

TAJNIKI JĘZYKA JAVA 7!

 PRENTICE  
HALL

# JAVA®

## Techniki zaawansowane

WYDANIE IX



CAY S. HORSTMANN · GARY CORNELL

 Helion

Tytuł oryginału: Core Java, Volume II - Advanced Features (9th Edition)

Tłumaczenie: Jaromir Senczyk

ISBN: 978-83-246-7765-8

Authorized translation from the English language edition, entitled CORE JAVA, VOLUME II – ADVANCED FEATURES, Ninth Edition; ISBN 013708160X; by Cay S. Horstmann; and Gary Cornell; published by Pearson Education, Inc, publishing as Prentice Hall.

Copyright © 2013 by Oracle and/or its affiliates, 500 Oracle Parkway, Redwood Shores, CA 94065.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education Inc.

Polish language edition published by HELION S.A. Copyright © 2013.

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

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

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

Wydawnictwo HELION  
ul. Kościuszki 1c, 44-100 GLIWICE  
tel. 032 231 22 19, 032 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/javtz9.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

[http://helion.pl/user/opinie/javtz9\\_ebook](http://helion.pl/user/opinie/javtz9_ebook)

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

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

# Spis treści

<b>Przedmowa .....</b>	<b>11</b>
<b>Podziękowania .....</b>	<b>15</b>
<b>Rozdział 1. Strumienie i pliki .....</b>	<b>17</b>
1.1. Strumienie .....	17
1.1.1. Odczyt i zapis bajtów .....	18
1.1.2. Zoo pełne strumieni .....	20
1.1.3. Łączenie filtrów strumieni .....	24
1.2. Strumienie tekstowe .....	28
1.2.1. Zapisywanie tekstu .....	28
1.2.2. Wczytywanie tekstu .....	31
1.2.3. Zapis obiektów w formacie tekstowym .....	31
1.2.4. Zbiory znaków .....	34
1.3. Odczyt i zapis danych binarnych .....	39
1.3.1. Strumienie plików o swobodnym dostępie .....	42
1.4. Archiwa ZIP .....	46
1.5. Strumienie obiektów i serializacja .....	49
1.5.1. Format pliku serializacji obiektów .....	54
1.5.2. Modyfikowanie domyślnego mechanizmu serializacji .....	60
1.5.3. Serializacja singletonów i wyliczeń .....	62
1.5.4. Wersje .....	63
1.5.5. Serializacja w roli klonowania .....	65
1.6. Zarządzanie plikami .....	68
1.6.1. Ścieżki dostępu .....	68
1.6.2. Odczyt i zapis plików .....	70
1.6.3. Kopiowanie, przenoszenie i usuwanie plików .....	72
1.6.4. Tworzenie plików i katalogów .....	72
1.6.5. Informacje o plikach .....	74
1.6.6. Przeglądanie plików w katalogu .....	75
1.6.7. Systemy plików ZIP .....	78
1.7. Mapowanie plików w pamięci .....	79
1.7.1. Struktura bufora danych .....	86
1.7.2. Blokowanie plików .....	88
1.8. Wyrażenia regularne .....	90

<b>Rozdział 2. Język XML .....</b>	<b>101</b>
2.1. Wprowadzenie do języka XML .....	102
2.1.1. Struktura dokumentu XML .....	104
2.2. Parsowanie dokumentów XML .....	107
2.3. Kontrola poprawności dokumentów XML .....	118
2.3.1. Definicje typów dokumentów .....	119
2.3.2. XML Schema .....	126
2.3.3. Praktyczny przykład .....	129
2.4. Wyszukiwanie informacji i XPath .....	142
2.5. Przestrzenie nazw .....	148
2.6. Parsery strumieniowe .....	150
2.6.1. Wykorzystanie parsera SAX .....	151
2.6.2. Wykorzystanie parsera StAX .....	156
2.7. Tworzenie dokumentów XML .....	160
2.7.1. Dokumenty bez przestrzeni nazw .....	160
2.7.2. Dokumenty z przestrzenią nazw .....	160
2.7.3. Zapisywane dokumentu .....	161
2.7.4. Przykład: tworzenie pliku SVG .....	162
2.7.5. Tworzenie dokumentu XML za pomocą parsera StAX .....	165
2.8. Przekształcenia XSL .....	172
<b>Rozdział 3. Programowanie aplikacji sieciowych .....</b>	<b>183</b>
3.1. Połączenia z serwerem .....	183
3.1.1. Limity czasu gniazd .....	187
3.1.2. Adresy internetowe .....	189
3.2. Implementacja serwerów .....	191
3.2.1. Obsługa wielu klientów .....	194
3.2.2. Połączenia częściowo zamknięte .....	196
3.3. Przerwianie działania gniazd sieciowych .....	198
3.4. Połączenia wykorzystujące URL .....	204
3.4.1. URL i URI .....	205
3.4.2. Zastosowanie klasy URLConnection do pobierania informacji .....	207
3.4.3. Wysyłanie danych do formularzy .....	216
3.5. Wysyłanie poczty elektronicznej .....	222
<b>Rozdział 4. Programowanie baz danych: JDBC .....</b>	<b>227</b>
4.1. Architektura JDBC .....	228
4.1.1. Typy sterowników JDBC .....	228
4.1.2. Typowe zastosowania JDBC .....	229
4.2. Język SQL .....	231
4.3. Instalacja JDBC .....	235
4.3.1. Adresy URL baz danych .....	237
4.3.2. Pliki JAR zawierające sterownik .....	237
4.3.3. Uruchamianie bazy danych .....	237
4.3.4. Rejestracja klasy sterownika .....	238
4.3.5. Nawiązywanie połączenia z bazą danych .....	239
4.4. Wykonywanie poleceń języka SQL .....	242
4.4.1. Zarządzanie połączonymi, poleceniami i zbiorami wyników .....	245
4.4.2. Analiza wyjątków SQL .....	246
4.4.3. Wypełnianie bazy danych .....	248

4.5.	Wykonywanie zapytań .....	252
4.5.1.	Polecenia przygotowane .....	252
4.5.2.	Odczyt i zapis dużych obiektów .....	258
4.5.3.	Sekwencje sterujące .....	260
4.5.4.	Zapytania o wielu zbiorach wyników .....	261
4.5.5.	Pobieranie wartości kluczy wygenerowanych automatycznie .....	262
4.6.	Przewijalne i aktualizowalne zbiorы wyników zapytań .....	263
4.6.1.	Przewijalne zbiorы wyników .....	263
4.6.2.	Aktualizowalne zbiorы rekordów .....	265
4.7.	Zbiorы rekordów .....	270
4.7.1.	Tworzenie zbiorów rekordów .....	270
4.7.2.	Buforowane zbiorы rekordów .....	271
4.8.	Metadane .....	274
4.9.	Transakcje .....	283
4.9.1.	Punkty kontrolne .....	284
4.9.2.	Aktualizacje wsadowe .....	284
4.9.3.	Zaawansowane typy języka SQL .....	287
4.10.	Zaawansowane zarządzanie połączniami .....	288
<b>Rozdział 5. Internacjonalizacja .....</b>	<b>291</b>	
5.1.	Lokalizatory .....	292
5.2.	Formaty liczb .....	297
5.2.1.	Waluty .....	302
5.3.	Data i czas .....	304
5.4.	Porządek alfabetyczny .....	311
5.4.1.	Moc uporządkowania .....	312
5.4.2.	Rozkład .....	313
5.5.	Formatowanie komunikatów .....	318
5.5.1.	Formatowanie z wariantami .....	320
5.6.	Pliki tekstowe i zbiorы znaków .....	322
5.6.1.	Internacjonalizacja a pliki źródłowe programów .....	322
5.7.	Komplety zasobów .....	324
5.7.1.	Wyszukiwanie kompletów zasobów .....	324
5.7.2.	Pliki właściwości .....	325
5.7.3.	Klasy kompletów zasobów .....	326
5.8.	Kompletny przykład .....	328
<b>Rozdział 6. Zaawansowane możliwości pakietu Swing .....</b>	<b>343</b>	
6.1.	Listy .....	343
6.1.1.	Komponent JList .....	344
6.1.2.	Modele list .....	349
6.1.3.	Wstawianie i usuwanie .....	354
6.1.4.	Odrysowywanie zawartości listy .....	355
6.2.	Tabele .....	359
6.2.1.	Najprostsze tabele .....	359
6.2.2.	Modele tabel .....	363
6.2.3.	Wiersze i kolumny .....	367
6.2.4.	Rysowanie i edycja komórek .....	383
6.3.	Drzewa .....	394
6.3.1.	Najprostsze drzewa .....	395
6.3.2.	Przeglądanie węzłów .....	410

6.3.3.	Rysowanie węzłów .....	412
6.3.4.	Nasłuchiwanie zdarzeń w drzewach .....	415
6.3.5.	Własne modele drzew .....	421
6.4.	Komponenty tekstowe .....	429
6.4.1.	Śledzenie zmian zawartości komponentów tekstowych .....	430
6.4.2.	Sformatowane pola wejściowe .....	433
6.4.3.	Komponent JSpinner .....	449
6.4.4.	Prezentacja HTML za pomocą JEditorPane .....	457
6.5.	Wskaźniki postępu .....	463
6.5.1.	Paski postępu .....	463
6.5.2.	Monitory postępu .....	466
6.5.3.	Monitorowanie postępu strumieni wejścia .....	469
6.6.	Organizatory komponentów i dekoratory .....	474
6.6.1.	Panele dzielone .....	475
6.6.2.	Panele z zakładkami .....	478
6.6.3.	Panele pulpitu i ramki wewnętrzne .....	483
6.6.4.	Rozmieszczenie kaskadowe i sąsiadujące .....	487
6.6.5.	Zgłaszanie weta do zmiany właściwości .....	495
<b>Rozdział 7. Zaawansowane możliwości biblioteki AWT .....</b>	<b>505</b>	
7.1.	Potokowe tworzenie grafiki .....	506
7.2.	Figury .....	508
7.2.1.	Wykorzystanie klas obiektów graficznych .....	511
7.3.	Pola .....	523
7.4.	Ślad pędzla .....	524
7.5.	Wypełnienia .....	532
7.6.	Przekształcenia układu współrzędnych .....	534
7.7.	Przycinanie .....	539
7.8.	Przezroczystość i składanie obrazów .....	541
7.9.	Wskaźówki operacji graficznych .....	549
7.10.	Czytanie i zapisywanie plików graficznych .....	555
7.10.1.	Wykorzystanie obiektów zapisu i odczytu plików graficznych .....	555
7.10.2.	Odczyt i zapis plików zawierających sekwencje obrazów .....	560
7.11.	Operacje na obrazach .....	565
7.11.1.	Dostęp do danych obrazu .....	565
7.11.2.	Filtrowanie obrazów .....	571
7.12.	Drukowanie .....	580
7.12.1.	Drukowanie grafiki .....	580
7.12.2.	Drukowanie wielu stron .....	589
7.12.3.	Podgląd wydruku .....	591
7.12.4.	Usługi drukowania .....	599
7.12.5.	Usługi drukowania za pośrednictwem strumieni .....	603
7.12.6.	Atrybuty drukowania .....	604
7.13.	Schowek .....	610
7.13.1.	Klasy i interfejsy umożliwiające przekazywanie danych .....	611
7.13.2.	Przekazywanie tekstu .....	612
7.13.3.	Interfejs Transferable i formaty danych .....	615
7.13.4.	Przekazywanie obrazów za pomocą schowka .....	617
7.13.5.	Wykorzystanie schowka systemowego do przekazywania obiektów Java .....	621
7.13.6.	Zastosowanie lokalnego schowka do przekazywania referencji obiektów .....	624

7.14.	Mechanizm „przeciągnij i upuść” .....	625
7.14.1.	Przekazywanie danych pomiędzy komponentami Swing .....	627
7.14.2.	Źródła przeciąganych danych .....	631
7.14.3.	Cele upuszczanych danych .....	633
7.15.	Integracja z macierzystą platformą .....	641
7.15.1.	Ekran powitalny .....	641
7.15.2.	Uruchamianie macierzystych aplikacji pulpitu .....	646
7.15.3.	Zasobnik systemowy .....	651

**Rozdział 8. JavaBeans .....657**

8.1.	Dlaczego ziarnka? .....	658
8.2.	Proces tworzenia ziarenek JavaBeans .....	660
8.3.	Wykorzystanie ziarenek do tworzenia aplikacji .....	662
8.3.1.	Umieszczanie ziarenek w plikach JAR .....	663
8.3.2.	Korzystanie z ziarenek .....	664
8.4.	Wzorce nazw właściwości ziarenek i zdarzeń .....	669
8.5.	Typy właściwości ziarenek .....	673
8.5.1.	Właściwości proste .....	673
8.5.2.	Właściwości indeksowane .....	674
8.5.3.	Właściwości powiązane .....	674
8.5.4.	Właściwości ograniczone .....	676
8.6.	Klasa informacyjna ziarnka .....	683
8.7.	Edytory właściwości .....	687
8.7.1.	Implementacja edytora właściwości .....	690
8.8.	Indywidualizacja ziarnka .....	697
8.8.1.	Implementacja klasy indywidualizacji .....	699
8.9.	Trwałość ziarenek JavaBeans .....	705
8.9.1.	Zastosowanie mechanizmu trwałości JavaBeans dla dowolnych danych .....	709
8.9.2.	Kompletny przykład zastosowania trwałości JavaBeans .....	715

**Rozdział 9. Bezpieczeństwo .....727**

9.1.	Ładowanie klas .....	728
9.1.1.	Hierarchia klas ładowania .....	730
9.1.2.	Zastosowanie procedur ładujących w roli przestrzeni nazw .....	732
9.1.3.	Implementacja własnej procedury ładującej .....	733
9.2.	Weryfikacja kodu maszyny wirtualnej .....	738
9.3.	Menedżery bezpieczeństwa i pozwolenia .....	742
9.3.1.	Bezpieczeństwo na platformie Java .....	744
9.3.2.	Pliki polityki bezpieczeństwa .....	747
9.3.3.	Tworzenie własnych klas pozwoleń .....	755
9.3.4.	Implementacja klasy pozwoleń .....	756
9.4.	Uwierzytelnianie użytkowników .....	762
9.4.1.	Moduły JAAS .....	767
9.5.	Podpis cyfrowy .....	776
9.5.1.	Skróty wiadomości .....	777
9.5.2.	Podpisywanie wiadomości .....	779
9.5.3.	Weryfikacja podpisu .....	781
9.5.4.	Problem uwierzytelniania .....	784
9.5.5.	Podpisywanie certyfikatów .....	786
9.5.6.	Żądania certyfikatu .....	787

---

9.6.	Podpisywanie kodu .....	788
9.6.1.	Podpisywanie plików JAR .....	789
9.6.2.	Certyfikaty twórców oprogramowania .....	793
9.7.	Szyfrowanie .....	795
9.7.1.	Szyfrowanie symetryczne .....	795
9.7.2.	Generowanie klucza .....	797
9.7.3.	Strumienie szyfrujące .....	801
9.7.4.	Szyfrowanie kluczem publicznym .....	803

**Rozdział 10. Skrypty, kompilacja i adnotacje ..... 807**

10.1.	Skrypty na platformie Java .....	807
10.1.1.	Wybór silnika skryptów .....	808
10.1.2.	Wykonywanie skryptów i wiązania zmiennych .....	809
10.1.3.	Przekierowanie wejścia i wyjścia .....	811
10.1.4.	Wyoływanie funkcji i metod skryptów .....	812
10.1.5.	Kompilacja skryptu .....	814
10.1.6.	Przykład: skrypty i graficzny interfejs użytkownika .....	815
10.2.	Interfejs kompilatora .....	819
10.2.1.	Kompilacja w najprostszym sposobie .....	820
10.2.2.	Stosowanie zadań kompilacji .....	820
10.2.3.	Przykład: dynamiczne tworzenie kodu w języku Java .....	826
10.3.	Stosowanie adnotacji .....	830
10.3.1.	Przykład: adnotacje obsługi zdarzeń .....	832
10.4.	Składnia adnotacji .....	837
10.5.	Adnotacje standardowe .....	841
10.5.1.	Adnotacje kompilacji .....	842
10.5.2.	Adnotacje zarządzania zasobami .....	842
10.5.3.	Metaadnotacje .....	843
10.6.	Przetwarzanie adnotacji w kodzie źródłowym .....	845
10.7.	Inżynieria kodu bajtowego .....	851
10.7.1.	Modyfikacja kodu bajtowego podczas ładowania .....	857

**Rozdział 11. Obiekty rozproszone ..... 861**

11.1.	Role klienta i serwera .....	862
11.2.	Wyołania zdalnych metod .....	864
11.2.1.	Namiastka i szeregowanie parametrów .....	864
11.3.	Model programowania RMI .....	866
11.3.1.	Interfejsy i implementacje .....	866
11.3.2.	Rejestr RMI .....	868
11.3.3.	Przygotowanie wdrożenia .....	871
11.3.4.	Rejestrowanie aktywności RMI .....	874
11.4.	Parametry zdalnych metod i wartości zwracane .....	876
11.4.1.	Przekazywanie obiektów zdalnych .....	876
11.4.2.	Przekazywanie obiektów, które nie są zdalne .....	876
11.4.3.	Dynamiczne ładowanie klas .....	878
11.4.4.	Zdalne referencje obiektów o wielu interfejsach .....	883
11.4.5.	Zdalne obiekty i metody equals, hashCode oraz clone .....	884
11.5.	Aktywacja zdalnych obiektów .....	884

---

<b>Rozdział 12. Metody macierzyste .....</b>	<b>891</b>
12.1. Wywołania funkcji języka C z programów w języku Java .....	892
12.2. Numeryczne parametry metod i wartości zwarcane .....	898
12.2.1. Wykorzystanie funkcji printf do formatowania liczb .....	899
12.3. Łańcuchy znaków jako parametry .....	900
12.4. Dostęp do składowych obiektu .....	906
12.4.1. Dostęp do pól instancji .....	906
12.4.2. Dostęp do pól statycznych .....	910
12.5. Sygnatury .....	911
12.6. Wywoływanie metod języka Java .....	912
12.6.1. Wywoływanie metod obiektów .....	912
12.6.2. Wywoływanie metod statycznych .....	916
12.6.3. Konstruktory .....	917
12.6.4. Alternatywne sposoby wywoływania metod .....	917
12.7. Tablice .....	919
12.8. Obsługa błędów .....	923
12.9. Interfejs programowy wywołań języka Java .....	927
12.10. Kompletny przykład: dostęp do rejestru systemu Windows .....	932
12.10.1. Rejestr systemu Windows .....	933
12.10.2. Interfejs dostępu do rejestru na platformie Java .....	934
12.10.3. Implementacja dostępu do rejestru za pomocą metod macierzystych ....	935
<b>Skorowidz .....</b>	<b>949</b>



# Przedmowa

## Do Czytelnika

Książka, którą przekazujemy, stanowi drugi tom dziewiątego wydania *Java. Techniki zaawansowane* w pełni zaktualizowanego dla Java SE 7. Pierwszy tom (*Java. Podstawy*) prezentuje podstawowe cechy języka Java, natomiast niniejszy omawia zaawansowane możliwości, które można wykorzystać do tworzenia profesjonalnych aplikacji. Książka ta *adresowana jest do programistów, którzy zamierzają użyć technologii Java w pracy nad poważnymi projektami*.

Jeśli jesteś już doświadczonym programistą języka Java i korzystanie z zaawansowanych możliwości języka, na przykład klas wewnętrznych, nie sprawia Ci trudności, to nie musisz czytać pierwszego tomu książki, aby przejść do lektury bieżącego. (Chociaż autorzy odwołują się często do zawartości pierwszego tomu i mają nadzieję, że jest już własnością czytelników, to podstawowy materiał przygotowujący do lektury niniejszego tomu można znaleźć w każdej książce omawiającej wyczerpujący sposób podstawy języka Java).

Z procesem tworzenia każdej książki w nieunikniony sposób związane są różnego rodzaju błędy. Na stronie <http://www.horstmann.com/corejava.html> umieściliśmy odpowiedzi na pytania najczęściej zadawane przez czytelników oraz informacje o zauważonych dotąd błędach<sup>1</sup>. Na końcu tej strony umieściliśmy też formularz, który można wykorzystać, aby przesłać nam informacje o innych błędach bądź sugestie dotyczące kolejnych wydań.

## O książce

Rozdziały niniejszej książki są w większości przypadków zupełnie niezależne od siebie i w praktyce można rozpocząć lekturę od najbardziej interesującego materiału, a pozostałe rozdziały przeczytać w dowolnej kolejności.

---

<sup>1</sup> Errata do polskiego wydania książki jest dostępna na stronie <http://helion.pl/ksiazki/javt9.htm> — przyp. tłum.

**Rozdział 1.** omawia obsługę wejścia i wyjścia, która na platformie Java odbywa się za pomocą tak zwanych strumieni. Strumienie pozwalają w jednolity sposób komunikować się z różnymi źródłami danych, takimi jak pliki, połączenia sieciowe czy bloki pamięci. W rozdziale tym szczegółowo przedstawimy klasy umożliwiające odczyt i zapis danych strumieni, które ułatwiają posługiwanie się kodem Unicode. Omówimy również sposób działania mechanizmu serializacji obiektów, który znakomicie ułatwia zapamiętywanie i ładowanie dowolnych obiektów. Na koniec poświęcimy sporo uwagi bibliotece ulepszonych klas wejścia i wyjścia NIO2 wprowadzonej w wersji Java SE 7; umożliwiają one efektywniejsze operacje na plikach, a także bibliotece wyrażeń regularnych.

**Rozdział 2.** poświęcony jest językowi XML. Pokazujemy w nim sposoby parsowania dokumentów XML, tworzenia dokumentów XML i zastosowania przekształceń XSL. Zagadnienia te ilustrujemy przydatnym przykładem opisu układu komponentów Swing za pomocą XML-a. Rozdział ten został zaktualizowany ze względu na interfejs programowy XPath, który znakomicie ułatwia „odnajdywanie igieł w stogach XML”.

**Rozdział 3.** opisuje tworzenie *aplikacji sieciowych*. Platforma Java umożliwia w niezwykle prosty sposób tworzenie złożonych programów pracujących w sieci. Przedstawiamy w nim sposoby tworzenia połączeń z serwerami, implementacji własnych serwerów oraz połączenia z użyciem protokołu HTTP.

W **rozdziale 4.** przedstawiamy interfejs programowy JDBC™ umożliwiający wykorzystanie relacyjnych baz danych w aplikacjach tworzonych w języku Java. Omawiamy jedynie najistotniejszy z praktycznego punktu widzenia podziób możliwość interfejsu JDBC, ponieważ jest on na tyle rozbudowany, że można mu poświęcić osobną książkę. Rozdział kończymy krótkim omówieniem interfejsu JNDI (*Java Naming and Directory Interface*).

**Rozdział 5.** opisuje zagadnienie, którego znaczenie, jak sądzimy, może tylko wzrastać: *internacjonalizację* aplikacji. Język Java od samego początku wykorzystuje kod Unicode, jednak jego możliwości w zakresie internacjonalizacji są dużo większe. Umożliwia to tworzenie aplikacji, które nie tylko potrafią pokonać bariery między różnymi platformami, ale także bariery językowe. Jako przykład prezentujemy aplet kalkulatora emerytalnego pracującego w języku angielskim, niemieckim lub chińskim — w zależności od lokalizacji maszyny, na którą go załadowano.

**Rozdział 6.** zawiera materiał dotyczący zaawansowanych możliwości biblioteki *Swing*, który nie zmieścił się już w pierwszym tomie książki. Poświęcony jest on przede wszystkim omówieniu skomplikowanych komponentów drzew i tabel. Przedstawia także wykorzystanie paneli edycji, implementację w języku Java interfejsu użytkownika zawierającego wiele okien dokumentów, wskaźniki postępu używane w programach wielowątkowych oraz integrację z pulpitem poprzez zastosowanie ekranów tytułowych czy wykorzystanie zasobnika systemu. Podobnie jak w innych rozdziałach, koncentrujemy się na zagadnieniach najbardziej przydatnych w praktyce programisty, ponieważ encyklopedyczne omówienie całej biblioteki Swing zajęłoby wiele tomów i byłoby interesujące jedynie z punktu widzenia klasyfikacji, a nie praktycznych zastosowań.

**Rozdział 7.** omawia interfejs programowy Java 2D, który umożliwia aplikacjom tworzenie realistycznych rysunków. Rozdział ten przedstawia także niektóre bardziej zaawansowane możliwości biblioteki *AWT*, które wydały nam się zbyt specjalistyczne, by przedstawić je w pierwszym tomie książki, ale jednak powinny znaleźć się wśród narzędzi wykorzystywanych

przez każdego programistę. Do możliwości tych należą tworzenie wydruków i mechanizmy „wytnij-i-wklej” oraz „przeciagnij-i-upuść”.

**Rozdział 8.** zawiera podstawowe wiadomości dotyczące architektury komponentów *Java-Beans™*. Prezentuje sposoby tworzenia własnych ziarnek, które mogą później być wykorzystywane przez innych programistów posługujących się narzędziami wizualnego tworzenia aplikacji. Rozdział kończymy przykładem wykorzystania mechanizmu trwałości JavaBeans, który w przeciwieństwie do serializacji, nadaje się do długotrwałego przechowywania obiektów.

**Rozdział 9.** omawia *model bezpieczeństwa* zastosowany w języku Java. Platforma Java została zaprogramowana od podstaw z myślą o bezpieczeństwie wykonania aplikacji i appletów. W rozdziale tym pokazujemy, w jaki sposób zostało to osiągnięte. Prezentujemy też sposoby tworzenia własnych procedur ładowania klas oraz menedżerów bezpieczeństwa wykorzystywanych przez wyspecjalizowane aplikacje. Przedstawiamy także możliwości interfejsu programowego bezpieczeństwa, który umożliwia realizację podpisywania kodu klas, podpisywania wiadomości, autoryzację i uwierzytelnianie, a także szyfrowanie. Rozdział kończymy przykładem wykorzystania algorytmów szyfrowania AES i RSA.

**Rozdział 10.** omawia trzy techniki przetwarzania kodu. Interfejsy programowe skryptów i kompilatora pozwalają programom na wywoływanie kodu napisanego w językach skryptowych takich jak JavaScript czy Groovy oraz na kompilowanie kodu w języku Java. Z kolei adnotacje umożliwiają dodawanie dowolnych informacji (czasami zwanych metadanymi) do kodu programów w języku Java. Przedstawiamy sposób działania procesorów adnotacji wykorzystujących adnotacje na poziomie kodu źródłowego. A także zastosowanie adnotacji do zmiany zachowania klas już podczas ich działania. Wartość adnotacji mierzona jest dostępnością odpowiednich narzędzi umożliwiających ich przetwarzanie. Mamy nadzieję, że rozdział ten pomoże Ci wybrać narzędzia odpowiednie do Twoich potrzeb.

