejąca klasa (użyj nr #4) |
| | 40 E8 6A 00 00 00 00 00 | Wartość pola pensja — double |
| | 73 | Wartość pola dzienZatrudnienia — nowy obiekt (nr #12) |

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

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

Powinniśmy zapamiętać, że:

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

2.3.3. Modyfikowanie domyślnego mechanizmu serializacji

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

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

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

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

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

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

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

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

Implementując metodę `readObject`, odwrócimy cały proces:

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

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

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

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

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

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

```
public void readExternal(ObjectInput s)
    throws IOException
{
    name = s.readUTF();
    salary = s.readDouble();
    hireDay = LocalDate.ofEpochDay(s.readLong());
}
public void writeExternal(ObjectOutput s)
    throws IOException
{
    s.writeUTF(name);
    s.writeDouble(salary);
    s.writeLong(hireDay.toEpochDay());
}
```



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

2.3.4. Serializacja singletonów i wyliczeń

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

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

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

```

    private Orientation(int v) { value = v; }
    private int value;
}

```

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

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

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

```

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

```

Okaże się, że porównanie

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

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

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

```

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

```

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

2.3.5. Wersje

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

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

```
serialver Employee
```

otrzymujemy:

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

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

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

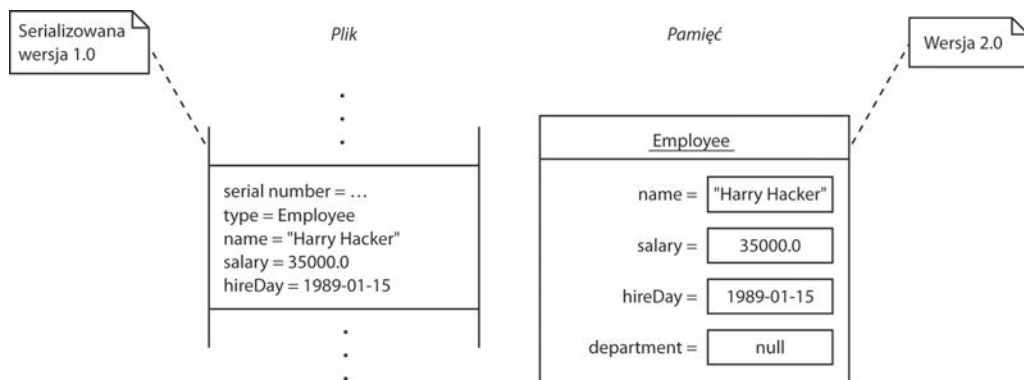
Klasa posiadająca statyczne pole składowe o nazwie `serialVersionUID` nie obliczy własnego „odcisku palca”, ale skorzysta z już istniejącej wartości.

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

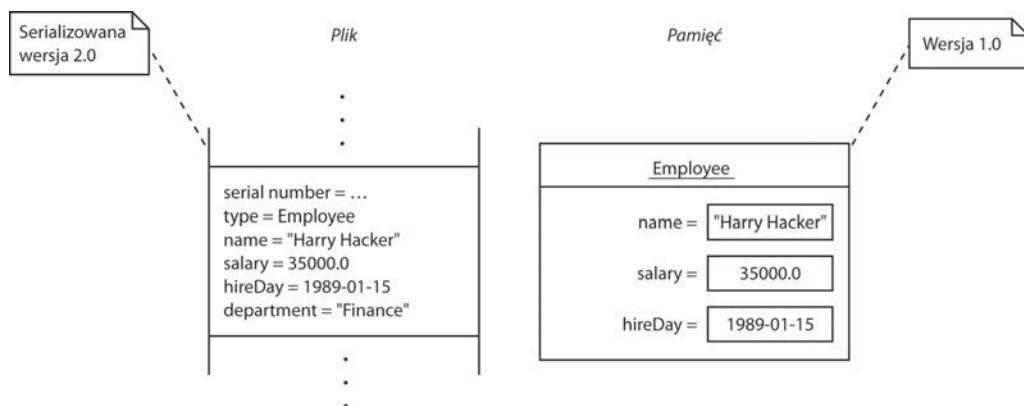
Jeśli zmienią się tylko metody danej klasy, sposób odczytu danych nie ulegnie zmianie. Jednakże jeżeli zmieni się pole składowe, możemy mieć pewne problemy. Dla przykładu, stary obiekt może posiadać więcej lub mniej pól składowych niż aktualny albo też typy danych mogą się różnić. W takim wypadku wejściowy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

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

Oto przykład. Załóżmy, że zapisaliśmy na dysku pewną liczbę obiektów klasy `Employee`, używając przy tym oryginalnej (1.0) wersji klasy. Teraz wprowadzamy nową wersję 2.0 klasy `Employee`, dodając do niej pole składowe `department`. Rysunek 2.7 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 być bezbolesne — odbiorca wciąż posiada dane, którymi potrafi manipulować. Nadawanie wartości null nie jest już tak bezpieczne. Wiele klas inicjalizuje wszystkie pola składowe, nadając im w konstruktorach niezerowe wartości, tak więc metody mogą być nieprzygotowane do obsługi wartości null. Od projektanta klasy zależy, czy zaimplementuje w metodzie `readObject` dodatkowy kod poprawiający wyniki wczytywania różnych wersji danych, czy też dołączy do metod obsługujących wartości null.



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.

2.3.6. Serializacja w roli klonowania

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

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

Listing 2.4. `serialClone/SerialCloneTest.java`

```

1 package serialClone;
2
3 /**
4  * @version 1.22 2018-05-01
5  * @author Cay Horstmann
6  */
7
8 import java.io.*;
9 import java.util.*;
10 import java.time.*;
11
12 public class SerialCloneTest
13 {
14     public static void main(String[] args) throws CloneNotSupportedException
15     {
16         var harry = new Employee("Harry Hacker", 35000, 1989, 10, 1);
17         // Klonuje obiekt harry
18         var harry2 = (Employee) harry.clone();
19
20         // Modyfikuje obiekt harry
21         harry.raiseSalary(10);
22
23         // Teraz obiekt harry i jego klon różnią się
24         System.out.println(harry);
25         System.out.println(harry2);
26     }
27 }
28
29 /**
30  * Klasa, której metoda clone wykorzystuje serializację.
31  */
32 class SerialCloneable implements Cloneable, Serializable
33 {
34     public Object clone() throws CloneNotSupportedException
35     {
36         try {
37             // Zapisuje obiekt w tablicy bajtów
38             var bout = new ByteArrayOutputStream();
39             try (var out = new ObjectOutputStream(bout))
40             {
41                 out.writeObject(this);
42             }

```

```
43
44     // Wczytuje klon obiektu z tablicy bajtów
45     try (var bin = new ByteArrayInputStream(bout.toByteArray()))
46     {
47         var in = new ObjectInputStream(bin);
48         return in.readObject();
49     }
50 }
51 catch (IOException | ClassNotFoundException e)
52 {
53     var e2 = new CloneNotSupportedException();
54     e2.initCause(e);
55     throw e2;
56 }
57 }
58 }
59
60 /**
61  * Znana już klasa Employee.
62  * tym razem jako pochodna klasy Serializable.
63  */
64 class Employee extends Serializable
65 {
66     private String name;
67     private double salary;
68     private LocalDate hireDay;
69
70     public Employee(String n, double s, int year, int month, int day)
71     {
72         name = n;
73         salary = s;
74         hireDay = LocalDate.of(year, month, day);
75     }
76
77     public String getName()
78     {
79         return name;
80     }
81
82     public double getSalary()
83     {
84         return salary;
85     }
86
87     public LocalDate getHireDay()
88     {
89         return hireDay;
90     }
91
92     /**
93     * Metoda zwiększająca wynagrodzenie tego pracownika.
94     * @byPercent procentowa wartość wzrostu wynagrodzenia
95     */
96     public void raiseSalary(double byPercent)
97     {
98         double raise = salary * byPercent / 100;
99         salary += raise;

```

```

100 }
101
102 public String toString()
103 {
104     return getClass().getName()
105         + "[name=" + name
106         + ",salary=" + salary
107         + ",hireDay=" + hireDay
108         + "];";
109 }
110 }

```

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

2.4. Zarządzanie plikami

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

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

2.4.1. 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 wyrzuca wyjątek `InvalidPathException`.

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

```
String baseDir = props.getProperty("base.dir")
// Może zawierać łańcuch 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`.
- W przeciwnym razie wynik ma postać połączenia „`p` potem `q`” wykonanego zgodnie z regułami systemu 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 = Paths.get("work");
Path workPath = basePath.resolve(workRelative);
```

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

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

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

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

utworzy ścieżkę `/opt/myapp/temp`.

Działaniem odwrotnym do rozwiązywania ścieżki jest jej *relatywizacja*. Wywołanie `p.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 /
```

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

```
var in = new Scanner(Paths.get("/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.Paths 7

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

API java.nio.file.Path 7

- `Path resolve(Path other)`
- `Path resolve(String other)`
jeśli `other` jest ścieżką bezwzględną, zwraca `other`; w przeciwnym razie zwraca ścieżkę powstałą z połączenia `this` i `other`.
- `Path resolveSibling(Path other)`
- `Path resolveSibling(String other)`
jeśli `other` jest ścieżką bezwzględną, zwraca `other`; w przeciwnym razie zwraca ścieżkę powstałą z połączenia ścieżki nadrzędnej dla `this` i `other`.
- `Path relativize(Path other)`
zwraca ścieżkę względną, która rozwiązana względem `this` daje w wyniku `other`.
- `Path normalize()`
usuwa nadmiarowe elementy ścieżki, takie jak `.` i `..`.
- `Path toAbsolutePath()`
zwraca ścieżkę bezwzględną odpowiadającą danej ścieżce.
- `Path getParent()`
zwraca ścieżkę nadrzędną lub `null`, gdy ścieżka nadrzędna nie istnieje.
- `Path getFileName()`
zwraca ostatni komponent ścieżki lub `null`, gdy ścieżka nie ma komponentów.

- `Path getRoot()`
zwraca komponent korzenia danej ścieżki lub `null`, gdy ścieżka nie ma takiego komponentu.
- `toFile()`
tworzy obiekt `File` dla danej ścieżki.

API `java.io.File 1.0`

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

2.4.2. Odczyt i zapis plików

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

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

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

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

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

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

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

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

Aby dopisać łańcuch do pliku, zastosujemy poniższe wywołanie.

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

Kolekcję wierszy możemy zapisać w następujący sposób:

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

Te proste metody zostały udostępnione z myślą o obsłudze plików tekstowych o umiarkowanych rozmiarach. Jeśli przetwarzane pliki są znacznych rozmiarów lub zawierają dane binarne, to nadal z powodzeniem możemy używać poznanych wcześniej strumieni wejścia-wyjścia oraz obiektów `Reader/Writer`:

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

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

API `java.nio.file.Files` 7

- `static byte[] readAllBytes(Path path)`
- `static List<String> readAllLines(Path path, Charset charset)`
wzyciują zawartość pliku.
- `static Path write(Path path, byte[] contents, OpenOption... options)`
- `static Path write(Path path, Iterable<? extends CharSequence> contents, OpenOption options)`
zapisują podaną zawartość w pliku i zwracają ścieżkę.
- `static InputStream newInputStream(Path path, OpenOption... options)`
- `static OutputStream newOutputStream(Path path, OpenOption... options)`
- `static BufferedReader newBufferedReader(Path path, Charset charset)`
- `static BufferedWriter newBufferedWriter(Path path, Charset charset, OpenOption... options)`
otwierają plik do odczytu lub zapisu.

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

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

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

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

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

API java.nio.file.Files 7

- `static Path createFile(Path path, FileAttribute<?>... attrs)`
 - `static Path createDirectory(Path path, FileAttribute<?>... attrs)`
 - `static Path createDirectories(Path path, FileAttribute<?>... attrs)`
tworzą plik lub katalog. Metoda `createDirectories` tworzy również katalogi pośrednie.
 - `static Path createTempFile(String prefix, String suffix, FileAttribute<?>... attrs)`
 - `static Path createTempFile(Path parentDir, String prefix, String suffix, FileAttribute<?>... attrs)`
 - `static Path createTempDirectory(String prefix, FileAttribute<?>... attrs)`
 - `static Path createTempDirectory(Path parentDir, String prefix, FileAttribute<?>... attrs)`
- tworzą plik lub katalog tymczasowy w lokalizacji przewidzianej dla takich plików lub w podanym katalogu nadrzędnym. Zwracają ścieżkę dostępu do utworzonego pliku lub katalogu.

2.4.4. Kopiowanie, przenoszenie i usuwanie plików

Aby skopiować plik z jednej lokalizacji do innej, wywołamy:

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

Aby przenieść plik (czyli skopiować go i usunąć oryginał), użyjemy następującego wywołania:

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

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

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

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

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

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


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

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

Aby usunąć plik, wywołujemy po prostu:

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

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

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

Metody te możemy również wykorzystać do usunięcia pustego katalogu.

Lista wszystkich opcji, których można używać w operacjach na plikach, została podana w tabeli 2.3.

Tabela 2.3. Standardowe opcje stosowane w operacjach na plikach

| Opcja | Opis |
|--|---|
| StandardOpenOption — do stosowania w metodach <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 przenoszą from do lokalizacji to. Zwracają to.
- static long copy(InputStream from, Path to, CopyOption... options)
- static long copy(Path from, OutputStream to, CopyOption... options)
kopiuje dane ze strumienia wejściowego do pliku lub z pliku do strumienia wyjściowego, zwracając przy tym liczbę skopiowanych bajtów.
- static void delete(Path path)
- static boolean deleteIfExists(Path path)
zwracają podany plik lub pusty katalog. Metoda delete wyrzuca wyjątek, jeśli plik lub katalog nie istnieje. W takim przypadku metoda deleteIfExists zwraca wartość false.

2.4.5. 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.

2.4.6. Przeglądanie zawartości katalogu

Stacyczna 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.relative(p));
        if (Files.isDirectory(p))
            Files.createDirectory(q);
        else
            Files.copy(p, q);
    }
    catch (IOException ex)
    {
        throw new UncheckedIOException(ex);
    }
});
```

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

2.4.7. Stosowanie strumieni katalogów

Jak przekonaaliśmy 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.

Przełądane pliki możemy filtrować za pomocą wzorca:

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

Dostępne wzorce zostały przedstawione w tabeli 2.4.

Tabela 2.4. Wzorce filtrowania plików

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



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

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

- gdy napotkany zostanie 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 wyrzuci wyjątek, przeglądanie zostanie zakończone, a metoda `walkFileTree` wyrzuci ten wyjątek.



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

Klasa `SimpleFileVisitor` implementuje interfejs `FileVisitor`. Metody — oprócz `visitFileFailed` — nie podejmują żadnych działań, powodując tym samym kontynuację przeglądania. Metoda `visitFileFailed` wyrzuca wyjątek będący przyczyną błędu i w ten sposób kończy przeglądanie.

Poniżej przedstawiamy fragment kodu wyświetlający wszystkie podkatalogi danego katalogu.