**Rozdział 11.** poświęcony jest tematyce *obiektów rozproszonych*. Omówiony zostaje szczegółowo *interfejs zdalnych wywołań metod RMI (Remote Method Invocation)*. Pozwala on korzystać z usług obiektów rozproszonych na różnych maszynach pracujących w sieci.

**Rozdział 12.** prezentuje *metody rodzime*, które pozwalają na korzystanie z możliwości specyficznych tylko dla wybranej platformy, na przykład Microsoft Windows, jednak za cenę utraty możliwości wykonywania aplikacji na innych platformach. Mimo to uważamy, że każdy programista tworzący aplikacje w języku Java powinien opanować tę technikę. Podczas tworzenia profesjonalnych aplikacji często zdarza się bowiem, że jesteśmy zmuszeni skorzystać ze specyficznych interfejsów programowych wybranej platformy. Zagadnienia te ilustrujemy przykładem dostępu do rejestru systemu Windows.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem: <ftp://ftp.helion.pl/przyklady/javtz9.zip>.

## Konwencje typograficzne

Podobnie jak w wielu książkach komputerowych przykłady kodu programów pisane są czcionką o stałej szerokości znaków.



Taką ikoną opatrzone są uwagi.



Tą ikoną opatrzone są wskazówki.



Takiej ikony używamy, aby ostrzec przed jakimś niebezpieczeństwem.



W książce pojawia się wiele uwag wyjaśniających różnice pomiędzy Java a językiem C++. Jeśli nie znasz się na programowaniu w C++ lub na myśl o przykrych wspomnieniach z nim związanych dostajesz gęsiej skórki, możesz je pominąć.



Interfejs programowania aplikacji 1.2

Java posiada bardzo dużą bibliotekę programistyczną, czyli API (ang. *Application Programming Interface*). Kiedy po raz pierwszy używamy jakiegoś wywołania API, na końcu sekcji umieszczamy jego krótki opis. Opisy te są nieco nieformalne, ale staraliśmy się, aby zawierały więcej potrzebnych informacji niż te, które można znaleźć w oficjalnej dokumentacji API w internecie. Mając na uwadze dobrze czytelników, którzy nie używają najnowszej wersji Javy, każdą uwagę na temat API opatrzyliśmy numerem wersji, w której opisywana własność została wprowadzona.

Programy, których kod źródłowy można znaleźć w internecie, są oznaczane jako listingi, np.:

---

**Listing 1.1. ScriptTest.java**

---

Wszystkie programy przedstawione na listingach można znaleźć pod adresem:  
<ftp://ftp.helion.pl/przyklady/javtz9.zip>.

# Podziękowania

Pisanie książki jest zawsze ogromnym wysiłkiem, a opracowanie kolejnego wydania nie okazuje się łatwiejsze, zwłaszcza wobec tak gwałtownego rozwoju technologii Java. Proces ten wymaga poświecenia wielu osób i dlatego mam wielką przyjemność złożyć podziękowania całemu zespołowi tworzącemu *Core Java*.

Wiele osób w wydawnictwie Prentice Hall dołożyło starań, aby proces tworzenia niniejszej książki zakończył się sukcesem, ale ich wysiłek pozostał anonimowy. Chciałbym jednak, aby wszyscy oni wiedzieli, jak bardzo doceniam ich pracę. Po raz kolejny redaktorem książki był Greg Doench, który wspaniale wykonał zadanie polegające na koordynacji wszystkich aspektów tego złożonego projektu i w ten sposób pozwolił mi pozostać nieświadomym istnienia osób, które bezpośrednio zrealizowały ten projekt. Podziękowania za nadzór nad produkcją książki należą się Julie Nahil. Za skład odpowiedzialni byli Dmitry Kirsanov i Alina Kirsanova.

Podziękowania należą się czytelnikom wcześniejszych wydań, którzy wykryli w nich błędy i przesłali wiele pomysłów na ulepszenie książki. Swoją wdzieczność wyrażam także doskonałemu zespołowi korektorskiemu, który dokładnie przeanalizował rękopis książki, co pozwoliło uniknąć wielu istotnych błędów.

Swoją uwagę temu i wcześniejszym wydaniom książki poświęcili: Chuck Allison (redaktor, *C/C++ User Journal*), Lance Anderson (Oracle), Alec Beaton (PointBase, Inc.), Cliff Berg (iSavvix Corporation), Joshua Bloch, David Brown, Corky Cartwright, Frank Cohen (Push-ToTest), Chris Crane (devXSolution), dr Nicholas J. De Lillo (Manhattan College), Rakesh Dhoopar (Oracle), Robert Evans (starszy wykładowca, The Johns Hopkins University Applied Physics Lab), David Geary (Sabreware), Brian Goetz (główny konsultant, Quiotix Corp.), Angela Gordon, Dan Gordon, Rob Gordon, John Gray (University of Hartford), Cameron Gregory (olabs.com), Marty Hall (The Johns Hopkins University Applied Physics Lab), Vincent Hardy, Dan Harkey (San Jose State University), William Higgins (IBM), Vladimir Ivanovic (PointBase), Jerry Jackson (ChannelPoint Software), Tim Kimmet (Preview Systems), Chris Laffra, Charlie Lai, Angelika Langer, Doug Langston, Hang Lau (McGill University), Mark Lawrence, Doug Lea (SUNY Oswego), Gregory Longshore, Bob Lynch (Lynch Associates), Philip Milne (konsultant), Mark Morrissey (The Oregon Graduate Institute), Mahesh Neelakanta (Florida Atlantic University), Hao Pham, Paul

Philion, Blake Ragsdell, Ylber Ramadani (Ryerson University), Stuart Reges (University of Arizona), Rich Rosen (Interactive Data Corporation), Peter Sanders (ESSI University, Nicea, Francja), Dr. Paul Sanghera (San Jose State University i Brooks College), Paul Sevinc (Teamup AG), Devang Shah, Richard Slywczak (NASA/Glenn Research Center), Bradley A. Smith, Steven Stelting, Christopher Taylor, Luke Taylor (Valtech), George Thiruvathukal, Kim Topley (autor książki *Core JFC*), Janet Traub, Paul Tyma (konsultant), Peter van der Linden, Burt Walsh, Joe Wang (Oracle) i Dan Xu(Oracle).

*Cay Horstmann  
San Francisco, Kalifornia  
grudzień 2012*

# 1

## Strumienie i pliki

W tym rozdziale:

- strumienie,
- strumienie tekstowe,
- odczyt i zapis danych binarnych,
- archiwa ZIP,
- strumienie obiektów i serializacja,
- zarządzanie plikami,
- pliki mapowane w pamięci,
- wyrażenia regularne.

W tym rozdziale omówimy interfejsy programowe związane z obsługą wejścia i wyjścia programów. Przedstawimy sposoby dostępu do plików i katalogów oraz sposoby zapisywania do i wczytywania informacji z plików w formacie tekstowym i binarnym. W rozdziale przedstawiony jest również mechanizm serializacji obiektów, który umożliwia przechowywanie obiektów z taką łatwością, z jaką przechowujesz tekst i dane numeryczne. Następnie omówimy szereg ulepszeń, które do obsługi wejścia i wyjścia wprowadził pakiet `java.nio` udostępniony w wersji Java SE 1.4, a także najświeższe rozszerzenia i ulepszenia wprowadzone w Java 7. Rozdział zakończymy przedstawieniem problematyki wyrażeń regularnych, mimo że nie jest ona bezpośrednio związana ze strumieniami i plikami. Nie potrafiliśmy jednak znaleźć dla niej lepszego miejsca w książce. W naszym wyborze nie byliśmy zresztą osamotnieni, ponieważ zespół Javy dołączył specyfikację interfejsów programowych związanych z przetwarzaniem wyrażeń regularnych do specyfikacji ulepszonej obsługi wejścia i wyjścia.

### 1.1. Strumienie

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

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

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

### 1.1.1. 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 przesyłają 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 z klawiatury.

Klasa `InputStream` posiada również nieabstrakcyjne metody pozwalające pobrać lub zignorować tablicę bajtów. Metody te wywołują abstrakcyjną metodę `read`, tak więc podklasy muszą przesyłać tylko tę jedną metodę.

Analogicznie, klasa `OutputStream` definiuje metodę abstrakcyjną

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

która wysyła jeden bajt do aktualnego wyjścia.

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

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

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

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

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

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

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

#### `java.io.InputStream` 1.0

- `abstract int read()`

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

- `int read(byte[] b)`

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

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

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

*Parametry:*    `b`                         tablica, w której zapisywane są dane.

`off`                         indeks tablicy `b`, pod którym powinien zostać umieszczony pierwszy wczytany bajt.

`len`                         maksymalna liczba wczytywanych bajtów.

- `long skip(long n)`

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

- `int available()`

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

- `void close()`

zamyka strumień wejścia.

- `void mark(int readlimit)`

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

- void reset()
 

wraca do ostatniego znacznika. Późniejsze wywołania read będą powtórnie czytać pobrane już bajty. Jeżeli znacznik nie istnieje, strumień nie zostanie zresetowany.
- boolean markSupported()
 

zwraca true, jeżeli strumień obsługuje znaczniki.

**API java.io.OutputStream 1.0**

- abstract void write(int n)
 

zapisuje jeden bajt.
- void write(byte[] b)
 

zapisują wszystkie bajty tablicy b lub pewien ich zakres.

*Parametry:*

b	tablica, z której pobierane są dane.
off	indeks tablicy b, spod którego powinien zostać pobrany pierwszy zapisywany bajt.
len	liczba zapisywanych bajtów.
- void close()
 

opróżnia i zamyka strumień wyjścia.
- void flush()
 

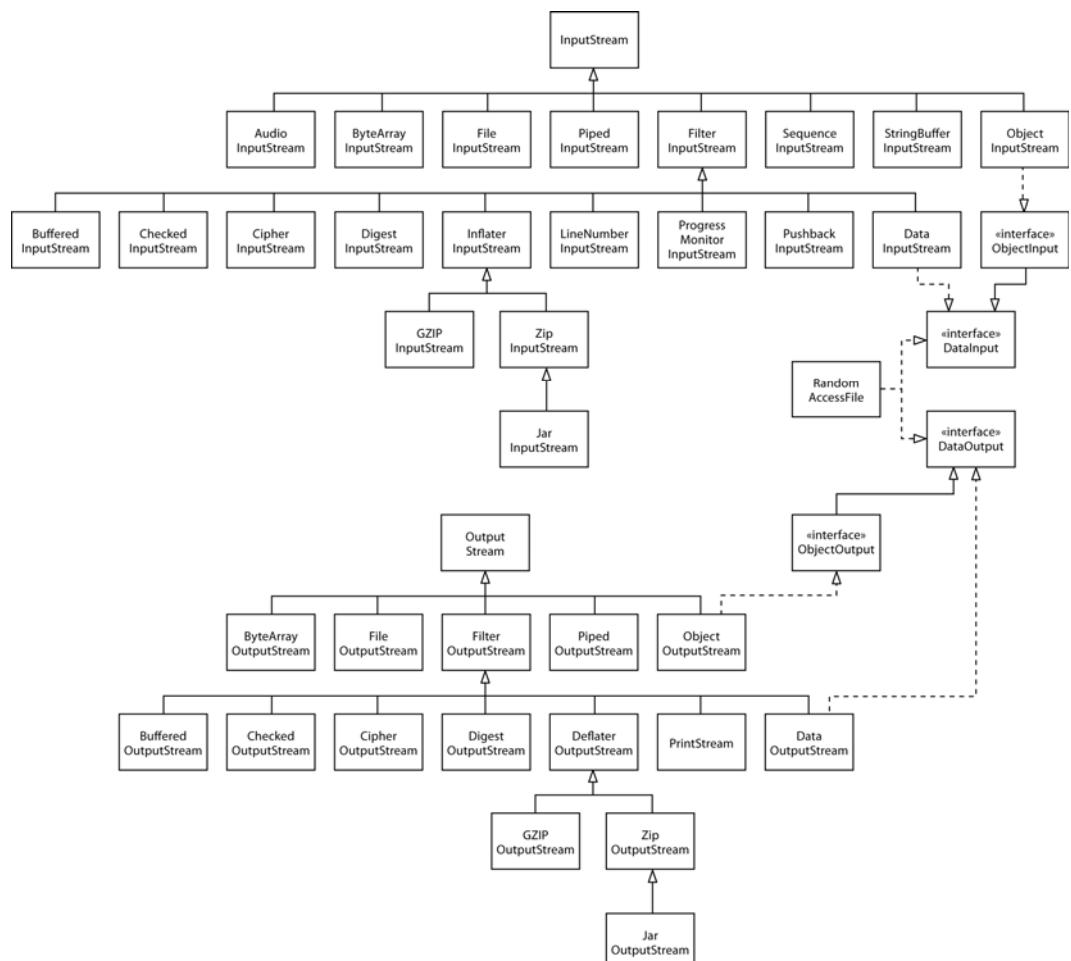
opróżnia strumień wyjścia, czyli wysyła do odbiorcy wszystkie dane znajdujące się w buforze.

## 1.1.2. 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 (patrz rysunki 1.1 i 1.2).

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

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



Rysunek 1.1. Hierarchia strumieni wejścia i wyjścia

```

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

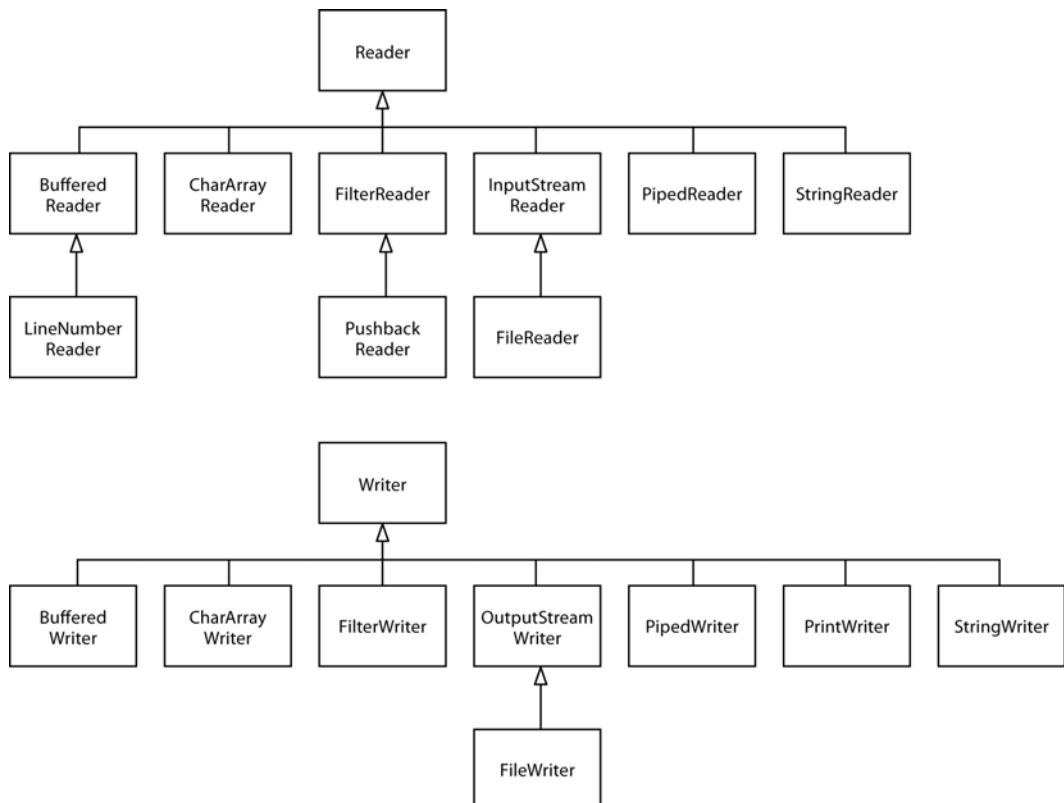
Metoda `read` zwraca albo kod znaku Unicode (jako liczbę z przedziału od 0 do 65535), albo -1, jeżeli natrafi na koniec pliku. Metoda `write` jest wywoływana dla podanego kodu znaku Unicode (więcej informacji na temat kodów Unicode znajdziesz w rozdziale 3. książki *Java. Podstawy*).

Dostępne są również cztery dodatkowe interfejsy: `Closeable`, `Flushable`, `Readable` i `Appendable` (patrz rysunek 1.3). Pierwsze dwa z nich są wyjątkowo proste i zawierają odpowiednio metody:

```

void close() throws IOException
  i
void flush()
  
```

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



**Rysunek 1.2.** Hierarchia klas Reader i Writer



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`.

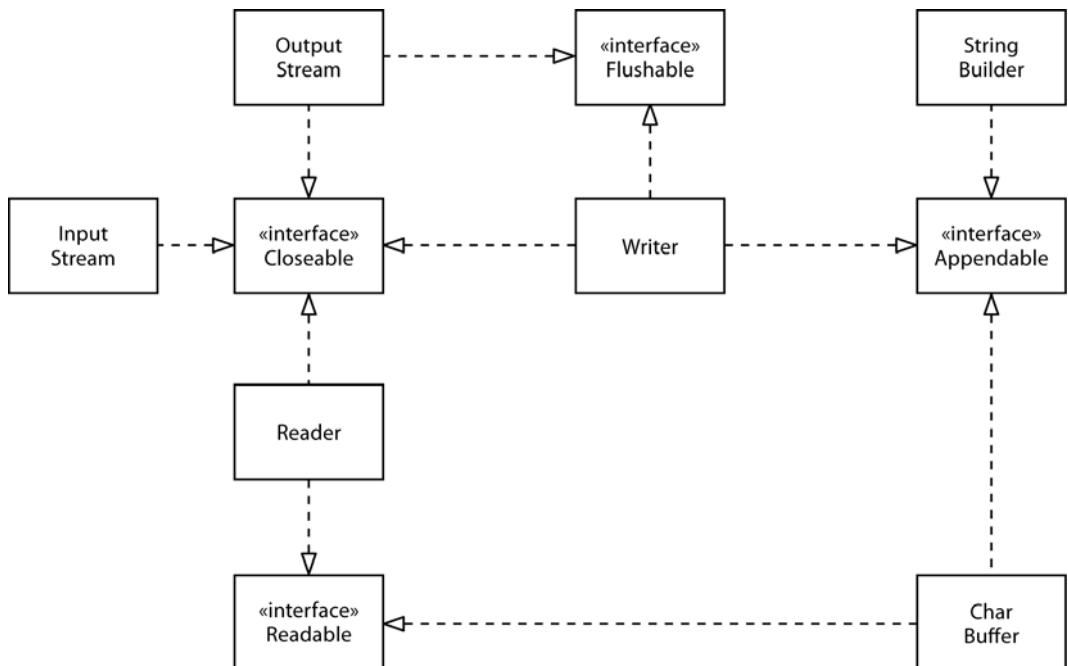
Interfejs `Readable` ma tylko jedną metodę

```
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 1.7.1, „Struktura bufora danych”).

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

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



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

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

Spośród klas strumieni jedynie klasa Writer implementuje interfejs Appendable.

#### API `java.io.Closeable 5.0`

- `void close()`

zamyka obiekt 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)`

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

**API java.lang.Appendable 5.0**

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

**API java.lang.CharSequence 1.4**

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

### 1.1.3. Łączenie filtrów strumieni

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

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

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



Ponieważ wszystkie klasy w `java.io` uznają relatywne ścieżki dostępu za rozpoznające się od aktualnego katalogu roboczego, powinieneś wiedzieć, co to za katalog. Możesz pobrać tę informację poleciением `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 ścieżek 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 przeowywany jako stała `java.io.File.separator`.

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

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

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

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

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

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

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

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

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

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

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

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

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

Teraz możemy odczytać wartość następnego bajta:

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

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

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

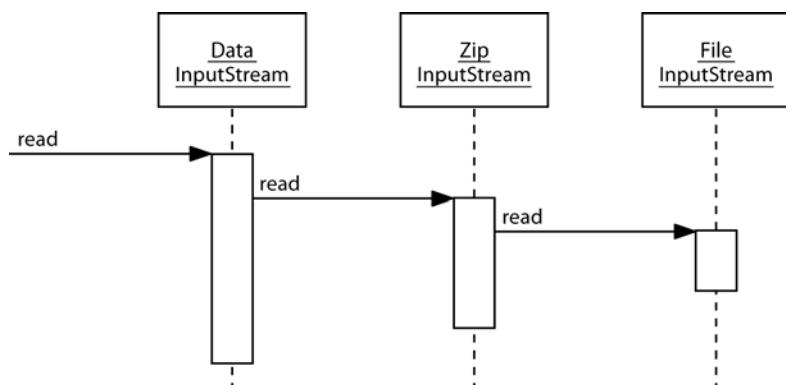
Ale wczytywanie i powtórne wstawianie bajtów to *jedynie* metody obsługiwane przez klasę Pushback-InputStream. Jeżeli chcemy podejrzeć bajty, a także wczytywać liczby, potrzebujemy referencji zarówno do PushbackInputStream, jak i do DataInputStream.

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

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

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

**Rysunek 1.4.**  
*Sekwencja filtrowanych strumieni*



Aby dowiedzieć się więcej o obsłudze formatu ZIP, zajrzyj do podrozdziału 1.4, „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.BufferedReader 1-8

- #### ■ `BufferedInputStream(InputStream in)`

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

API java.io.BufferedOutputStream 1-8

- #### ■ BufferedOutputStream(OutputStream out)

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

API java.io.PushbackInputStream 1-9

- PushbackInputStream(InputStream in)
  - PushbackInputStream(InputStream in, int size)  
tworzą strumień umożliwiający podgląd kolejnego bloku rozmiarze.
  - void unread(int b)

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

Parametry:      **b**                          zwracany bait

## 1.2. Strumienie tekstowe

Zapisując dane, możemy wybierać pomiędzy formatem binarnym i tekstowym. Dla przykładu: jeżeli liczba całkowita 1234 zostanie zapisana w postaci binarnej, w pliku pojawi się sekwencja bajtów 00 00 04 D2 (w notacji szesnastkowej). W formacie tekstowym liczba ta zostanie zapisana jako łańcuch "1234". Mimo iż zapis danych w postaci binarnej jest szybki i efektywny, to uzyskany wynik jest kompletnie nieczytelny dla ludzi. W poniższym podrozdziale skoncentrujemy się na *tekstowym* wejściu-wyjściu, a odczyt i zapis danych binarnych omówimy w podrozdziale 1.3., „Odczyt i zapis danych binarnych”.

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

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

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

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

Obiekt wejścia korzysta z domyślnego kodowania lokalnego systemu, na przykład ISO 8859-1 stosowanego w Europie Zachodniej. Możemy wybrać inny sposób kodowania, podając jego nazwę w konstruktorze `InputStreamReader`, na przykład:

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

Więcej informacji na temat kodowania znaków znajdziesz w punkcie 1.2.4., „Zbiory znaków”.

### 1.2.1. Zapisywanie tekstu

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

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

stanowi odpowiednik instrukcji

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

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

Spójrzmy na poniższy kod:

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

Rezultatem jego wykonania będzie wysłanie napisu

Harry Hacker 75000.0

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

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

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

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

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



Weterani Javy prawdopodobnie zastanawiają się, co się stało z klasą `PrintStream` i obiektem `System.out`. W języku Java 1.0 klasa `PrintStream` obcinała znaki Unicode do znaków ASCII, po prostu opuszczając górny bajt (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, 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 przeciwnieństwie do `PrintWriter` pozwalają wysyłać bajty za pomocą metod `write(int)` i `write(byte[])`.

#### `java.io.PrintWriter 1.1`

■ `PrintWriter(Writer out)`

- `PrintWriter(Writer out, boolean autoFlush)`  
tworzy nowy obiekt klasy `PrintWriter`.  
*Parametry:*    `out`                obiekt zapisu tekstu.  
                    `autoFlush` true oznacza, że metody `println` będą opróżniać bufor (domyślnie: false).
- `PrintWriter(OutputStream out)`
- `PrintWriter(OutputStream out, boolean autoFlush)`  
tworzy nowy obiekt klasy `PrintWriter` na podstawie istniejącego obiektu typu `OutputStream`, poprzez utworzenie pośredniczącego obiektu klasy `OutputStreamWriter`.
- `PrintWriter(String filename)`
- `PrintWriter(File file)`  
tworzy nowy obiekt klasy `PrintWriter` zapisujący dane do pliku poprzez utworzenie pośredniczącego obiektu klasy `FileWriter`.
- `void print(Object obj)`  
zapisuje łańcuch zwracany przez metodę `toString` danego obiektu.  
*Parametry:*    `obj`                drukowany obiekt.
- `void print(String p)`  
zapisuje łańcuch Unicode.
- `void println(String p)`  
zapisuje łańcuch zakończony znakiem końca wiersza. Jeżeli automatyczne opróżnianie jest włączone, opróżnia bufor strumienia.
- `void print(char[] p)`  
zapisuje tablicę znaków Unicode.
- `void print(char c)`  
zapisuje znak Unicode.
- `void print(int i)`
- `void print(long l)`
- `void print(float f)`
- `void print(double d)`
- `void print(boolean b)`  
zapisuje podaną wartość w formacie tekstowym.
- `void printf(String format, Object... args)`  
zapisuje podane wartości według łańcucha formatującego. Specyfikację łańcucha formatującego znajdziesz w rozdziale 3. książki *Java. Podstawy*.

- boolean checkError()

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

## 1.2.2. Wczytywanie tekstu

Wiemy już, że:

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

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

```
BufferedReader in = new BufferedReader(
    new InputStreamReader(new FileInputStream("employee.txt"), "UTF-8"));
```

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

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

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

## 1.2.3. Zapis obiektów w formacie tekstowym

W tym podrozdziale przeanalizujemy działanie przykładowego programu, który będzie zapisywać tablicę obiektów typu `Employee` w pliku tekstowym. Dane każdego obiektu zostaną zapisane w osobnym wierszu. Wartości pól składowych zostaną oddzielone od siebie separatorami. Jako separatorka 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 nasłańcuchów?

Oto przykładowy zbiór danych obiektów:

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

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

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

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

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

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

Kompletny program został przedstawiony na listingu 1.1. Metoda statyczna

```
void writeData(Employee[] e, PrintWriter out)
najpierw zapisuje rozmiar tablicy, a następnie każdy z rekordów. Metoda statyczna
Employee[] readData(BufferedReader in)
najpierw wczytuje rozmiar tablicy, a następnie każdy z rekordów. Wymaga to zastosowania
pewnej sztuczki:
```

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

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

### **Listing 1.1. *textfile/TextFileTest.java***

```
package textfile;

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

/**
 * @version 1.13 2012-05-30
 * @author Cay Horstmann
 */
public class TextFileTest
{
    public static void main(String[] args) throws IOException
    {
        Employee[] staff = new Employee[3];

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

        // zapisuje wszystkie rekordy pracowników w pliku employee.dat
        try (PrintWriter out = new PrintWriter("employee.dat", "UTF-8"))
        {
            writeData(staff, out);
        }

        // wczytuje wszystkie rekordy do nowej tablicy
        try (Scanner in = new Scanner(
                new FileInputStream("employee.dat"), "UTF-8"))
        {
            Employee[] newStaff = readData(in);

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

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

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

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

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

/*
 * Zapisuje dane obiektu klasy Employee
 * do obiektu klasy PrintWriter
 * @param out obiekt klasy PrintWriter
 */
public static void writeEmployee(PrintWriter out, Employee e)
{
    GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(e.getHireDay());
    out.println(e.getName() + "|" + e.getSalary() + "|" + calendar.get(Calendar.
        YEAR) + "|"
        + (calendar.get(Calendar.MONTH) + 1) + "|" + calendar.get(Calendar.DAY_
        OF_MONTH));
}

/*
 * Wczytuje dane obiektu klasy Employee
 * @param in obiekt klasy Scanner
 */
public static Employee readEmployee(Scanner in)
{
    String line = in.nextLine();
    String[] tokens = line.split("\\|");
    String name = tokens[0];
    double salary = Double.parseDouble(tokens[1]);
    int year = Integer.parseInt(tokens[2]);
    int month = Integer.parseInt(tokens[3]);
    int day = Integer.parseInt(tokens[4]);
    return new Employee(name, salary, year, month, day);
}
```

---

## 1.2.4. Zbiory znaków

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

Zbiór znaków stanowi odwzorowanie pomiędzy dwubajtowymi kodami Unicode i sekwencjami bajtów stosowanymi w lokalnych kodowaniach znaków. Jednym z najpopularniejszych kodowań znaków jest ISO 8859-1, które koduje za pomocą jednego bajta pierwszych 256 znaków z zestawu Unicode. Coraz większe znaczenie zyskuje również ISO 8859-15, w którym zastąpiono część mniej przydatnych znaków kodu ISO 8859-1 akcentowanymi znakami języka francuskiego i fińskiego, a przede wszystkim zamiast znaku waluty międzynarodowej ☷ umieszczono symbol euro (€) o kodzie 0xA4. Innymi przykładami kodowań znaków są kodowania o zmiennej liczbie bajtów stosowane dla języków japońskiego i chińskiego.

Klasa Charset używa nazw zbiorów znaków zgodnie ze standardem określonym przez IANA Character Set Registry (<http://www.iana.org/assignments/character-sets>). Nazwy te różnią się nieco od nazw stosowanych w poprzednich wersjach. Na przykład „oficjalną” nazwę ISO 8859-1 jest teraz "ISO-8859-1" zamiast "ISO8859\_1" preferowaną w Java SE 1.3 i wcześniejszych wersjach.

Aby można było zachować zgodność z innymi konwencjami nazw, każdy zbiór znaków może mieć wiele synonimów. Na przykład zbiór ISO 8859-1 ma poniższe synonimy:

```
ISO8859-1
ISO_8859_1
ISO8859_1
ISO_8859-1
ISO_8859-1:1987
8859_1
latin1
11
cSISOLatin1
iso_ir-100
cp819
IBM819
IBM-819
819
```

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

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

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

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

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



Doskonały opis kodowania ISO 8859 znajdziesz na stronie <http://czyborra.com/charsets/iso8859.html>.

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

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

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

**Tabela 1.1.** Kodowania znaków wymagane na platformie Java

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

**Tabela 1.2.** Podstawowe kodowania znaków

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

**Tabela 1.3.** Rozszerzone kodowania znaków

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

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

Dysponując zbiorem znaków, możemy użyć go do konwersjiłańcuchów Unicode i sekwencji bajtów. Oto przykład kodowaniałańcucha Unicode:

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

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

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

#### **java.nio.charset.Charset 1.4**

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

#### **java.nio.ByteBuffer 1.4**

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

#### **java.nio.CharBuffer**

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

■ `String toString()`

zwraca łańcuch, który tworzą kody zarządzane przez ten bufor.

## 1.3. Odczyt i zapis danych binarnych

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

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

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



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

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

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

**Tabela 1.4.** Kodowanie UTF-8

Zakres znaków	Kodowanie
0...7F	0a <sub>6</sub> a <sub>5</sub> a <sub>4</sub> a <sub>3</sub> a <sub>2</sub> a <sub>1</sub> a <sub>0</sub>
80...7FF	110a <sub>10</sub> a <sub>9</sub> a <sub>8</sub> a <sub>7</sub> a <sub>6</sub> 10a <sub>5</sub> a <sub>4</sub> a <sub>3</sub> a <sub>2</sub> a <sub>1</sub> a <sub>0</sub>
800...FFFF	1110a <sub>15</sub> a <sub>14</sub> a <sub>13</sub> a <sub>12</sub> 10a <sub>11</sub> a <sub>10</sub> a <sub>9</sub> a <sub>8</sub> a <sub>7</sub> a <sub>6</sub> 10a <sub>5</sub> a <sub>4</sub> a <sub>3</sub> a <sub>2</sub> a <sub>1</sub> a <sub>0</sub>
10000...10FFFF	11110a <sub>20</sub> a <sub>19</sub> a <sub>18</sub> 10a <sub>17</sub> a <sub>16</sub> a <sub>15</sub> a <sub>14</sub> a <sub>13</sub> a <sub>12</sub> 10a <sub>11</sub> a <sub>10</sub> a <sub>9</sub> a <sub>8</sub> a <sub>7</sub> a <sub>6</sub> 10a <sub>5</sub> a <sub>4</sub> a <sub>3</sub> a <sub>2</sub> a <sub>1</sub> a <sub>0</sub>

**Tabela 1.5.** Kodowanie UTF-16

Zakres znaków	Kodowanie
0...FFFF	a <sub>15</sub> a <sub>14</sub> a <sub>13</sub> a <sub>12</sub> a <sub>11</sub> a <sub>10</sub> a <sub>9</sub> a <sub>8</sub> a <sub>7</sub> a <sub>6</sub> a <sub>5</sub> a <sub>4</sub> a <sub>3</sub> a <sub>2</sub> a <sub>1</sub> a <sub>0</sub>
10000...10FFFF	110110b <sub>19</sub> b <sub>18</sub> b <sub>17</sub> b <sub>16</sub> a <sub>15</sub> a <sub>14</sub> a <sub>13</sub> a <sub>12</sub> a <sub>11</sub> a <sub>10</sub> 110111a <sub>9</sub> a <sub>8</sub> a <sub>7</sub> a <sub>6</sub> a <sub>5</sub> a <sub>4</sub> a <sub>3</sub> a <sub>2</sub> a <sub>1</sub> a <sub>0</sub> gdzie b <sub>19</sub> b <sub>18</sub> b <sub>17</sub> b <sub>16</sub> =a <sub>20</sub> a <sub>19</sub> a <sub>18</sub> a <sub>17</sub> a <sub>16</sub> - 1



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

Aby odczytać dane, korzystamy z poniższych metod interfejsu DataInput:

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

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

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

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

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

#### API java.io.DataInput 1.0

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

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

#### API `java.io.DataOutput 1.0`

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

### 1.3.1. Strumienie plików o swobodnym dostępie

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

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

Otwarcie istniejącego pliku przy użyciu RandomAccessFile nie powoduje jego skasowania.

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

Metoda getFilePointer zwraca aktualne położenie wskaźnika pliku.

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

Prześledzmy 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. Założymy na przykład, że chcemy ustawić wskaźnik pliku na trzecim rekordzie. Musimy zatem wyznaczyć bajt, na którym należy ustawić ten wskaźnik, a następnie możemy już wczytać rekord.

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

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

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

Aby określić całkowitą liczbę bajtów w pliku, używamy metody length. Calkowitą 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 bardziej skomplikowana. Stworzymy zatem dwie metody pomocnicze pozwalające zapisywać i wczytywać łańcuchy o ustalonym rozmiarze.

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

```

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

```

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

```

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

```

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

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

```

DataIO.writeFixedString(name, NAME_SIZE, out);
out.writeDouble(salary);
GregorianCalendar calendar = new GregorianCalendar();
calendar.setTime(hireDay);
out.writeInt(calendar.get(Calendar.YEAR));
out.writeInt(calendar.get(Calendar.MONTH) + 1);
out.writeInt(calendar.get(Calendar.DAY_OF_MONTH));

```

Odczyt rekordu jest równie prosty.

```

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

```

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

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

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

---

**Listing 1.2.** randomAccess/RandomAccessTest.java

```
package randomAccess;

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

/**
 * @version 1.12 2012-05-30
 * @author Cay Horstmann
 */
public class RandomAccessTest
{
    public static void main(String[] args) throws IOException
    {
        Employee[] staff = new Employee[3];

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

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

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

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

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

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

```

        }

    }

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

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

    /**
     * Wczytuje dane pracownika
     * @param in obiekt typu DataInput
     * @return pracownik
     */
    public static Employee readData(DataInput in) throws IOException
    {
        String name = DataIO.readFixedString(Employee.NAME_SIZE, in);
        double salary = in.readDouble();
        int y = in.readInt();
        int m = in.readInt();
        int d = in.readInt();
        return new Employee(name, salary, y, m - 1, d);
    }
}

```

**API java.io.RandomAccessFile 1.0**

- RandomAccessFile(String file, String mode)

- RandomAccessFile(File file, String mode)

*Parametry:* file plik, który ma zostać otwarty.  
                  tryb "r" dla samego odczytu, "rw" dla odczytu i zapisu,  
                  "rws" dla odczytu i zapisu danych wraz  
                  z synchronicznym zapisem danych i metadanych  
                  dla każdej aktualizacji, "rwd" dla odczytu i zapisu  
                  danych wraz z synchronicznym zapisem tylko  
                  samych danych.

- long getFilePointer()

zwraca aktualne położenie wskaźnika pliku.

- void seek(long pos)  
zmienia położenie wskaźnika pliku, przesuwając go o pos bajtów od początku pliku.
- long length()  
zwraca długość pliku w bajtach.

## 1.4. 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. Metoda `read` klasy `ZipInputStream` zwraca wartość `-1`, gdy napotka koniec pozycji archiwum, a nie koniec całego pliku ZIP. Aby odczytać kolejną pozycję archiwum, musimy wtedy wywołać metodę `closeEntry`. Oto typowy kod wczytujący zawartość pliku ZIP:

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

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

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



Nie należy zamykać strumienia archiwum ZIP po wczytaniu pojedynczej pozycji lub przekazywać go metodzie, która go zamknie. W takim przypadku nie będziemy mogli odczytać kolejnych pozycji archiwum ZIP.

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 rozpoczęć zapis nowego pliku w archiwum, wywołujemy metodę `putNextEntry` klasy `ZipOutputStream`. Następnie wysyłamy dane do strumienia ZIP. Po zakończeniu zapisu pliku wywołujemy metodę `closeEntry`. Wymienione operacje powtarzamy dla wszystkich plików, które chcemy skompresować w archiwum. Oto schemat kodu:

```

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

```



Pliki JAR (omówione w rozdziale 10. 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 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 połaczenie sieciowe. Na przykład za każdym razem, gdy mechanizm ładowania klas jakiegoś apletu wczytuje plik JAR, tak naprawdę wczytuje i dekompresuje dane pochodzące z sieci.



W punkcie 1.6.7 „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.

#### **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().

#### **java.util.zip.ZipOutputStream 1.1**

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

- void closeEntry()  
zamyka aktualnie otwartą pozycję archiwum ZIP. Aby otworzyć następną pozycję, wywołujemy metodę putNextEntry.
- void setLevel(int level)  
określa domyślny stopień kompresji następnych pozycji archiwum o trybie DEFLATED. Domyślną wartością jest Deflater.DEFAULT\_COMPRESSION. Wyrzuca wyjątek IllegalArgumentException, jeżeli podany stopień jest nieprawidłowy.  
*Parametry:* level stopień kompresji, od 0 (NO\_COMPRESSION) do 9 (BEST\_COMPRESSION).
- void setMethod(int method)  
określa domyślną metodę kompresji dla danego ZipOutputStream dla wszystkich pozycji archiwum, dla których metoda kompresji nie została określona.  
*Parametry:* method metoda kompresji, DEFLATED lub STORED.

 **java.util.zip.ZipEntry 1.1**

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

**API java.util.ZipFile 1.1**

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

## 1.5. Strumienie obiektów i serializacja

Korzystanie z rekordów o stałej długości jest dobrym rozwiązańiem, 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 będąceinstancjami klas pochodnych, np. klasy `Manager`.

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

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

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

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

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

Aby z powrotem załadować obiekty, używamy strumienia ObjectInputStream:

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

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

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

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

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

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

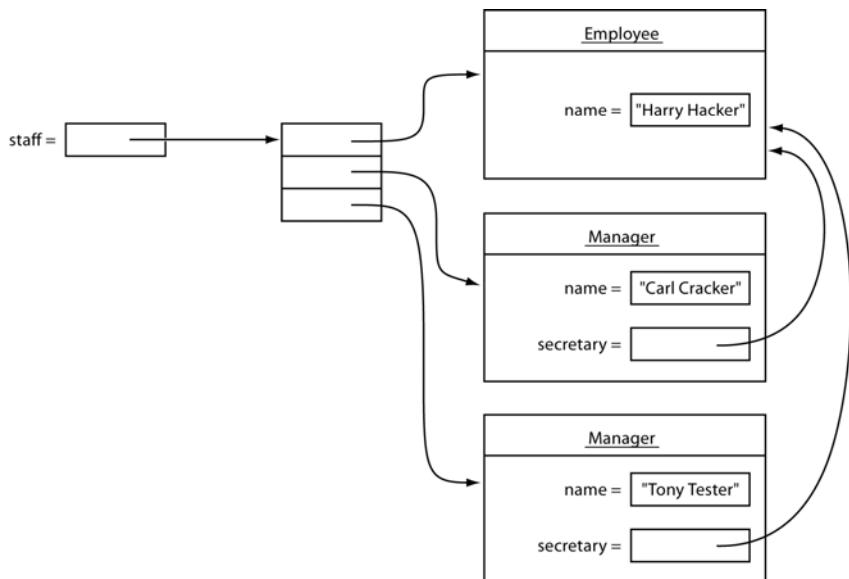
```
class Manager extends Employee
{
    .
    .
    private Employee secretary;
}
```

Każdy obiekt typu Manager przechowuje teraz referencję do obiektu klasy Employee opisującego asystenta, a nie osobną kopię tego obiektu. Oznacza to, że dwóch menedżerów może mieć tego samego asystenta, tak jak zostało to przedstawione na rysunku 1.5 i w poniższym kodzie:

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

**Rysunek 1.5.**

Dwóch menedżerów może mieć wspólnego asystenta



Teraz założymy, że zapisujemy dane pracowników na dysk. Oczywiście nie możemy zapisać i przywrócić adresów obiektów asystentów w pamięci, ponieważ po ponownym załadunku 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 1.6).
- 2 Jeśli referencja do obiektu została napotkana po raz pierwszy, obiekt zostaje zapisany w strumieniu.
- 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.

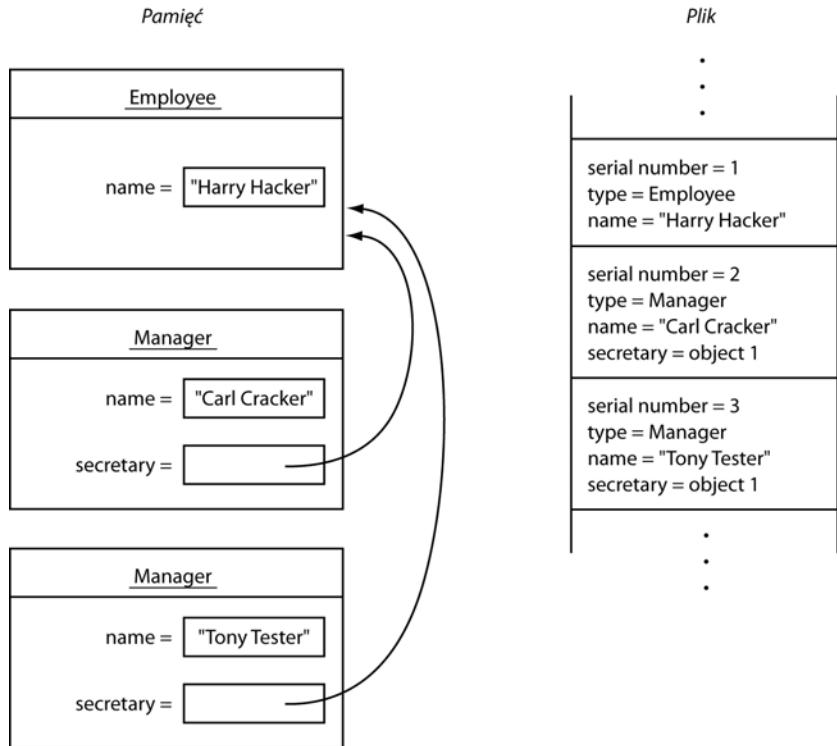
- 1 Gdy obiekt pojawia się w strumieniu po raz pierwszy, Java tworzy go, inicjuje danymi ze strumienia i zapamiętuje związek pomiędzy numerem i referencją do obiektu.
- 2 Gdy natrafi na znacznik „ten sam obiekt, co pod numerem seryjnym x”, sprawdza, gdzie znajduje się obiekt o danym numerze, i nadaje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



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

**Rysunek 1.6.**

Przykład serializacji obiektów



Listing 1.3 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 1.3.** *objectStream/ObjectStreamTest.java*

```
package objectStream;

import java.io.*;

/**
 * @version 1.10 17 Aug 1998
 * @author Cay Horstmann
 */
class ObjectStreamTest
{
    public static void main(String[] args) throws IOException, ClassNotFoundException
    {
        Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
        Manager carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
        carl.setSecretary(harry);
        Manager tony = new Manager("Tony Tester", 40000, 1990, 3, 15);
        tony.setSecretary(harry);

        Employee[] staff = new Employee[3];
    }
}
```

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

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

try (ObjectInputStream in = new ObjectInputStream(new
    ↪FileInputStream("employee.dat")))
{
    // wczytuje wszystkie rekordy do nowej tablicy

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

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

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

**API** **java.io.ObjectOutputStream 1.1**

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

**API** **java.io.ObjectInputStream 1.1**

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

## 1.5.1. Format pliku serializacji obiektów

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

Każdy plik zaczyna się dwubajtową „magiczną liczbą”:

AC ED

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

00 05

(w tym podroziale 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 stu procentową pewność, że jeżeli zachowana informacja zmieni się, zmianie ulegnie również jej „odcisk palca”. (aby dowiedzieć się więcej o SHA, zajrzyj np. do *Cryptography and Network Security: Principle and Practice*, autorstwa Williama Stallingsa, wydanej przez Prentice Hall). Jednakże Java korzysta jedynie z pierwszych ośmiu bajtów kodu SHA. Mimo to nadal jest bardzo prawdopodobne, że „odcisk” zmieni się, jeżeli ulegną zmianie pola składowe lub metody.

W chwili odczytu 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 1.5.4., „Wersje”.

Oto w jaki sposób przechowywany jest identyfikator klasy:

72

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

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

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

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

Każdy deskryptor pola składowego składa się z następujących elementów:

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

Kod typu może mieć jedną z następujących wartości:

- B byte
- C char
- D double
- F float

- I int
- J long
- L obiekt
- S short
- Z boolean
- [ tablica

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

Dla przykładu, pole salary klasy Employee zostanie zapisane jako:

D 00 06 salary

A oto kompletny opis klasy Employee:

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

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

71 numer seryjny (4 bajty)

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

Obiekt jest przechowywany w następującej postaci:

73 opis klasy dane obiektu

Dla przykładu, oto zapis obiektu klasy Employee:

40 E8 6A 00 00 00 00	Wartość pola salary — double
73	Wartość pola hireDay — nowy obiekt

71 00 7E 00 08	Istniejąca klasa java.util.Date
77 08 00 00 00 91 1B 4E B1 80 78	Zawartość zewnętrzna — szczegóły poniżej
74 00 0C Harry Hacker	Wartość pola name — String

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

Tablice są zapisywane w następujący sposób:

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

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

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

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

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

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

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

71 00 7E 00 08

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

Referencja null jest zapisywana jako

70

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

AC ED 00 05	Nagłówek pliku
75	Tablica staff (nr #1)
72 00 0C [LEmployee:	Nowa klasa, długość łańcucha, nazwa klasy Employee[] (nr #0)
FC BF 36 11 C5 91 11 C7 02	„Odcisk” oraz flagi
00 00	Liczba pól składowych
78	Znacznik końca
70	Brak klasy bazowej
00 00 00 03	Liczba elementów tablicy
73	staff[0] — nowy obiekt (nr #7)
72 00 08 Manager	Nowa klasa, długość łańcucha, nazwa klasy (nr #2)
36 06 AE 13 63 8F 59 B7 02	„Odcisk” oraz flagi
00 01	Liczba pól składowych
L 00 08 secretary	Typ i nazwa pola składowego
74 00 0B LEmployee:	Nazwa klasy pola składowego — String (nr #3)
78	Znacznik końca
72 00 09 Employee	Nadklasa — nowa klasa, długość łańcucha, nazwa klasy (nr #4)
E6 D2 86 7D AE AC 18 1B 02	„Odcisk” oraz flagi
00 03	Liczba pól składowych
D 00 06 salary	Typ i nazwa pola składowego
L 00 11 hireDay	Typ i nazwa pola składowego
74 00 10 Ljava/util/Date:	Nazwa klasy pola składowego — String (nr #5)
L 00 08 name	Typ i nazwa pola składowego
74 00 12 Ljava/lang/String;	Nazwa klasy pola składowego — String (nr #6)

78	Znacznik końca
70	Brak klasy bazowej
40 F3 88 00 00 00 00 00	Wartość pola salary — double
73	Wartość pola hireDay — nowy obiekt (nr #9)
72 00 0E java.util.Date	Nowa klasa, długość łańcucha, nazwa klasy (nr #8)
68 6A 81 01 4B 59 74 19 03	„Odcisk” oraz flagi
00 00	Brak zmiennych składowych
78	Znacznik końca
70	Brak klasy bazowej
77 08	Zawartość zewnętrzna, liczba bajtów
00 00 00 83 E9 39 E0 00	Data
78	Znacznik końca
74 00 0C Carl Cracker	Wartość pola name — String (nr #10)
73	Wartość pola secretary — nowy obiekt (nr #11)
71 00 7E 00 04	Istniejąca klasa (użyj nr #4)
40 E8 6A 00 00 00 00 00	Wartość pola pensja — double
73	Wartość pola dzienZatrudnienia — nowy obiekt (nr #12)
71 00 7E 00 08	Istniejąca klasa (użyj nr #8)
77 08	Zawartość zewnętrzna, liczba bajtów
00 00 00 91 1B 4E B1 80	Data
78	Znacznik końca
74 00 0C Harry Hacker	Wartość pola name — String (nr #13)
71 00 7E 00 0B	staff[1] — istniejący obiekt (użyj nr #11)
73	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 strumień obiektów posiada szczegółowy opis wszystkich obiektów, które zawiera, co pozwala mu odtwarzać te obiekty, jak i tablice obiektów.

Powinniśmy zapamiętać, że:

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

## 1.5.2. 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 oznać je słowem kluczowym `transient`. Słowem tym należy również oznać 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`, odwrócićmy 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 upłyнуły 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 przeciwnieństwie do omówionych wcześniej metod `readObject` i `writeObject`, te metody są całkowicie odpowiedzialne za zapisanie i odczytanie obiektu, *łącznie z danymi klasą bazową*. Mechanizm serializacji zapisuje jedynie klasę obiektu w strumieniu. Odtwarzając obiekt implementujący interfejs `Externalizable`, strumień obiektów wywołuje domyślny konstruktor, a następnie metodę `readExternal`. Oto jak możemy zaimplementować te metody w klasie `Employee`:

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

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



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.

### 1.5.3. Serializacja singletonów i wyliczeń

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

Jeśli używamy w programach konstrukcji enum wprowadzonej w Java SE 5.0, to nie musimy przejmować się serializacją — wszystko będzie działać poprawnie. Założymy 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.writeObject();
out.close();
ObjectInputStream in = . . .;
Orientation saved = (Orientation) in.read();
```

Okaże się, że porównanie

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

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

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

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

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

## 1.5.4. Wersje

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

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

```
serialver Employee
```

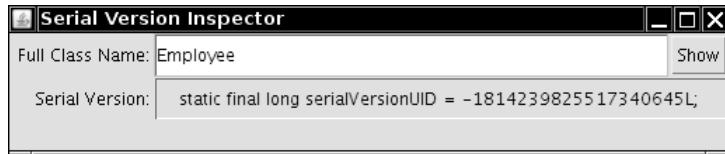
otrzymujemy:

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

Jeżeli uruchomimy serialver z opcją -show, program wyświetli okno dialogowe (rysunek 1.7).

**Rysunek 1.7.**

Wersja graficzna programu serialver



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

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

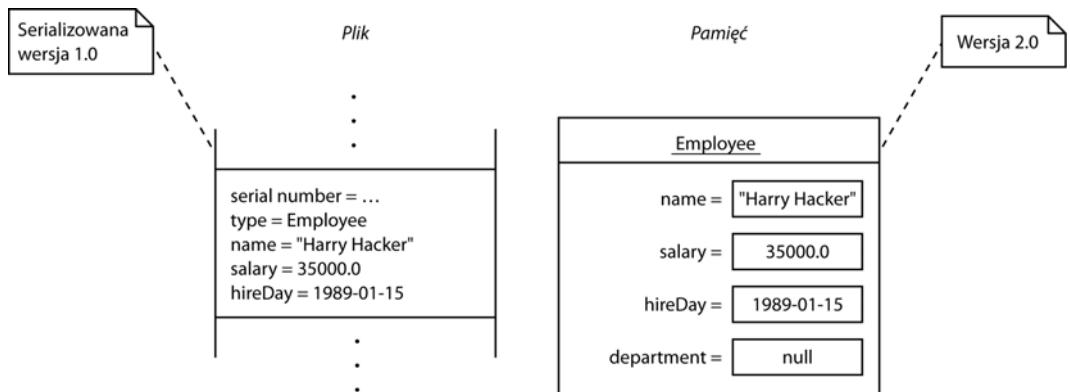
Klasa posiadająca statyczne pole składowe o nazwie serialVersionUID nie obliczy własnego „odcisku palca”, ale skorzysta z już istniejącej wartości.