```
Files.walkFileTree(Paths.get("/"), new SimpleFileVisitor<Path>()
{
    public FileVisitResult preVisitDirectory(Path path,
        BasicFileAttributes attrs) throws IOException
    {
        System.out.println(path);
        return FileVisitResult.CONTINUE;
    }
    public FileVisitResult postVisitDirectory(Path dir, IOException exc)
    {
        return FileVisitResult.CONTINUE;
    }
    public FileVisitResult visitFileFailed(Path path, IOException exc)
        throws IOException
    {
        return FileVisitResult.SKIP_SUBTREE;
    }
}):
```

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

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

Pozostałe metody interfejsu `FileVisitor` przydają się, gdy 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, BasicFileAttributes attrs)`
wywoływane przed i po wizycie w katalogu. Domyślna implementacja nie podejmuje żadnych działań, tym samym powodując kontynuację przeglądania.
- static `FileVisitResult visitFileFailed(T path, IOException exc)`
wywoływana, gdy podczas próby uzyskania informacji o danym pliku został wyrzucony wyjątek. Domyślna implementacja ponownie wyrzuca ten wyjątek, powodując zakończenie przeglądania. Jeśli przeglądanie ma być kontynuowane, należy zastąpić metodę własną implementacją.

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

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

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

Metoda `fs.getPath` działa analogicznie do `Paths.get` dla dowolnego systemu plików.

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

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

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

API `java.nio.file.FileSystems` 7

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

przełąda zainstalowanych dostawców systemów plików oraz systemy plików, które może załadować `loader` (jeśli jest różny od `null`). Zwraca system plików utworzony przez pierwszego dostawcę, który akceptuje podaną ścieżkę. Domyślnie istnieje dostawca systemu plików ZIP akceptujący nazwy plików kończące się rozszerzeniem `.zip` lub `.jar`.

API `java.nio.file.FileSystem` 7

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

tworzy ścieżkę, łącząc podane łańcuchy.

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

Tabela 2.5. Czasy wykonywania operacji na pliku

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

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

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

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

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

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

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

- `FileChannel.MapMode.READ_ONLY`: otrzymany bufor umożliwia wyłącznie odczyt danych. Jakakolwiek próba zapisu do bufora spowoduje wyrzucenie wyjątku `ReadOnlyBufferException`.

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

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

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

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

Alternatywnie możemy również wykorzystać dostęp swobodny:

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

Możemy także czytać tablice bajtów, stosując metody:

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

Dostępne są również poniższe metody:

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

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

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

Aby poznać bieżący sposób uporządkowania bajtów w buforze, wywołujemy:

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



Ta para metod nie stosuje konwencji nazw `set/get`.

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

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

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

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

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

Listing 2.5. `memoryMap/MemoryMapTest.java`

```
1 package memoryMap;
2
3 import java.io.*;
4 import java.nio.*;
5 import java.nio.channels.*;
6 import java.nio.file.*;
7 import java.util.zip.*;
8
9 /**
10  * Program obliczający sumę kontrolną CRC pliku.
11  * Uruchamianie: java memoryMap.MemoryMapTest nazwapliku
12  * @version 1.02 2018-05-01
13  * @author Cay Horstmann
14  */
15 public class MemoryMapTest
16 {
17     public static long checksumInputStream(Path filename) throws IOException
18     {
19         try (InputStream in = Files.newInputStream(filename))
20         {
21             var crc = new CRC32();
22
23             int c;
24             while ((c = in.read()) != -1)
25                 crc.update(c);
26             return crc.getValue();
27         }
28     }
29
30     public static long checksumBufferedInputStream(Path filename) throws IOException
31     {
32         try (var in = new BufferedInputStream(Files.newInputStream(filename)))
33         {
34             var crc = new CRC32();
35
36             int c;
37             while ((c = in.read()) != -1)
38                 crc.update(c);
```

```
39         return crc.getValue();
40     }
41 }
42
43 public static long checksumRandomAccessFile(Path filename) throws IOException
44 {
45     try (var file = new RandomAccessFile(filename.toFile(), "r"))
46     {
47         long length = file.length();
48         var crc = new CRC32();
49
50         for (long p = 0; p < length; p++)
51         {
52             file.seek(p);
53             int c = file.readByte();
54             crc.update(c);
55         }
56         return crc.getValue();
57     }
58 }
59
60 public static long checksumMappedFile(Path filename) throws IOException
61 {
62     try (FileChannel channel = FileChannel.open(filename))
63     {
64         var crc = new CRC32();
65         int length = (int) channel.size();
66         MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ_ONLY, 0, length);
67
68         for (int p = 0; p < length; p++)
69         {
70             int c = buffer.get(p);
71             crc.update(c);
72         }
73         return crc.getValue();
74     }
75 }
76
77 public static void main(String[] args) throws IOException
78 {
79     System.out.println("Strumień wejściowy:");
80     long start = System.currentTimeMillis();
81     Path filename = Paths.get(args[0]);
82     long crcValue = checksumInputStream(filename);
83     long end = System.currentTimeMillis();
84     System.out.println(Long.toHexString(crcValue));
85     System.out.println((end - start) + " milisekund");
86
87     System.out.println("Buforowany strumień wejściowy:");
88     start = System.currentTimeMillis();
89     crcValue = checksumBufferedInputStream(filename);
90     end = System.currentTimeMillis();
91     System.out.println(Long.toHexString(crcValue));
92     System.out.println((end - start) + " milisekund");
93
94     System.out.println("Plik o dostępie swobodnym:");
95     start = System.currentTimeMillis();
```

```

96     crcValue = checksumRandomAccessFile(filename);
97     end = System.currentTimeMillis();
98     System.out.println(Long.toHexString(crcValue));
99     System.out.println((end - start) + " milisekund");
100
101     System.out.println("Plik mapowany w pamięci:");
102     start = System.currentTimeMillis();
103     crcValue = checksumMappedFile(filename);
104     end = System.currentTimeMillis();
105     System.out.println(Long.toHexString(crcValue));
106     System.out.println((end - start) + " milisekund");
107 }
108 }

```

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 wyrzucony 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 wystarczającego miejsca, to nie są zapisywane żadne bajty i zostaje wyrzucony 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`.

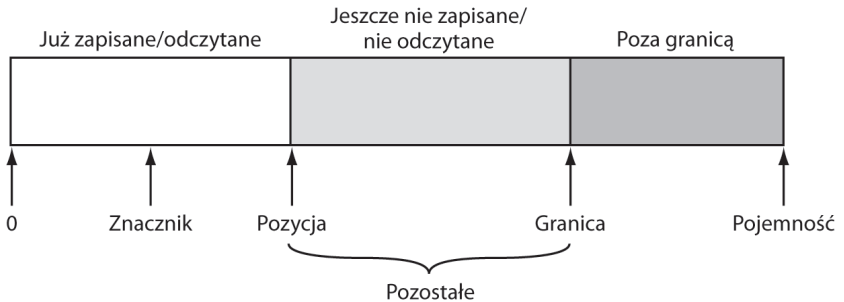


Klasa `StringBuffer` nie jest związana z omawianą tutaj hierarchią klas.

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:

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

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

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

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

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

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

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

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

API java.nio.Buffer 1.4

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

2.6. Blokowanie plików

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

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

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

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

lub

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

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

Można również zablokować dostęp do fragmentu pliku za pomocą wywołania

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

lub

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

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



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

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

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

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

- W niektórych systemach blokady plików mają jedynie charakter *pomocniczy*. Nawet jeśli aplikacji nie uda się zdobyć blokady, to może zapisywać dane w pliku „zablokowanym” wcześniej przez inną aplikację.
- W niektórych systemach nie jest możliwe zablokowanie dostępu do mapy pliku w pamięci.

- Blokady plików są przydzielane na poziomie maszyny wirtualnej Java. Jeśli zatem dwa programy działają na tej samej maszynie wirtualnej, to nie mogą uzyskać blokady tego samego pliku. Metody `lock` i `tryLock` wyrzucą wyjątek `OverlappingFileLockException` w sytuacji, gdy maszyna wirtualna jest już w posiadaniu blokady danego pliku.
- W niektórych systemach zamknięcie kanału zwalnia wszystkie blokady pliku będące w posiadaniu maszyny wirtualnej Java. Dlatego też należy unikać wielu kanałów dostępu do tego samego, zablokowanego pliku.
- Działanie blokad plików w sieciowych systemach plików zależy od konkretnego systemu i dlatego należy unikać stosowania blokad w takich systemach.

API java.nio.channels.FileChannel 1.4

- `FileLock lock()`
uzyskuje wyłączną blokadę pliku. Blokuje działanie programu do momentu uzyskania blokady.
- `FileLock tryLock()`
uzyskuje wyłączną blokadę całego pliku lub zwraca `null`, jeśli nie może uzyskać blokady.
- `FileLock lock(long position, long size, boolean shared)`
- `FileLock tryLock(long position, long size, boolean shared)`
uzyskuje blokadę dostępu do fragmentu pliku. Pierwsza wersja blokuje działanie programu do momentu uzyskania blokady, a druga zwraca natychmiast wartość `null`, jeśli nie może uzyskać od razu blokady. Parametr `shared` ma wartość `true` dla blokady współdzielonej, `false` dla wyłącznej.

API java.nio.channels.FileLock 1.4

- `void close() 1.7`
zwalnia blokadę.

2.7. Wyrażenia regularne

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

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.

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

Tabela 2.6. Składnia wyrażeń regularnych

| Składnia | Objaśnienie | Przykład |
|--|--|---|
| Znaki | | |
| c, ale żadne z: . * + ? { () [\ ^ \$ | Znak c. | J |
| . | Dowolny znak oprócz kończącego wiersz (lub dowolny znak, jeśli znacznik DOTALL został ustawiony). | |
| \x{p} | Znak Unicode o szesnastkowej wartości p. | \x{1D546} |
| \uhhhh, \xhh, \0o, \0oo, \0ooo | Znak o kodzie, którego wartość została podana w notacji szesnastkowej lub ósemkowej. | \uFEFF |
| \a, \e, \f, \n, \r, \t | Znaki sterujące: alertu (\x{7}), sekwencji sterującej (\x{1B}), końca strony (\x{B}), nowego wiersza (\x{A}), powrotu karetki (\x{D}) i tabulacji (\x{9}). | \n |
| \cc, gdzie c należy do [A-Z] bądź jest jednym ze znaków @ [\] ^ _ ? | Znak sterujący odpowiadający znakowi c. | \cH to znak cofnięcia (ang. <i>backspace</i> ; \x{8}) |
| \c, gdzie c nie należy do [A-Za-z0-9] | Znak c. | \\ |
| \Q . . . \E | Wszystko pomiędzy początkiem i końcem cytatu. | \Q(. . .)\E pasuje do łańcucha (. . .) |
| Klasy znaków | | |
| [C ₁ C ₂ . . .], gdzie C _i jest znakiem, zakresem znaków c – d lub klasą znaków | Dowolny ze znaków reprezentowanych przez C ₁ C ₂ . . . | [0-9+ -] |
| [^ . . .] | Dopełnienie klasy znaków. | [^\d\s] |
| [. . .&&. . .] | Część wspólna (przecięcie) dwóch klas znaków. | [\p{L}&&^[A-Za-z]] |
| \p{. . .}, \P{. . .} | Predefiniowana klasa znaków (patrz tabela 2.7); dopełnienie predefiniowanej klasy znaków. | \p{L} odpowiada literze Unicode, podobnie jak \pL — można pominąć nawiasy klamrowe wokół pojedynczej litery |
| \d, \D | Cyfry ([0-9] lub \p{Digit} w przypadku użycia flagi UNICODE_CHARACTER_CLASS); dopełnienie, czyli znak, który nie jest cyfrą. | \d+ — sekwencja cyfr |
| \w, \W | Znak słowa ([a-zA-Z0-9_] lub znak słowa w Unicode, jeśli została użyta flaga UNICODE_CHARACTER_CLASS). | |

Tabela 2.6. Składnia wyrażeń regularnych — ciąg dalszy

| Składnia | Objaśnienie | Przykład |
|---|--|--|
| Klasy znaków | | |
| <code>\s, \S</code> | Znak odstępu (<code>[\t\n\r\f\v\x0B]</code> lub <code>\p{IsWhiteSpace}</code> , jeśli została użyta flaga <code>UNICODE_CHARACTER_CLASS</code>); znak, który nie jest odstępem. | <code>\s*, \S*</code> — przecinek opcjonalnie otoczony znakami odstępu |
| <code>\h, \v, \H, \V</code> | Odstęp w poziomie, odstęp w pionie; dopełnienia tych znaków. | |
| Sekwencje i alternatywy | | |
| <code>XY</code> | Dowolny łańcuch z <i>X</i> , po którym następuje dowolny łańcuch z <i>Y</i> . | <code>[1-9][0-9]*</code> — liczba dodatnia bez początkowego zera |
| <code>X Y</code> | Dowolny łańcuch z <i>X</i> lub <i>Y</i> . | <code>http ftp</code> |
| Grupowanie | | |
| <code>(X)</code> | Przechwycenie dopasowania <i>X</i> . | <code>'([^\']**)'</code> — tekst w apostrofach |
| <code>\n</code> | Dopasowanie <i>n</i> -tej grupy. | <code>(["']).*\1</code> — pasuje do "Fred" lub 'Fred', lecz nie do "Fred" |
| <code>(?<nazwa>X)</code> | Przechwytyje dopasowanie <i>X</i> , nadając mu podaną nazwę. | <code>'(?<id>[A-Za-z0-9]+)'</code> przechwytyje dopasowanie, nadając mu nazwę <code>id</code> |
| <code>\k<nazwa></code> | Grupa o podanej nazwie. | <code>\k<id></code> dopasowuje grupę o nazwie <code>id</code> |
| <code>(?:X)</code> | Zastosowane nawiasów bez przechwytywania <i>X</i> . | W wyrażeniu regularnym <code>(?:http ftp)://(.*)</code> dopasowanie po <code>://</code> jest grupą <code>\1</code> |
| <code>(?f₁f₂...:X)</code> , <code>(?f₁...f_k...:X)</code> , gdzie <i>f_i</i> należy do <code>[dimsuX]</code> | Dopasowuje <i>X</i> , lecz go nie przechwytyje, używając lub nie używając (jeśli zastosowano -) określonych flag. | <code>(?:jpe?g)</code> — w dopasowaniu nie będzie uwzględniana wielkość liter |
| Inne <code>(? ...)</code> | Patrz dokumentacja API Pattern. | |
| Kwantyfikatory | | |
| <code>X?</code> | Opcjonalnie <i>X</i> . | <code>\x?</code> to opcjonalny znak + |
| <code>X*, X+</code> | <i>X</i> , 0 lub więcej razy oraz <i>X</i> co najmniej jeden raz. | <code>[1-9][0-9]+</code> to liczba całkowita większa od 0. |
| <code>X{n} X{n,} X{n,m}</code> | <i>X</i> <i>n</i> razy, co najmniej <i>n</i> razy, pomiędzy <i>n</i> i <i>m</i> razy. | <code>[0-7]{1,3}</code> od jednej do trzech cyfr ósemkowych. |

Tabela 2.6. Składnia wyrażeń regularnych — ciąg dalszy

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

Tabela 2.7. Wstępnie zdefiniowane nazwy klas znaków

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

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

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

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

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

Niestety, składnia wyrażeń regularnych nie jest całkowicie ustandaryzowana. Istnieje zgodność w zakresie podstawowych konstrukcji, ale diabeł tkwi w szczegółach. Klasy języka Java związane z przetwarzaniem wyrażeń regularnych używają składni podobnej do zastosowanej w języku Perl. Wszystkie konstrukcje tej składni zostały przedstawione w tabeli 2.6. 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. Dopasowywanie wyrażeń regularnych do łańcucha

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

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

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

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

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

Ewentualnie można je także podawać w wyrażeniu:

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

Obsługiwane są poniższe znaczniki:

- `Pattern.CASE_INSENSITIVE` lub `i` — dopasowanie niezależnie od wielkości liter. Domyślnie dotyczy to tylko znaków US ASCII.
- `Pattern.UNICODE_CASE` lub `U` — zastosowane w połączeniu z `CASE_INSENSITIVE`; dotyczy wszystkich znaków Unicode.

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

Ostatnich dwóch znaczników nie można używać wewnątrz wyrażeń regularnych.

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

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

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

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

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

zwracają indeks początkowy i końcowy podanej grupy.

Dopasowany łańcuch możemy pobrać, wywołując

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

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

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

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

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

dla danych

```
11:59am
```

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

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

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

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

Listing 2.6. `regex/RegexTest.java`

```

1 package regex;
2
3 import java.util.*;
4 import java.util.regex.*;
5
6 /**
7  * Program testujący zgodność z wyrażeniem regularnym. Wprowadź wzorzec
8  * i dopasowywany łańcuch. Jeśli wzorzec zawiera grupy, to po dopasowaniu program
9  * wyświetli ich granice.
10 * @version 1.03 2018-05-01
11 * @author Cay Horstmann
12 */
13 public class RegexTest
14 {
15     public static void main(String[] args) throws PatternSyntaxException
16     {
17         var in = new Scanner(System.in);
18         System.out.println("Wpisz wyrażenie regularne: ");
19         String patternString = in.nextLine();
20
21         Pattern pattern = Pattern.compile(patternString);
22
23         while (true)
24         {
25             System.out.println("Wpisz łańcuch znaków: ");
26             String input = in.nextLine();
27             if (input == null || input.equals("")) return;
28             Matcher matcher = pattern.matcher(input);
29             if (matcher.matches())
30             {
31                 System.out.println("Dopasowano");
32                 int g = matcher.groupCount();
33                 if (g > 0)
34                 {
35                     for (int i = 0; i < input.length(); i++)
36                     {
37                         // Wyświetla puste grupy
38                         for (int j = 1; j <= g; j++)
39                             if (i == matcher.start(j) && i == matcher.end(j))
40                                 System.out.print("(");

```

```

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

```

2.7.3. Znajdowanie wielu dopasowań

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

```

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

```

W ten sposób można kolejno przetworzyć wszystkie dopasowania. Jak pokazałem na przykładzie, można pobrać zarówno dopasowany łańcuch znaków, jak i jego położenie w łańcuchu wejściowym.

Jeszcze bardziej eleganckim rozwiązaniem jest wywołanie metody `results` w celu pobrania strumienia 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(Matcher::group)
    .collect(Collectors.toList());

```

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

```
var in = new Scanner(path, StandardCharsets.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*