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

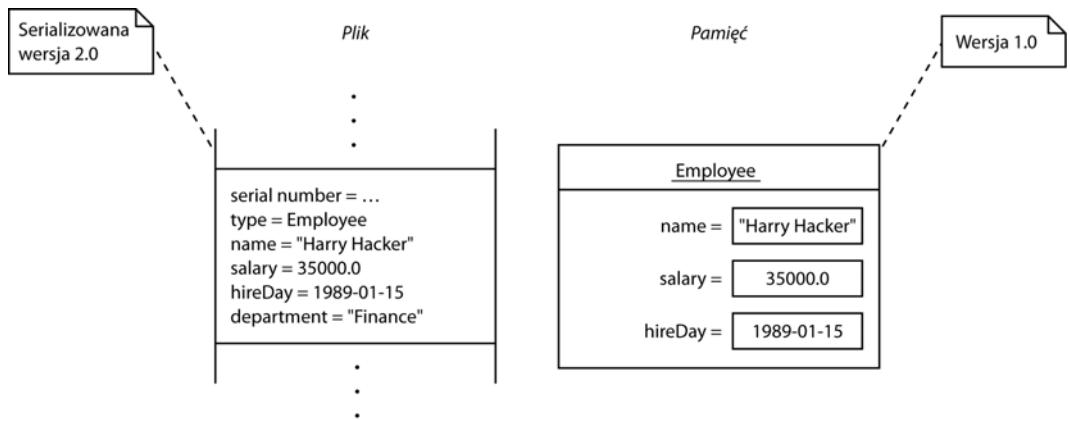
Jeśli zmienią się tylko metody danej klasy, sposób odczytu danych nie ulegnie zmianie. Jednakże jeżeli zmieni się pole składowe, możemy mieć pewne problemy. Dla przykładu, stary obiekt może posiadać więcej lub mniej pól składowych niż aktualny, albo też typy danych mogą się różnić. W takim wypadku strumień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

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

Oto przykład. Założmy, że zapisaliśmy na dysku pewną liczbę obiektów klasy Employee, używając przy tym oryginalnej (1.0) wersji klasy. Teraz wprowadzamy nową wersję 2.0 klasy Employee, dodając do niej pole składowe department. Rysunek 1.8 przedstawia, co się dzieje, gdy obiekt wersji 1.0 jest wczytywany przez program korzystający z obiektów 2.0. Pole department otrzymuje wartość null. Rysunek 1.9 ilustruje odwrotną sytuację — program korzystający z obiektów 1.0 wczytuje obiekt 2.0. Dodatkowe pole department jest ignorowane.



**Rysunek 1.8.** Odczytywanie obiektu o mniejszej liczbie pól



**Rysunek 1.9.** Odczytywanie obiektu o większej liczbie pól

Czy ten proces jest bezpieczny? To zależy. Opuszczenie 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 konstruktorek niezerowe wartości, tak więc metody mogą być nieprzygotowane do obsługiwanego wartości `null`. Od projektanta klasy zależy, czy zaimplementuje w metodzie `readObject` dodatkowy kod poprawiający wyniki wczytywania różnych wersji danych, czy też dołączy do metod obsługi wartości `null`.

## 1.5.5. Serializacja w roli klonowania

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

Kod z listingu 1.4 udowadnia, że aby otrzymać metodę `clone` „za darmo”, wystarczy rozszerzyć klasę `Serializable`.

**Listing 1.4.** *serialClone/SerialCloneTest.java*

```
package serialClone;

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

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

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

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

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

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

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

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

```
        }
    }
}

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

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

    public String getName()
    {
        return name;
    }

    public double getSalary()
    {
        return salary;
    }

    public Date getHireDay()
    {
        return hireDay;
    }

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

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

---

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

# 1.6. Zarządzanie plikami

Potrafimy już zapisywać i wczytywać dane z pliku. Jednakże obsługa plików to coś więcej niż tylko operacje zapisu i odczytu. Klasy Path i 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 zajmują się zawartością plików, natomiast klasy omawiane w tym podrozdziale związane są z organizacją przechowywania plików na dysku.

Klasy Path i Files wprowadzono w wersji Java SE 7. Posługiwaniem 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.

## 1.6.1. 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 względową. W przypadku ścieżki bezwzględnej założyliśmy, że komputer wykorzystuje system plików zorganizowany na wzór systemu UNIX.

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

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

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

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



Ścieżka nie musi odpowiadać istniejącemu plikowi. Stanowi raczej abstrakcyjną sekwencję nazw. W następnym podrozdziale pokażemy, że chcąc utworzyć plik, najpierw konstruujemy ścieżkę, a dopiero potem wywołujemy metodę tworzącą plik 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.relativize(r) daje w wyniku ścieżkę q, gdy r jest wynikiem rozwiązywania p względem q. Na przykład wynikiem relatywizacji ścieżki `/home/cay` względem `/home/fred/myprog` jest ścieżka `./fred/myapp`. W tym przykładzie zakładamy, że .. oznacza katalog nadzędny 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/cay/./fred./myprog` będzie ścieżka `/home/fred/myprog`.

Metoda toAbsolutePath daje w wyniku bezwzględną ścieżkę dla ścieżki podanej, rozpoczynającą się komponentem korzenia.

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

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



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

#### `java.nio.file.Paths`

- 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 wzglną, 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 nadrzędna 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.

## 1.6.2. Odczyt i zapis plików

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

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

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

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

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

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

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

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

Aby dopisać łańcuch do pliku, zastosujemy poniższe wywołanie.

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

Kolekcję wierszy możemy zapisać w następujący sposób:

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

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

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

### 1.6.3. 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);
```

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.

#### java.nio.file.Files ▾

- static Path copy(Path from, Path to, CopyOption... options)
- staticPath move(Path from, Path to, CopyOption... options)
- kopiują lub przenoszą from do lokalizacji to. Zwracają to.
- 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.

### 1.6.4. Tworzenie plików i katalogów

Aby utworzyć nowy katalog, wywołamy:

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

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

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

Pusty plik tworzymy, wywołując:

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

Powyzsza 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 uprawnień. Ponieważ jednak szczegóły zależą od wykorzystywanego systemu plików, nie będziemy ich tutaj omawiać.

#### java.nio.file.Files ▶

- `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 nadziednym. Zwracają ścieżkę dostępu do utworzonego pliku lub katalogu.

## 1.6.5. 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:

- czasы: 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.

### `java.nio.file.Files`

- 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.

 **java.nio.file.attribute.BasicFileAttributes** ↗

- FileTime creationTime()
- FileTime lastAccessTime()
- FileTime lastModifiedTime()
- boolean isRegularFile()
- boolean isDirectory()
- boolean isSymbolicLink()
- long size()
- Object fileKey()
  - zwracającą żądanego atrybut.

## 1.6.6. Przeglądanie plików w katalogu

Klasa `File` miała metodę zwracającą tablicę wszystkich plików w wybranym katalogu. Jak łatwo się domyślić, rodziło to spore problemy w przypadku bardzo dużej liczby plików w katalogu. Z tego powodu klasa `Files` dysponuje metodą udostępniającą iterator. Poniżej przedstawiamy sposób jego użycia:

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

Instrukcja `try` zarządzająca zasobami gwarantuje, że strumień katalogu zostanie poprawnie zamknięty.

Przeglądanie plików nie odbywa się w żadnym z góry określonym porządku.

Przeglądane pliki możemy filtrować za pomocą wzorca:

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

Dostępne wzorce zostały przedstawione w tabeli 1.6.

**Tabela 1.6.** Wzorce filtrowania plików

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



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

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

- gdy napotkany zostaje plik lub katalog: `FileVisitResult visitFile(T path, BasicFileAttributes attrs);`
- zanim zostanie przetworzony katalog: `FileVisitResult preVisitDirectory(T dir, IOException ex);`
- po przetworzeniu katalogu: `FileVisitResult postVisitDirectory(T dir, IOException ex);`
- gdy podczas próby odwiedzenia pliku lub katalogu wystąpi błąd, na przykład na skutek próby otwarcia katalogu bez wystarczających uprawnień: `FileVisitResult visitFailed(T path, IOException ex).`

W każdym z tych przypadków możemy określić, czy chcemy:

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

Jeśli którakolwiek z metod wyrzuci wyjątek, przeglądanie zostanie zakończone, a metoda walkFileTree wyrzuci ten wyjątek.



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

Klasa SimpleFileVisitor implementuje interfejs FileVisitor. Metody — oprócz visitFile →Failed — 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(dir, new SimpleFileVisitor<Path>()
{
    public FileVisitResult visitFile(Path path, BasicFileAttributes attrs) throws
        IOException
    {
        if (attrs.isDirectory())
            System.out.println(path);
        return FileVisitResult.CONTINUE;
    }
    public FileVisitResult visitFileFailed(Path path, IOException exc) throws IOException
    {
        return FileVisitResult.CONTINUE;
    }
});
```

Zauważmy, że w tym przypadku konieczne jest zastąpienie metody visitFileFailed własną wersją. W przeciwnym razie przeglądanie zakończy się w momencie, gdy napotka katalog, którego nie mamy prawa otworzyć.

Zwróćmy również uwagę, że atrybuty ścieżki są przekazywane jako parametr. Metoda walkFileTree musiała już wcześniej wykonać odpowiednie wywołanie systemowe, aby rozróżnić pliki i katalogi. Dzięki temu sami nie musimy wykonywać kolejnego wywołania.

Pozostałe metody interfejsu FileVisitor przydają się, gdy odwiedzając lub opuszczając katalog, musimy wykonać pewne działania. Na przykład kopując drzewo katalogów, musimy najpierw skopiować bieżący katalog, zanim umieścimy w nim kopie plików. Gdy usuwamy drzewo katalogów, bieżący katalog usuwamy po usunięciu z niego wszystkich plików.

#### java.nio.file.Files ▶

- DirectoryStream<Path> newDirectoryStream(Path path)
  - 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.

- Path walkFileTree(Path start, FileVisitor<? super Path> visitor)  
odwiedza wszystkie elementy podrzędne danej ścieżki, stosując do nich obiekt visitor.

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

- FileVisitResult visitFile(T path, BasicFileAttributes attrs)  
wywoływana, gdy odwiedzany jest plik lub katalog, zwraca CONTINUE, SKIP\_SUBTREE, SKIP\_SIBLINGS lub TERMINATE. Domyślna implementacja nie podejmuje żadnych działań, tym samym powodując kontynuację przeglądania.
- FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)
- FileVisitResult postVisitDirectory(T dir, BasicFileAttributes attrs)  
wywoływane przed i po wizycie w katalogu. Domyślna implementacja nie podejmuje żadnych działań, tym samym powodując kontynuację przeglądania.
- 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.

## 1.6.7. 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 podrozdziale 1.4., „Archiwa ZIP”, wymagająca użycia całego zestawu odpowiednich klas.

#### **API** java.nio.file.FileSystem 7

- static FileSystem newFileSystem(Path path, ClassLoader loader)

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

#### **API** java.nio.file.FileSystem 7

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

tworzy ścieżkę, łącząc podane łańcuchy.

## 1.7. Mapowanie plików w pamięci

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

Na końcu tego podrozdziału zamieściliśmy program, który oblicza sumę kontrolną CRC32 dla pliku, używając standardowych operacji wejścia i wyjścia, a także pliku mapowanego w pamięci. Na jednej i tej samej maszynie otrzymaliśmy wyniki jego działania przedstawione w tabeli 1.7 dla pliku *rt.jar* (37 MB) znajdującego się w katalogu *jre/lib* pakietu JDK.

**Tabela 1.7.** 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 zauważyc, na naszym komputerze mapowanie pliku dało nieco lepszy wynik niż zastosowanie buforowanego wejścia i znacznie lepszy niż użycie klasy RandomAccessFile.

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

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

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

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

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

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

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

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

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

Alternatywnie możemy również wykorzystać dostęp swobodny:

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

Możemy także czytać tablice bajtów, stosując metody:

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

Dostępne są również poniższe metody:

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

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

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

Aby poznać bieżący sposób uporządkowania bajtów w buforze, wywołujemy:

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



Ta para metod nie stosuje konwencji nazw set/get.

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

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

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

Program przedstawiony na listingu 1.5 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 1.5.** *memoryMap/MemoryMapTest.java*

```
package memoryMap;

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

```
import java.nio.file.*;
import java.util.zip.*;

/**
 * Program obliczający sumę kontrolną CRC pliku.
 * Uruchamianie: java memoryMap.MemoryMapTest nazwapliku
 * @version 1.01 2012-05-30
 * @author Cay Horstmann
 */
public class MemoryMapTest
{
    public static long checksumInputStream(Path filename) throws IOException
    {
        try (InputStream in = Files.newInputStream(filename))
        {
            CRC32 crc = new CRC32();

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

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

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

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

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

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

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

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

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

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

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

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



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

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

Program uruchamiamy w następujący sposób:

```
java memoryMap.MemoryMapTest nazwapliku
```

**API java.io.FileInputStream 1.0**

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

zwraca kanał dostępu do strumienia.

**API java.io.FileOutputStream 1.0**

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

zwraca kanał dostępu do strumienia.

**API java.io.RandomAccessFile 1.0**

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

zwraca kanał dostępu do pliku.

**API java.nio.channels.FileChannel 1.4**

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

otwiera kanał dla pliku o podanej ścieżce dostępu. Domyślnie kanał zostaje otwarty dla odczytu.

*Parametry:* path ścieżka dostępu do pliku

options wartości WRITE, APPEND, TRUNCATE\_EXISTING, CREATE należące do typu wyliczeniowego StandardOpenOption

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

*Parametry:* mode jedna ze stałych READ\_ONLY, READ\_WRITE lub PRIVATE zdefiniowanych w klasie FileChannel.MapMode

position początek mapowanego fragmentu  
size rozmiar mapowanego fragmentu

**API java.nio.Buffer 1.4**

- `boolean hasRemaining()`

zwraca wartość true, jeśli bieżąca pozycja bufora nie osiągnęła jeszcze jego końca.

- `int limit()`

zwraca pozycję końcową bufora; jest to pierwsza pozycja, na której nie są już dostępne kolejne dane bufora.

**API java.nio.ByteBuffer 1.4**

- `byte get()`

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

- `byte get(int index)`  
pobiera bajt o podanym indeksie.
- `ByteBuffer put(byte b)`  
umieszcza bajt na bieżącej pozycji bufora i przesuwa pozycję do kolejnego bajta.
- `ByteBuffer put(int index, byte b)`  
umieszcza bajt na podanej pozycji bufora. Zwraca referencję do bufora.
- `ByteBuffer get(byte[] destination)`
- `ByteBuffer get(byte[] destination, int offset, int length)`  
wypełnia tablicę bajtów lub jej zakres bajtami z bufora i przesuwa pozycję bufora o liczbę wczytanych bajtów. Jeśli bufor nie zawiera wystarczającej liczby bajtów, to nie są one w ogóle wczytywane i zostaje wyrzucony wyjątek `BufferUnderflowException`. Zwracającą referencję do bufora.

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

- `ByteBuffer put(byte[] source)`
- `ByteBuffer put(byte[] source, int offset, int length)`  
umieszcza w buforze wszystkie bajty z tablicy lub jej zakresu i przesuwa pozycję bufora o liczbę umieszczonej bajtów. Jeśli w buforze nie ma wystarczającego miejsca, to nie są zapisywane żadne bajty i zostaje wyrzucony wyjątek `BufferOverflowException`. Zwracającą referencję do bufora.

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

- `Xxx getXxx()`
- `Xxx getXxx(int index)`
- `ByteBuffer putXxx(xxx value)`
- `ByteBuffer putXxx(int index, xxx value)`

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

- `ByteBuffer order(ByteOrder order)`
- `ByteOrder order()`  
określa lub pobiera uporządkowanie bajtów w buforze. Wartością parametru `order` jest stała `BIG_ENDIAN` lub `LITTLE_ENDIAN` zdefiniowana w klasie `ByteOrder`.
- `static ByteBuffer allocate(int capacity)`  
tworzy bufor o podanej pojemności.

- static ByteBuffer wrap(byte[] values)  
tworzy bufor w oparciu o podaną tablicę.
- CharBuffer asCharBuffer()  
tworzy nowy bufor na podstawie istniejącego, z którym ma wspólną zawartość, ale jednocześnie ma własną pozycję, ograniczenie i znacznik.

 **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.

## 1.7.1. Struktura bufora danych

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

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



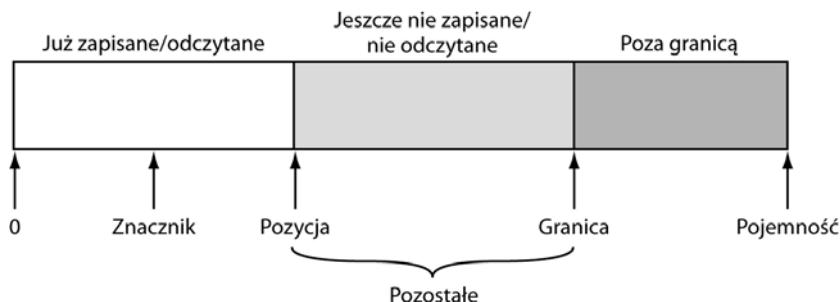
Klasa StringBuffer nie jest związana z omawianą tutaj hierarchią klas.

W praktyce najczęściej używane są klasy ByteBuffer i CharBuffer. Na rysunku 1.10 pokazaliśmy, że bufor jest scharakteryzowany przez:

- *pojemność*, która nigdy nie ulega zmianie;
- *pozycję* wskazującą następną wartość do odczytu lub zapisu;

**Rysunek 1.10.**

Bufor



- *granice*, poza którą odczyt i zapis nie mają sensu;
- opcjonalny *znacznik* dla powtarzających się operacji odczytu lub zapisu.

Wymienione wartości spełniają następujący warunek:

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

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

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

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

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

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

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

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

#### **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 granice 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)`  
zwracając i określając pozycję bufora.
- `int capacity()`  
zwraca pojemność bufora.

## 1.7.2. 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 żądalismy 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.

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

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

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

- `void close() 1.7`  
zwalnia blokadę.

## 1.8. Wyrażenia regularne

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

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

Oto prosty przykład. Z wyrażeniem regularnym

`[Jj]ava.+`

może zostać uzgodniony dowolny łańcuch znaków następującej postaci:

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

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

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

- Przez *klasę znaków* rozumiemy zbiór alternatywnych znaków ujęty w nawiasy kwadratowe, na przykład [Jj], [0-9], [A-Za-z] czy [^0-9]. Znak - oznacza zakres (czyli wszystkie znaki, których kody Unicode leżą w podanych granicach), a znak ^ oznacza dopełnienie (wszystkie znaki oprócz podanych).
- 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 1.8 i 1.9.
- Większość znaków oznacza samą siebie, tak jak znaki ava w poprzednim przykładzie.
- Symbol . oznacza dowolny znak (z wyjątkiem, być może, znaków końca wiersza, co zależy od stanu odpowiedniego znacznika).

**Tabela 1.8.** Składnia wyrażeń regularnych

Składnia	Objaśnienie
<b>Znaki</b>	
c	Znak c.
\unnnn, \xnnn, \0n, \0nn, \0nnn	Znak o kodzie, którego wartość została podana w notacji szesnastkowej lub ósemkowej.
\t, \n, \r, \f, \a, \e	Znaki sterujące tabulatora, nowego wiersza, powrotem karetki, końca strony, alertu i sekwencji sterującej.
\cc	Znak sterujący odpowiadający znakowi c.
<b>Klasy znaków</b>	
[C <sub>1</sub> C <sub>2</sub> . . .]	Dowolny ze znaków reprezentowanych przez C <sub>1</sub> C <sub>2</sub> . . ., gdzie C <sub>i</sub> jest znakiem, zakresem znaków (c <sub>1</sub> -c <sub>2</sub> ) lub klasą znaków.
[^ . . .]	Dopełnienie klasy znaków.
[. . . && . . .]	Część wspólna (przecięcie) dwóch klas znaków.

**Tabela 1.8.** Składnia wyrażeń regularnych — ciąg dalszy

Składnia	Objaśnienie
<b>Wstępnie zdefiniowane klasy znaków</b>	
.	Dowolny znak oprócz kończącego wiersz (lub dowolny znak, jeśli znacznik DOTALL został ustawiony).
\d	Cyfra [0-9].
\D	Znak, który nie jest cyfrą [^0-9].
\s	Znak odstępu [ \t\n\r\f\x0B].
\S	Znak, który nie jest odstępem.
\w	Znak słowa [a-zA-Z0-9_].
\W	Znak inny niż znak słowa.
\p{nazwa}	Klasa znaków o podanej nazwie (patrz tabela 1.9).
\P{nazwa}	Dopełnienie klasy znaków o podanej nazwie.
<b>Granice dopasowania</b>	
^ \$	Początek, koniec wejścia (lub początek, koniec wiersza w trybie wielowierszowym).
\b	Granica słowa.
\B	Granica inna niż słowa.
\A	Początek wejścia.
\z	Koniec wejścia.
\Z	Koniec wejścia oprócz ostatniego zakończenia wiersza.
\G	Koniec poprzedniego dopasowania.
<b>Kwantyfikatory</b>	
X?	Opcjonalnie X.
X*	X, 0 lub więcej razy.
X+	X, 1 lub więcej razy.
X{n} X{n,} X{n,m}	X n razy, co najmniej n razy, pomiędzy n i m razy.
<b>Przyrostki kwantyfikatora</b>	
?	Powoduje dopasowanie najmniejszej liczby wystąpień.
+	Powoduje dopasowanie największej liczby wystąpień, nawet kosztem ogólnego powodzenia dopasowania.
<b>Operacje na zbiorach</b>	
XY	Dowolny łańcuch z X, po którym następuje dowolny łańcuch z Y.
X Y	Dowolny łańcuch z X lub Y.

**Tabela 1.8.** Składnia wyrażeń regularnych — ciąg dalszy

Składnia	Objaśnienie
<b>Grupowanie</b>	
(X)	Grupa.
\n	Dopasowanie <i>n</i> -tej grupy.
<b>Sekwencje sterujące</b>	
\c	Znak <i>c</i> (nie może być znakiem alfabetu).
\Q...\\E	Cytat... dosłownie.
(?...)	Specjalna konstrukcja — patrz opis klasy Pattern.

**Tabela 1.9.** Wstępnie zdefiniowane nazwy klas znaków

Nazwa klasy znaków	Objaśnienie
Lower	Małe litery ASCII [a-z]
Upper	Duże litery ASCII [A-Z]
Alpha	Litery alfabetu ASCII [A-Za-z]
Digit	Cyfry ASCII [0-9]
Alnum	Litery alfabetu bądź cyfry ASCII [A-Za-z0-9]
Xdigit	Cyfry szesnastkowe [0-9A-Fa-f]
Print lub Graph	Znaki ASCII posiadające reprezentację graficzną (na wydruku) [\x21-\x7E]
Punct	Znaki, które nie należą do znaków alfanumerycznych, bądź cyfry [\p{Print}&&!P{Alnum}]
ASCII	Wszystkie znaki ASCII [\x00-\x7F]
Cntrl	Znaki sterujące ASCII [\x00-\x1F]
Blank	Spacja lub tabulacja [\t]
Space	Odstęp [\t\n\r\f\0x0B]
javaLowerCase	Mała litera, zgodnie z wynikiem metody Character.isLowerCase()
javaUpperCase	Duża litera, zgodnie z wynikiem metody Character.isUpperCase()
javaWhitespace	Odstęp, zgodnie z wynikiem metody Character.isWhiteSpace()
javaMirrored	Ekwivalent wyniku metody Character.isMirrored()
InBloku	Blok jest nazwą bloku znaków Unicode z usuniętymi spacjami, na przykład BasicLatin lub Mongolian.
IsSkrypt	Skrypt jest nazwą skryptu Unicode, na przykład Common, z którego usunięto odstęp.
Kategoria lub IsKategoria	Kategoria jest nazwą kategorii znaków Unicode, na przykład L (litera) czy Sc (symbol waluty).
IsWłaściwość	Właściwość jest jedną z wartości Alphabetic, Ideographic, Letter, Lowercase, Uppercase, Titlecase, Punctuation, Control, White_Space, Digit, Hex_Digit, Noncharacter_Code_Point, Assigned.

- \ 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).
  - Domyslnie kwantyfikator dopasowuje największą możliwą liczbę wystąpień, która gwarantuje ogólne powodzenie dopasowania. Zachowanie to możemy zmodyfikować za pomocą przyrostka ? (dopasowanie najmniejszej liczby wystąpień) i przyrostka + (dopasowanie największej liczby wystąpień, nawet jeśli nie gwarantuje ono ogólnego powodzenia dopasowania).
- Na przykład łańcuch cab może zostać dopasowany do wyrażenia [a-z]\*ab, ale nie do [a-z]\*+ab. W pierwszym przypadku wyrażenie [a-z]\* dopasuje jedynie znak c, wobec czego znaki ab zostaną dopasowane do reszty wzorca. Jednak wyrażenie [a-z]\*+ dopasuje znaki cab, wobec czego reszta wzorca pozostanie bez dopasowania.
- *Grupy* pozwalają definiować podwyrażenia. Grupy ujmujemy w znaki nawiasów ( ); na przykład ([+-]?)([0-9]+). Możemy następnie zażądać dopasowania do wszystkich grup lub do wybranej grupy, do której odwołujemy się przez \n, gdzie n jest numerem grupy (numeracja rozpoczyna się od \1).

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

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

Niestety, składnia wyrażeń regularnych nie jest całkowicie ustandaryzowana. Istnieje zgodność w zakresie podstawowych konstrukcji, ale diabeł tkwi w szczegółach. Klasy języka Java związane z przetwarzaniem wyrażeń regularnych używają składni podobnej do zastosowanej w języku Perl. Wszystkie konstrukcje tej składni zostały przedstawione w tabeli 1.8. Więcej informacji na temat składni wyrażeń regularnych znajdziesz w dokumentacji klasy Pattern lub książce *Wyrażenia regularne* autorstwa J.E.F. Friedla (Wydawnictwo Helion, 2001).

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

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

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

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

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

Obsługiwanych jest sześć następujących znaczników:

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

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

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

zwracają indeks początkowy i końcowy podanej grupy.

Dopasowany łańcuch możemy pobrać, wywołując

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

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

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

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

dla danych

```
11:59am
```

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

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

Program przedstawiony na listingu 1.6 umożliwia wprowadzenie wzorca, a następnie łańcucha, którego dopasowanie zostanie sprawdzone. Jeśli łańcuch pasuje do wzorca zawierającego grupy, to program wyświetla granice grup w postaci nawiasów, na przykład:

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

**Listing 1.6.** regex/RegexTest.java

```
package regex;

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

/**
 * Program testujący zgodność z wyrażeniem regularnym.
 * Wprowadź wzorzec i dopasowywany łańcuch.
 * Jeśli wzorzec zawiera grupy, program
 * wyświetli ich granice po użyciu lańcucha ze wzorcem.
 * @version 1.02 2012-06-02
 * @author Cay Horstmann
 */
public class RegexTest
{
    public static void main(String[] args) throws PatternSyntaxException
    {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter pattern: ");
        String patternString = in.nextLine();

        Pattern pattern = Pattern.compile(patternString);

        while (true)
        {
            System.out.println("Enter string to match: ");
            String input = in.nextLine();
            if (input == null || input.equals("")) return;
            Matcher matcher = pattern.matcher(input);
            if (matcher.matches())
            {
                System.out.println("Match");
                int g = matcher.groupCount();
                if (g > 0)
                {
                    for (int i = 0; i < input.length(); i++)
                    {
                        // Wyświetla puste grupy
                        for (int j = 1; j <= g; j++)
                            if (i == matcher.start(j) && i == matcher.end(j))
                                System.out.print("(");
                        // Wyświetla (dla grup rozpoczęjących się tutaj
                        for (int j = 1; j <= g; j++)
                            if (i == matcher.start(j) && i != matcher.end(j))
                                System.out.print(')');
                        System.out.print(input.charAt(i));
                        // Wyświetla (dla grup kończących się tutaj
                        for (int j = 1; j <= g; j++)
                            if (i + 1 != matcher.start(j) && i + 1 == matcher.end(j))
                                System.out.print(')');
                    }
                    System.out.println();
                }
            }
            else
                System.out.println("No match");
        }
    }
}
```

```

        }
    }
}
```

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

```

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

    ...
}
```

Program przedstawiony na listingu 1.7 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://www.horstmann.com
```

#### **Listing 1.7.** *match/HrefMatch.java*

```

package match;

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

/**
 * Program wyświetlający wszystkie adresy URL na stronie WWW
 * poprzez dopasowanie wyrażenia regularnego
 * opisującego znacznik <a href=...> języka HTML.
 * Uruchamianie: java match.HrefMatch adresURL
 * @version 1.01 2004-06-04
 * @author Cay Horstmann
 */
public class HrefMatch
{
    public static void main(String[] args)
    {
        try
        {
            // pobiera URL z wiersza poleceń lub używa domyślnego
            String urlString;
            if (args.length > 0) urlString = args[0];
            else urlString = "http://java.sun.com";

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

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

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

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

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

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

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

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

Metoda `replaceFirst` zastępuje jedynie pierwsze wystąpienie wzorca.

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

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

#### java.util.regex.Pattern 1.4

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

kompiluje łańcuch wyrażenia regularnego, tworząc obiekt wzorca przyspieszający przetwarzanie.

*Parametry:*

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

■ `Matcher matcher(CharSequence input)`

tworzy obiekt pozwalający odnajdywać dopasowania do wzorca w łańcuchu wejściowym.

■ `String[] split(CharSequence input)`

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

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

*Parametry:*

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

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

■ `boolean matches()`

zwraca `true`, jeśli łańcuch wejściowy pasuje do wzorca.

■ `boolean lookingAt()`

zwraca `true`, jeśli początek łańcucha wejściowego pasuje do wzorca.

■ `boolean find()`

■ `boolean find(int start)`

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

*Parametry:* `start` indeks, od którego należy rozpoczęć poszukiwanie

■ `int start()`

■ `int end()`

zwraca pozycję początkową dopasowania lub następną pozycję za dopasowaniem.

■ `String group()`

zwraca bieżące dopasowanie.

■ `int groupCount()`

zwraca liczbę grup we wzorcu wejściowym.

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

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

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

- String group(int groupIndex)
- zwraca łańcuch dopasowany do podanej grupy.
- Parametry:* groupIndex indeks grupy (wartości indeksu rozpoczynają się od 1) lub 0 dla oznaczenia całego dopasowania
- String replaceAll(String replacement)
- String replaceFirst(String replacement)

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

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

- static String quoteReplacement(String str) 5.0  
cytuje wszystkie znaki \ i \$ w łańcuchu str.
- Matcher reset()
- Matcher reset(CharSequence input)  
resetuje stan obiektu Matcher. Druga wersja powoduje przejście obiektu Matcher do pracy z innymi danymi wejściowymi. Obie wersje zwracają this.

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

# 2

## Język XML

W tym rozdziale:

- Wprowadzenie do języka XML.
- Parsowanie dokumentów XML.
- Kontrola poprawności dokumentów XML.
- Wyszukiwanie informacji i XPath.
- Przestrzenie nazw.
- Parsery strumieniowe.
- Tworzenie dokumentów XML.
- Przekształcenia XSL.

We wstępie książki *Essential XML* jej autor Don Box (Addison-Wesley, 2000) stwierdza pół żartem: „język XML zastąpił język Java, wzorce projektowe i technologię obiektową w roli panaceum na problem głodu na świecie oferowanego przez przemysł produkcji oprogramowania”. Rzeczywiście, jak pokażemy to w bieżącym rozdziale, język XML jest doskonałym sposobem opisu struktur informacji. Dostępne narzędzia XML ułatwiają przetwarzanie tych informacji. XML nie posiada jednakmagicznych możliwości działania. Aby efektywnie z niego korzystać, potrzebne są standardy zastosowań i biblioteki kodu. Język XML nie stanowi konkurencyjnej propozycji w stosunku do języka Java, a oba uzupełniają się doskonale. W drugiej połowie lat 90. IBM, Apache i inne firmy oraz organizacje opracowały szereg doskonałych bibliotek języka Java, które służyły do przetwarzania języka XML. Wiele z nich dostępnych jest w standardowej edycji platformy Java.

W rozdziale tym przedstawimy wprowadzenie do języka XML oraz omówimy możliwości biblioteki języka Java związane z przetwarzaniem języka XML. Będziemy przy tym starali się odpowiedzieć na pytanie, czy duże nadzieje związane z językiem XML są rzeczywiście uzasadnione, czy też raczej należy pozostać przy tradycyjnych, sprawdzonych sposobach rozwiązywania pewnych klas problemów.

## 2.1. Wprowadzenie do języka XML

W rozdziale 10. książki *Java. Podstawy* pokazaliśmy kilka przykładów zastosowania plików właściwości do zapisu konfiguracji programów. Plik właściwości zawiera pary nazwa-wartość, takie jak poniżej.

```
fontname=Times Roman  
fontsize=12  
windowsize=400 200  
color=0 50 100
```

Do ich odczytu możemy wykorzystać metody klasy Properties. Choć jest to przydatne rozwiążanie, to jednak w wielu sytuacjach okazuje się zbyt ograniczone. Często informacja, którą chcielibyśmy zapisać w takim pliku, posiada pewną strukturę, a jej zapis w pliku właściwości okazuje się mało wygodny. Rozważmy na przykład zapis rodzaju i wielkości czcionki. Może on wyglądać tak:

```
font=Times Roman 12
```

Wymagać on będzie jednak dokładnego parsowania wartości właściwości w celu wykrycia końca nazwy czcionki i początku opisu jej rozmiaru.

Pliki właściwości posiadają płaską strukturę. Programiści starają się często obejść to ograniczenie, stosując odpowiednie nazwy klucza, co pokazuje poniższy przykład.

```
title.fontname=Helvetica  
title.fontsize=36  
body.fontname=Times Roman  
body.fontsize=12
```

Innym ograniczeniem plików właściwości jest konieczność stosowania unikalnych kluczy. Aby przechować w takim pliku sekwencję wartości, musimy znowu zastosować specjalne klucze.

```
menu.item.1=Times Roman  
menu.item.2=Helvetica  
menu.item.3=Goudy Old Style
```

Wszystkie te problemy rozwiązuje zastosowanie formatu XML, który potrafi wyrazić hierarchiczną strukturę danych i jest przez to bardziej uniwersalny od „płaskiego” pliku właściwości.

Plik XML opisujący konfigurację programu może na przykład wyglądać następująco:

```
<configuration>  
  <title>  
    <font>  
      <name>Helvetica</name>  
      <size>36</size>  
    </font>  
  </title>  
  <body>  
    <font>  
      <name>Times Roman</name>  
      <size>12</size>
```

```

        </font>
    </body>
    <window>
        <width>400</width>
        <heigth>200</height>
    </window>
    <color>
        <red>0</red>
        <green>50</green>
        <blue>100</blue>
    </color>
    <menu>
        <item>Times Roman</item>
        <item>Helvetica</item>
        <item>Goudy Old Style</item>
    </menu>
</configuration>
```

Jak łatwo zauważyc, format XML pozwala wyrazić hierarchiczną strukturę danych i powtórzenia elementów bez niepotrzebnych kombinacji.

Format pliku XML jest przejrzysty i zbliżony do plików HTML. Języki XML i HTML stanowią bowiem pochodne języka SGML (*Standard Generalized Markup Language*).

Język SGML opracowano w latach 70. w celu opisu struktury złożonych dokumentów. Był z powodzeniem stosowany w gałęziach przemysłu wymagających zarządzania olbrzymimi ilościami dokumentacji, na przykład w przemyśle lotniczym. Ponieważ jednak język SGML jest dość skomplikowany, to nie zyskał większego grona zwolenników. Złożoność języka SGML wynika w znacznej mierze z próby osiągnięcia dwóch sprzecznych celów. Z jednej strony próbuje on zapewnić, że dokumenty są zawsze sformatowane zgodnie ze swoim typem, a z drugiej — ograniczyć wysiłek związany z wprowadzaniem danych przez zastosowanie skrótów. Język XML zaprojektowano jako uproszczoną wersję języka SGML z myślą o zastosowaniach w Internecie. Jak często się zdarza, prostsze okazało się lepsze i język XML spotkał się z entuzjastycznym przyjęciem oraz szybko zdobył duże grono zwolenników, co nie udało się językowi SGML.



Bardzo udaną wersję opisu standardu XML opracował Tim Bray i jest ona dostępna na stronie <http://www.xml.com/axml/axml.html>.

Mimo że języki XML i HTML posiadają wspólne korzenie, to istnieje między nimi wiele różnic.

- W przeciwieństwie do języka HTML język XML uwzględnia wielkość znaków. Dlatego też na przykład znaczniki `<H1>` i `<h1>` będą w nim rozróżniane.
- W języku HTML możemy opuścić znaczniki końca akapitu `</p>` lub końca elementu listy `</li>`, jeśli z kontekstu wynika, w którym miejscu powinny one wystąpić. W języku XML nie wolno opuszczać znaczników końca.
- W języku XML elementy posiadające pojedynczy znacznik, któremu nie odpowiada żaden znacznik końca, muszą kończyć się znakiem ukośnika, na przykład ``. Dzięki temu parser języka XML uzyskuje informację, że nie powinien szukać znacznika końca.

- W języku XML wartości atrybutów muszą być ujęte w znaki cudzysłowu, które w tym przypadku są opcjonalne w języku HTML. Na przykład zapis <applet code="MyApplet.class" width=300 height=300> jest dozwolony w języku HTML, ale zabroniony w języku XML. W języku XML należałoby uzupełnić go znakami cudzysłowu: width="300" height="300".
- W języku HTML mogą istnieć atrybuty bez wartości, na przykład <input type="radio" name="language" value="Java" checked>. W języku XML wszystkie atrybuty muszą posiadać wartość, na przykład checked="true" lub (co mniej sensowne) checked="checked".

## 2.1.1. Struktura dokumentu XML

Dokument XML powinien rozpoczynać się nagłówkiem w postaci

```
<?xml version="1.0"?>
```

lub

```
<?xml version="1.0" encoding="UTF-8"?>
```

Chociaż użycie nagłówka jest opcjonalne, to jednak ze wszech miar zalecane.



Ponieważ język SGML został utworzony w celu przetwarzania dokumentów, to także pliki XML zwykło nazywać się *dokumentami*, mimo że większość z nich opisuje zbiory danych, których nie nazwalibyśmy w ten sposób.

Po nagłówku następuje zwykle *deklaracja typu dokumentu (DTD)*, na przykład:

```
<!DOCTYPE web-app PUBLIC
        "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
        "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

Deklaracje typu dokumentu stanowią ważny mechanizm służący do zapewnienia poprawności dokumentu, ale także są opcjonalne. Omówimy je w dalszej części rozdziału.

Na dokument XML składa się także *element korzenia*, który może zawierać inne elementy. Na przykład:

```
<?xml version="1.0"?>
<!DOCTYPE configuration . . .>
<configuration>
    <title>
        <font>
            <name>Helvetica</name>
            <size>36</size>
        </font>
    </title>
    .
</configuration>
```

Każdy z elementów może zawierać *element podrzędny* i (lub) tekst. W powyższym przykładzie element font posiada dwa elementy podrzędne, name i size. Element name zawiera tekst "Helvetica".



Dobrym rozwiązaniem jest takie zorganizowanie struktury dokumentu XML, by każdy element zawierał albo *wyłącznie* elementy podrzędne, albo *wyłącznie* tekst. Innymi słowy powinno unikać się sytuacji przedstawionej poniżej.

```
<font>
    Helvetica
    <size>36</size>
</font>
```

Specyfikacja XML określa ją mianem *mieszanej zawartości*. Jak pokażemy później, unikając mieszanej zawartości, możemy uprościć parsowanie dokumentów XML.

Elementy mogą też posiadać atrybuty, na przykład:

```
<size unit="pt">36</size>
```

Pomiędzy projektantami języka XML nie ma zgody co do tego, kiedy należy stosować elementy, a kiedy ich atrybuty. Wydaje się na przykład, że czcionkę łatwiej opisać za pomocą atrybutów

```
<font name="Helvetica" size="36"/>
```

niż za pomocą elementów

```
<font>
    <name>Helvetica</name>
    <size>36</size>
</font>
```

Jednak atrybuty okazują się mniej uniwersalne od elementów. Założymy na przykład, że opis rozmiaru czcionki chcemy wyposażyć w jednostkę. Korzystając z atrybutów, umieścimy ją w wartości atrybutu.

```
<font name="Helvetica" size="36 pt"/>
```

Jednak wtedy będziemy zmuszeni parsować wartość atrybutu, a stosując język XML, chcieliśmy przecież tego uniknąć. Lepszym rozwiązaniem jest zatem dodanie atrybutu do elementu size:

```
<font>
    <name>Helvetica</name>
    <size unit="pt">36</size>
</font>
```

W praktyce stosuje się zasadę, która głosi, że atrybuty należy wykorzystywać jedynie wtedy, gdy modyfikują sposób interpretacji wartości, a nie do specyfikacji wartości. Rozróżnienie tych dwóch sytuacji może w niektórych przypadkach okazać się trudne i wtedy lepiej zrezygnować z użycia atrybutów na rzecz elementów. Wiele użytecznych dokumentów XML w ogóle nie wykorzystuje atrybutów.



W języku HTML zasada wykorzystania atrybutów jest prosta: jeśli element nie jest prezentowany na stronie internetowej, to powinien być atrybutem. Rozważmy na przykład poniższe hiperłącze.

```
<a href="http://java.sun.com" >Java Technology</a>
```

Łańcuch Java Technology wyświetlany jest na stronie, ale adres URL łączy już nie. Zasada ta nie jest niestety użyteczna w przypadku języka XML, ponieważ w większości przypadków dane zawarte w plikach XML nie są przewidziane do prezentacji użytkownikom.

Elementy i tekst stanowią podstawę zawartości plików XML. Zastosowanie znajduje też kilka innych rodzajów instrukcji, które przedstawiamy poniżej.

- *Referencje znaków* posiadają postać `&#d;` lub `&#h;`. W obu przypadkach kod znaku jest kodem Unicode, *d* reprezentuje jego postać dziesiętną, a *h* — szesnastkową. Na przykład znak é możemy zapisać za pomocą jednej z przedstawionych poniżej referencji:

```
&#233;  
&#xD9;
```

- *Referencje bytów* posiadają postać `&name;`. Poniższe referencje

```
&lt;  
&gt;  
&amp;  
&quot;  
&apos;
```

posiadają predefiniowane znaczenie: znaku mniejszości, znaku większości, ampersandu, znaku cudzysłowu i znaku apostrofu. Każda definicja typu dokumentu (DTD) może definiować własne referencje bytów.

- *Sekcje CDATA* ograniczone są znacznikami postaci `<![CDATA[ i ]]>`. Reprezentują one specjalną formę danych znakowych. Wykorzystujemy je do reprezentacji łańcuchów, zawierających znaki `<` i `>`, które nie mają być interpretowane jako znaczniki, na przykład:

```
<![CDATA[< &gt; are my favorite delimiters]]>
```

Sekcje CDATA nie mogą zawierać podłańcucha `]]>`. Należy korzystać z nich ostrożnie, ponieważ stanowią furtkę nadużywaną do wprowadzania tradycyjnych danych do dokumentów XML.

- *Instrukcje przetwarzania* ograniczone są za pomocą łańcuchów `<? i ?>`, na przykład:

```
<?xml-stylesheet href="mystyle.css" type="text/css"?>
```

Każdy dokument XML rozpoczyna się następującą instrukcją przetwarzania:

```
<?xml version="1.0"?>
```

- *Komentarze* ograniczone są za pomocą łańcuchów `<!-- i -->`, na przykład

```
<!--To jest komentarz. -->
```

Komentarze nie mogą zawierać podłańcucha `--`. Stanowią informację dla programistów i nie powinny zawierać ukrytych poleceń aplikacji, dla których należy stosować instrukcje przetwarzania.

## 2.2. Parsowanie dokumentów XML

Aby przetworzyć dokument XML, musimy najpierw go *parsować*. Parser jest programem, który wczytuje zawartość pliku, stwierdza poprawność jej formatu i rozkłada ją na elementy składowe, które udostępnia programiście. Istnieją dwa rodzaje parserów XML:

- parsery drzewiaste, jak na przykład DOM (*Document Object Model*), które umieszczają elementy dokumentu XML w strukturze drzewa,
- parsery strumieniowe, jak na przykład SAX (*Simple API for XML*), które generują zdarzenia podczas czytania dokumentu XML.

W większości przypadków parser DOM jest łatwiejszy w użyciu i omówimy go w pierwszej kolejności. Parser strumieniowy stosujemy przede wszystkim, przetwarzając obszerne dokumenty XML, dla których reprezentacja za pomocą drzewa zajęłaby zbyt wiele pamięci lub jeśli interesuje nas tylko kilka elementów dokumentu XML, a ich kontekst nie ma dla nas znaczenia. Więcej informacji na ten temat umieściliśmy w podrozdziale 2.6, „Parsery strumieniowe”.

Interfejs parsera DOM został ustandaryzowany przez konsorcjum W3C (*World Wide Web Consortium*). Pakiet org.w3c.dom zawiera definicję typów interfejsu, takich jak Document i Element. Inni dostawcy, na przykład organizacja Apache czy firma IBM, utworzyli parsery DOM, których klasy implementują te interfejsy. Interfejs programowy języka Java do przetwarzania dokumentów XML (JAXP) umożliwia połączenie z dowolnym z tych parserów. Ale również pakiet JDK zawiera własny parser DOM, który będziemy wykorzystywać w bieżącym rozdziale.

Aby wczytać dokument XML, potrzebny jest obiekt klasy DocumentBuilder, który uzyskamy z fabryki DocumentBuilderFactory w następujący sposób:

```
DocumentBuilderFactory factory
    = DocumentBuilderFactory.newInstance();
DocumentBuilder builder
    = factory.newDocumentBuilder();
```

Możemy teraz wczytać dokument XML z lokalnego pliku:

```
File f = . . . ;
Document doc = builder.parse(f);
```

lub z lokalizacji określonej za pomocą adresu URL:

```
URL u = . . . ;
Document doc = builder.parse(u);
```

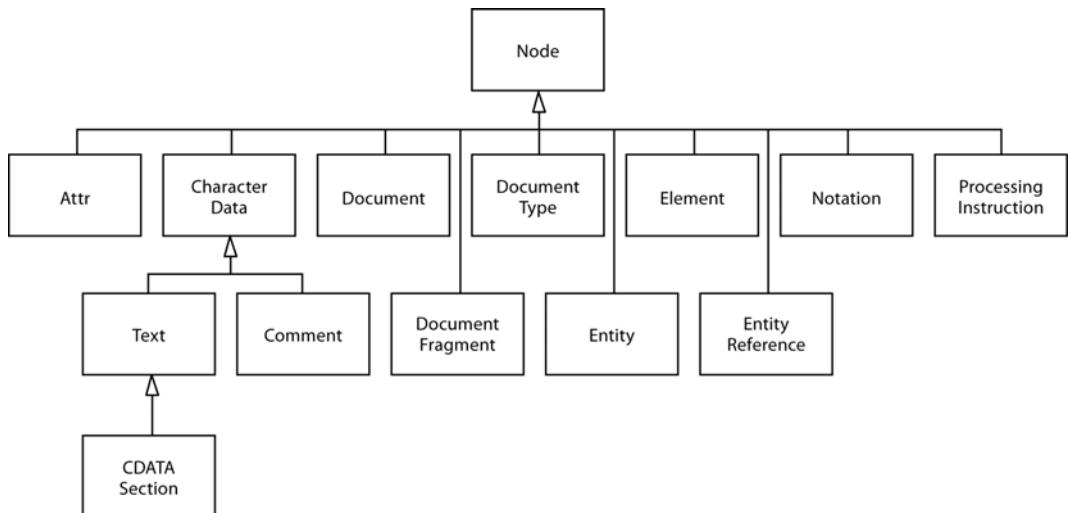
A nawet dowolnego strumienia wejściowego:

```
InputStream in = . . . ;
Document doc = builder.parse(in);
```



Jeśli do wczytania dokumentu XML wykorzystamy strumień wejściowy, to parser nie będzie w stanie zlokalizować innych plików, do których referencje zawiera dokument XML, na przykład plików DTD, nawet umieszczonych w tym samym katalogu co dokument XML. Rozwiązanie tego problemu polega na instalacji specjalnego obiektu implementującego interfejs EntityResolver.

Obiekt typu `Document` reprezentuje dokument XML za pomocą struktury drzewiastej utworzonej w pamięci. Składa się ona z obiektów, które implementują interfejs `Node` i jego różne interfejsy pochodne. Rysunek 2.1 prezentuje hierarchię dziedziczenia tych interfejsów.



**Rysunek 2.1.** Interfejs `Node` i jego pochodne

Analizę zawartości dokumentu XML rozpoczynamy od wywołania metody `getDocumentElement()`. Zwraca ona element znajdujący się w korzeniu struktury dokumentu.

```
Element root = doc.getDocumentElement();
```

Na przykład przetwarzając dokument o następującej zawartości:

```
<?xml version="1.0"?>
<font>
  .
</font>
```

wywołanie metody `getDocumentElement()` zwróci element `font`.

Metoda `getTagName()` zwraca nazwę znacznika elementu. Dla poprzedniego przykładu pliku wywołanie `root.getTagName()` zwróci łańcuch "font".

Aby pobrać elementy podrzędne danego elementu (które mogą być elementami, tekstem, komentarzem lub innymi rodzajami węzłów), korzystamy z metody `getChildNodes()`. Zwraca ona kolekcję typu `NodeList`. Typ ten zdefiniowany został przed wprowadzeniem standardowych kolekcji języka Java i dlatego też dysponuje innym protokołem dostępu. Metoda `item()` zwraca element kolekcji o podanym indeksie, a metoda `getLength()` — całkowitą liczbę elementów kolekcji. Korzystając z tych metod, możemy przeglądać elementy kolekcji w następujący sposób:

```
NodeList children = root.getChildNodes();
for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    .
}
```

Elementy podrzędne powinniśmy analizować niezwykle dokładnie. Założymy na przykład, że przetwarzamy następujący dokument:

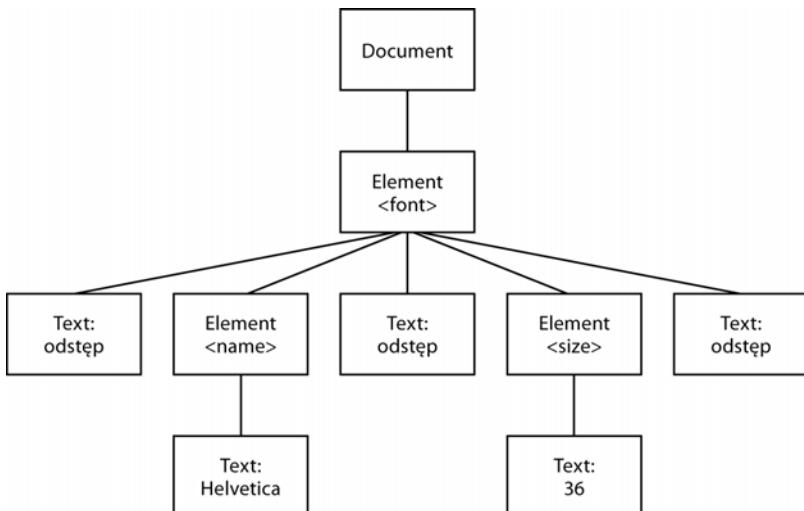
```
<font>
    <name>Helvetica</name>
    <size>36</size>
</font>
```

Moglibyśmy spodziewać się, że element `font` posiada dwa elementy podrzędne. Jednak parser przedstawi nam aż pięć takich elementów. Oto one.

- Odstęp między `<font>` i `<name>`.
- Element `name`.
- Odstęp między `</name>` i `<size>`.
- Element `size`.
- Odstęp między `</size>` i `</font>`.

Rysunek 2.2 prezentuje drzewo tego dokumentu utworzone przez parser DOM.

**Rysunek 2.2.**  
Drzewo utworzone  
przez parser DOM



Jeśli interesują nas tylko elementy podrzędne, to możemy zignorować odstępy w poniższy sposób:

```
for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    if (child instanceof Element)
    {
        Element childElement = (Element)child;
        ...
    }
}
```

Teraz możemy już przetwarzać dalej tylko dwa elementy: `name` i `size`.

W kolejnym podrozdziale pokażemy jeszcze doskonalsze rozwiązywanie tego problemu polegające na zastosowaniu deklaracji typu dokumentu (DTD). Informuje ona parser, które elementy nie posiadają węzłów tekstowych jako elementów podrzędnych, co pozwala mu zignorować odstępy.

Analizując elementy `name` i `size`, będziemy chcieli uzyskać zawarte w nich łańcuchy. Łańcuchy te umieszczone są w węzłach podrzędnych typu `Text`. Ponieważ wiemy, że węzły te są jedynymi węzłami podrzędnymi, to do ich uzyskania możemy wykorzystać metodę `getFirstChild` bez konieczności przeglądania kolejnych węzłów. Metoda `getData` zwróci łańcuch zawarty w węźle typu `Text`.

```
for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    if (child instanceof Element)
    {
        Element childElement = (Element)child;
        Text textNode = (Text) childElement.getFirstChild();
        String text = textNode.getData().trim();
        if (childElement.getTagName().equals("name"))
            name = text;
        else if (childElement.getTagName().equals("size"))
            size = Integer.parseInt(text);
    }
}
```



Dla wartości zwracanej przez metodę `getData` warto wywołać zawsze metodę `trim`. Jeśli bowiem autor pliku XML umieścił znaczniki początkowy i końcowy w osobnych wierszach, na przykład

```
<size>
  36
</size>
```

to parser włączy wszystkie znaki odstępu i nowego wiersza do danych węzła tekstu. Wywołanie metody `trim` umożliwi nam pozbycie się tych zbędnych znaków otaczających właściwe dane.

Pobranie ostatniego z węzłów podrzędnych umożliwia metoda `getLastChild`, a kolejnego węzła siostrzanego dla danego węzła — metoda `getNextSibling`. Korzystając z tej ostatniej, możemy przeglądać węzły podrzędne w następujący sposób:

```
for (Node childNode = element.getFirstChild();
     childNode != null;
     childNode = childNode.getNextSibling())
{
    ...
}
```

Aby przejrzeć atrybuty węzła, wywołujemy metodę `getAttributes`. Zwraca ona obiekt typu `NamedNodeMap`, który zawiera obiekty typu `Node` opisujące atrybuty. Węzły umieszczone w obiekcie typu `NamedNodeMap` przeglądamy w ten sam sposób co węzły kolekcji `NodeList`. Metody `getNodeName` i `getNodeValue` zwracają nazwę i wartość atrybutu.