```
1 package match;
2
3 import java.io.*;
4 import java.net.*;
5 import java.nio.charset.*;
6 import java.util.regex.*;
7
8 /**
9  * Program wyświetlający wszystkie adresy URL na stronie WWW poprzez dopasowanie
10 * wyrażenia regularnego opisującego znacznik <a href=...> języka HTML.
11 * Uruchamianie: java match.HrefMatch adresURL
12 * @version 1.03 2018-03-19
13 * @author Cay Horstmann
14 */
15 public class HrefMatch
16 {
17     public static void main(String[] args)
18     {
19         try
20         {
21             // Pobiera URL z wiersza poleceń lub używa domyślnego
22             String urlString;
23             if (args.length > 0) urlString = args[0];
24             else urlString = "http://openjdk.java.net/";
25
26             // Wczytuje zawartość strony
27             InputStream in = new URL(urlString).openStream();
28             var input = new String(in.readAllBytes(), StandardCharsets.UTF_8);
29
30             // Wyszukuje wszystkie wystąpienia wzorca
31             var patternString = "<a\\s+href\\s*=\\s*(\"[^\"]*)\"|\"[^\s>]*\"\\s*>";
32             Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);
33             pattern.matcher(input)
34                 .results()
35                 .map(MatchResult::group)
36                 .forEach(System.out::println);
37         }
38         catch (IOException | PatternSyntaxException e)
39         {
40             e.printStackTrace();
41         }
42     }
43 }
```

2.7.4. Podział w miejscach wystąpienia separatora

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

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

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

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

Jeśli ani wstępna kompilacja wyrażenia regularnego, ani pobieranie leniwe nie mają znaczenia, to można także skorzystać z metody `String.split`:

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

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

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

2.7.5. Zastępowanie dopasowań

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

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

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

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

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.

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

rozbija 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.

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.
- `int start()`
- `int end()`
zwraca pozycję początkową dopasowania lub następną pozycję za dopasowaniem.
- `String group()`
zwraca bieżące dopasowanie.

- `int groupCount()`
zwraca liczbę grup we wzorcu wejściowym.
- `int start(int groupIndex)`
- `int start(String name)` **8**
- `int end(int groupIndex)`
- `int end(String name)` **8**
zwraca pozycję początkową grupy lub następną pozycję za grupą dla danej grupy bieżącego dopasowania. Grupa jest określana jako indeks liczbowy zaczynający się od 1, 0 dla całego dopasowania lub łańcuch znaków dla grupy nazwanej.
- `String group(int groupIndex)`
- `String group(String name)` **7**
zwraca łańcuch dopasowany do podanej grupy, określonej jako indeks liczbowy zaczynający się od 1, 0 dla całego dopasowania lub łańcuch znaków dla grupy nazwanej.
- `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)`
zwraca dopasowany łańcuch lub łańcuch dopasowany przez określoną grupę.
- `int start()`
- `int end()`
- `int start(int group)`

- `int end(int group)`

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

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

- @ActionListenerFor, 450
 - @BugReport, 450
 - @Deprecated, 447, 448
 - @Documented, 447, 450
 - @Inherited, 447, 450
 - @LogEntry, 456
 - @Override, 447
 - @Persistent, 451
 - @Retention, 447, 449
 - @Serializable, 451
 - @SuppressWarnings, 447, 448
 - @Target, 447, 449
- A**
- abort(), 540
 - absolute(), 311
 - AbstractCellEditor, 597, 598
 - AbstractTableModel, 572, 592
 - accept(), 229, 231
 - acceptChanges(), 315, 316
 - ActionListener, 436
 - ActionListenerFor, 436, 442
 - ActionListenerInstaller, 435
 - processAnnotations(), 435
 - actionPerformed(), 436
 - add(), 609, 731
 - addActionListener(), 435, 436
 - addAttribute(), 220
 - addBatch(), 329
 - addCellEditorListener(), 601
 - addClass(), 626
 - addColumn(), 582, 587
 - addSelectionListener(), 621
 - addTableModelListener(), 632, 635
 - adnotacje
 - @ActionListenerFor, 435, 450
 - @Deprecated, 447, 448
 - @Documented, 447, 450
 - @Inherited, 447, 450
 - @interface, 435
 - @LogEntry, 456
 - @Override, 447, 448
 - @Persistent, 451
 - @Retention, 435, 447, 449
 - @Serializable, 451
 - @SuppressWarnings, 447, 448
 - @Target, 435, 447, 449
 - ActionListenerFor, 436, 442
 - cykliczne zależności, 443
 - deklaracja elementu, 440
 - elementy, 434
 - format, 441
 - interfejs, 434, 440
 - kod bajtowy, 456
 - kolejność elementów, 442
 - metaadnotacje, 449
 - metody, 434
 - modyfikacja kodu bajtowego
 - podczas ładowania, 461
 - obsługa zdarzeń, 435
 - pojedyncze wartości, 442
 - przetwarzanie, 452
 - regularne, 448
 - składnia, 440
 - skrót, 442
 - standardowe, 447
 - typy elementów, 441, 449
 - znacznikowe, 442
- adres internetowy, 227
 - adres localhost, 231
 - adres URI, 242
 - adres URL, 191, 242, 280, 515
 - AES, 555, 557, 562
 - AffineTransform, 665, 666
 - getRotateInstance(), 666, 667
 - getScaleInstance(), 667
 - getShearInstance(), 667
 - getTranslateInstance(), 667
 - setToRotation(), 667
 - setToScale(), 667
 - setToShear(), 667
 - setToTranslation(), 667
 - AffineTransformOp, 695, 701
 - TYPE_BILINEAR, 695
 - TYPE_NEAREST_NEIGHBOR, 695
 - afterLast(), 312
 - aktualizacje wsadowe, 327
 - aktualizowalne zbiory wyników zapytań, 306, 309
 - alfa, 670, 671
 - algorytmy
 - AES, 555, 557, 562
 - DES, 555
 - DSA, 544
 - kryptograficzne, 494
 - MD5, 541
 - RSA, 545, 563
 - SHA1, 541
 - zyfrowania, 555
 - z kluczem symetrycznym, 562
 - AllPermission, 517, 520
 - AllPermissions, 511
 - AlphaComposite, 673
 - getInstance(), 678
 - analiza wyjątków SQL, 289
 - animacje GIF, 685
 - AnnotatedElement, 436, 439
 - getAnnotation(), 439
 - getAnnotations(), 440
 - getDeclaredAnnotations(), 440
 - isAnnotationPresent(), 439
 - Annotation, 440, 441
 - annotationType(), 441
 - equals(), 441
 - hashCode(), 441
 - toString(), 441
 - Apache, 332
 - aplety, 540
 - lokalizacja, 396
 - aplikacje interaktywne, 236
 - aplikacje klient-serwer, 274
 - append(), 650, 652

- appendChild(), 203, 205
 - applyPattern(), 385
 - Arc2D, 639, 641
 - Arc2D.Double, 651
 - architektura JDBC, 272
 - architektura trójwarstwowa, 274
 - ArcMaker, 651
 - Area, 653
 - ARGB, 690, 694
 - Array, 330
 - ARRAY, 330
 - ArrayIndexOutOfBoundsException, 765
 - ArrayList, 633
 - ArrayStoreException, 765
 - ASCII, 357
 - ASP, 251
 - atrybuty, 159, 173
 - atrybuty drukowania, 725, 729
 - dokumenty, 725
 - hierarchia, 726
 - klasy, 728
 - usługi drukowania, 726
 - wydruck, 726
 - zbiór, 727, 728
 - żądanie wydruku, 725
 - AttributeSet, 726
 - ATTLIST, 173, 180
 - Attribute, 726, 727, 731
 - getCategory(), 731
 - getName(), 731
 - Attributes, 198
 - getLength(), 198
 - getLocalName(), 198
 - getQName(), 198
 - getURI(), 198
 - getValue(), 198
 - AttributeSet, 731
 - add(), 731
 - get(), 731
 - remove(), 731
 - toArray(), 732
 - AttributesImpl, 220
 - addAttribute(), 220
 - AudioPermission, 517
 - AuthPermission, 517
 - AuthTest.policy, 530
 - AWT, 635
 - drukowanie, 703
 - figury, 638
 - filtrowanie obrazów, 694
 - Graphics, 635, 638
 - obszar przycięcia, 636
 - operacje na obrazach, 688
 - pliki graficzne, 678
 - pola, 652
 - potokowe tworzenie grafiki, 635
 - przekształcenia układu
 - współrzędnych, 636, 663
 - przezroczystość, 670
 - prycinanie, 668
 - rysowanie figur, 636
 - składanie obrazów, 670
 - śląd pędzla, 636, 653
 - wypełnianie obszaru, 636, 661
 - zasady składania obrazów, 637
 - AWTPermission, 516
- ## B
- Banner, 712
 - BasicPermission, 515
 - BasicStroke, 653, 655, 659, 660
 - baza danych, 271
 - adres URL, 280
 - aktualizowalne zbiory wyników
 - zapytań, 309
 - JNDI, 331
 - kolumny, 275
 - kursory, 308
 - łączenie tabel, 277
 - metadane, 317
 - model dostępu, 272
 - modyfikacja danych, 278
 - ODBC, 272
 - przewijanie zbioru rekordów, 308
 - rekordy, 275
 - rekordy wstawiania, 310
 - spójność, 326
 - SQL, 271
 - tabele, 275
 - transakcje, 326
 - uruchamianie, 281
 - wstawianie danych, 279
 - wypełnianie, 292
 - zapytania, 276
 - zarządzanie połączeniami, 331
 - zbiory rekordów, 313
 - BCEL, 456
 - BEA WebLogic, 332
 - beforeFirst(), 312
 - bezpieczeństwo, 493
 - hierarchia klas pozwoleń, 510
 - JAAS, 526
 - Java 2, 509
 - klasy pozwoleń, 519
 - kryptografia klucza
 - publicznego, 563
 - ładowanie klas, 494
 - menedżer bezpieczeństwa, 494, 508
 - piaskownica, 509
 - pliki polityki, 509
 - podpis cyfrowy, 540
 - podpisywanie kodu, 552
 - polityka bezpieczeństwa, 510
 - pozwolenia, 508, 510
 - przydzielanie praw, 509
 - skrótowy wiadomości, 541
 - szyfrowanie, 555
 - uwierzytelnianie użytkowników, 526
 - uwierzytelnianie wiadomości, 548
 - weryfikacja kodu maszyny
 - wirtualnej, 504
 - źródło kodu, 509
 - biblioteka AWT, 635
 - biblioteki
 - BCEL, 456
 - DLL, 737
 - JCE, 561
 - BigDecimal, 330
 - Blob, 302, 330
 - BLOB, 279, 301, 330
 - boolean, 330
 - BOOLEAN, 279, 330
 - breadthFirstEnumeration(), 616, 620
 - Buffer, 236
 - BufferedImage, 662, 673, 688, 693
 - getColorModel(), 693
 - getRaster(), 693
 - BufferedImageOp, 688, 694, 695, 701
 - filter(), 701
 - buforowane zbiory rekordów, 314
 - ButtonFrame, 435
 - ByteLookupTable, 699, 702
- ## C
- C, 734
 - łańcuchy znakowe platformy
 - Java, 744
 - obsługa błędów, 764, 768
 - tablice, 760
 - wywoływanie metod języka
 - Java, 754, 759
 - wywoływanie metod
 - statycznych, 757
 - C++, 736
 - CachedRowSet, 313, 314, 316
 - acceptChanges(), 316
 - execute(), 316
 - getTableName(), 316
 - populate(), 316
 - setTableName(), 316

- Caesar, 502
- Callback, 537
- CallbackHandler, 539
 - handle(), 539
- CallNonVirtualXxxMethod(), 758, 759
- CallNonVirtualXxxMethodA(), 759
- CallNonVirtualXxxMethodV(), 759
- CallStaticObjectMethod(), 757
- CallStaticXxxMethod(), 757, 760
- CallStaticXxxMethodA(), 760
- CallStaticXxxMethodV(), 760
- CallXxxMethod(), 759
- CallXxxMethodA(), 759
- CallXxxMethodV(), 759
- cancelCellEditing(), 598, 599, 600
- cancelRowUpdates(), 312
- canInsertImage(), 684, 688
- CDATA, 160, 174
- CellEditor, 600
 - addCellEditorListener(), 601
 - cancelCellEditing(), 600
 - getCellEditorValue(), 601
 - isCellEditable(), 600
 - removeCellEditorListener(), 601
 - shouldSelectCell(), 600
 - stopCellEditing(), 600
- certyfikaty, 509
 - podpisywanie, 550
 - żądanie, 551
- certyfikaty X.509
 - keytool, 546
 - komponenty, 546
 - składnica kluczy, 546
 - sprawdzanie wiarygodności, 547
 - wydawcy certyfikatów, 547
 - zarządzanie, 546
- CGI, 251
- Channels, 241
 - newInputStream(), 241
 - newOutputStream(), 236, 241
- CHAR, 279, 330
- CHARACTER, 279
- CharacterData, 169
 - getData(), 169
- characters(), 194, 198
- checkExit(), 508, 509, 511
- checkLogin(), 533
- checkPermission(), 512, 520
- children(), 616
- ChoiceFormat, 387
- Chromaticity, 729
- Cipher, 555, 560
 - doFinal(), 560
 - getBlockSize(), 560
 - getInstance(), 560
 - getOutputSize(), 560
 - init(), 560
 - update(), 560
- CipherInputStream, 561, 562
 - read(), 562
- CipherOutputStream, 562
 - flush(), 562
 - write(), 562
- Class, 436, 503, 512, 617, 727
 - getClassLoader(), 503
 - getProtectionDomain(), 512
- ClassLoader, 495, 498, 503
 - defineClass(), 503
 - findClass(), 503
 - getParent(), 503
 - getSystemClassLoader(), 503
- ClassNameTreeCellRenderer, 620
- ClassTreeFrame, 620
- CLEAR, 672
- clearParameters(), 301
- clip(), 636, 668, 669
- Clob, 303, 330
- CLOB, 279, 301, 330
 - close(), 229, 231, 287, 288
 - closePath(), 643, 652
- CodeSource, 512
 - getCertificates(), 512
 - getLocation(), 512
- CollationKey, 383
 - compareTo(), 383
- Collator, 383
 - compare(), 383
 - equals(), 383
 - getAvailableLocales(), 383
 - getCollationKey(), 383
 - getDecomposition(), 383
 - getInstance(), 383
 - getStrength(), 383
 - PRIMARY, 378
 - setDecomposition(), 383
 - setStrength(), 383
- Collections
 - sort(), 378
- Color, 661, 670, 690
 - getRGB(), 694
- ColorConvertOp, 699
- ColorModel, 691, 694
 - getDataElements(), 694
 - getRGB(), 694
- ColorSupported, 729
- column(), 288
- commit(), 327, 328, 540
- Comparator, 378
- compare(), 383
- compareTo(), 383
- Compression, 729
- CONCUR_READ_ONLY, 307
- CONCUR_UPDATABLE, 307, 309
- connect(), 226, 244, 250
- Connection, 289, 301, 303, 311, 324
 - close(), 287
 - commit(), 328
 - createStatement(), 287, 311
 - getAutoCommit(), 328