```

NamedNodeMap attributes = element.getAttributes();
for (int i = 0; i < attributes.getLength(); i++)
{
    Node attribute = attributes.item(i);
    String name = attribute.getNodeName();
    String value = attribute.getNodeValue();
    . .
}

```

Jeśli znamy nazwę atrybutu, to jego wartość możemy też pobrać bezpośrednio:

```
String unit = element.getAttribute("unit");
```

Zaprezentowane dotąd sposoby analizy drzewa utworzonego przez parser DOM zilustrujemy teraz programem, którego kod źródłowy zawiera listing 2.1. Pozycja menu *File/Open* umożliwia wczytanie dokumentu XML. Obiekt typu `DocumentBuilder` parsuje plik XML i tworzy obiekt typu `Document`. Program prezentuje powstałą w ten sposób strukturę drzewiastą za pomocą komponentu `JTree` (patrz rysunek 2.3).

### **Listing 2.1.** *dom/TreeViewer.java*

```

package dom;

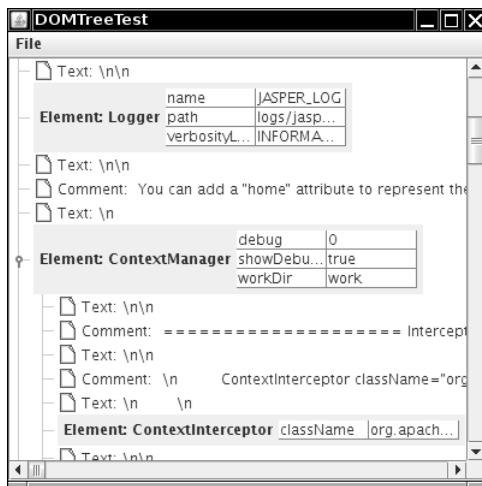
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;
import javax.swing.tree.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.w3c.dom.CharacterData;

/**
 * Program prezentujący strukturę dokumentu XML w postaci drzewa.
 * @version 1.12 2012-06-03
 * @author Cay Horstmann
 */
public class TreeViewer
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new DOMTreeFrame();
                frame.setTitle("TreeViewer");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

```

**Rysunek 2.3.**

*Rezultat parsowania dokumentu XML*



```
/**
 * Ramka zawierająca komponent drzewa
 * prezentujący zawartość dokumentu XML.
 */
class DOMTreeFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 400;

    private DocumentBuilder builder;

    public DOMTreeFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        JMenu fileMenu = new JMenu("File");
        JMenuItem openItem = new JMenuItem("Open");
        openItem.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                openFile();
            }
        });
        fileMenu.add(openItem);
        JMenuItem exitItem = new JMenuItem("Exit");
        exitItem.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                System.exit(0);
            }
        });
        fileMenu.add(exitItem);

        JMenuBar menuBar = new JMenuBar();
        menuBar.add(fileMenu);
        setJMenuBar(menuBar);
    }
}
```

```

    }

    /**
     * Otwiera plik i ładuje dokument.
     */
    public void openFile()
    {
        JFileChooser chooser = new JFileChooser();
        chooser.setCurrentDirectory(new File("dom"));

        chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
        {
            public boolean accept(File f)
            {
                return f.isDirectory() || f.getName().toLowerCase().endsWith(".xml");
            }
        });

        public String getDescription()
        {
            return "XML files";
        }
    });
    int r = chooser.showOpenDialog(this);
    if (r != JFileChooser.APPROVE_OPTION) return;
    final File file = chooser.getSelectedFile();

    new SwingWorker<Document, Void>()
    {
        protected Document doInBackground() throws Exception
        {
            if (builder == null)
            {
                DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
                builder = factory.newDocumentBuilder();
            }
            return builder.parse(file);
        }

        protected void done()
        {
            try
            {
                Document doc = get();
                JTree tree = new JTree(new DOMTreeModel(doc));
                tree.setCellRenderer(new DOMTreeCellRenderer());

                setContentPane(new JScrollPane(tree));
                validate();
            }
            catch (Exception e)
            {
                JOptionPane.showMessageDialog(DOMTreeFrame.this, e);
            }
        }
    }.execute();
}
}

```

```
/*
 * Model drzewa opisujący strukturę dokumentu XML.
 */
class DOMTreeModel implements TreeModel
{
    private Document doc;

    /**
     * Tworzy model drzewa dokumentu.
     * @param doc dokument
     */
    public DOMTreeModel(Document doc)
    {
        this.doc = doc;
    }

    public Object getRoot()
    {
        return doc.getDocumentElement();
    }

    public int getChildCount(Object parent)
    {
        Node node = (Node) parent;
        NodeList list = node.getChildNodes();
        return list.getLength();
    }

    public Object getChild(Object parent, int index)
    {
        Node node = (Node) parent;
        NodeList list = node.getChildNodes();
        return list.item(index);
    }

    public int getIndexOfChild(Object parent, Object child)
    {
        Node node = (Node) parent;
        NodeList list = node.getChildNodes();
        for (int i = 0; i < list.getLength(); i++)
            if (getChild(node, i) == child) return i;
        return -1;
    }

    public boolean isLeaf(Object node)
    {
        return getChildCount(node) == 0;
    }

    public void valueForPathChanged(TreePath path, Object newValue)
    {
    }

    public void addTreeModelListener(TreeModelListener l)
    {
    }

    public void removeTreeModelListener(TreeModelListener l)
```

```

    {
}

}

/***
 * Klasa wyświetlająca węzeł dokumentu.
 */
class DOMTreeCellRenderer extends DefaultTreeCellRenderer
{
    public Component getTreeCellRendererComponent(JTree tree, Object value, boolean
→selected,
        boolean expanded, boolean leaf, int row, boolean hasFocus)
    {
        Node node = (Node) value;
        if (node instanceof Element) return elementPanel((Element) node);

        super.getTreeCellRendererComponent(tree, value, selected, expanded, leaf, row,
→hasFocus);
        if (node instanceof CharacterData) setText(characterString((CharacterData) node));
        else setText(node.getClass() + ": " + node.toString());
        return this;
    }

    public static JPanel elementPanel(Element e)
    {
        JPanel panel = new JPanel();
        panel.add(new JLabel("Element: " + e.getTagName()));
        final NamedNodeMap map = e.getAttributes();
        panel.add(new JTable(new AbstractTableModel()
        {
            public int getRowCount()
            {
                return map.getLength();
            }

            public int getColumnCount()
            {
                return 2;
            }

            public Object getValueAt(int r, int c)
            {
                return c == 0 ? map.item(r).getNodeName() : map.item(r).getNodeValue();
            }
        }));
        return panel;
    }

    public static String characterString(CharacterData node)
    {
        StringBuilder builder = new StringBuilder(node.getData());
        for (int i = 0; i < builder.length(); i++)
        {
            if (builder.charAt(i) == '\r')
            {
                builder.replace(i, i + 1, "\\\r");
                i++;
            }
        }
    }
}

```

```

        }
        else if (builder.charAt(i) == '\n')
        {
            builder.replace(i, i + 1, "\\\n");
            i++;
        }
        else if (builder.charAt(i) == '\t')
        {
            builder.replace(i, i + 1, "\\\t");
            i++;
        }
    }
    if (node instanceof CDATASection) builder.insert(0, "CDATASection: ");
    else if (node instanceof Text) builder.insert(0, "Text: ");
    else if (node instanceof Comment) builder.insert(0, "Comment: ");

    return builder.toString();
}
}

```

Prezentowane przez program drzewo pokazuje, w jaki sposób elementy podrzędne dokumentu XML otoczone są przez tekst zawierający odstępy i komentarze. Dla lepszej przejrzystości prezentuje ono znaki nowej linii i powrotu karetki jako \n i \r. (W przeciwnym razie znaki te reprezentowane byłyby symbolem pustego prostokąta, stosowanego przez bibliotekę Swing w przypadku wszystkich znaków, które nie posiadają reprezentacji graficznej).

W rozdziale 6. poznamy techniki wykorzystywane przez program w celu prezentacji drzewa i tabel atrybutów. Klasa DOMTreeModel implementuje interfejs TreeModel. Metoda getRoot zwraca element znajdujący się w korzeniu struktury dokumentu. Metoda getChild pobiera listę węzłów podrzędnych i zwraca element o podanym indeksie. Podczas wyświetlania zawartości drzewa prezentowane są następujące informacje:

- dla elementów — nazwa etykiety znacznika i tabela wszystkich atrybutów,
- dla danych znakowych — interfejs (Text, Comment lub CDATASection) i dane, w których wystąpienia znaków nowego wiersza i powrotu karetki reprezentowane są odpowiednio jako \n i \r,
- dla wszystkich innych typów węzłów — nazwa klasy, po której następuje rezultat wywołania metody `toString`.

#### javax.xml.parsers.DocumentBuilderFactory 1.4

- static DocumentBuilderFactory newInstance()
   
zwraca instancję klasy DocumentBuilderFactory.
- DocumentBuilder newDocumentBuilder()
   
zwraca instancję klasy DocumentBuilder.

#### java.xml.parsers.DocumentBuilder 1.4

- Document parse(File f)
- Document parse(String url)

- Document parse(InputStream in)

parsują dokument XML pobrany z podanego pliku, adresu URL bądź strumienia wejściowego. Zwracają sparsowany dokument.

#### org.w3c.dom.Document 1.4

- Element getDocumentElement()

zwraca element znajdujący się w korzeniu struktury dokumentu XML.

#### org.w3c.dom.Element 1.4

- String getTagName()

zwraca nazwę elementu.

- String getAttribute(String name)

zwraca wartość atrybutu o podanej nazwie lub pusty łańcuch, jeśli atrybut o takiej nazwie nie istnieje.

#### org.w3c.dom.Node 1.4

- NodeList getChildNodes()

zwraca listę węzłów zawierającą wszystkie węzły podrzędne danego węzła.

- Note getChild()

- Note getLastChild()

zwracają pierwszy lub ostatni z węzłów podrzędnych danego węzła lub wartość null, jeśli węzeł nie posiada węzłów podrzędnych.

- Node getNextSibling()

- Node getPreviousSibling()

zwracają następny lub poprzedni węzeł siostrzany danego węzła lub wartość null, jeśli węzeł nie posiada węzłów siostrzanych.

- Node getParentNode()

zwraca węzeł nadrzędny danego węzła lub wartość null, jeśli dany węzeł reprezentuje dokument.

- NamedNodeMap getAttributes()

zwraca mapę węzłów zawierającą węzły typu Attr reprezentujące atrybuty danego węzła.

- String getNodeName()

zwraca nazwę węzła. Jeśli węzeł jest typu Attr, jest to nazwa atrybutu.

- String getNodeValue()

zwraca wartość węzła. Jeśli węzeł jest typu Attr, to jest to wartość atrybutu.

**API** *org.w3c.dom.CharacterData 1.4*

- `String getData()`  
zwraca tekst przechowywany w węźle.

**API** *org.w3c.dom.NodeList 1.4*

- `int getLength()`  
zwraca liczbę węzłów na liście.
- `Node item(int index)`  
zwraca węzeł o podanym indeksie. Indeks ten musi być z zakresu od 0 do `getLength - 1`.

**API** *org.w3c.dom.NamedNodeMap 1.4*

- `int getLength()`  
zwraca liczbę węzłów na mapie.
- `Node item(int index)`  
zwraca węzeł o podanym indeksie. Indeks ten musi być z zakresu od 0 do `getLength - 1`.

## 2.3. Kontrola poprawności dokumentów XML

W poprzednim podrozdziale pokazaliśmy, w jaki sposób przeglądać drzewistą strukturę dokumentu wygenerowaną przez parser DOM. Jeśli zastosujemy takie rozwiązanie w praktyce, to okaże się, że wymaga ono sporo programowania i rozbudowanej kontroli błędów. Nie tylko będziemy musieli poradzić sobie z odstępami między elementami, ale także sprawdzać, czy dokument zawiera spodziewane węzły. Założymy na przykład, że wczytaliśmy poniższy element.

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

Pobierzmy pierwszy element podrzędny. Okaże się on węzłem tekstowym zawierającym odstęp "\n ". Pominiemy zatem dalej wszystkie węzły tekstowe i znajdziemy pierwszy węzeł reprezentujący element. Musimy sprawdzić, czy nazwą jego znacznika jest "name" a później, czy posiada on pojedynczy węzeł podrzędny typu Text. Ten ostatni warunek musimy sprawdzić dla wszystkich węzłów podrzędnych, ponieważ autor dokumentu mógł zmienić porządek tych węzłów lub dodać nowe. Napisanie odpowiedniego kodu jest pracochłonne i łatwo w nim o pomyłkę.

Jednak jedną z podstawowych zalet stosowania parsera dokumentów XML jest to, że może on automatycznie sprawdzić poprawność struktury dokumentu. Dzięki temu późniejsza jej analiza jest znacznie łatwiejsza. Jeśli na przykład element `font` przeszedł pomyślnie weryfikację, to możemy od razu pobrać dwa jego elementy podrzędne drugiego stopnia, rzutując je na typ `Text` i pobrać łańcuchy znaków bez żadnej dodatkowej kontroli.

Aby określić strukturę dokumentu, dostarczamy plik definicji typu dokumentu (DTD) lub definicję w języku XML Schema. Plik DTD lub definicja XML Schema zawiera reguły opisujące format dokumentu przez określenie dopuszczalnych elementów podrzędnych i atrybutów każdego z elementów. Przykładowa reguła może wyglądać następująco:

```
<!ELEMENT font (name,size)>
```

Reguła ta wymaga, aby element `font` posiadał zawsze dwa elementy podrzędne — `name` i `size`. W języku XML Schema te same ograniczenia moglibyśmy wyrazić w poniższy sposób:

```
<xsd:element name="font">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="xsd:int"/>
  </xsd:sequence>
</xsd:element>
```

Za pomocą XML Schema można wyrazić bardziej złożone warunki kontroli poprawności (na przykład fakt, że element `size` musi zawierać wartość całkowitą) niż za pomocą plików DTD. W przeciwieństwie do DTD, XML Schema używa składni XML, co może być zaletą w sytuacji, gdy musimy przetwarzanie pliki XML Schema.

Język XML Schema zaprojektowano z myślą o zastąpieniu plików DTD. Jednak nadal są one szeroko wykorzystywane. Język XML Schema jest skomplikowany i nie zyskał powszechniej akceptacji. Niektórzy użytkownicy języka XML sfrustrowani poziomem złożoności języka XML Schema używają alternatywnych rozwiązań kontroli poprawności plików XML. Najpopularniejszym z nich jest Relax NG (<http://www.relaxng.org>).

W kolejnym podrozdziale omówimy szczegółowo pliki DTD. Następnie krótko przedstawimy podstawy obsługi XML Schema. Na koniec pokażemy przykład kompletnej aplikacji ilustrujący sposób, w jaki zastosowanie kontroli poprawności upraszcza przetwarzanie dokumentów XML.

### 2.3.1. Definicje typów dokumentów

Istnieje kilka sposobów dostarczenia definicji typu dokumentu (DTD). Możemy umieścić ją w dokumencie XML w poniższy sposób:

```
<?XML VERSION="1.0"?>
<!DOCTYPE configuration [
  <!ELEMENT configuration . . .>
  następne reguły
]
<configuration>
  .
  .
  </configuration>
```

Reguły w tym przypadku umieszczone są wewnątrz deklaracji DOCTYPE w bloku ograniczonym znakami nawiasów prostokątnych [ . . . ]. Typ dokumentu musi zgadzać się z nazwą elementu znajdującego się w korzeniu struktury dokumentu, czyli configuration w naszym przykładzie.

Rozwiążanie polegające na umieszczaniu definicji typu dokumentu w samym dokumencie XML stosuje się rzadko, ponieważ definicja ta może być sporych rozmiarów. Lepszym wyjściem jest więc umieszczenie jej osobno. W dokumencie określamy wtedy jej lokalizację za pomocą deklaracji SYSTEM zawierającej adres URL definicji typu dokumentu, na przykład:

```
<!DOCTYPE configuration SYSTEM "config.dtd">
```

lub

```
<!DOCTYPE configuration SYSTEM "http://myserver.com/config.dtd">
```



Jeśli określamy względne położenie definicji typu dokumentu (na przykład "config.dtd"), to musimy przekazać parserowi obiekt typu File lub Uri, a nie typu InputStream. Jeśli musimy wczytać definicję typu dokumentu, korzystając z strumienia wejściowego, to należy zastosować obiekt implementujący interfejs EntityResolver. Zagadnienie to omawia kolejna uwaga.

Istnieje także mechanizm identyfikacji popularnych definicji typów dokumentów, który pochodzi jeszcze z języka SGML. Poniżej prezentujemy odpowiedni przykład.

```
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

Jeśli procesor języka XML potrafi zlokalizować popularny typ dokumentu, to nie musimy nawet podawać mu odpowiedniego adresu URL.



Kiedy, stosując parser DOM, chcemy wykorzystać identyfikator PUBLIC, musimy wywołać najpierw metodę setEntityResolver klasy DocumentBuilder w celu zainstalowania obiektu implementującego interfejs EntityResolver. Interfejs ten posiada pojedynczą metodę resolveEntity. Poniżej prezentujemy typowy sposób jego implementacji.

```
class MyEntityResolver implements EntityResolver
{
    public InputSource resolveEntity(String publicID,
                                     String systemID)
    {
        if (publicID.equals(znany identyfikator))
            return new InputSource(dane DTD);
        else
            return null; //wykły sposób działania
    }
}
```

Obiekt InputSource możemy skonstruować na podstawie obiektu InputStream, Reader lub String.

Wiemy już, w jaki sposób parser lokalizuje definicję typu dokumentu, przejdźmy zatem do szczegółowego omówienia reguł.

Reguła ELEMENT określa elementy podrzędne danego elementu. Konstruujemy ją za pomocą wyrażenia regularnego złożonego z komponentów przedstawionych w tabeli 2.1.

**Tabela 2.1.** Reguły określające zawartość elementów

Reguła	Znaczenie
$E^*$	0 lub więcej wystąpień $E$
$E^+$	1 lub więcej wystąpień $E$
$E^?$	0 lub 1 wystąpienie $E$
$E_1 E_2  \dots  E_n$	jeden z elementów $E_1, E_2, \dots, E_n$
$E_1, E_2, \dots, E_n$	$E_1$ , po którym następują $E_2, \dots, E_n$
#PCDATA	tekst
(#PCDATA  $E_1 E_2  \dots  E_n$ ) $^*$	0 lub więcej wystąpień tekstu, po którym występują $E_1, E_2, \dots, E_n$ w dowolnym porządku (zawartość mieszana)
ANY	dowolny element podrzędny
EMPTY	nie jest dozwolony jakikolwiek element podrzędny

Poniżej prezentujemy kilka prostych, ale typowych przykładów reguły tego typu. Następująca reguła określa, że element menu zawiera 0 lub więcej elementów item.

```
<!ELEMENT menu (item)*>
```

Poniższy zestaw reguł stwierdza, że czcionka opisana jest przez nazwę, po której następuje jej rozmiar, a każdy z elementów opisujących czcionkę zawiera tekst.

```
<!ELEMENT font (name,size)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT size (#PCDATA)>
```

Skrót PCDATA oznacza *parsowane dane znakowe*. Dane określone są jako „parsowane”, gdyż parser interpretuje tekst w poszukiwaniu znaków < oznaczających początek nowego znacznika lub znaków & oznaczających początek bytu.

Specyfikacja elementu może zawierać złożone i zagnieździone wyrażenia regularne. Na przykład następująca reguła może opisywać strukturę rozdziału naszej książki:

```
<!ELEMENT chapter (intro, (heading, (para|image|table|note)+)+)>
```

Każdy rozdział składa się z wprowadzenia, po którym następuje jeden lub więcej podrozdziałów posiadających tytuł i jeden lub więcej paragrafów, rysunków, tabel oraz uwag.

Często pojawia się jednak sytuacja, w której reguły okazują się niewystarczająco uniwersalne. Jeśli element ma zawierać tekst, to dopuszczać one tylko dwa przypadki. Element może zawierać tylko i wyłącznie tekst, co określa następująca reguła:

```
<!ELEMENT name (#PCDATA)>
```

lub też może zawierać *dowolną kombinację tekstu i znaczników w dowolnej kolejności*, na przykład:

```
<!ELEMENT para (#PCDATA|em|strong|code)*>
```

Niedozwolone jest tworzenie innych rodzajów reguł zawierających skrót #PCDATA. Poniższa reguła jest więc niedozwolona.

```
<!ELEMENT captionedImage (image, #PCDATA)>
```

Regułę taką możemy wyeliminować, wprowadzając dodatkowy element `caption` lub zezwalając na dowolną kombinację znacznika `image` i tekstu.

Ograniczenie te ułatwiają implementację parsera XML podczas analizy *mieszanej zawartości* (tekstu i znaczników). Zezwalając na mieszaną zawartość elementu, tracimy możliwość jej precyzyjnej kontroli. Dlatego też najlepiej projektować definicje typu dokumentów w taki sposób, aby wszystkie elementy zawierały albo tylko inne elementy, albo sam tekst.



W rzeczywistości nie jest prawdą, że reguły opisujące postać elementów mogą zawierać dowolne wyrażenia regularne. Parser XML może odrzucić pewne zbiory złożonych reguł, których efektem byłoby „niedeterministyczne” parsowanie. Przykładem takiego wyrażenia może być  $((x,y)|(x,z))$ , ponieważ gdy parser napotka  $x$ , to nie będzie wiedzieć, którą z możliwości należy zastosować. Reguły te możemy przekształcić do deterministycznej postaci  $(x,(y|z))$ , ale przekształcenie takie w ogólnym przypadku nie zawsze jest możliwe. Przykładem wyrażenia, które nie może być sprowadzone do deterministycznej postaci, jest na przykład  $((x,y)^*|x?)$ . Implementacja parsera dostępna w JDK nie informuje o takich problemach i wybiera zawsze pierwszą pasującą możliwość, co może prowadzić do odrzucenia poprawnych konstrukcji. Jednak zezwala mu na to standard XML, który zakłada, że definicje typów dokumentów powinny być jednoznaczne.

W praktyce problemy te mają niewielkie znaczenie, ponieważ większość definicji DTD jest na tyle prosta, że trudno w nich o niejednoznaczność.

Za pomocą reguł możemy także określić dozwolone atrybuty elementu. Ogólna składnia wygląda w tym przypadku następująco:

```
<!ATTLIST element attribute type default>
```

Tabela 2.2 prezentuje dozwolone typy atrybutów, a tabela 2.3 ich wartości domyślne.

**Tabela 2.2.** Typy atrybutów

Typ	Znaczenie
CDATA	Dowolny łańcuch znaków.
$(A_1 A_2 \dots A_n)$	Jeden z atrybutów $A_1 A_2 \dots A_n$ będących łańcuchami znaków.
NMTOKEN, NMOKENS	Jeden lub więcej tokenów nazw.
ID	Unikalny identyfikator.
IDREF, IDREFS	Jedna lub więcej referencji unikalnego identyfikatora.
ENTITY, ENTITIES	Jeden lub więcej nieparsowanych bytów.

Najczęściej spotykane są dwa sposoby specyfikacji atrybutów przedstawione poniżej:

```
<!ATTLIST font style (plain|bold|italic|bold-italic) plain>
<!ATTLIST size unit CDATA #IMPLIED>
```

**Tabela 2.3.** Wartości domyślne atrybutów

Wartość domyślna	Znaczenie
#REQUIRED	Atrybut jest obowiązkowy.
#IMPLIED	Atrybut jest opcjonalny.
<i>A</i>	Atrybut jest opcjonalny; jeśli nie jest wyspecyfikowany, to parser założy, że jest równy <i>A</i> .
#FIXED <i>A</i>	Atrybut musi być niewyspecyfikowany lub równy <i>A</i> . W obu przypadkach parser przyjmuje, że jest on równy <i>A</i> .

Pierwszy z nich opisuje atrybut style elementu font. Dozwolone są cztery wartości atrybutu, a wartością domyślną jest plain. Drugi ze sposobów przedstawia regułę stwierdzającą, że atrybut unit elementu size może zawierać dowolną sekwencję danych znakowych.



Do opisu danych zalecamy wykorzystanie elementów, a nie atrybutów. Zgodnie z tym zaleceniem styl czcionki powinien być osobnym elementem, na przykład <font> ➔<style> plain</style>...<font>. Atrybuty posiadają jednak niezaprzeczalną zaletę w przypadku typów wyliczeniowych, ponieważ parser może sprawdzić, czy posiadają dozwoloną wartość. W przypadku gdy styl czcionki będzie atrybutem, to parser sprawdzi, czy atrybut posiada jedną z czterech dozwolonych wartości oraz dostarczy wartości domyślnej, gdy wartość atrybutu nie została określona.

Sposób obsługi atrybutów typu CDATA różni się nieco od sposobu przetwarzania skrótu #PCDATA i nie jest w ogóle związany z przetwarzaniem sekcji <![CDATA[...]]>. Wartość atrybutu typu CDATA jest najpierw *normalizowana*, czyli parser przetwarza wszystkie zawarte w niej referencje znaków i bajtów (na przykład &233; lub &lt;:) oraz zastępuje wszelkie odstępy znakiem spacji.

Typ NMTOKEN jest podobny do CDATA z tą różnicą, że użycie większości znaków nieposiadających interpretacji graficznej jest zabronione, a parser usuwa wszelkie odstępy poprzedzające i kończące token. Typ NMTOKENS stanowi listę tokenów nazw oddzielonych znakami odstępu.

Typ ID przydaje się często i określa token nazwy, która musi być unikalna w całym dokumencie. Jej unikalność sprawdzana jest przez parser. Zastosowanie identyfikatorów pokażemy w kolejnym przykładzie. Typ IDREF jest referencją identyfikatora występującego w tym samym dokumencie, co także sprawdzane jest przez parser. Typ IDREFS stanowi listę referencji identyfikatorów oddzielonych znakami odstępu.

Typ ENTITY stanowi spadek po języku SGML i jest w praktyce rzadko stosowany. Przykład jego zastosowania znaleźć można w specyfikacji standardu na stronie <http://www.xml.com/axml/axml.html>.