 - getMetaData(), 324
 - prepareStatement(), 301, 311
 - releaseSavepoint(), 329
 - rollback(), 329
 - setAutoCommit(), 328
 - setSavepoint(), 329
- Constructor, 436
- containsAll(), 523
- ContentHandler, 194, 195, 197
 - characters(), 198
 - endDocument(), 197
 - endElement(), 198
 - startDocument(), 197
 - startElement(), 198
- CONTIGUOUS_TREE_SELECTION, 621
- ConvolveOp, 700, 702
- Copies, 726, 728, 729
- CopiesSupported, 726
- COREJAVA, 280
- CREATE TABLE, 279, 286, 292
- createBlob(), 303
- createClob(), 303
- createElement(), 202, 204
- createImageInputStream(), 683, 686
- createImageOutputStream(), 684, 686
- createPrintJob(), 722
- createStatement(), 285, 287, 307, 311, 326, 327, 328
- createTextNode(), 202, 204
- CubicCurve2D, 638, 643, 651
- Currency, 370
 - getCurrencyCode(), 370
 - getDefaultFractionsDigits(), 371
 - getInstance(), 370
 - getSymbol(), 371
 - toString(), 370
- curveTo(), 643, 652
- Cygwin, 737, 773
- czas, 371
- czcionki, 652, 668, 669
 - obrrys, 668

D

- DA, 728
- DafaultTreeModel, 611
- data, 358, 371
- database.properties, 331
- DatabaseMetaData, 289, 308, 313, 317, 318, 329
 - getJDBCMinorVersion(), 325
 - getJDBCMajorVersion(), 325
 - getMaxConnections(), 325
 - getMaxStatements(), 325
 - getTables(), 325
 - supportsBatchUpdates(), 329
 - supportsResultSetConcurrency(), 313
 - supportsResultSetType(), 313
- DataSource, 331
- Date, 330
- DATE, 279, 330
- DateFormat, 373
 - getDateInstance(), 377
 - getTimeInstance(), 377
- DateTimeAtCompleted, 729
- DateTimeAtCreation, 729
- DateTimeAtProcessing, 729
- DDL, 287
- DEC, 330
- DECIMAL, 279, 330
- decode(), 259
- DefaultCellEditor, 596, 599, 600, 612
- DefaultHandler, 195
- DefaultMutableTreeNode, 603, 607, 609, 616, 620
 - add(), 609
 - breadthFirstEnumeration(), 620
 - depthFirstEnumeration(), 620
 - postOrderEnumeration(), 620
 - preOrderEnumeration(), 620
 - setAllowsChildren(), 609
- defaultPage(), 710
- DefaultTreeCellRenderer, 618, 619, 620, 621
 - setClosedIcon(), 621
 - setLeafIcon(), 621
 - setOpenIcon(), 621
- DefaultTreeModel, 602, 609, 611, 616, 628
 - insertNodeInto(), 616
 - nodeChanged(), 616
 - reload(), 616
 - removeNodeFromParent(), 616
 - setAsksAllowsChildren(), 609
- defineClass(), 499, 503
- definicja typu dokumentu, 170
- deklaracja typu dokumentu, 158
- DELETE, 286
- deleteRow(), 310, 312
- depthFirstEnumeration(), 616, 620
- depthFirstTraversal(), 617
- DES, 555, 556
- deskryptory wdrożeń, 433
- Destination, 729
- DestroyJavaVM(), 769, 773
- DialogCallbackHandler, 536
- digest(), 543
- DISCONTIGUOUS_TREE_SELECTION, 621
- DLL, 737
- doAs(), 527, 528, 531
- doAsPrivileged(), 527, 528, 531
- Doc, 720
- DocAttribute, 726, 728
- DocAttributeSet, 727
- DocFlavor, 720
- DocPrintJob, 720, 722
 - getAttributes(), 732
 - print(), 722
- DOCTYPE, 170, 204
- Document, 161, 168, 204
 - createElement(), 204
 - createTextNode(), 204
 - getDocumentElement(), 168
- DocumentBuilder, 161, 167, 176, 202, 204
 - parse(), 167
 - setEntityResolver(), 176
 - setErrorHandler(), 176
- DocumentBuilderFactory, 167, 175, 177, 192, 193
 - isIgnoringElementContentWhitespace(), 177
 - isNamespaceAware(), 193
 - isValidating(), 177
 - newDocumentBuilder(), 167
 - newInstance(), 167
 - setIgnoringElementContentWhitespace(), 177
 - setNamespaceAware(), 193
 - setValidating(), 177
- DocumentName, 729
- doFinal(), 556, 560
- dokumenty XML, 158
 - analiza zawartości, 161
 - atrybuty, 159
 - CDATA, 160
 - deklaracja typu dokumentu, 158
 - DOCTYPE, 170
 - DTD, 160
 - element korzenia, 158
 - elementy, 162
 - elementy podrzędne, 162, 167
 - komentarze, 160
 - kontrola poprawności, 169
 - nagłówki, 158
 - NodeList, 162
 - parsowanie, 160
 - PCDATA, 172
 - pobieranie węzłów, 164
 - przeglądanie atrybutów, 164
 - tworzenie, 202
 - tworzenie drzewa DOM, 202
 - wczytywanie, 161
 - wyszukiwanie informacji, 186
 - XML Schema, 178
 - XPath, 186
- DOM, 160
 - drzewo, 163
 - tworzenie drzewa, 202
- domena ochronna, 511
- DOMResult, 217, 220
- DOMSource, 205, 216
- domyślne obiekty rysujące, 575
- domyślny edytor komórki, 612
- dostęp do kolumn tabeli, 576
- dostęp do składowych, 747
 - pola instancji, 751
 - pola statyczne, 751
- double, 330
- DOUBLE, 279, 330
- draw(), 637, 638
- draw3DRect(), 638
- drawArc(), 638
- drawLine(), 638
- drawOval(), 638
- drawPolygon(), 638
- drawPolyline(), 638
- drawRect(), 635
- drawRectangle(), 638
- drawRoundRect(), 638
- DriverManager, 285, 331
 - getConnection(), 285
 - setLogWriter(), 285
- DROP TABLE, 286
- drukowanie, 703
 - algorytm rozmieszczenia materiału, 712
 - atrybuty, 725
 - format strony, 705
 - liczba stron, 704
 - marginasy, 705
 - PostScript, 723
 - Printable, 703

przycięcie kontekstu
 graficznego, 705
 rozmieszczenie materiału
 na stronach, 712
 transparent, 712
 usługa, 720
 wiele stron, 711
 wymiary strony, 705
 zadania, 703
 drukowanie grafiki, 703
 drzewa, 601
 domyślny edytor komórki, 612
 dziedziczenie, 618
 edycja węzłów, 612
 ikony liści, 607
 JTree, 601, 602
 kolejność przeglądania węzłów,
 617
 korzeń, 601, 604
 las, 601, 606
 liście, 601
 model, 602
 modyfikacja ścieżek, 609
 modyfikacje, 609
 nasłuchiwanie zdarzeń, 621
 obiekt nasłuchujący wyboru, 621
 obiekt rysujący komórki, 618
 obiekt użytkownika, 603
 powiązania między węzłami, 603
 przeglądanie od końca, 617
 przeglądanie w głąb, 617
 przeglądanie węzłów, 616
 przewijalny panel, 612
 rozwijanie ścieżek, 612
 rysowanie węzłów, 618
 struktura, 633
 ścieżki, 610
 tworzenie modeli, 627
 węzły, 601
 węzły nadrzędne, 601
 węzły podrzędne, 601
 wstawianie węzłów, 612
 wygląd, 605, 618
 zdarzenia, 621
 zmiana struktury węzła, 611
 zwiniecie, 606
 DST, 672
 DST_ATOP, 672
 DST_IN, 672
 DST_OUT, 672
 DST_OVER, 672
 DTD, 160, 170, 175
 DTDHandler, 195
 dziedziczenie, 618

E

edycja komórek, 590, 591
 edytor rejestru, 774
 Element, 168, 205
 getAttribute(), 168
 getTagName(), 168
 setAttribute(), 205
 setAttributeNS(), 205
 ELEMENT, 170, 172
 elipsy, 650
 Ellipse2D, 638, 639, 640
 EmployeeReader, 216
 encode(), 259
 endDocument(), 194, 197
 endElement(), 194, 198
 ENTITY, 174
 EntityResolver, 161, 176, 195
 resolveEntity(), 176
 EntryLogger, 462
 Enumeration, 286
 EnumSyntax, 728
 env, 743, 769
 equals(), 383, 441, 523
 error(), 176, 177
 ErrorHandler, 176, 177, 195
 error(), 177
 fatalError(), 177
 warning(), 177
 evaluate(), 187, 190
 EventHandler, 437
 EventListenerList, 632
 ExceptionCheck(), 768
 ExceptionClear(), 768
 ExceptionOccured(), 765, 768
 ExecSQL, 293
 execute(), 287, 306, 314, 316
 executeBatch(), 328, 329
 executeQuery(), 286, 287, 296, 301
 executeUpdate(), 286, 287, 297,
 301, 306, 327
 exitInternal(), 509
 extern "C", 736

F

fatalError(), 176, 177
 Fidelity, 729
 Field, 436, 757
 figury, 636, 638, 639, 650
 łuk, 640
 odcinki, 643
 prostokąt, 640
 przycinanie, 668
 punkty kontrolne, 650

rysowanie, 636
 tworzenie, 651
 wielokąty, 643
 wypełnianie, 638, 661
 File, 603
 FileInputStream, 511
 FilePermission, 510, 516
 FileReader, 511
 fill(), 637, 638
 fillOval(), 635
 filter(), 701
 FilteredRowSet, 313
 filtr RowFilter, 580
 filtrowanie obrazów, 694
 interpolacja, 695
 negatyw, 699
 rozmycie, 699
 wykrywanie krawędzi, 700
 filtrowanie wierszy, 580
 findClass(), 499, 503
 FindClass(), 751
 Finishings, 729
 first(), 312
 FIXED, 174
 float, 330
 FLOAT, 279, 330
 flush(), 562
 FontType, 179
 Format, 386
 format liczby, 364
 format(), 369, 385
 formatowanie komunikatów, 384
 warianty, 386
 formatowanie liczb, 364
 formularze, 251
 HTML, 251
 metoda GET, 252
 przetwarzanie danych, 251
 serwlety, 251
 skrypty CGI, 251
 Submit, 251
 wysyłanie informacji
 do serwera, 252
 fprintf(), 765
 fprintf(), 754
 FROM, 277
 FTP, 246
 funkcje języka C, 734

G

GeneralPath, 638, 643, 650, 651,
 652, 659, 669
 append(), 652
 closePath(), 652

- GeneralPath
 - curveTo(), 652
 - lineTo(), 652
 - moveTo(), 652
 - quadTo(), 652
- generateKey(), 557, 561
- generowanie klucza, 557
- generowanie liczb losowych, 557
- geometria pól, 652
- GET, 252
- get(), 731
- getAddress(), 227, 228
- getAdvance(), 670
- getAllByName(), 227, 228
- getAllowsChildren(), 609
- getAnnotation(), 436, 439
- getAnnotations(), 440
- GetArrayLength(), 761, 763
- getAscent(), 670
- getAttribute(), 168
- getAttributes(), 164, 168, 732
- getAutoCommit(), 328
- getAvailableIDs(), 377
- getAvailableLocales(), 365, 366, 383
- getBinaryStream(), 303
- getBlob(), 302
- getBlockSize(), 560
- GetBooleanArrayElements(), 762
- getBundle(), 392, 393, 395
- getByName(), 227, 228
- GetByteArrayElements(), 777
- getBytes(), 302
- getCategory(), 727, 731
- getCellEditorValue(), 596, 598, 599, 601
- getCellSelectionEnabled(), 587
- getCertificates(), 512
- getCharacterStream(), 303
- getChild(), 628, 634
- getChildAt(), 616
- getChildCount(), 611, 616, 634
- getChildNodes(), 168
- getClassLoader(), 495, 503
- getClip(), 669
- getClob(), 302
- getCodeSource(), 512
- getCollationKey(), 379, 383
- getColorModel(), 690, 693
- getColumn(), 587
- getColumnClass(), 575, 586
- getColumnCount(), 326, 572, 573, 574
- getColumnDisplaySize(), 326
- getColumnLabel(), 326
- getColumnName(), 326, 575
- getColumnNumber(), 177
- getColumnSelectionAllowed(), 587
- getCommand(), 316
- getConcurrency(), 311
- getConnection(), 283, 285, 293, 331
- getConnectTimeout(), 249
- getContent(), 250
- getContentEncoding(), 247, 250
- getContentLength(), 247, 250
- getContentType(), 247, 250
- getContextClassLoader(), 504
- getCountry(), 363
- getCurrencyCode(), 370
- getCurrencyInstance(), 364, 369
- getData(), 164, 169
- getDataElements(), 690, 693, 694
- getDate(), 247, 250
- getDateInstance(), 377
- getDeclaredAnnotations(), 440
- getDecomposition(), 383
- getDefault(), 361, 363
- getDefaultEditor(), 599
- getDefaultFractionsDigits(), 371
- getDefaultName(), 539
- getDefaultRenderer(), 599
- getDescent(), 670
- getDisplayCountry(), 364
- getDisplayLanguage(), 363
- getDisplayName(), 362, 363, 366
- getDocumentElement(), 161, 162, 168
- getDoInput(), 249
- getDoOutput(), 249
- getDouble(), 286, 287
- GetDoubleField(), 748
- getErrorCode(), 291
- getErrorStream(), 258
- getExpiration(), 247, 250
- getFieldDescription(), 626
- GetFieldID(), 748, 751
- getFields(), 633
- getFileSuffixes(), 687
- getFirstChild(), 168
- getFontRenderContext(), 668
- getFontRendererContext(), 670
- getFormatNames(), 687
- getHeaderField(), 244, 246, 250
- getHeaderFieldKey(), 244, 246, 250
- getHeaderFields(), 244, 246, 250
- getHeight(), 687, 705, 711
- getHostAddress(), 228
- getHostName(), 228
- getIdentifier(), 589
- getIfModifiedSince(), 249
- getImageableHeight(), 705, 711
- getImageableWidth(), 705, 711
- getImageableX(), 711
- getImageableY(), 711
- getImageReadersByFormatName(), 685
- getImageReadersByMIMEType(), 679, 685
- getImageReadersBySuffix(), 679, 685
- getImageWritersByFormatName(), 685
- getImageWritersByMIMEType(), 685
- getImageWritersBySuffix(), 685
- getIndexofChild(), 628, 634
- getInputStream(), 225, 229, 245, 253
- getInstance(), 370, 383, 542, 555, 560, 561, 673, 678
- GetIntField(), 748, 777
- getJDBCMajorVersion(), 325
- getJDBCMinorVersion(), 325
- getKeys(), 395
- getLanguage(), 363
- getLastChild(), 164, 168
- getLastModified(), 247, 250
- getLastPathComponent(), 610, 615
- getLastSelectedPathComponent(), 610, 615
- getLeading(), 670
- getLength(), 162, 169, 198
- getLineNumber(), 177
- getLocale(), 385
- getLocalHost(), 227, 228
- getLocalName(), 192, 193, 198
- getLocation(), 512
- getLogger(), 457
- getMaxConnections(), 325
- getMaximumFractionDigits(), 369
- getMaximumIntegerDigits(), 369
- getMaxStatement(), 289
- getMaxStatements(), 325
- getMetaData(), 324, 325
- GetMethodID(), 758, 759, 760
- getMIMETypes(), 687
- getMinimumFractionDigits(), 369
- getMinimumIntegerDigits(), 369
- getModel(), 589
- getMoreResults(), 305
- getName(), 523, 525, 531, 532, 539, 720, 731
- getNameSpaceURI(), 192, 193
- getNextException(), 291
- getNextSibling(), 164, 168
- getNodeName(), 164, 168, 192
- getNodeValue(), 164, 168

- getNumberInstance(), 364
 - getNumImages(), 686
 - getNumThumbnails(), 683, 687
 - getObject(), 395
 - GetObjectArrayElement(), 761, 763
 - GetObjectClass(), 748, 751
 - GetObjectField(), 748
 - getOrientation(), 711
 - getOriginatingProvider(), 679, 687, 688
 - getOutline(), 669
 - getOutputSize(), 560
 - getOutputStream(), 225, 229, 245, 253
 - getPageCount(), 712
 - getParent(), 503, 616
 - getParentNode(), 168
 - getPassword(), 316, 539
 - getPath(), 627
 - getPaths(), 627
 - getPathToRoot(), 612
 - getPercentInstance(), 364
 - getPixel(), 689, 693, 694
 - getPixels(), 693
 - getPointCount(), 650
 - getPreviousSibling(), 168
 - getPrincipals(), 531
 - getPrinterJob(), 703, 710
 - getPrintService(), 723
 - getPrompt(), 539
 - getProperty(), 757
 - getProtectionDomain(), 512
 - getQName(), 198
 - getRaster(), 688, 693
 - getReaderFormatNames(), 686
 - getReaderMIMETypes(), 686
 - getRequestProperties(), 250
 - getResultSet(), 287
 - getRGB(), 690, 694
 - getRoot(), 634
 - getRotateInstance(), 665, 666, 667
 - getRow(), 311
 - getRowCount(), 572, 573, 574
 - getRowHeight(), 586
 - getRowMargin(), 586
 - getRowSelectionAllowed(), 587
 - getSavepointId(), 329
 - getSavepointName(), 329
 - getScaleInstance(), 665, 667
 - getSelectedColumns(), 582
 - getSelectedNode(), 611
 - getSelectionMode(), 578, 586
 - getSelectionPath(), 615, 622, 626, 627
 - getSelectionPaths(), 622, 627
 - getShearInstance(), 665, 667
 - getSQLState(), 291
 - GetStaticFieldID(), 751, 752
 - GetStaticMethodID(), 757, 760
 - GetStaticXxxField(), 751, 752
 - getStrength(), 383
 - getString(), 286, 317, 395
 - getStringArray(), 395
 - GetStringChars(), 745
 - GetStringLength(), 744
 - GetStringRegion(), 744
 - GetStringUTFChars(), 743, 744, 746, 777
 - GetStringUTFLength(), 744
 - GetStringUTFRegion(), 744
 - getStringValue(), 589
 - getSubject(), 531
 - GetSuperClass(), 788
 - getSymbol(), 371
 - getSystemClassLoader(), 503
 - getTableCellEditorComponent(), 597, 598, 600
 - getTableCellRendererComponent(), 597, 599
 - getTableName(), 316
 - getTables(), 325
 - getTagName(), 162, 168
 - getTimeInstance(), 377
 - getTranslateInstance(), 665, 667
 - getTreeCellRendererComponent(), 619, 620
 - getType(), 311
 - getUpdateCount(), 288
 - getURI(), 198
 - getURL(), 315
 - getUsername(), 315
 - getValue(), 198, 589, 728, 775, 776
 - getValueAt(), 572, 575, 596
 - getValueCount(), 589
 - getVendorName(), 687
 - getVersion(), 679, 687
 - getWidth(), 687, 705, 711
 - getWriterFormatNames(), 682, 686
 - getWriterMIMETypes(), 686
 - GetXxxArrayElements(), 763
 - GetXxxArrayRegion(), 762, 763
 - GetXxxField(), 751
 - GIF, 678, 720
 - gniazda, 224
 - adresy internetowe, 227
 - kanaly, 236
 - limity czasu, 225
 - nawiazanie polaczenia, 236
 - otwieranie, 224
 - polaczenia czesciowo zamkniete, 235
 - zamykanie, 231
 - gniazdka, 229
 - Gnu C, 737
 - GradientPaint, 661, 662
 - grafika, 635
 - drukowanie, 703
 - filtrowanie obrazow, 694
 - Java 2D, 635
 - klasy obiektow graficznych, 640
 - operacje na obrazach, 688
 - potok rysowania, 637
 - potokowe tworzenie, 635
 - prostokat ograniczajacy, 639
 - przekształcenia układu wspolrzednych, 663
 - przezroczystosc, 670
 - prycinanie, 668
 - skladanie obrazow, 670
 - slad pedzla, 653
 - wspolrzedne, 639
 - wypelnianie obszaru, 661
 - zbiór Mandelbrota, 690
 - Graphics, 635, 638, 669
 - getClip(), 669
 - setClip(), 669
 - Graphics2D, 636, 638, 660, 662, 663, 667, 669, 671, 678
 - clip(), 669
 - draw(), 638
 - fill(), 638
 - getFontRendererContext(), 670
 - rotate(), 668
 - scale(), 668
 - setComposite(), 678
 - setPaint(), 662
 - setStroke(), 660
 - setTransform(), 667
 - shear(), 668
 - transform(), 667
 - translate(), 668
 - gzip, 250
- ## H
- handle(), 539
 - handleGetObject(), 395
 - hashCode(), 441
 - HashPrintRequestAttributeSet, 703, 727
 - hasła, 533, 536
 - hasMoreElements(), 775, 776, 786
 - hierarchia klas ładowania, 495

- hierarchia klas pozwoleń, 510
 - hierarchia klas Shape, 638
 - host, 222
 - HTML, 157, 223
 - HTTP, 274
 - nagłówki żądań, 245
 - odpowiedzi, 246
 - URLConnection, 258
 - getErrorStream(), 258
- I**
- ICC, 690
 - ID, 174
 - identyfikator URI, 728
 - IDREF, 174
 - IIOImage, 688
 - IIOServiceProvider, 687
 - getVendorName(), 687
 - getVersion(), 687
 - IllegalArgumentException, 765
 - IllegalStateException, 683
 - image/gif, 250
 - ImageInputStream, 683
 - ImageIO, 662, 678, 679, 685
 - createImageInputStream(), 686
 - createImageOutputStream(), 686
 - getImageReadersByFormat
 - ↳Name(), 685
 - getImageReadersByMimeType(), 685
 - getImageReadersBySuffix(), 685
 - getImageWritersByFormat
 - ↳Name(), 685
 - getImageWritersByMimeType(), 685
 - getImageWritersBySuffix(), 685
 - getReaderFormatNames(), 686
 - getReaderMIMETypes(), 686
 - getWriterFormatNames(), 686
 - getWriterMIMETypes(), 686
 - read(), 685
 - write(), 685
 - ImageOutputStream, 686
 - ImageReader, 686
 - getHeight(), 687
 - getNumImages(), 686
 - getNumThumbnails(), 687
 - getOriginatingProvider(), 687
 - getWidth(), 687
 - read(), 686
 - readThumbnail(), 687
 - ImageReaderWriterSpi, 687
 - getFileSuffixes(), 687
 - getFormatNames(), 687
 - getMIMETypes(), 687
 - ImageWriteParam, 684
 - ImageWriter, 684, 687
 - canInsertImage(), 688
 - getOriginatingProvider(), 688
 - setOutput(), 687
 - write(), 687
 - writeInsert(), 687
 - IMPLIED, 174
 - implies(), 512, 520, 525
 - import, 497
 - IndexOutOfBoundsException, 683
 - InetAddress, 227, 228
 - getAddress(), 227, 228
 - getAllByName(), 227, 228
 - getByName(), 227, 228
 - getHostAddress(), 228
 - getHostName(), 228
 - getLocalHost(), 227, 228
 - InetSocketAddress, 241
 - isUnresolved(), 241
 - init(), 511, 560, 561
 - InitialContext, 331
 - initialize(), 535, 539
 - InputSource, 176
 - InputStream, 229, 241, 242
 - INSERT, 279, 286
 - INSERT INTO, 292
 - insertNodeInto(), 611, 616
 - insertRow(), 310, 312
 - instalacja JDBC, 280
 - Instrumentation, 462
 - int, 330
 - INT, 279, 330
 - INTEGER, 279, 330
 - IntegerSyntax, 728
 - interfejs
 - ActionListener, 436
 - adnotacje, 434
 - AnnotatedElement, 436
 - Annotation, 440
 - AttributeSet, 726
 - Attribute, 726
 - BufferedImageOp, 688, 694
 - CachedRowSet, 314
 - Callback, 537
 - Comparator, 378
 - ContentHandler, 194, 195
 - DataSource, 331
 - Doc, 720
 - EntityResolver, 161
 - ErrorHandler, 176
 - Instrumentation, 462
 - JNDI, 331
 - JNI, 740
 - MutableTreeNode, 603
 - Node, 161
 - Pageable, 711
 - Paint, 661
 - Printable, 703
 - PrintRequestAttributeSet, 703
 - PrivilegedExceptionAction, 528
 - PrivilegedAction, 527
 - ResultSet, 313
 - RowSet, 313
 - Shape, 650, 653
 - Source, 216
 - Stroke, 653
 - SupportedValuesAttribute, 726
 - TableCellEditor, 597
 - TableCellRenderer, 590
 - TreeCellRenderer, 619
 - TreeModel, 602, 628
 - TreeSelectionListener, 621
 - interfejs programowy wywołań
 - języka Java, 768, 773
 - interfejs ResultSet, 306
 - internacjonalizacja
 - czas, 371
 - data, 358, 371
 - komplety zasobów, 391
 - komunikaty, 384
 - liczby, 364
 - lokalizatory, 358
 - łańcuchy znaków, 393
 - porządek alfabetyczny, 377
 - waluta, 364, 369
 - zbiory znaków, 388
 - interpolacja, 695
 - interrupt(), 236
 - InterruptedException, 226
 - intValue(), 777
 - inżynieria kodu bajtowego, 456
 - BCEL, 456
 - modyfikacja kodu podczas ładowania, 461
 - IPv6, 227
 - isAfterLast(), 312
 - isAnnotationPresent(), 439
 - IsAssignableFrom(), 777, 788
 - isBeforeFirst(), 312
 - isCellEditable(), 575, 591, 592, 597, 600
 - isClosed(), 226
 - isConnected(), 226
 - isEchoOn(), 539
 - isFirst(), 312
 - isGroupingUsed(), 369
 - isIgnoringElementContent
 - ↳Whitespace(), 177

- isInputShutdown(), 235
 - isLast(), 312
 - isLeaf(), 607, 608, 609, 632, 634
 - isNamespaceAware(), 193, 197
 - isOutputShutdown(), 235
 - isParseIntegerOnly(), 369
 - isUnresolved(), 241
 - isValidating(), 177, 197
 - Item, 462
 - item(), 169
 - Iterator, 286
- J**
- JAAS, 526, 531
 - moduł logowania, 532
 - moduły, 531
 - uwierzalnianie oparte na rolach, 532
 - jaas.config, 530
 - JAR, 242
 - jarray, 761, 776
 - jarsigner, 548, 554
 - java, 565
 - Java 2, 509
 - Java 2D, 635
 - figury, 637, 638
 - geometria pól, 652
 - klasy obiektów graficznych, 640
 - krzywe, 638, 642
 - krzywe Beziera, 643
 - linie, 643
 - łuk, 640
 - operacje na polach, 653
 - poła, 652
 - prostokąt, 640
 - przezroczystość, 670
 - punkty kontrolne, 650
 - składanie obrazów, 670
 - śląd pędzla, 653
 - wartość alfa, 670
 - współrzędne, 639
 - wypełnianie obszaru, 661
 - Java Authentication and Authorization Service, 526
 - java.net.Socket, 225
 - java.nio, 234, 236
 - java.security, 494, 540
 - java.text, 364
 - jawah, 747
 - javap, 753
 - JavaServer Faces, 251
 - javax.imageio, 678
 - javax.security.auth.login.Login
 - ↪Context, 530
 - javax.security.auth.Subject, 531
 - javax.sql.rowset, 313
 - JAXP, 161
 - JCE, 561
 - JCheckBox, 596
 - jclass, 748, 758
 - JComboBox, 596, 650
 - JComponent, 609
 - putClientProperty(), 609
 - JDBC, 271
 - adres URL baz danych, 280
 - aktualizacja danych, 271
 - aktualizacja wsadowa, 327
 - aktualizowalne zbiory wyników zapytań, 306, 309
 - architektura, 272
 - instalacja, 280
 - JNDI, 331
 - klient-serwer, 274
 - menedżer sterowników, 283
 - metadane, 317
 - nawiązywanie połączenia, 281
 - polecenia, 289
 - polecenia przygotowane, 296
 - połączenia krótkotrwałe, 289
 - przewijalne zbiory wyników zapytań, 306, 307
 - przewijanie zbioru rekordów, 308
 - pula połączeń, 332
 - rekordy wstawiania, 310
 - serwer, 274
 - SQL, 271, 274
 - sterowniki, 272
 - transakcje, 326
 - warstwa pośrednia, 274
 - wykonywanie zapytań, 295
 - wypełnianie bazy danych, 292
 - zamykanie zbioru wyników, 289
 - zapytania, 271
 - zarządzanie połączeniami, 289
 - zastosowania, 274
 - zbiory rekordów, 306, 313
 - zbiory wyników, 289
 - źródło danych, 280
 - JDBC 3, 331
 - JDBC API, 272
 - jdbc.password, 293
 - jdbc.property, 282
 - jdbc.url, 293
 - jdbc.username, 293
 - JdbcRowSet, 313
 - język
 - C, 734
 - C++, 736
 - DDL, 287
 - HTML, 157, 223
 - SGML, 157
 - SQL, 271, 274
 - XML, 155, 157
 - XML Schema, 178
 - XPath, 186
 - język SQL, 285
 - jfieldID, 748
 - JLabel, 618
 - JNDI, 331
 - nawiązywanie połączenia, 332
 - pula połączeń, 332
 - zarządzanie nazwami użytkowników, 332
 - źródła danych, 331, 332
 - JndiLoginModule, 527
 - JNI, 740, 742
 - konwencja wywołań, 742
 - wywołania funkcji, 743
 - JNI_CreateJavaVM(), 769, 773
 - JNI_TRUE, 777
 - JNICALL, 746
 - JNIEXPORT, 746
 - JobAttributes, 731
 - JobHoldUntil, 729
 - JobImpressions, 729
 - JobImpressionsCompleted, 729
 - jobject, 748, 776
 - JobKOctets, 729
 - JobKOctetsProcessed, 729
 - JobMediaSheets, 729
 - JobMediaSheetsCompleted, 729
 - JobMessageFromOperator, 729
 - JobName, 729
 - JobOriginatingUserName, 729
 - JobPriority, 729
 - JobSheets, 729
 - JobState, 729
 - JobStateReason, 729
 - JobStateReasons, 729
 - JoinRowSet, 313
 - JPanel, 707
 - JPEG, 678, 679
 - JScrollPane, 570
 - jstring, 742, 758, 776
 - JTable, 567, 571, 576, 586, 597, 599
 - addColumn(), 587
 - getCellSelectionEnabled(), 587
 - getColumnSelectionAllowed(), 587
 - getDefaultEditor(), 599
 - getDefaultRenderer(), 599
 - getRowHeight(), 586
 - getRowMargin(), 586
 - getRowSelectionAllowed(), 587
 - getSelectionModel(), 586

- JTable
 - moveColumn(), 587
 - removeColumn(), 587
 - setAutoResizeMode(), 586
 - setColumnSelectionAllowed(), 587
 - setRowHeight(), 586
 - setRowMargin(), 586
 - setRowSelectionAllowed(), 587
 - JTextArea, 520
 - JTextField, 596
 - JTree, 601, 602, 604, 608, 610, 615, 627
 - getLastSelectedPathComponent(), 615
 - getSelectionPath(), 615, 627
 - getSelectionPaths(), 627
 - makeVisible(), 615
 - scrollPathToVisible(), 615
 - setRootVisible(), 608
 - setShowsRootHandles(), 608
 - jvalue, 760
 - jvm, 769
- K**
- kalkulator emerytalny, 396
 - kanaly, 236
 - SocketChannel, 236
 - Kernel, 700, 702
 - KeyGenerator, 557, 561
 - generateKey(), 561
 - getInstance(), 561
 - init(), 561
 - KeyPairGenerator, 563
 - KeyStoreLoginModule, 527
 - keytool, 546, 551, 554
 - klasa DefaultCellEditor, 596
 - klasa JTree, 610
 - klasy, 498
 - AbstractCellEditor, 597
 - AbstractTableModel, 572
 - abstrakcyjne, 619
 - ActionListenerInstaller, 435
 - AffineTransform, 665
 - AffineTransformOp, 695
 - AllPermission, 520
 - AllPermissions, 511
 - AlphaComposite, 673
 - Arc2D, 641
 - ArcMaker, 651
 - Area, 653
 - Banner, 712
 - BasicPermission, 515
 - BasicStroke, 653, 655
 - BufferedImage, 662, 688
 - ButtonFrame, 435
 - ChoiceFormat, 387
 - Cipher, 555
 - CipherInputStream, 561, 562
 - ClassLoader, 495, 498
 - ClassNameTreeCellRenderer, 620
 - ClassTreeFrame, 620
 - Collator, 378
 - Color, 661
 - ColorConvertOp, 699
 - ColorModel, 691
 - ConvolveOp, 700
 - Copies, 726, 728
 - CopiesSupported, 726
 - CubicCurve2D, 643
 - Currency, 370
 - DatabaseMetaData, 308, 318
 - DateFormat, 373
 - DefaultCellEditor, 612
 - DefaultHandler, 195
 - DefaultMutableTreeNode, 603
 - DefaultTreeModel, 602, 611, 628
 - DialogCallbackHandler, 536
 - DocFlavor, 720
 - DocPrintJob, 720
 - Document, 161
 - DocumentBuilder, 161, 202
 - domena ochronna, 511
 - DOMResult, 217
 - DOMSource, 216
 - DriverManager, 331
 - Ellipse2D, 640
 - EntryLogger, 462
 - EnumSyntax, 728
 - File, 603
 - FilePermission, 510
 - GeneralPath, 643
 - GradientPaint, 661
 - Graphics, 635, 638
 - Graphics2D, 636
 - HashPrintRequestAttributeSet, 703
 - ImageInputStream, 683
 - ImageIO, 678
 - ImageWriter, 684
 - InetAddress, 227
 - informacyjne ziarnka, 433
 - InputStream, 242
 - IntegerSyntax, 728
 - JComboBox, 650