DTD może także zawierać własne definicje bajtów i skrótów, które zastępowane są podczas parsowania dokumentu. Dobrym przykładem ich zastosowania do opisu interfejsu użytkownika jest przeglądarka Mozilla/Netscape 6. Stosowane przez nią opisy mają postać XML i zawierają definicje bajtów w rodzaju:

```
<!ENTITY back.label "Back">
```

Referencja powyższego bytu może zostać następnie wykorzystana w następujący sposób.

```
<menuitem label=&back.label; />
```

Parser zastąpi ją odpowiednim łańcuchem. Dzięki temu przygotowanie innej wersji językowej aplikacji wymagać będzie jedynie przetłumaczenia łańcucha umieszczonego w definicji bytu. Istnieją też inne zastosowania bytów, bardziej skomplikowane i rzadziej stosowane. Ich opis zawiera wspomniana już specyfikacja standardu XML.

Na tym kończymy nasze wprowadzenie do tematyki definicji typów dokumentów. Wiemy już, w jaki sposób ich używać. W tym celu musimy jednak odpowiednio skonfigurować parser. Najpierw poinformujemy więc fabrykę DocumentBuilderFactory, aby wykorzystywała weryfikację poprawności dokumentów.

```
factory.setValidating(true);
```

Dzięki temu wszystkie obiekty uzyskane za pomocą tej fabryki będą sprawdzać poprawność dokumentów, korzystając z DTD. Najprostszym a przy tym bardzo użytecznym przykładem wykorzystania weryfikacji dokumentów jest pominięcie odstępów przy tworzeniu struktury dokumentu. Rozważmy poniższy fragment dokumentu XML.

```
<font>
<name>Helvetica</name>
<size>36</size>
</font>
```

Parser, który nie będzie korzystał z możliwości weryfikacji dokumentu XML, uzna, że odstępy między elementami font, name i size są także elementami struktury dokumentu, ponieważ nie wie, czy elementy podrzędne mogą być dowolnej postaci (czyli ANY), czy też może poniższej:

```
(name,size)
(#PCDATA,name,size)*
```

Jeśli definicja typu dokumentu określi, że elementy podrzędne są w postaci (name,size), to parser będzie dzięki temu wiedzieć, że odstępy pomiędzy nimi nie są elementami zawierającymi tekst. Wystarczy wywołać metodę

```
factory.setIgnoringElementContentWhitespace(true);
```

i odstępy przestaną się pojawiać jako węzły w strukturze dokumentu. Dodatkowo DTD zapewnia, że w przypadku poprawnego dokumentu węzeł node posiada dwa węzły podrzędne. Nie musimy więc mozolnie analizować ich już za pomocą przedstawionej poniżej pętli:

```
for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    if (child instanceof Element)
    {
        Element childElement = (Element)child;
        if (childElement.getTagName().equals("name")) . . .
        else if (childElement.getTagName().equals("size")) . . .
    }
}
```

Wystarczy jedynie pobrać pierwszy i drugi element podrzędny:

```
Element nameElement = (Element)children.item(0);
Element sizeElement = (Element)children.item(1);
```

Właśnie dlatego definicje typu dokumentu są tak przydatne. Dzięki ich zastosowaniu nie musimy sprawdzać w programie struktury dokumentu. Zadanie to wykona za nas parser, zanim jeszcze zaczniemy korzystać z dokumentu.



Wielu programistów rozpoczynających dopiero przygodę z językiem XML unika na początku korzystania z DTD i tworzy niepotrzebnie rozbudowany kod analizy drzewa DOM. Aby przekonać ich do stosowania DTD, wystarczy zaprezentować pokazane powyżej alternatywne fragmenty kodu.

Jeśli parser zgłosi wystąpienie błędu w strukturze dokumentu, to nasza aplikacja musi odpowiednio zareagować — zapisać błąd w dzienniku, poinformować o nim użytkownika bądź wyrzucić wyjątek i przerwać parsowanie. W tym celu zawsze gdy korzystamy z kontroli poprawności dokumentu, powinniśmy zainstalować obiekt obsługi błędów. Obiekt ten musi implementować interfejs `ErrorHandler`, który zawiera trzy następujące metody:

```
void warning(SAXParseException exception)
void error(SAXParseException exception)
void fatalError(SAXParseException exception)
```

Obiekt obsługi błędów instalujemy za pomocą metody `setErrorHandler` klasy `DocumentBuilder`:

```
builder.setErrorHandler(handler);
```

#### javax.xml.parsers.DocumentBuilder 1.4

- `void setEntityResolver(EntityResolver resolver)`  
instaluje obiekt `EntityResolver` w celu lokalizacji bytów w parsowanych dokumentach XML.
- `void setErrorHandler(ErrorHandler handler)`  
instaluje obiekt obsługi błędów parsowania.

#### org.xml.sax.EntityResolver 1.4

- `public InputSource resolveEntity(String publicID, String systemID)`  
zwraca źródło zawierające dane określone za pomocą identyfikatora lub wartość `null`, jeśli obiekt nie potrafi określić lokalizacji danych. Parametr `publicID` może być wartością `null`, jeśli nie został dostarczony żaden publiczny identyfikator.

#### org.xml.sax.InputSource 1.4

- `InputSource(InputStream in)`
- `InputSource(Reader in)`
- `InputSource(String systemID)`

Tworzą obiekt klasy `InputSource`, korzystając ze strumienia wejściowego, obiektu typu `Reader` bądź identyfikatora systemowego (zwykle w postaci względnego lub bezwzględnego adresu URL).

**API org.xml.sax.ErrorHandler 1.4**

- void fatalError(SAXParseException exception)
- void error(SAXParseException exception)
- void warning(SAXParseException exception)

Metody te należy zastąpić w celu obsługi błędów parsowania.

**API org.xml.sax.SAXParseException 1.4**

- int getLineNumber()
- int getColumnNumber()

zwracają numer wiersza i kolumny, znajdujące się na końcu parsowanej informacji, która spowodowała wystąpienie błędu.

**API javax.xml.parsers.DocumentBuilderFactory 1.4**

- boolean isvalidating()
- void setValidating(boolean value)

sprawdzają lub ustawiają właściwość fabryki polegającą na kontroli poprawności dokumentów. Jeśli właściwość ta posiada wartość true, to parsery pobierane z fabryki sprawdzają poprawność struktury dokumentów XML.

- boolean isIgnoringElementContentWhitespace()
- setIgnoringElementContentWhitespace(boolean value)

sprawdzają lub ustawiają właściwość fabryki polegającą na ignorowaniu odstępów między elementami dokumentu, które nie posiadają mniej więcej zawartości (czyli nie zawierają i elementów i #PCDATA). Jeśli właściwość ta posiada wartość true, to parsery pobierane z fabryki zachowują się właśnie w taki sposób.

### 2.3.2. XML Schema

Ponieważ język XML Schema jest bardziej skomplikowany od składni plików DTD, to omówimy tutaj jedynie jego podstawy. Więcej informacji można znaleźć na stronie <http://www.w3.org/TR/xmlschema-0>.

Do pliku XML Schema odwołujemy się w dokumencie XML dodając odpowiednie atrybuty do elementu korzenia, na przykład:

```
<?xml version="1.0"?>
<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="config.xsd">
</configuration>
```

Deklaracja ta oznacza, że do kontroli poprawności dokumentu XML powinien zostać użyty plik *condif.xsd*. Jeśli dokument używa przestrzeni nazw, to składnia jest nieco bardziej skomplikowana — szczegóły można odnaleźć na wspomnianej wyżej stronie WWW. (Przedrostek *xsi* jest synonimem przestrzeni nazw — więcej na temat przestrzeni nazw w podrozdziale 2.5, „Przestrzenie nazw”).

Schemat definiuje *typ* każdego elementu. Typ ten może być *prosty*, na przykład łańcuch znaków wraz z ograniczeniami formatowania, lub *złożony*. Niektóre typy proste zostały wbudowane w język XML Schema, na przykład

```
xsd:string  
xsd:int  
xsd:boolean
```



Do oznaczenia przestrzeni nazw XSL Schema Definition używamy prefiku *xsd*:. Niektórzy autorzy używają zamiast niego prefiku *xs*.

Możemy również zdefiniować własne typy proste. Poniżej przykład definicji typu wyliczeniowego:

```
<xsd:simpleType name="StyleType">  
  <xsd:restriction base="xsd:string">  
    <xsd:enumeration value="PLAIN" />  
    <xsd:enumeration value="BOLD" />  
    <xsd:enumeration value="ITALIC" />  
    <xsd:enumeration value="BOLD_ITALIC" />  
  </xsd:restriction>  
</xsd:simpleType>
```

Definiując element określamy jego typ:

```
<xsd:element name="name" type="xsd:string">  
<xsd:element name="size" type="xsd:int">  
<xsd:element name="style" type="StyleType">
```

Typ ogranicza zawartość elementu. Na przykład elementy

```
<size>10</size>  
<style>PLAIN</style>
```

są prawidłowe, ale parser odrzuci poniższe elementy

```
<size>default</size>  
<style>SLANTED</style>
```

Typy złożone tworzymy z innych typów, na przykład:

```
<xsd:complexType name="FontType">  
  <xsd:sequence>  
    <xsd:element ref="name"/>  
    <xsd:element ref="size"/>  
    <xsd:element ref="style"/>  
  </xsd:sequence>  
</xsd:complexType>
```

Typ `FontType` stanowi sekwencję elementów `name`, `size` i `style`. Powyższa definicja typu używa atrybutu `ref` w odniesieniu do definicji znajdujących się w innej części tego samego schematu. Definicje możemy również zagnieździć w następujący sposób:

```
<xsd:complexType name="FontType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="xsd:int"/>
    <xsd:element name="style" type="StyleType">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="PLAIN" />
          <xsd:enumeration value="BOLD" />
          <xsd:enumeration value="ITALIC" />
          <xsd:enumeration value="BOLD_ITALIC" />
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

Zwrócmy uwagę na *anonimową definicję typu* dla elementu `style`.

Konstrukcja `xsd:sequence` stanowi odpowiednik konkatenacji w plikach DTD. Natomiast konstrukcja `xsd:choice` jest odpowiednikiem operatora `|`. Na przykład

```
<xsd:complexType name="contactinfo">
  <xsd:choice>
    <xsd:element ref="email"/>
    <xsd:element ref="phone"/>
  <xsd:choice>
</xsd:complexType>
```

jest odpowiednikiem typu `email|phone` zdefiniowanego w pliku DTD.

Powtórzenia elementów możemy określić za pomocą atrybutów `minoccurs` i `maxoccurs`. Na przykład odpowiednikiem typu `item*` zdefiniowanego w pliku DTD będzie

```
<xsd:element name="item" type="..." minoccurs="0" maxoccurs="unbounded">
```

W definicjach `complexType` możemy określać atrybuty dodając elementy `xsd:attribute`:

```
<xsd:element name="size">
  <xsd:complexType>
    .
    .
    <xsd:attribute name="unit" type="xsd:string" use="optional" default="cm"/>
  </xsd:complexType>
</xsd:element>
```

Powyższa definicja jest odpowiednikiem następującej instrukcji DTD:

```
<!ATTLIST size unit CDATA #IMPLIED "cm">
```

Definicje elementów i typów umieszczamy wewnątrz elementu `xsd:schema`:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  .
  .
</xsd:schema>
```

Parsowanie dokumentu XML posiadającego schemat jest podobne do parsowania dokumentu, dla którego dostarczono plik DTD. Występują jednak trzy zasadnicze różnice:

- Należy włączyć obsługę przestrzeni nazw nawet, gdy nie są one używane w dokumencie XML.
 

```
factory.setNamespaceAware(true);
```
- Musimy przygotować fabrykę do obsługi schematów za pomocą następującego fragmentu kodu:
 

```
final String JAXP_SCHEMA_LANGUAGE =
    "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
final String W3C_XML_SCHEMA = "http://www.w3.org/2001/XMLSchema";
factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
```

- Parser *nie pomija odstępów*. Jest to oczywista niedogodność, ale nie ma zgody co do tego, czy wynika ona z błędu implementacji. Kod przedstawiony na listingu 2.4 zawiera obejście tej niedogodności.

### 2.3.3. Praktyczny przykład

W podrozdziale tym przedstawimy praktyczny przykład zastosowania języka XML. W 9. rozdziale książki *Java. Podstawy* przedstawiliśmy klasę `GridBagLayout` — jeden z bardziej przydatnych menedżerów układu komponentów Swing. W praktyce programiści często unikają jego stosowania ze względu na znaczny stopień jego złożoności i dość wysoki nakład pracy potrzebny do jego wykorzystania. Dlatego też zamiast tworzyć w programie rozbudowane i powtarzające się wielokrotnie instrukcje określające układ komponentów, lepiej umieścić odpowiedni opis w pliku tekstowym. Do opisu układu komponentów wykorzystamy język XML i zaprezentujemy sposób parsowania dokumentów opisujących interfejs użytkownika.

Menedżer klasy `GridBagLayout` wykorzystuje siatkę złożoną z wierszy i kolumn przypominającą tabelę HTML. Podobnie więc do opisu tabeli zastosowanego w języku HTML, także i my przedstawimy układ komponentów jako sekwencję wierszy zawierających komórki:

```
<gridbag>
  <row>
    <cell>...</cell>
    <cell>...</cell>
    .
  </row>
  <row>
    <cell>...</cell>
    <cell>...</cell>
    .
  </row>
</gridbag>
```

Plik definicji typu dokumentu `gridbag.dtd` będzie zawierać następujące reguły:

```
<!ELEMENT gridbag (row)*>
<!ELEMENT row (cell)*>
```

Niektóre z komórek mogą rozciągać się na wiele wierszy i kolumn. W przypadku menedżera GridBagLayout efekt taki osiąga się przez nadanie parametrom ograniczeń komórki gridwidth i gridheight wartości większych od 1. Do zapisu tego faktu wykorzystamy atrybuty o takich samych nazwach:

```
<cell gridwidth="2" gridheight="2">
```

Atrybuty zastosujemy też w przypadku innych parametrów ograniczeń, takich jak fill, anchor, gridx, gridy, weightx, weighty, ipadx i ipady. (Nie będziemy obsługiwać ograniczeń insets, ponieważ ich wartość nie jest typem prostym. Rozszerzenie programu o ich obsługę jest jednak dość proste). Na przykład:

```
<cell fill="HORIZONTAL" anchor="NORTH">
```

Dla większości z tych atrybutów określmy takie same wartości domyślne jakie przypisuje im domyślny konstruktor klasy GridBagConstraints:

```
<!ATTLIST cell gridwidth CDATA "1">
<!ATTLIST cell gridheight CDATA "1">
<!ATTLIST cell fill (NONE|BOTH|HORIZONTAL|VERTICAL) "NONE">
<!ATTLIST cell anchor (CENTER|NORTH|NORTHEAST|EAST|SOUTHEAST|SOUTH|SOUTHWEST|WEST|
→NORTHWEST) "CENTER">
.
```

Wartości atrybutów gridx i gridy wymagają szczególnej uwagi, ponieważ ręczna ich specyfikacja jest pracochłonna i łatwo podczas niej o błędy. Określenie wartości tych atrybutów będzie więc opcjonalne:

```
<!ATTLIST cell gridx CDATA #IMPLIED>
<!ATTLIST cell gridy CDATA #IMPLIED>
```

Jeśli ich wartości nie zostaną podane, to program ustali je, posługując się następującą heurystyką: w kolumnie 0 domyślną wartością gridx jest 0. W pozostałych kolumnach wartością tego atrybutu będzie suma wartości gridx i gridwidth dla poprzedniej kolumny. Natomiast domyślna wartość atrybutu gridy będzie zawsze równa numerowi wiersza. Dzięki temu w najczęstszych przypadkach gdy komponent rozciąga się na kilka wierszy, nie musimy określać wartości ograniczeń gridx i gridy. Natomiast w przypadku gdy komponent zajmuje kilka kolumn, należy określić wartość ograniczenia gridx.



Czytelnicy zorientowani w możliwościach menedżera GridBagLayout mogą zastanawiać się, dlaczego nie skorzystaliśmy z mechanizmu RELATIVE i REMAINDER umożliwiającego menedżerowi automatyczne ustalenie wartości parametrów gridx i gridy. Przynajmniejmy, że próbowaliśmy, ale w żaden sposób nie udało nam się uzyskać układu komponentów okna dialogowego wyboru czcionki pokazanego na rysunku 2.4. Przeprowadzona analiza kodu źródłowego klasy GridBagLayout także nie wskazuje, by było to istotnie możliwe.

Program parsuje atrybuty i na ich podstawie ustala ograniczenia siatki komponentów. Na przykład odczyt szerokości siatki wymaga pojedynczej instrukcji:

```
constraints.gridwidth
= Integer.parseInt(e.getAttribute("gridwidth"));
```

Tworząc program, nie musimy martwić się o brakujące atrybuty, ponieważ parser dostarczy nam w takim wypadku wartości domyślnej.

Aby sprawdzić, czy zostały określone wartości atrybutu gridx lub gridy, wywołujemy metodę getAttribute i testujemy, czy zwróciła pusty łańcuch.

```
String value = e.getAttribute("gridy");
if (value.length() == 0) // użyj wartości domyślnej
    constraints.gridy = r;
else
    constraints.gridx = Integer.parseInt(value);
```

Zdecydowaliśmy, że komórka siatki będzie mogła zawierać dowolny obiekt. Pozwoli nam to specyfikować typy, które nie są komponentami, na przykład opisujące granice siatki. Jedynym wymaganiem odnośnie do takiego obiektu będzie zgodność z konwencją JavaBeans: posiadanie przez obiekt domyślnego konstruktora oraz właściwości dostępnych za pomocą odpowiednich metod get i set. (JavaBeans omawiamy bardziej szczegółowo w rozdziale 8.).

Komponent JavaBeans będziemy opisywać za pomocą nazwy klasy i opcjonalnego zestawu właściwości:

```
<!ELEMENT bean (class, property*)>
<!ELEMENT class (#PCDATA)>
```

Właściwość zawierać będzie nazwę i wartość.

```
<!ELEMENT property (name, value)>
<!ELEMENT name (#PCDATA)>
```

Wartość może być liczbą całkowitą, wartością logiczną, łańcuchem lub innym komponentem.

```
<!ELEMENT value (int|string|boolean|bean)>
<!ELEMENT int (#PCDATA)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT boolean (#PCDATA)>
```

Poniżej prezentujemy opis typowego komponentu na przykładzie etykiety klasy JLabel, której właściwość text zawiera łańcuch "Face: ".

```
<bean>
    <class>javax.swing.JLabel</class>
    <property>
        <name>text</name>
        <value><string>Face: </string></value>
    </property>
</bean>
```

Otoczanie łańcucha za pomocą znacznika `<string>` może wydawać się niepotrzebne. Można przecież użyć dla łańcuchów skrótu `#PCDATA`, a znaczniki stosować dla innych typów. Jednak w praktyce oznaczać to będzie zgodę na mieszaną zawartość elementów i osłabienie reguły opisującej element `value` do postaci:

```
<!ELEMENT value (#PCDATA|int|boolean|bean)*>
```

Reguła ta pozwala na dowolne wymieszanie tekstu i znaczników.

Program nadaje wartość właściwościom komponentu, korzystając z jego klasy informacyjnej BeanInfo. Klasa ta umożliwia utworzenie wyliczenia deskryptorów właściwości komponentu, wśród których wyszukujemy właściwość o interesującej nas nazwie i następnie nadajemy jej wartość.

Nasz przykładowy program wczytuje z dokumentu wystarczającą ilość informacji, aby utworzyć i uporządkować komponenty interfejsu użytkownika. Jednak interfejs ten nie będzie działał tak długo, aż nie wyposażymy go w odpowiednie obiekty nasłuchujące zdarzeń. Aby dodać takie obiekty do poszczególnych komponentów, musimy odnaleźć dany komponent. Z tego właśnie powodu do każdego z komponentów dodajemy opcjonalny atrybut typu ID:

```
<!ATTLIST bean id ID #IMPLIED>
```

Poniżej zamieszczamy przykład opisu listy rozwijalnej posiadającej identyfikator.

```
<bean id="face">
    <class>javax.swing.JComboBox</class>
</bean>
```

Przypomnijmy w tym miejscu, że parser sprawdza unikalność identyfikatorów.

Obiekty nasłuchujące możemy zainstalować w następujący sposób:

```
gridbag = new GridBagPane("fontdialog.xml");
setContentPane(gridbag);
JComboBox face = (JComboBox)gridbag.get("face");
face.addListener(listener);
```



W naszym przykładzie używamy języka XML jedynie do opisu układu komponentów interfejsu użytkownika. Instalację obiektów nasłuchujących musi dopisać programista w kodzie programu w języku Java. Ty, Czytelniku, możesz pójść jeszcze dalej i spróbować rozszerzyć opis w języku XML o odpowiedni kod. Najlepszym rozwiązaniem będzie w takim przypadku wykorzystanie języka skryptów, na przykład JavaScript. Jeśli zdecydujesz się na takie rozwiązanie, to zapoznaj się z możliwościami interpretera Rhino (<http://www.mozilla.org/rhino>).

Program z listingu 2.2 pokazuje sposób wykorzystania klasy GridBagPane do automatycznego zainicjowania menedżera układu komponentów. Układ ten określa dokument XML pokazany na listingu 2.4. Rysunek 2.4 pokazuje rezultat wykorzystania opisu. Program inicjuje jedynie zawartość list rozwijalnych (która są zbyt skomplikowane, aby ich opisu dokonać za pomocą mechanizmu właściwości komponentów stosowanego przez klasę GridBagPane) oraz instaluje obiekty nasłuchujące. Klasa GridBagPane, której kod źródłowy pokazuje listing 2.3, parsuje plik XML, tworzy komponenty i ich układ. Listing 2.5 zawiera plik definicji typu dokumentu.

### **Listing 2.2. *read/GridBagTest.java***

```
package read;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;

/**
 * Program prezentujący możliwość wykorzystania dokumentów XML
```

```

* do opisu układu komponentów interfejsu użytkownika.
* @version 1.11 2012-06-03
* @author Cay Horstmann
*/
public class GridBagTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFileChooser chooser = new JFileChooser(".");
                chooser.showOpenDialog(null);
                File file = chooser.getSelectedFile();
                JFrame frame = new FontFrame(file);
                frame.setTitle("GridBagTest");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

<**
* Ramka zawierająca okno dialogowe wyboru czcionki
* opisane przez dokument XML.
* @param filename nazwa pliku zawierającego opis komponentów interfejsu użytkownika
*/
class FontFrame extends JFrame
{
    private GridBagPane gridbag;
    private JComboBox<String> face;
    private JComboBox<String> size;
    private JCheckBox bold;
    private JCheckBox italic;

    @SuppressWarnings("unchecked")
    public FontFrame(File file)
    {
        gridbag = new GridBagPane(file);
        add(gridbag);

        face = (JComboBox<String>) gridbag.get("face");
        size = (JComboBox<String>) gridbag.get("size");
        bold = (JCheckBox) gridbag.get("bold");
        italic = (JCheckBox) gridbag.get("italic");

        face.setModel(new DefaultComboBoxModel<String>(new String[] { "Serif",
            "SansSerif", "Monospaced", "Dialog", "DialogInput" }));
        size.setModel(new DefaultComboBoxModel<String>(new String[] { "8",
            "10", "12", "15", "18", "24", "36", "48" }));

        ActionListener listener = new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                setSample();
            }
        };
    }
}

```

```
        }
    };

    face.addActionListener(listener);
    size.addActionListener(listener);
    bold.addActionListener(listener);
    italic.addActionListener(listener);

    setSample();
    pack();
}

/**
 * Metoda konfigurująca tekst prezentowany
 * za pomocą wybranej czcionki.
 */
public void setSample()
{
    String fontFace = face.getItemAt(face.getSelectedIndex());
    int fontSize = Integer.parseInt(size.getItemAt(size.getSelectedIndex()));
    JTextArea sample = (JTextArea) gridbag.get("sample");
    int fontStyle = (bold.isSelected() ? Font.BOLD : 0)
        + (italic.isSelected() ? Font.ITALIC : 0);

    sample.setFont(new Font(fontFace, fontStyle, fontSize));
    sample.repaint();
}
}
```

---

**Listing 2.3.** *read/GridBagPane.java*

---

```
package read;

import java.awt.*;
import java.beans.*;
import java.io.*;
import java.lang.reflect.*;
import javax.swing.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

/**
 * Panel wykorzystujący plik XML zawierający
 * opis komponentów i ich układu na siatce typu GridBagLayout.
 */
public class GridBagPane extends JPanel
{
    private GridBagConstraints constraints;

    /**
     * Tworzy obiekty klasy GridBagPane.
     * @param filename nazwa pliku XML opisującego komponenty
     * i ich układ
     */
    public GridBagPane(File file)
    {
        setLayout(new GridBagLayout());
        constraints = new GridBagConstraints();
    }
}
```

```

try
{
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    factory.setValidating(true);

    if (file.toString().contains("-schema"))
    {
        factory.setNamespaceAware(true);
        final String JAXP_SCHEMA_LANGUAGE =
            "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
        final String W3C_XML_SCHEMA = "http://www.w3.org/2001/XMLSchema";
        factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
    }

    factory.setIgnoringElementContentWhitespace(true);

    DocumentBuilder builder = factory.newDocumentBuilder();
    Document doc = builder.parse(file);
    parseGridbag(doc.getDocumentElement());
}
catch (Exception e)
{
    e.printStackTrace();
}
}

/**
 * Zwraca komponent o podanej nazwie
 * @param name nazwa komponentu
 * @return komponent o podanej nazwie lub wartość null,
 * jeśli żaden z komponentów panelu nie posiada takiej nazwy
 */
public Component get(String name)
{
    Component[] components = getComponents();
    for (int i = 0; i < components.length; i++)
    {
        if (components[i].getName().equals(name)) return components[i];
    }
    return null;
}

/**
 * Parsuje element panelu.
 * @param e element panelu
 */
private void parseGridbag(Element e)
{
    NodeList rows = e.getChildNodes();
    for (int i = 0; i < rows.getLength(); i++)
    {
        Element row = (Element) rows.item(i);
        NodeList cells = row.getChildNodes();
        for (int j = 0; j < cells.getLength(); j++)
        {
            Element cell = (Element) cells.item(j);
            parseCell(cell, i, j);
        }
    }
}