 - JLabel, 618
 - JPanel, 707
 - JTable, 576
 - JTextArea, 520
 - JTree, 601
 - Kernel, 700
 - KeyGenerator, 557
 - KeyPairGenerator, 563
 - komplety zasobów, 393
 - Line2D, 640
 - ListResourceBundle, 394
 - Locale, 361, 364
 - LoginContext, 526
 - LookupOp, 699
 - ładowanie, 494
 - MessageDigest, 542
 - MessageFormat, 384, 387
 - Number, 365
 - NumberFormat, 364, 365
 - PageFormat, 705
 - Permission, 519
 - Policy, 510
 - pozwolenie, 519
 - PreparedStatement, 297
 - PrinterJob, 703
 - PrintService, 720, 723
 - PrintServiceLookup, 720
 - PrintWriter, 253
 - PrivilegedAction, 527
 - Properties, 156
 - Random, 557
 - Raster, 690
 - Reader, 216
 - Rectangle2D, 640
 - RescaleOp, 698
 - ResourceBundles, 393
 - ResultSetMetaData, 318
 - RetinaScanCallback, 537
 - RoundRectangle2D, 640
 - rozwiązywanie, 494
 - Runtime, 508
 - SAXSource, 217
 - Scanner, 236
 - SecureRandom, 557
 - SecurityManager, 511
 - ServerSocket, 229, 231
 - ShapeMaker, 650
 - ShapePanel, 650
 - SimpleDateFormat, 385
 - SimpleLoginModule, 535
 - SimplePrincipal, 533
 - Socket, 225
 - SocketChannel, 236
 - StreamPrintService, 723
 - StreamPrintServiceFactory, 723
 - StreamSource, 216
 - szyfrowanie plików, 503
 - TableColumn, 577

- TableColumnModel, 576
 - TextLayout, 669
 - TexturePaint, 661, 662
 - ThreadedEchoHandler, 232
 - TreeNode, 610
 - TreePath, 610
 - TreeSelectionModel, 621
 - UnixNumericGroupPrincipal, 527
 - UnixPrincipal, 526
 - URI, 242
 - URL, 242
 - URLConnection, 242
 - WordCheckPermission, 521
 - WritableRaster, 690
 - klasy kolumn, 575
 - klient, 223
 - klucz, 557
 - klucz prywatny, 544
 - klucz publiczny, 544, 562
 - klucze, 556
 - klucze wygenerowane automatycznie, 306
 - kod ASCII, 357
 - kod bajtowy, 456
 - kod języków, 359
 - kod macierzysty, 733
 - kod Unicode, 357
 - kodowanie znaków, 388
 - kolory, 670
 - RGB, 670, 689
 - kolumny, 275, 575
 - ukrywanie, 582
 - wyświetlanie, 582
 - komórki
 - edycja, 591
 - rysowanie, 590
 - kompilator, 494
 - Gnu C, 737
 - komplety zasobów, 391
 - implementacja klas, 394
 - klasy, 393
 - ładowanie, 392
 - pliki właściwości, 393
 - komponent JTable, 567
 - komponenty
 - JTree, 601
 - komunikaty, 384
 - formatowanie z wariantami, 386
 - indeks znacznika, 385
 - konstruktory, 758
 - kontekst graficzny, 664, 705, 706
 - kontekst tworzenia czcionki, 668
 - kontrola dostępu, 493
 - kontrola poprawności dokumentów XML, 169
 - atributy, 173
 - ATTLIST, 173
 - byty, 174
 - CDATA, 174
 - definicja typu dokumentu, 170
 - DOCTYPE, 170
 - DTD, 170
 - ELEMENT, 170, 172
 - parser XML, 173
 - reguły zawartości elementów, 172
 - skrót, 174
 - typy atrybutów, 173
 - wartości domyślne atrybutów, 174
 - warunki kontroli, 170
 - XML Schema, 170, 178
 - kontrola pozwoleń, 512
 - kończenie pracy maszyny wirtualnej, 509
 - korzeń, 601
 - Krb5LoginModule, 527
 - kryptografia klucza publicznego, 544, 563
 - krzywe, 638, 642, 650
 - Beziera, 643
 - drugiego stopnia, 643
 - punkty kontrolne, 642
 - trzeciego stopnia, 643
 - kursory, 308
- ## L
- las, 601, 606, 607
 - last(), 312
 - layoutPages(), 712
 - LD_LIBRARY_PATH, 773
 - length(), 302
 - liczby, 364
 - formatowanie, 364
 - formaty, 365
 - lokalizatory, 365
 - losowe, 557
 - LIKE, 278
 - Line2D, 638, 640
 - lineTo(), 643, 652
 - linie, 643
 - Linux, 278, 772
 - ListResourceBundle, 394
 - ListSelectionModel, 588
 - setSelectionMode(), 588
 - liście, 601
 - loadClass(), 499
 - loadLibrary(), 737, 739
 - Locale, 360, 361, 363, 364
 - getCountry(), 363
 - getDefault(), 363
 - getDisplayCountry(), 364
 - getDisplayLanguage(), 363
 - getDisplayName(), 363
 - getLanguage(), 363
 - setDefault(), 363
 - toString(), 364
 - localFile, 515
 - LoggingPermission, 517
 - logika biznesowa, 528
 - login(), 530, 540
 - LoginContext, 526, 527, 530, 536
 - getSubject(), 531
 - login(), 530
 - logout(), 530
 - LoginModule, 539
 - abort(), 540
 - commit(), 540
 - initialize(), 539
 - login(), 540
 - logout(), 540
 - logout(), 530, 540
 - logowanie, 527, 538
 - lokalizacja, 396
 - kod, 509
 - zasoby, 392
 - lokalizatory, 358
 - czas, 371
 - data, 371
 - domyślny, 361
 - języki, 361
 - liczby, 365
 - lookup(), 331
 - LookupOp, 699, 701
 - lookupPrintServices(), 720
 - LookupTable, 699
- ## Ł
- ładowanie klas, 494
 - ClassLoader, 498
 - implementacja procedury ładującej, 498
 - klasy systemowe, 495
 - maszyna wirtualna, 494
 - procedury, 495
 - przestrzeń nazw, 497
 - łańcuch zaufania, 549
 - łańcuchy, 387
 - łączenie tabel, 277
 - łuk, 640, 650
- ## M
- macierz przekształceń, 665
 - mailto, 242
 - main(), 494
 - makeShape(), 651

- makeVisible(), 612, 615
 - maszyna wirtualna, 504
 - MD5, 541
 - MediaName, 729
 - MediaSize, 729
 - MediaSizeName, 730
 - MediaTray, 730
 - menedżer bezpieczeństwa, 494, 508, 514
 - checkExit(), 508
 - domyślny, 509
 - pliki polityki, 514
 - pozwolenia, 508
 - SecurityManager, 511
 - MessageDigest, 542, 543
 - digest(), 543
 - reset(), 543
 - update(), 543
 - MessageFormat, 384, 385, 387
 - applyPattern(), 385
 - format(), 384, 385
 - getLocale(), 385
 - setLocale(), 385
 - metaadnotacje, 449
 - metadane, 317
 - DatabaseMetaData, 318
 - pozyskiwanie, 317
 - ResultSetMetaData, 318
 - Metal, 605
 - Method, 436, 757
 - metoda
 - andFilter, 581
 - dateFilter, 580
 - notFilter, 581
 - orFilter, 581
 - regexFilter, 581
 - metoda iterator(), 290
 - metody
 - main(), 494
 - sygnatury, 752
 - metody macierzyste
 - alternatywne sposoby
 - wywoływania metod, 758
 - dostęp do pól instancji, 747, 751
 - dostęp do pól statycznych, 751
 - funkcje języka C, 734
 - implementacja, 736
 - interfejs programowy wywołań
 - języka Java, 768
 - jarray, 761
 - język C++, 736
 - JNI, 740
 - konstruktory, 758
 - łańcuchy znaków, 742, 752
 - obsługa błędów, 764, 768
 - parametry numeryczne, 740
 - przeciążanie identyfikatorów, 735
 - rejestr systemu Windows, 773
 - składowe obiektu, 747
 - sygnatury, 752
 - tablice, 760, 763
 - wartości zwracane, 740
 - wyrzucanie wyjątków, 764
 - wywoływanie metod języka
 - Java, 754, 759
 - wywoływanie metod obiektów, 754
 - wywoływanie metod
 - statycznych, 757
 - Microsoft ASP, 251
 - MIME, 679
 - MissingResourceException, 392
 - model drzewa, 602
 - model kolorów, 689
 - model tabeli, 568, 571
 - moduł JAAS, 531
 - moduł logowania, 527, 532
 - hasła, 533, 536
 - logowanie, 538
 - nazwa użytkownika, 536
 - options, 535
 - SimpleLoginModule, 533
 - moduł uwierzytelniania, 527
 - modyfikacja kodu bajtowego
 - podczas ładowania, 461
 - modyfikacja kodu maszyny
 - wirtualnej, 507
 - modyfikatory, 434
 - mostek JDBC/ODBC, 272
 - moveColumn(), 587
 - moveTo(), 643, 652
 - moveToCurrentRow(), 310, 312
 - moveToInsertRow(), 310, 312
 - MultipleDocumentHandling, 730
 - MutableTreeNode, 603, 609
 - setUserObject(), 609
- N**
- nagłówek, 591
 - nagłówki żądań HTTP, 245
 - NameCallback, 537, 539
 - getDefaultName(), 539
 - getName(), 539
 - getPrompt(), 539
 - setName(), 539
 - NamedNodeMap, 164, 169
 - getLength(), 169
 - item(), 169
 - negatyw obrazu, 699
 - NetPermission, 516
 - NewByteArray(), 776
 - newDocument(), 202
 - newDocumentBuilder(), 167
 - newInputStream(), 241
 - newInstance(), 167, 189, 197, 205
 - NewObject(), 760, 764
 - NewObjectA(), 760
 - NewObjectV(), 760
 - newOutputStream(), 236, 241
 - newSAXParser(), 197
 - NewString(), 744
 - NewStringUTF(), 744, 776
 - newXPath(), 189
 - NewXxxArray(), 762
 - next(), 288
 - nextElement(), 775, 776, 786
 - NMTOKEN, 174
 - NMTOKENS, 174
 - Node, 161, 162, 168, 193
 - appendChild(), 205
 - getAttributes(), 168
 - getChildNodes(), 168
 - getFirstChild(), 168
 - getLastChild(), 168
 - getLocalName(), 193
 - getNameSpaceURI(), 193
 - getNextSibling(), 168
 - getNodeName(), 168
 - getNodeValue(), 168
 - getParentNode(), 168
 - getPreviousSibling(), 168
 - nodeChanged(), 611, 616
 - NodeList, 162, 169
 - getLength(), 169
 - item(), 169
 - NOT LIKE, 278
 - NTLoginModule, 527
 - NullPointerException, 765
 - Number, 365
 - NumberFormat, 364, 365, 368
 - format(), 369
 - getAvailableLocales(), 366
 - getCurrencyInstance(), 369
 - getMaximumFractionDigits(), 369
 - getMaximumIntegerDigits(), 369
 - getMinimumFractionDigits(), 369
 - getMinimumIntegerDigits(), 369
 - isGroupingUsed(), 369
 - isParseIntegerOnly(), 369
 - parse(), 369
 - setGroupingUsed(), 369
 - setMaximumFractionDigits(), 369
 - setMaximumIntegerDigits(), 369
 - setMinimumFractionDigits(), 369

setMinimumIntegerDigits(), 369
 setParseIntegerOnly(), 369
 NumberOfDocuments, 730
 NumberOfInterveningJobs, 730
 NumberUp, 730
 NUMERIC, 279, 330
 numeryczne parametry metod, 740

O

obiekt SQLException, 290
 obiekty
 dostęp do składowych, 747
 użytkownika, 603
 obiekty binarne, 301
 obiekty rysujące, 590
 obiekty znakowe, 301
 obrazy
 dostęp do danych, 688
 filtrowanie, 694
 model kolorów, 689, 690
 modyfikacja pikseli, 689
 negatyw, 699
 obrót, 695
 operacje, 688
 próbki piksela, 689
 przejrzystość, 670
 przyrostowe tworzenie, 688
 rozmycie, 699
 sekwencje, 683
 wczytywanie, 678
 wykrywanie krawędzi, 700
 zapisywanie, 678
 zbiór Mandelbrota, 690
 obrót, 663, 695
 obrys czcionek, 668
 obrys figury, 636
 obrys tekstu, 669
 obsługa błędów, 764
 obsługa zdarzeń
 adnotacje, 435
 ODBC, 272
 odcinki, 643
 odcisk palca, 541
 odczyt dużych obiektów, 301
 odszyfrowywanie danych, 561
 odwołanie transakcji, 326
 okna dialogowe
 drukowanie, 710
 openConnection(), 244, 249
 openInputStream(), 250
 openOutputStream(), 250
 openStream(), 242, 245, 249
 operacje na obrazach, 688
 ORDER BY, 286

OrientationRequested, 730
 OutOfMemoryError, 765
 OutputDeviceAssigned, 730
 OutputStream, 229, 241

P

Package, 436
 Pageable, 711
 PageAttributes, 731
 pageDialog(), 706, 710
 PageFormat, 705, 710, 711
 getHeight(), 711
 getImageableHeight(), 711
 getImageableWidth(), 711
 getImageableX(), 711
 getImageableY(), 711
 getOrientation(), 711
 getWidth(), 711
 PageRanges, 730
 PagesPerMinute, 730
 PagesPerMinuteColor, 730
 Paint, 661
 paint(), 636
 paintComponent(), 636, 659, 707
 pakiet OpenSSL, 551
 pakiet Swing, 567
 pakiety, 497, 498
 parse(), 167, 197, 220, 365, 369
 ParseException, 365
 parser
 SAX, 193
 parser XML, 160
 DOM, 160
 implementacja, 173
 SAX, 160
 parsowanie dokumentów XML,
 160
 DOM, 160, 163
 SAX, 160
 XML Schema, 180
 PasswordCallback, 537, 539
 getPassword(), 539
 getPrompt(), 539
 isEchoOn(), 539
 setPassword(), 539
 pathFromAncestorEnumeration(),
 617
 PCDATA, 172, 175
 PDLOverrideSupported, 730
 Permission, 519, 525
 getName(), 525
 implies(), 525
 piaskownica, 509
 piksel, 670
 docelowy, 671
 PJA, 728
 PKCS#5, 556
 plik java.policy, 513
 pliki, 518
 class, 494
 JAR, 242
 pomocnicze, 433
 pozwoleń, 510
 właściwości, 156, 393
 XML, 156
 XML Schema, 170
 zasoby, 391
 pliki graficzne, 678
 animacje GIF, 685
 formaty, 678
 ImageIO, 679
 interfejs dostawcy, 679
 JPEG, 679
 MIME, 679
 obiekt odczytu, 679
 obiekty, 679
 sekwencje obrazów, 683
 zapisywanie, 678, 682
 pliki JAR, 281
 pliki polityki, 509
 baza kodu, 514
 grant, 514
 implementacja klasy pozwoleń,
 520
 klasy pozwoleń, 519
 menedżer bezpieczeństwa, 514
 niezależne od platformy
 systemowej, 519
 odczyt, 519
 operacje sieciowe, 518
 pozwolenia, 515, 523
 składnica kluczy, 554
 system plików, 518
 testowanie aplikacji, 514
 właściwości systemowe, 519
 zapis, 519
 pliki polityki bezpieczeństwa, 512
 PNG, 678
 pobieranie wartości kluczy, 306
 pochylanie, 663
 poczta elektroniczna, 221, 266
 SMTP, 266
 wysyłanie, 266
 podpis cyfrowy, 540, 550
 aplety, 540
 generator liczb losowych, 557
 MD5, 541
 MessageDigest, 542
 podpisywanie wiadomości, 544
 SHA1, 541

- podpis cyfrowy
 - skrót wiadomości, 541
 - weryfikacja, 545
- podpisywanie certyfikatów, 550
- podpisywanie kodu, 493, 552
 - jarsigner, 554
- podpisywanie wiadomości, 544
- Point2D, 650, 669
- pola, 652
 - add, 652
 - exclusiveOr, 653
 - intersect, 653
 - operacje, 653
 - subtract, 652
 - śląd pędzla, 653
 - tworzenie, 653
- polecenia przygotowane, 296
- polecenia SQL, 285
- Policy, 510
- polityka bezpieczeństwa, 509, 510
 - pliki, 512
- połączenia bazodanowe
 - JNDI, 331
- połączenia sieciowe
 - częściowo zamknięte, 235
 - gniazda, 224
 - klient, 223
 - porty, 222
 - serwer, 221
 - strumienie, 225
 - TCP, 225
 - UDP, 225
 - URL, 242
 - wysyłanie danych do formularzy, 251
- połączenie z bazą danych, 283
- populate(), 316
- porty, 222
- porządek alfabetyczny, 377
- POST, 252
- postOrderEnumeration(), 620
- postOrderTraversal(), 617
- PostScript, 723
- potok rysowania, 637
- potokowe tworzenie grafiki, 635
- pozwolenia, 508, 510, 523
 - implementacja klasy, 520
 - klasy, 519
 - operacje sieciowe, 518
 - parametry, 516, 517
 - pliki polityki, 515
 - stos wywołań metod, 511
 - system plików, 518
- PRA, 728
- premain(), 462
- preOrderEnumeration(), 620
- preOrderTraversal(), 617
- PreparedStatement, 296, 297, 301
 - clearParameters(), 301
 - executeQuery(), 301
 - executeUpdate(), 301
 - setXxx(), 301
- prepareStatement(), 301, 307, 311
- PresentationDirection, 730
- previous(), 311
- Principal, 531
 - getName(), 531
- Print, 707
- print(), 703, 705, 710, 711, 722, 754
- Printable, 703, 704, 710
 - print(), 710
- printDialog(), 703, 704, 710
- PrinterException, 704
- PrinterInfo, 730
- PrinterIsAcceptingJobs, 730
- PrinterJob, 703, 704, 705, 706, 710, 711
 - defaultPage(), 710
 - getPrinterJob(), 710
 - pageDialog(), 710
 - print(), 711
 - printDialog(), 710
 - setPrintable(), 710
- PrinterLocation, 730
- PrinterMakeAndModel, 730
- PrinterMessageFromOperator, 730
- PrinterMoreInfo, 730
- PrinterMoreInfoManufacturer, 730
- PrinterName, 730
- PrinterResolution, 730
- PrinterState, 730
- PrinterStateReason, 730
- PrinterStateReasons, 730
- PrinterURI, 730
- printf(), 734, 752
- PrintJob, 703
- PrintJobAttribute, 726, 728
- PrintQuality, 728, 730
- PrintRequestAttribute, 726, 728
- PrintRequestAttributeSet, 703
- PrintService, 720, 722, 723, 732
 - createPrintJob(), 722
 - getAttributes(), 732
- PrintServiceAttribute, 726, 728
- PrintServiceLookup, 720, 722
- PrintWriter, 229, 253, 754
- PrivilegedAction, 527, 531
 - run(), 531
- PrivilegedExceptionAction, 528, 531
 - run(), 531
- PrivilegedAction, 527, 528, 537
- problem uwierzytelniania, 548
- procedury ładowania klas
 - implementacja, 498
- processor XSLT, 213
- processAnnotations(), 436
- profil ICC, 690
- programowanie baz danych, 271
- programowanie transakcji, 326
- Properties, 156
- PropertyPermission, 516
- prostokąt, 640, 650
 - ograniczający, 639
- ProtectionDomain, 512
 - getCodeSource(), 512
 - implies(), 512
- protokoły
 - HTTP, 245
 - SMTP, 266
 - TCP, 225
 - UDP, 225
- prywatne metody macierzyste, 509
- przeglądarka klas, 621
- przekształcenia afiniczne, 665, 695
- przekształcenia figur, 636
- przekształcenia układu
 - współrzędnych, 636, 663
 - macierz przekształceń, 665
 - obrót, 663
 - pochylenie, 663
 - przesunięcie, 663
 - skalowanie, 663
 - składanie przekształceń, 664
- przekształcenia XSL
 - Java, 216
 - style, 213
 - szablon przekształcenia, 213
 - XSLT, 213
- przestrzeń nazw, 190, 497
 - alias, 191
 - definiowanie, 192
 - identyfikatory, 191
 - URI, 190
 - XML, 190
- przesunięcie, 663
- przetwarzanie adnotacji, 452
- przetwarzanie dokumentów XML, 161
- przewijalne zbiory wyników zapytań, 306, 307
- przezroczystość, 670
- przycinanie, 636, 668
 - określanie obszaru, 669

przyrostowe tworzenie obrazów, 688
 PSA, 728
 public, 434
 pula połączeń, 332
 punkty kontrolne transakcji, 327
 putClientProperty(), 606, 609

Q

QuadCurve2D, 638
 QuadCurve2D.Double, 651
 quadTo(), 643, 652
 QueuedJobCount, 731

R

Random, 557
 RandomAccessFile, 686
 Raster, 690, 693
 getDataElements(), 693
 getPixel(), 693
 getPixels(), 693
 read(), 236, 562, 679, 683, 685, 686
 ReadableByteChannel, 236
 Reader, 216
 readThumbnail(), 684, 687
 REAL, 279, 330
 Rectangle2D, 638, 639, 640
 ReferenceUriSchemesSupported,
 731
 ReflectPermission, 517
 refleksja, 450
 REG_BINARY, 776
 REG_DWORD, 776
 REG_SZ, 776
 rejestr systemu Windows, 773
 edytor rejestru, 774
 funkcje kontroli typów, 788
 interfejs dostępu, 775
 klucze, 774
 metody macierzyste, 776
 pobieranie wartości, 776
 przeglądanie nazw kluczy, 777
 uchwyt klucza, 777
 węzły, 774
 Win32RegKey, 775
 Win32RegKeyName
 ↳ Enumeration, 777
 zapisywanie wartości, 777
 rejestracja klasy sterownika, 282
 rekordy, 275
 wstawiania, 310
 relative(), 308, 312
 relativize(), 244
 releaseSavepoint(), 327, 329

ReleaseStringChars(), 745
 ReleaseStringUTFChars(), 744, 746
 ReleaseXxxArrayElements(), 763
 reload(), 611, 616
 REMARKS, 325
 remove(), 731
 removeCellEditorListener(), 601
 removeColumn(), 582, 587
 removeNodeFromParent(), 611, 616
 removeTreeModelListener(), 632,
 635
 RequestingUserName, 731
 REQUIRED, 174
 RescaleOp, 698, 701
 reset(), 543
 resolve(), 244
 resolveEntity(), 176
 ResourceBundle, 392, 395
 getBundle(), 392, 395
 getKeys(), 395
 getObject(), 395
 getString(), 395
 getStringArray(), 395
 handleGetObject(), 395
 ResourceBundles, 393
 Result, 216
 ResultSet, 286, 288, 289, 302, 306,
 313, 325
 absolute(), 311
 afterLast(), 312
 beforeFirst(), 312
 cancelRowUpdates(), 312
 close(), 288
 column(), 288
 CONCUR_READ_ONLY, 307
 CONCUR_UPDATABLE, 309
 deleteRow(), 312
 first(), 312
 getConcurrency(), 311
 getMetaData(), 325
 getRow(), 311
 getType(), 311
 getXxx(), 288
 insertRow(), 312
 isAfterLast(), 312
 isBeforeFirst(), 312
 isFirst(), 312
 isLast(), 312
 last(), 312
 moveToCurrentRow(), 312
 moveToInsertRow(), 312
 next(), 288
 previous(), 311
 relative(), 312

TYPE_SCROLL_INSENSITIVE,
 307, 309
 updateRow(), 312
 updateXxx(), 312
 ResultSetMetaData, 318, 326
 getColumnCount(), 326
 getColumnDisplaySize(), 326
 getColumnLabel(), 326
 getColumnName(), 326
 RetentionPolicy, 449, 450
 RetinaScanCallback, 537
 return, 764
 RGB, 670, 689
 role, 532
 rollback(), 327, 329
 rotate(), 664, 668
 RoundRectangle2D, 638, 639, 640
 RoundRectangle2D.Double, 651
 RowSet, 313, 315
 execute(), 316
 getCommand(), 316
 getPassword(), 316
 getURL(), 315
 getUsername(), 315
 setCommand(), 316
 setPassword(), 316
 setURL(), 315
 setUsername(), 315
 rozmycie obrazu, 699
 rozwiązywanie klasy, 494
 RSA, 545, 563
 klucze, 563
 run(), 531
 Runtime, 508
 RuntimePermission, 516
 rysowanie
 figur, 636
 potok, 637
 węzłów, 618
 rysowanie komórek, 590
 rysunki, 635

S

Savepoint, 329
 getSavepointId(), 329
 getSavepointName(), 329
 SAX, 160, 193
 wyszukiwanie elementów, 195
 SAXParseException, 177
 getColumnNumber(), 177
 getLineNumber(), 177
 SAXParser, 195
 parse(), 197

- SAXParserFactory, 195, 197
 - isNamespaceAware(), 197
 - isValidating(), 197
 - newInstance(), 197
 - newSAXParser(), 197
 - setNamespaceAware(), 197
 - setValidating(), 197
- SAXSource, 216, 217, 220
- scale(), 663, 664, 668
- Scanner, 229, 236
- scrollPathToVisible(), 612, 615
- SecretKeySpec, 561
- SecureRandom, 557
- SecurityException, 509, 511
- SecurityManager, 511, 512
 - checkPermission(), 512
- SecurityPermission, 517
- sekwencje obrazów, 683
- sekwencje sterujące, 303
- SELECT, 277
 - FROM, 277
 - LIKE, 278
 - NOT LIKE, 278
 - WHERE, 277
- Serializable, 450, 451
- SerializablePermission, 517
- ServerSocket, 229, 231
 - accept(), 231
 - close(), 231
- serwer, 221, 274
 - aplikacji, 332
 - FTP, 246
 - gniazda, 229
 - HTTP, 229
 - implementacja, 228
 - obsługa wielu klientów, 231
 - połączenia, 221
 - wątki, 232
 - Web, 221, 252
 - wysyłanie danych do formularzy, 251
- serwlety, 251, 332
- set(), 637
- setAllowsChildren(), 608, 609
- setAsksAllowsChildren(), 608, 609
- setAttribute(), 203, 205
- setAttributeNS(), 205
- setAutoCommit(), 326, 328
- setAutoResizeMode(), 586
- SetByteArrayRegion(), 776
- setCellEditor(), 600
- setCellRenderer(), 600
- setCellSelectionEnabled(), 578
- setClip(), 668, 669, 705
- setClosedIcon(), 621
- setColumnSelectionAllowed(), 578, 587
- setCommand(), 314, 316
- setComposite(), 637, 673, 678
- setConnectTimeout(), 249
- setContentHandler(), 220
- setContextClassLoader(), 504
- setDataElements(), 690, 693
- setDecomposition(), 383
- setDefault(), 363
- setDefaultRenderer(), 591
- setDoInput(), 245, 249
- setDoOutput(), 245, 249, 253
- SetDoubleField(), 748
- setEditable(), 612
- setEntityResolver(), 176
- setErrorHandler(), 176
- setGroupingUsed(), 369
- setHeaderRenderer(), 591, 600
- setHeaderValue(), 591, 600
- setIfModifiedSince(), 249
- setIgnoringElementContent
 - ↳Whitespace(), 175, 177
- setInput(), 683
- SetIntField(), 748, 777
- setLeafIcon(), 621
- setLocale(), 361, 385
- setLogWriter(), 285
- setMaximumFractionDigits(), 369
- setMaximumIntegerDigits(), 369
- setMaxWidth(), 577, 588
- setMinimumFractionDigits(), 369
- setMinimumIntegerDigits(), 369
- setMinWidth(), 577, 588
- setModifiedSince(), 245
- setName(), 539
- setNamespaceAware(), 180, 192, 193, 195, 197
- SetObjectArrayElement(), 761, 765
- SetObjectField(), 748
- setOpenIcon(), 621
- setOutput(), 687
- setOutputProperty(), 205
- setPageable(), 711
- setPaint(), 636, 661, 662
- setParseIntegerOnly(), 369
- setPassword(), 314, 316, 539
- setPixel(), 689, 694
- setPixels(), 689, 694
- setPreferredWidth(), 577, 588
- setPrintable(), 710
- setReadTimeout(), 249
- setRenderingHints(), 636
- setRequestProperty(), 245, 246, 250
- setResizable(), 577, 588
- setRootVisible(), 606, 608
- setRowFilter(), 588
- setRowHeight(), 577, 578, 586
- setRowMargin(), 578, 586
- setRowSelectionAllowed(), 578, 587
- setSavepoint(), 327, 329
- setSecurityManager(), 514
- setSeed(), 557
- setSelectionMode(), 578, 588
- setShowsRootHandles(), 606, 608
- setSortable(), 588
- setSoTimeout(), 225, 226
- SetStaticXxxField(), 751
- setStrength(), 383
- setString(), 296
- setStroke(), 636, 653, 655, 659, 660
- setTableName(), 316
- setToRotation(), 665, 667
- setToScale(), 665, 667
- setToShear(), 665, 667
- setToTranslation(), 665, 667
- setTransform(), 665, 666, 667
- setURL(), 314, 315
- setUsername(), 314, 315
- setUserObject(), 603, 609
- setValidating(), 177, 197
- setValue(), 775, 776, 777
- setValueAt(), 575, 599
- setWidth(), 577, 588
- setXxx(), 301
- SetXxxArrayRegion(), 762, 764
- SetXxxField(), 751, 752
- Severity, 731
- SGML, 157
- SHA1, 541
- Shape, 650, 653
- ShapeMaker, 650
- ShapePanel, 650
- shear(), 664, 668
- SheetCollate, 731
- short, 330
- ShortLookupTable, 699
- shouldSelectCell(), 598, 600
- shutdownInput(), 235
- shutdownOutput(), 235
- Sides, 731
- sieć, 221
 - poczta elektroniczna, 266
 - przesyłanie danych, 229
- SimpleDateFormat, 385
- SimpleDoc, 720, 722
- SimpleLoginModule, 533, 535
- SimplePrincipal, 533
- SINGLE_TREE_SELECTION, 621
- skalowanie, 663

- składanie obrazów, 670
 - CLEAR, 672
 - DST, 672
 - DST_ATOP, 672
 - DST_IN, 672
 - DST_OUT, 672
 - DST_OVER, 672
 - piksel docelowy, 671
 - projektowanie reguły, 671
 - reguły Portera-Duffa, 672
 - reguły składania, 670
 - SRC, 672
 - SRC_ATOP, 672
 - SRC_IN, 672
 - SRC_OUT, 672
 - SRC_OVER, 672
 - XOR, 672
 - składanie przekształceń, 664
 - składnica kluczy, 546, 547, 548, 554
 - składowe obiektu, 747
 - skrót wiadomości, 541
 - skrypty CGI, 251
 - SMALLINT, 279, 330
 - SMTTP, 266
 - Socket, 225, 226, 235
 - connect(), 226
 - getInputStream(), 225
 - getOutputStream(), 225
 - isClosed(), 226
 - isConnected(), 226
 - isInputShutdown(), 235
 - isOutputShutdown(), 235
 - setSoTimeout(), 225, 226
 - shutdownInput(), 235
 - shutdownOutput(), 235
 - SocketChannel, 236, 241
 - SocketPermission, 516
 - SocketTimeoutException, 249
 - sort(), 378
 - sortowanie
 - porządek alfabetyczny, 377
 - sortowanie wierszy, 579
 - Source, 216
 - spot, 699
 - spójność bazy danych, 326
 - sprawdzanie uprawnień, 508
 - SQL, 271, 274, 285
 - aktualizowalne zbiory wyników
 - zapytań, 309
 - analiza wyjątków, 289
 - ARRAY, 330
 - CREATE TABLE, 279, 286
 - DDL, 287
 - DROP TABLE, 286
 - FROM, 277
 - funkcje, 279
 - INSERT, 279
 - LIKE, 278
 - łączenie tabel, 277
 - metadane, 317
 - modyfikacja danych, 278
 - NOT LIKE, 278
 - polecenia, 289
 - polecenia przygotowane, 296
 - sekwencje sterujące, 303
 - SELECT, 277
 - słowa kluczowe, 277
 - tworzenie tabel, 279
 - typy danych, 279, 330
 - typy wyjątków, 290
 - WHERE, 277, 278
 - wstawianie danych, 279
 - wykonywanie zapytań, 295
 - wynik zapytania, 276
 - wypełnianie bazy danych, 292
 - zapytania, 277
 - zarządzanie połączeniami, 289
 - zbiory wyników, 305
 - zbiór wyników, 289
 - SQLException, 291, 327
 - getErrorCode(), 291
 - getNextException(), 291
 - getSQLState(), 291
 - SQLPermission, 517
 - SRC, 672
 - SRC_ATOP, 672
 - SRC_IN, 672
 - SRC_OUT, 672
 - SRC_OVER, 672
 - sRGB, 690
 - startDocument(), 194, 197
 - startElement(), 194, 195, 198
 - Statement, 289, 305, 306, 307, 329
 - addBatch(), 329
 - close(), 288, 289
 - createStatement(), 285
 - execute(), 287
 - executeBatch(), 329
 - executeQuery(), 287
 - executeUpdate(), 286, 287
 - getResultSet(), 287
 - getUpdateCount(), 288
 - static, 434
 - stdarg.h, 760
 - sterowniki JDBC, 272
 - typ 1, 272
 - typ 2, 273
 - typ 3, 273
 - typ 4, 273
 - stopCellEditing(), 598, 599, 600
 - StreamPrintService, 723
 - StreamPrintServiceFactory, 723
 - getPrintService(), 723
 - StreamResult, 205
 - StreamSource, 216, 217, 219
 - String, 330, 754
 - Stroke, 653
 - strona WWW, 251
 - struktura dokumentu XML, 158
 - struktury danych
 - nieskończone, 634
 - strumienie, 225, 253
 - usługi drukowania, 722
 - strumienie szyfrujące, 561
 - CipherInputStream, 561
 - Subject, 531
 - doAs(), 531
 - doAsPrivileged(), 531
 - getPrincipals(), 531
 - SupportedValuesAttribute, 726
 - supportsBatchUpdates(), 329
 - supportsResultSetConcurrency(), 308, 313
 - supportsResultSetType(), 308, 313
 - SVG, 210
 - Swing
 - drzewa, 601
 - JTree, 601
 - tabele, 567
 - sygnatury metody, 752
 - SyncProviderException, 315, 316
 - System, 739
 - loadLibrary(), 737, 739
 - setSecurityManager(), 514
 - szyfr Cezara, 502
 - szyfrowanie, 502, 555
 - AES, 555, 557, 562
 - algorytmy, 555
 - Cipher, 555
 - DES, 555, 556
 - dopełnienie ostatniego bloku, 556
 - klucze, 556, 557
 - pliki klas, 503
 - RSA, 545, 563
 - strumienie, 561
 - symetryczne, 555
 - tryb pracy algorytmu, 556
 - szyfrowanie kluczem publicznym, 562
- Ś**
- ścieżka drzewa, 610
 - ścieżki drzewa, 610

- śląd pędzla, 636, 653
 - połączenia, 654
 - przerywany wzór, 655
 - szerokość, 654
 - wartość graniczna, 655
 - zakończenia, 654
- T**
- tabele, 275, 567
 - implementacja edytora komórek, 597
 - JTable, 567
 - kolumny, 575
 - model, 568, 571
 - model wyboru, 578
 - obiekt rysujący, 575
 - prezentacja danych, 575
 - przesuwanie kolumny, 570
 - szerokość kolumn, 570
 - tworzenie, 279
 - tworzenie edytorów, 597
 - ukrywanie kolumn, 582
 - widoki, 570
 - wiersze, 575
 - wybór kolumn, 578
 - wybór komórek, 578
 - wybór wierszy, 578
 - wysokość komórek, 577
 - wyświetlanie kolumn, 582
 - zmiana rozmiaru kolumn, 577
 - zmiana rozmiaru wierszy, 577
- TABLE_CAT, 325
- TABLE_NAME, 325
- TABLE_SCHEM, 325
- TABLE_TYPE, 325
- TableCellEditor, 597, 598, 600
 - getTableCellEditorComponent(), 600
- TableCellRenderer, 590, 591, 599
 - getTableCellRendererComponent(), 599
- TableColumn, 577, 587, 600
 - setMaxWidth(), 588
 - setMinWidth(), 588
 - setPreferredWidth(), 588
 - setResizable(), 588
 - setWidth(), 588
- TableColumnModel, 576, 587
 - getColumn(), 587
 - setCellEditor(), 600
 - setCellRenderer(), 600
 - setHeaderRenderer(), 600
 - setHeaderValue(), 600
- TableModel, 574, 586
 - getColumnCount(), 574
 - getColumnName(), 575
 - getRowCount(), 574
 - getValueAt(), 575
 - isCellEditable(), 575
 - setValueAt(), 575
- tablice, 758, 760
 - C, 760
 - C++, 761
 - jarrray, 761
 - rozmiar, 761
- TCP, 225
- testowanie, 433
- Text, 164
- text/plain, 250
- TextField, 365
- TextLayout, 669, 670
 - getAdvance(), 670
 - getAscent(), 670
 - getDescent(), 670
 - getLeading(), 670
- TexturePaint, 661, 662
- Thread, 504
 - getContextClassLoader(), 504
 - setContextClassLoader(), 504
- ThreadedEchoHandler, 232
- Throw(), 764, 768
- ThrowNew(), 764, 768
- Time, 330
- TIME, 279, 330
- Timestamp, 330
- TIMESTAMP, 279, 330
- TimeZone, 377
 - getAvailableIDs(), 377
- toArray(), 732
- Tomcat, 332
- toString(), 364, 370, 441, 603, 650
- transakcje, 326
 - aktualizacje wsadowe, 327
 - automatyczne zatwierdzenie, 326
 - odwołanie, 326
 - punkty kontrolne, 327
 - tworzenie, 326
- transform(), 205, 216, 636, 666, 667
- Transformer, 205
 - setOutputProperty(), 205
 - transform(), 205
- TransformerFactory, 205, 219
 - newInstance(), 205
- translate(), 664, 668, 712
- TreeCellRenderer, 618, 619, 620
 - getTreeCellRendererComponent(), 620
- TreeModel, 602, 609, 628, 634
 - addTreeModelListener(), 635
 - getChild(), 634
 - getChildCount(), 634
 - getIndexOfChild(), 634
 - getRoot(), 634
 - isLeaf(), 609, 634
 - removeTreeModelListener(), 635
 - valueForPathChanged(), 635
- TreeModelEvent, 635
- TreeModelListener, 635
 - treeNodesChanged(), 635
 - treeNodesInserted(), 635
 - treeNodesRemoved(), 635
 - treeStructureChanged(), 635
- TreeNode, 603, 608, 610, 616
 - children(), 616
 - getAllowsChildren(), 609
 - getChildAt(), 616
 - getChildCount(), 616
 - getParent(), 616
 - isLeaf(), 608
- treeNodesChanged(), 635
- treeNodesInserted(), 635
- treeNodesRemoved(), 635
- TreePath, 610, 615, 622
 - getLastPathComponent(), 615
- TreeSelectionEvent, 622, 627
 - getPath(), 627
 - getPaths(), 627
- TreeSelectionListener, 621, 627
 - valueChanged(), 627
- TreeSelectionModel, 621
- treeStructureChanged(), 635
- trim(), 164, 365
- tworzenie
 - certyfikatów, 546
 - dokumentów XML, 202
 - drzewa DOM, 202
 - grafiki, 635
 - klas pozwoleń, 519
- tworzenie zbiorów rekordów, 313
- TYPE_BILINEAR, 695
- TYPE_BYTE_GRAY, 691
- TYPE_FORWARD_ONLY, 307
- TYPE_INT_ARGB, 689
- TYPE_SCROLL_INSENSITIVE, 307, 309
- TYPE_SCROLL_SENSITIVE, 307
- typy danych C, 740
- typy danych Java, 330, 740
- typy danych SQL, 279, 330
- typy wyjątków SQL, 290

U

UDP, 225
 układ współrzędnych, 663
 Unicode, 357
 uniform resource identifiers, 242
 uniform resource locator, 242
 uniform resource name, 242
 UnixLoginModule, 527
 UnixNumericGroupPrincipal, 527
 UnixPrincipal, 526
 UnknownHostException, 224
 UPDATE, 286, 297
 update(), 543, 556, 560
 updateDouble(), 309
 updateRow(), 309, 310, 312
 uporządkowanie łańcuchów, 378
 uporządkowanie słownikowe, 378
 uprawnienia, 508
 URI, 190, 242, 243, 728
 absolutny, 243
 hierarchiczny, 243
 identyfikatory, 243
 nieprzenikalny, 243
 relatywizacja, 244
 rozwiązywanie, 243
 specyfikacja, 242
 względny, 243
 URL, 191, 242, 249, 514
 nagłówki żądań, 245
 openConnection(), 249
 openStream(), 249
 pobieranie informacji, 244
 URLConnection, 242, 244, 245, 249, 253
 connect(), 250
 getConnectionTimeout(), 249
 getContent(), 250
 getContentEncoding(), 250
 getContentLength(), 250
 getContentType(), 250
 getDate(), 250
 getDoInput(), 249
 getDoOutput(), 249
 getExpiration(), 250
 getHeaderField(), 250
 getHeaderFieldKey(), 250
 getHeaderFields(), 250
 getIfModifiedSince(), 249
 getLastModified(), 250
 getRequestProperties(), 250
 openConnection(), 244
 openInputStream(), 250
 openOutputStream(), 250
 setConnectTimeout(), 249
 setDoInput(), 245, 249
 setDoOutput(), 245, 249
 setIfModifiedSince(), 249
 setModifiedSince(), 245
 setReadTimeout(), 249
 setRequestProperty(), 245, 250
 URLDecoder, 259
 decode(), 259
 URLEncoder, 259
 encode(), 259
 URN, 242
 uruchamianie bazy danych, 281
 usługi drukowania, 720
 GIF, 720
 rodzaje dokumentów, 721
 strumienie, 722
 źródło danych, 721
 usługi sieciowe, 221
 usługi uwierzytelniania, 526
 UTF-16, 391, 742, 743
 UTF-32, 743
 UTF-8, 391, 742
 uwierzytelnianie
 oparte na rolach, 532
 uwierzytelnianie użytkowników, 526
 grant, 527
 JAAS, 526
 moduł logowania, 527
 moduły, 527
 nadzorczy, 527
 podmiot, 527
 pozwolenia, 528
 uwierzytelnianie wiadomości, 548
 klucze, 549
 łańcuch zaufania, 549
 podpis zaufanego pośrednika, 549
 zaufany pośrednik, 549

V

va_list, 760
 valueChanged(), 621, 627
 valueForPathChanged(), 632, 635
 VARCHAR, 279, 330
 Variable, 633, 634
 vm_args, 769

W

W3C, 194
 waluta, 364, 369
 formatowanie, 369
 identyfikatory, 370
 warning(), 176, 177
 warstwa pośrednia, 274
 wartość alfa, 670
 wątki, 232
 Web, 221, 251, 252
 WebRowSet, 313
 weryfikacja dokumentu XML, 175
 weryfikacja kodu maszyny
 wirtualnej, 504
 weryfikacja podpisu cyfrowego, 545
 weryfikacja podpisu plików JAR,
 548
 weryfikator kodu, 504, 507
 węzły, 601
 nadrzędne, 601
 podrzędne, 601
 przeglądanie, 616
 rysowanie, 618
 WHERE, 277, 278
 wielokąty, 638, 643, 650
 wiersze, 575
 filtrowanie, 580
 sortowanie, 579
 wybieranie, 578
 Win32RegKey, 775
 Win32RegKeyNameEnumeration,
 777
 Windows, 772, 773
 rejestr, 773
 WordCheckPermission, 520
 WritableByteChannel, 236
 WritableRaster, 688, 690, 693
 setDataElements(), 693
 setPixel(), 694
 setPixels(), 694
 write(), 236, 562, 679, 685, 687
 writeInsert(), 684, 687
 współrzędne, 639
 ekranowe, 663
 użytkownika, 663
 wygląd drzewa, 605
 wyjątek SQLException, 289
 wyjątki
 ArrayIndexOutOfBoundsException
 ↳ Exception, 765
 ArrayStoreException, 765
 IllegalStateException, 683
 IndexOutOfBoundsException,
 683
 MissingResourceException, 392
 ParseException, 365
 PrinterException, 704
 SecurityException, 509, 511
 SQLException, 327
 SyncProviderException, 315
 UnknownHostException, 224

wykonywanie zapytań SQL, 295
 wykrywanie krawędzi, 700
 wynik zapytania, 276
 wypełnianie obszaru, 636, 638, 661
 gradient, 661
 kolory, 661
 obrazek wzorca, 662
 prostokąt zakotwiczenia, 662
 wyrażenia XPath, 186
 wyrzucanie wyjątków, 764
 wysyłanie danych do formularzy, 251
 wysyłanie poczty elektronicznej, 266
 wyświetlanie nagłówka, 591
 wywołanie funkcji języka C, 734
 parametry, 737
 printf(), 734
 wywołanie metod języka Java, 754, 759
 wywołanie metod obiektów, 754
 wywołanie metod statycznych, 757
 wzorce model-widok-nadzorca, 602

X

XML, 156
 atributy, 158, 159
 ATTLIST, 173
 CDATA, 160, 174
 deklaracja typu dokumentu, 158
 DOM, 161
 element korzenia, 158
 instrukcje przetwarzania, 160
 JAXP, 161
 komentarze, 160
 kontrola poprawności dokumentów, 169
 mieszana zawartość, 159
 parser, 161
 parsowanie dokumentów, 160
 PCDATA, 172
 pliki, 156
 przestrzeń nazw, 190
 przetwarzanie dokumentów, 161

referencje bytów, 160
 referencje znaków, 160
 SAX, 161, 193
 struktura dokumentu, 158
 wartości atrybutów, 157
 wartości domyślne atrybutów, 174
 wielkość znaków, 157
 XPath, 186
 znaczniki, 157
 XML Schema, 170, 178, 191
 atributy, 180
 definicje elementów, 180
 parsowanie dokumentu XML, 180
 powtórzenia elementów, 179
 przestrzeń nazw, 178
 typ elementu, 178
 typy proste, 178
 typy złożone, 179
 xmlns, 191
 xmlns:alias, 192
 XMLReader, 220
 parse(), 220
 setContentHandler(), 220
 XPath, 186, 190, 213
 evaluate(), 190
 funkcje, 187
 wartość wyrażenia, 187
 wyrażenia, 186
 wyznaczanie wyrażeń, 187
 zbiór węzłów, 187
 XPathConstants, 187
 NODE, 187
 XPathFactory, 189
 newInstance(), 189
 newXPath(), 189
 xsd:attribute, 180
 xsd:boolean, 178
 xsd:choice, 179
 xsd:int, 178
 xsd:schema, 180
 xsd:sequence, 179
 xsd:string, 178
 XSL Schema Definition, 178
 xsl:apply-templates, 213
 xsl:output, 213
 xsl:template, 213

xsl:value-of, 214
 XSLT, 203, 213

Z

zadania drukowania, 703
 zapis dużych obiektów, 301
 zapytania SQL, 277
 przygotowane, 296
 zarządzanie połączeniami, 331
 zasada Gödla, 504
 zasada składania obrazów, 636, 637
 zasoby, 391
 lokalizacja, 392
 zbiory atrybutów drukowania, 727
 zbiory pozwoleń, 509
 zbiory rekordów, 306, 309, 313
 aktualizowalne, 309
 buforowane, 314
 CachedRowSet, 314
 modyfikacja, 315
 przewijalne, 307
 RowSet, 313
 sprawdzanie, 315
 zbiory znaków, 388
 zbiór Mandelbrota, 690
 zestawy znaków, 388
 zmiana rozmiaru kolumn, 577
 zmiana rozmiaru wierszy, 577
 zmienne
 LD_LIBRARY_PATH, 773
 znaczniki XML, 157
 znaki, 388

Ź

źródło danych, 332
 źródło danych JDBC, 280
 źródło danych JNDI, 331
 źródło kodu, 509
 certyfikaty, 509
 lokalizacja kodu, 509

Ż

żądania certyfikatu, 551

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: NOWOCZESNY STYL, NAJLEPSZE PRAKTYKI, SPRAWDZONE ROZWIĄZANIA!

Java jest dojrzałym językiem programowania, który pozwala na pisanie kodu dla wielu rodzajów komputerów służących do różnych celów i działających na różnych platformach. Jest świetnym wyborem dla programistów, którym zależy na tworzeniu bezpiecznych aplikacji o wyjątkowej jakości. Wokół Javy skupia się duża społeczność, dzięki której język ten wciąż się rozwija, unowocześnia i wzbogaca o nowe elementy. Osoby, które swoje zawodowe życie wiążą z pisaniem programów w Javie, muszą poznać zaawansowane zagadnienia i mniej oczywiste funkcjonalności Javy, również te niedawno zaimplementowane. To konieczność dla każdego profesjonalnego programisty Javy.

Oto kolejne, przejrane, zaktualizowane i uzupełnione wydanie znakomitego podręcznika dla zawodowych programistów Javy. Znalazł się tu dokładny opis sposobów tworzenia interfejsu użytkownika, stosowania rozwiązań korporacyjnych, sieciowych i zabezpieczeń, a także nowości wprowadzonych w JDK 11. Przedstawiono techniki programowania baz danych oraz umiędzynarodowienia aplikacji Javy. Sporo uwagi poświęcono bibliotece Swing oraz jej wykorzystaniu do tworzenia realistycznej grafiki i efektów specjalnych. Ponadto w książce zostały pokazane nowe możliwości języka — zademonstrowano, jak dzięki nim uzyskać wyjątkową jakość aplikacji, a zamieszczone przykłady opracowano pod kątem zrozumiałości i wartości praktycznej.

W TEJ KSIĄŻCE MIĘDZY INNYMI:

- API wejścia-wyjścia Javy, serializacja i wyrażenia regularne
- efektywne korzystanie z usług sieciowych
- klienci, serwery i pobieranie danych z internetu
- moduły platformy Javy
- nowoczesne mechanizmy bezpieczeństwa w Javie

CAY S. HORSTMANN pochodzi z Niemiec. Od niemal 30 lat wykłada informatykę na Uniwersytecie Stanowym w San Jose, współpracuje też z innymi uczelniami — w Niemczech, Szwajcarii czy Wietnamie. Wcześniej z sukcesem rozwinął startup internetowy. Zdobył tytuł Java Champion. Napisał kilkanaście książek dla profesjonalnych programistów i studentów informatyki.

 PRENTICE HALL
PEARSON EDUCATION

| | | |
|--|--|--|
|  Helion | <i>Sprawdź nasze szkolenia!</i>  SZKOLENIA AKADEMIA IT & BUSINESS WWW.SZKOLENIA.HELION.PL | KOD KORZYŚCI Sięgnij po więcej! ▶  |
|  helion.pl | | ISBN 978-83-283-6066-2 |
|  HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl | |  9 788328 360662 |
| INFORMATYKA W NAJLEPSZYM WYDANIU | | Cena: 149,00 zł |