```

```

    }

    /**
     * Parsuje element komórki.
     * @param e element komórki
     * @param r wiersz komórki
     * @param c kolumna komórki
     */
    private void parseCell(Element e, int r, int c)
    {
        // pobiera atrybuty

        String value = e.getAttribute("gridx");
        if (value.length() == 0) // stosuje wartość domyślną
        {
            if (c == 0) constraints.gridx = 0;
            else constraints.gridx += constraints.gridwidth;
        }
        else constraints.gridx = Integer.parseInt(value);

        value = e.getAttribute("gridy");
        if (value.length() == 0) // stosuje wartość domyślną
        constraints.gridy = r;
        else constraints.gridy = Integer.parseInt(value);

        constraints.gridwidth = Integer.parseInt(e.getAttribute("gridwidth"));
        constraints.gridheight = Integer.parseInt(e.getAttribute("gridheight"));
        constraints.weightx = Integer.parseInt(e.getAttribute("weightx"));
        constraints.weighty = Integer.parseInt(e.getAttribute("weighty"));
        constraints.ipadx = Integer.parseInt(e.getAttribute("ipadx"));
        constraints.ipady = Integer.parseInt(e.getAttribute("ipady"));

        // stosuje refleksję dla uzyskania wartości całkowitych pól statycznych
        Class<GridBagConstraints> cl = GridBagConstraints.class;

        try
        {
            String name = e.getAttribute("fill");
            Field f = cl.getField(name);
            constraints.fill = f.getInt(cl);

            name = e.getAttribute("anchor");
            f = cl.getField(name);
            constraints.anchor = f.getInt(cl);
        }
        catch (Exception ex) // metody refleksji mogą wyrzucić różne wyjątki
        {
            ex.printStackTrace();
        }

        Component comp = (Component) parseBean((Element) e.getFirstChild());
        add(comp, constraints);
    }

    /**
     * Pars element komponentu.
     * @param e element komponentu
     */

```

```

private Object parseBean(Element e)
{
    try
    {
        NodeList children = e.getChildNodes();
        Element classElement = (Element) children.item(0);
        String className = ((Text) classElement.getFirstChild()).getData();
        Class<?> cl = Class.forName(className);

        Object obj = cl.newInstance();

        if (obj instanceof Component) ((Component) obj).setName(e.getAttribute("id"));

        for (int i = 1; i < children.getLength(); i++)
        {
            Node propertyElement = children.item(i);
            Element nameElement = (Element) propertyElement.getFirstChild();
            String propertyName = ((Text) nameElement.getFirstChild()).getData();

            Element valueElement = (Element) propertyElement.getLastChild();
            Object value = parseValue(valueElement);
            BeanInfo beanInfo = Introspector.getBeanInfo(cl);
            PropertyDescriptor[] descriptors = beanInfo.getPropertyDescriptors();
            boolean done = false;
            for (int j = 0; !done && j < descriptors.length; j++)
            {
                if (descriptors[j].getName().equals(propertyName))
                {
                    descriptors[j].getWriteMethod().invoke(obj, value);
                    done = true;
                }
            }
        }
        return obj;
    }
    catch (Exception ex) // metody refleksji mogą wyrzekać różne wyjątki
    {
        ex.printStackTrace();
        return null;
    }
}

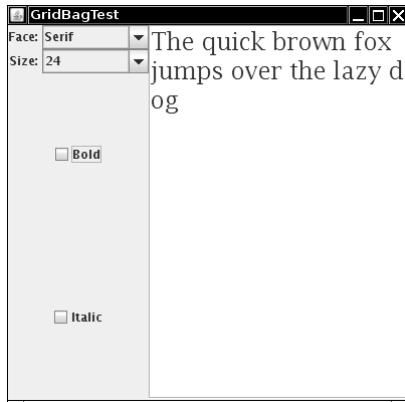
/**
 * Parsuje element wartości.
 * @param e element wartości
 */
private Object parseValue(Element e)
{
    Element child = (Element) e.getFirstChild();
    if (child.getTagName().equals("bean")) return parseBean(child);
    String text = ((Text) child.getFirstChild()).getData();
    if (child.getTagName().equals("int")) return new Integer(text);
    else if (child.getTagName().equals("boolean")) return new Boolean(text);

    else if (child.getTagName().equals("string")) return text;
    else return null;
}

```

**Rysunek 2.4.**

Układ okna dialogowego zdefiniowany za pomocą dokumentu XML

**Listing 2.4.** *read/fontdialog.xml*

```
<?xml version="1.0"?>
<!DOCTYPE gridbag SYSTEM "gridbag.dtd">
<gridbag>
    <row>
        <cell anchor="EAST">
            <bean>
                <class>javax.swing.JLabel</class>
                <property>
                    <name>text</name>
                    <value><string>Face: </string></value>
                </property>
            </bean>
        </cell>
        <cell fill="HORIZONTAL" weightx="100">
            <bean id="face">
                <class>javax.swing.JComboBox</class>
            </bean>
        </cell>
        <cell gridheight="4" fill="BOTH" weightx="100" weighty="100">
            <bean id="sample">
                <class>javax.swing.JTextArea</class>
                <property>
                    <name>text</name>
                    <value><string>The quick brown fox jumps over the lazy
                        ↵dog</string></value>
                </property>
                <property>
                    <name>editable</name>
                    <value><boolean>false</boolean></value>
                </property>
                <property>
                    <name>lineWrap</name>
                    <value><boolean>true</boolean></value>
                </property>
                <property>
                    <name>border</name>
                    <value>
                        <bean>
                            <class>javax.swing.border.EtchedBorder</class>
                        </bean>
                    </value>
                </property>
            </bean>
        </cell>
    </row>
</gridbag>
```

```

        </bean>
    </value>
</property>
</bean>
</cell>
</row>
<row>
<cell anchor="EAST">
<bean>
<class>javax.swing.JLabel</class>
<property>
<name>text</name>
<value><string>Size: </string></value>
</property>
</bean>
</cell>
<cell fill="HORIZONTAL" weightx="100">
<bean id="size">
<class>javax.swing.JComboBox</class>
</bean>
</cell>
</row>
<row>
<cell gridwidth="2" weighty="100">
<bean id="bold">
<class>javax.swing.JCheckBox</class>
<property>
<name>text</name>
<value><string>Bold</string></value>
</property>
</bean>
</cell>
</row>
<row>
<cell gridwidth="2" weighty="100">
<bean id="italic">
<class>javax.swing.JCheckBox</class>
<property>
<name>text</name>
<value><string>Italic</string></value>
</property>
</bean>
</cell>
</row>
</gridbag>
```

**Listing 2.5.** *read/gridbag.dtd*

```

<!ELEMENT gridbag (row)*>
<!ELEMENT row (cell)*>
<!ELEMENT cell (bean)>
<!ATTLIST cell gridx CDATA #IMPLIED>
<!ATTLIST cell gridy CDATA #IMPLIED>
<!ATTLIST cell gridwidth CDATA "1">
<!ATTLIST cell gridheight CDATA "1">
<!ATTLIST cell weightx CDATA "0">
<!ATTLIST cell weighty CDATA "0">
```

```
<!ATTLIST cell fill (NONE|BOTH|HORIZONTAL|VERTICAL) "NONE">
<!ATTLIST cell anchor
  CENTER|NORTH|NORTHEAST|EAST|SOUTHEAST|SOUTH|SOUTHWEST|WEST|NORTHWEST) "CENTER">
<!ATTLIST cell ipadx CDATA "0">
<!ATTLIST cell ipady CDATA "0">

<!ELEMENT bean (class, property*)>
<!ATTLIST bean id ID #IMPLIED>

<!ELEMENT class (#PCDATA)>
<!ELEMENT property (name, value)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT value (int|string|boolean|bean)>
<!ELEMENT int (#PCDATA)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT boolean (#PCDATA)>
```

---

Program może również przetwarzać schemat zamiast pliku DTD, jeśli wybierzemy plik, którego nazwa zawiera łańcuch -schema.

Schemat ten przedstawiony został na listingu 2.6.

---

**Listing 2.6.** *gridbag.xsd*

---

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="gridbag" type="GridBagType"/>

  <xsd:element name="bean" type="BeanType"/>

  <xsd:complexType name="GridBagType">
    <xsd:sequence>
      <xsd:element name="row" type="RowType" minOccurs="0"
        maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="RowType">
    <xsd:sequence>
      <xsd:element name="cell" type="CellType" minOccurs="0"
        maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="CellType">
    <xsd:sequence>
      <xsd:element ref="bean"/>
    </xsd:sequence>
    <xsd:attribute name="gridx" type="xsd:int" use="optional"/>
    <xsd:attribute name="gridy" type="xsd:int" use="optional"/>
    <xsd:attribute name="gridwidth" type="xsd:int" use="optional" default="1" />
    <xsd:attribute name="gridheight" type="xsd:int" use="optional" default="1" />
    <xsd:attribute name="weightx" type="xsd:int" use="optional" default="0" />
    <xsd:attribute name="weighty" type="xsd:int" use="optional" default="0" />
    <xsd:attribute name="fill" use="optional" default="NONE">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
```

```

        <xsd:enumeration value="NONE" />
        <xsd:enumeration value="BOTH" />
        <xsd:enumeration value="HORIZONTAL" />
        <xsd:enumeration value="VERTICAL" />
    </xsd:restriction>
</xsd:simpleType>
</xsd:attribute>
<xsd:attribute name="anchor" use="optional" default="CENTER">
    <xsd:simpleType>
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="CENTER" />
            <xsd:enumeration value="NORTH" />
            <xsd:enumeration value="NORTHEAST" />
            <xsd:enumeration value="EAST" />
            <xsd:enumeration value="SOUTHEAST" />
            <xsd:enumeration value="SOUTH" />
            <xsd:enumeration value="SOUTHWEST" />
            <xsd:enumeration value="WEST" />
            <xsd:enumeration value="NORTHWEST" />
        </xsd:restriction>
    </xsd:simpleType>
</xsd:attribute>
<xsd:attribute name="ipady" type="xsd:int" use="optional" default="0" />
<xsd:attribute name="ipadx" type="xsd:int" use="optional" default="0" />
</xsd:complexType>

<xsd:complexType name="BeanType">
    <xsd:sequence>
        <xsd:element name="class" type="xsd:string"/>
        <xsd:element name="property" type="PropertyType" minOccurs="0"
                     maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID" use="optional" />
</xsd:complexType>

<xsd:complexType name="PropertyType">
    <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="value" type="ValueType"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ValueType">
    <xsd:choice>
        <xsd:element ref="bean"/>
        <xsd:element name="int" type="xsd:int"/>
        <xsd:element name="string" type="xsd:string"/>
        <xsd:element name="boolean" type="xsd:boolean"/>
    </xsd:choice>
</xsd:complexType>
</xsd:schema>

```

Przykład ten ilustruje typowe zastosowanie języka XML. Jest wystarczająco użyteczny, aby opisać nawet bardzo złożone zależności. Dodatkową zaletą jest możliwość automatyzacji przez parser kontroli poprawności dokumentu oraz dostarczania wartości domyślnych atrybutów.

## 2.4. Wyszukiwanie informacji i XPath

Wyszukanie interesującej nas informacji w dokumencie XML może wymagać skomplikowanego przeglądania węzłów drzewa DOM. Zadaniem języka XPath jest uproszczenie dostępu do informacji zapisanej w węzłach wspomnianego drzewa. Założymy na przykład, że mamy następujący dokument XML:

```
<configuration>
    ...
    <database>
        <username>dbuser</username>
        <password>secret</password>
    ...
</database>
</configuration>
```

Nazwę użytkownika możemy pobrać z powyższego pliku za pomocą następującego wyrażenia XPath:

```
/configuration/database/username
```

Jest to rozwiązanie dużo prostsze niż z użyciem samego DOM, które wymaga:

- 1.** Pobrania węzła dokumentu.
- 2.** Przeglądania jego węzłów podrzędnych.
- 3.** Zlokalizowania elementu database.
- 4.** Pobrania pierwszego elementu podzielnego, czyli elementu username.
- 5.** Pobrania pierwszego elementu podzielnego tego elementu, czyli węzła Text.
- 6.** Pobrania danych tego węzła.

Wyrażenie XPath może opisywać *zbiór węzłów* w dokumencie XML. Na przykład wyrażenie XPath:

```
/gridbag/row
```

opisuje zbiór wszystkich elementów row, które są elementami podzielnymi elementu korzenia gridbag. Poszczególne elementy tego zbioru pobieramy za pomocą operatora [ ]. Na przykład:

```
/gridbag/row[1]
```

reprezentuje pierwszy element row (wartości indeksu rozpoczynają się od 1).

Za pomocą operatora @ pobieramy wartości atrybutów. Wyrażenie XPath:

```
/gridbag/row[1]/cell[1]/@anchor
```

opisuje atrybut anchor pierwszej komórki w pierwszym rzędzie. Wyrażenie:

```
gridbag/row/cell/@anchor
```

reprezentuje wszystkie węzły atrybutów anchor elementów cell wewnętrznych elementów row, które są węzłami podzielnymi korzenia gridbag.

Język XPath udostępnia szereg użytecznych funkcji. Na przykład:

```
count(/gridbag/row)
```

zwraca liczbę węzłów podrzędnych row węzła korzenia gridbag. W języku XPath można tworzyć wiele złożonych wyrażeń — więcej informacji na ten temat zawarto w specyfikacji dostępnej pod adresem <http://www.w3c.org/TR/xpath> lub we wprowadzeniu na stronie <http://www.zvon.org/xsl/XPathTutorial/General/examples.html>.

Java SE 5.0 dostarcza interfejsu programowego umożliwiającego posługiwanie się wyrażeniami XPath. Najpierw tworzymy obiekt XPath za pomocą fabryki XPathFactory:

```
XPathFactory xpfactory = XPathFactory.newInstance();
path = xpfactory.newXPath();
```

Wartość wyrażenia XPath wyznaczamy, wywołując metodę evaluate:

```
String username = path.evaluate("/configuration/database/username", doc);
```

Tego samego obiektu XPath możemy użyć do wyznaczania wartości wielu wyrażeń.

Przedstawiona wyżej wersja metody evaluate zwraca wynik w postaci łańcucha znaków. Jest odpowiednia do pobierania tekstu, na przykład węzła username z poprzedniego przykładu. Jeśli wyrażenie XPath zwraca zbiór węzłów, to używamy następującego wywołania:

```
NodeList nodes = (NodeList) path.evaluate("/gridbag/row", doc,
XPathConstants.NODESET);
```

Jeśli wynik wyrażenia jest pojedynczym węzłem, to używamy stałej XPathConstants.NODE:

```
Node node = (Node) path.evaluate("/gridbag/row[1]", doc,
XPathConstants.NODE);
```

Jeśli wynik jest liczbą, to stosujemy stałą XPathConstants.NUMBER:

```
int count = ((Number) path.evaluate("count(/gridbag/row)", doc,
XPathConstants.NUMBER)).intValue();
```

Wyszukiwania nie musimy zaczynać od korzenia dokumentu. Możemy rozpocząć je od dowolnego węzła lub listy węzłów. Na przykład, jeśli dysponujemy węzłem, który jest wynikiem poprzedniego wyrażenia, to możemy wywołać:

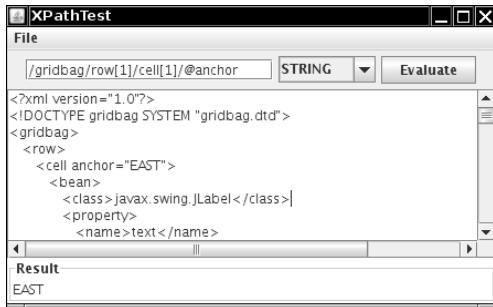
```
result = path.evaluate(expression, node);
```

Program przedstawiony na listingu 2.7 demonstruje wyznaczanie wartości wyrażeń XPath. Po załadowaniu pliku XML możemy wpisać wyrażenie XPath, wybrać jego typ i kliknąć przycisk *Evaluate*. Wynik wyrażenia zostaje wyświetlony w dolnej części ramki (patrz rysunek 2.5).

#### **Listing 2.7. xpath/XPathTester.java**

```
package xpath;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.nio.file.*;
```

**Rysunek 2.5.****Wyznaczanie wyrażeń XPath**

```

import javax.swing.*;
import javax.swing.border.*;
import javax.xml.namespace.*;
import javax.xml.parsers.*;
import javax.xml.xpath.*;
import org.w3c.dom.*;
import org.xml.sax.*;

/**
 * Program wyznaczający wyrażenia XPath.
 * @version 1.01 2007-06-25
 * @author Cay Horstmann
 */
public class XPathTester
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new XPathFrame();
                frame.setTitle("XPathTest");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

/**
 * Ramka prezentująca dokument XML oraz posiadającą pola
 * umożliwiające wpisanie wyrażenia i wyświetlenie wyniku.
 */
class XPathFrame extends JFrame
{
    private DocumentBuilder builder;
    private Document doc;
    private XPath path;
    private JTextField expression;
    private JTextField result;
    private JTextArea docText;
    private JComboBox<String> typeCombo;

    public XPathFrame()
    {
        ...
    }
}

```

```

{
    JMenu fileMenu = new JMenu("File");
    JMenuItem openItem = new JMenuItem("Open");
    openItem.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            openFile();
        }
    });
    fileMenu.add(openItem);

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

    JMenuBar menuBar = new JMenuBar();
    menuBar.add(fileMenu);
    setJMenuBar(menuBar);

    ActionListener listener = new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            evaluate();
        }
    };
    expression = new JTextField(20);
    expression.addActionListener(listener);
    JButton evaluateButton = new JButton("Evaluate");
    evaluateButton.addActionListener(listener);

    typeCombo = new JComboBox<String>(new String[] {
        "STRING", "NODE", "NODESET", "NUMBER", "BOOLEAN" });
    typeCombo.setSelectedItem("STRING");

    JPanel panel = new JPanel();
    panel.add(expression);
    panel.add(typeCombo);
    panel.add(evaluateButton);
    docText = new JTextArea(10, 40);
    result = new JTextField();
    result.setBorder(new TitledBorder("Result"));

    add(panel, BorderLayout.NORTH);
    add(new JScrollPane(docText), BorderLayout.CENTER);
    add(result, BorderLayout.SOUTH);

    try
    {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

```

```
        builder = factory.newDocumentBuilder();
    }
    catch (ParserConfigurationException e)
    {
        JOptionPane.showMessageDialog(this, e);
    }

    XPathFactory xpfactory = XPathFactory.newInstance();
    path = xpfactory.newXPath();
    pack();
}

< /**
 * Otwiera plik i ładuje dokument.
 */
public void openFile()
{
    JFileChooser chooser = new JFileChooser();
    chooser.setCurrentDirectory(new File("xpath"));

    chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
    {
        public boolean accept(File f)
        {
            return f.isDirectory() || f.getName().toLowerCase().endsWith(".xml");
        }

        public String getDescription()
        {
            return "XML files";
        }
    });
    int r = chooser.showOpenDialog(this);
    if (r != JFileChooser.APPROVE_OPTION) return;
    File file = chooser.getSelectedFile();
    try
    {
        docText.setText(new String(Files.readAllBytes(file.toPath())));
        doc = builder.parse(file);
    }
    catch (IOException e)
    {
        JOptionPane.showMessageDialog(this, e);
    }
    catch (SAXException e)
    {
        JOptionPane.showMessageDialog(this, e);
    }
}

public void evaluate()
{
    try
    {
        String typeName = (String) typeCombo.getSelectedItem();
        QName returnType = (QName) XPathConstants.class.getField(typeName).get(null);
        Object evalResult = path.evaluate(expression.getText(), doc, returnType);
        if (typeName.equals("NODESET"))

```

```
        NodeList list = (NodeList) evalResult;
        StringBuilder builder = new StringBuilder();
        builder.append("{");
        for (int i = 0; i < list.getLength(); i++)
        {
            if (i > 0) builder.append(", ");
            builder.append("") + list.item(i));
        }
        builder.append("}");
        result.setText("") + builder);
    }
    else result.setText("") + evalResult);
}
catch (XPathExpressionException e)
{
    result.setText("") + e);
}
catch (Exception e) //wyjatek refleksji
{
    e.printStackTrace();
}
}
```

API javax.xml.xpath.XPathFactory 5.0

- static XPathFactory newInstance()  
zwraca instancję fabryki XPathFactory tworzącej obiekty XPath.
  - XPath newXPath()  
tworzy obiekt XPath umożliwiający wyznaczanie wartości wyrażeń XPath.

API javax.xml.xpath.XPath 5.0

- `String evaluate(String expression, Object startingPoint)`  
wyznacza wartość wyrażenia, rozpoczynając od podanego punktu `startingPoint`.  
Punkt ten może być węzłem lub listą węzłów. Jeśli wynik wyrażenia jest węzłem  
lub zbiorem węzłów, to łańcuch zwrócony przez metodę zawiera dane wszystkich  
tekstowych węzłów podrzędnych.
  - `Object evaluate(String expression, Object startingPoint, QName resultType)`  
wyznacza wartość wyrażenia, rozpoczynając od podanego punktu `startingPoint`.  
Punkt ten może być węzłem lub listą węzłów. Parametr `resultType` jest jedną  
ze stałych `STRING`, `NODE`, `NODESET`, `NUMBER` lub `BOOLEAN` należącymi do klasy  
`XPathConstants`. Metoda zwraca obiekt klasy `String`, `Node`, `NodeList`, `Number`  
lub `Boolean`.

## 2.5. Przestrzeń nazw

Zastosowanie pakietów w języku Java pozwala uniknąć konfliktów nazw. Programiści mogą używać tych samych nazw dla różnych klas pod warunkiem, że nie znajdują się one w tym samym pakiecie. Język XML posiada podobny mechanizm *przestrzeni nazw* dla elementów i atrybutów.

Przestrzeń nazw określana jest za pomocą identyfikatora URI (*Uniform Resource Identifier*), na przykład:

```
http://www.w3.org/2001/XMLSchema  
uuid:1c759aed-b748-475c-ab68-10679700c4f2  
urn:com:books-r-us
```

Najczęściej spotykaną formą identyfikatora przestrzeni nazw jest adres URL. Zwróćmy jednak uwagę, że w takim przypadku adres URL wykorzystywany jest jako łańcuch identyfikatora, a nie jako adres określający położenie dokumentu. Na przykład poniższe identyfikatory przestrzeni nazw są uważane za *różne*

```
http://www.horstmann.com/corejava  
http://www.horstmann.com/corejava/index.html
```

mimo że serwer internetowy zwróciłby dla nich ten sam dokument.

Adres URL używany w roli identyfikatora przestrzeni nazw nie musi w ogóle opisywać położenia jakiegokolwiek dokumentu, ponieważ parser XML nie wykorzystuje go w tym celu. Jedynie jako pomoc dla programistów, którym może sprawiać trudność zrozumienie znaczenia nowej przestrzeni nazw, zwykło się umieszczać dokument wyjaśniający tę kwestię pod danym adresem. Na przykład jeśli wpiszemy w przeglądarce adres URL reprezentujący przestrzeń nazw dla XML Schema ([www.w3.org/2001/XMLSchema](http://www.w3.org/2001/XMLSchema)), to otrzymamy dokument opisujący ten standard.

Adresy URL wykorzystywane są w roli identyfikatorów przestrzeni nazw głównie ze względu na ich unikalność. Wybierając rzeczywisty adres URL, mamy zagwarantowaną jego unikalność przez system nazw domen internetowych. Unikalność adresu w obrębie danej domeny powinna natomiast zagwarantować organizacja będąca w jej posiadaniu. Identyfikatory przestrzeni nazw dokumentów XML wykorzystują więc tę samą zasadę, która leży u podstaw zastosowania odwróconych nazw domen w nazwach pakietów języka Java.

Oczywiście długie identyfikatory przestrzeni nazw są pożądane ze względu na swoją unikalność, ale mniej praktyczne w użyciu. Dlatego też język Java dysponuje mechanizmem import, który wymaga określenia długich nazw pakietów tylko raz, a później możemy już korzystać z krótkich nazw klas zawartych w tych pakietach. Język XML posiada podobny mechanizm:

```
<element xmlns="URIprzestrzeninazw">  
    element podrzędny  
</element>
```

Podany w ten sposób element i jego elementy podrzędne należą odtąd do określonej przestrzeni nazw.

Element podrzędny może posiadać własną przestrzeń nazw, co pokazano w poniższym przykładzie:

```
<element xmlns="URIprzestrzeninazw1">
    <elementpodrzędny xmlns="URIprzestrzeninazw2">
        element podrzędny niższego rzędu
    </elementpodrzędny>
    pozostałe elementy podrzędne
</element>
```

W przykładzie tym określono, że element podrzędny i jego elementy podrzędne należą do osobnej przestrzeni nazw.

Zaprezentowany powyżej mechanizm sprawdza się jedynie w przypadku pojedynczych przestrzeni nazw lub przestrzeni, które są zagnieżdżone w sobie w naturalny sposób. W pozostałych przypadkach lepiej skorzystać z innego mechanizmu, który nie posiada swojego odpowiednika w języku Java. Przestrzeń nazw możemy określić za pomocą *prefiksu* — krótkiego identyfikatora wybranego dla danego dokumentu. Poniżej typowy przykład jego zastosowania — prefiks xsd w pliku XML Schema:

```
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    <xsd:element name="gridbag" type="GridBagType"/>
    .
    .
    </xsd:schema>
```

Atrybut:

```
xmlns:prefix="namespaceURI"
```

definiuje przestrzeń nazw i prefiks. W naszym przykładzie prefiksem jest łańcuch xsd. W ten sposób xsd:schema oznacza „schema w przestrzeni nazw <http://www.w3.org/2001/XMLSchema>”.



Jedynie elementy podrzędne dziedziczą przestrzeń nazw po swoich elementach nadrzędnych. Atrybuty, których nie poprzedzono jawnie prefiksem, nie należą do żadnej przestrzeni nazw. Rozważmy poniższy przykład:

```
<configuration xmlns="http://www.horstmann.com/corejava"
    xmlns:si="http://www.bipm.fr/enus/3_SI/si.html"
    >
    <size value="210" si:unit="mm"/>
    .
    .
    </configuration>
```

Elementy configuration i size należą do przestrzeni nazw określonej przez identyfikator URI <http://www.horstmann.com/corejava>, a atrybut si:unit do przestrzeni nazw o identyfikatorze [http://www.bipm.fr/enus/3\\_SI/si.html](http://www.bipm.fr/enus/3_SI/si.html). Jednak atrybut value nie należy do żadnej przestrzeni nazw.

Sposobem korzystania z przestrzeni nazw przez parser możemy sterować. Domyślnie implementacja parsera DOM dostarczana w JDK nie wykorzystuje przestrzeni nazw.

Aby włączyć obsługę przestrzeni nazw, musimy wywołać metodę setNamespaceAware fabryki DocumentBuilderFactory:

```
factory.setNamespaceAware(true);
```

Odtąd wszystkie parsery pobrane z fabryki factory będą korzystać z przestrzeni nazw. Każdy węzeł utworzonego przez nie drzewa dokumentu posiadać będzie trzy właściwości:

- pełną nazwę z prefiksem, zwracaną przez metody `getnodeName`, `getTagName` itd.,
- identyfikator URI przestrzeni nazw zwracany przez metodę `getNamespaceURI`.
- nazwę lokalną pozbawioną prefiku lub przestrzeni nazw i zwracaną przez metodę `getLocalName`,

A oto przykład. Założymy, że parser napotkał poniższy element dokumentu:

```
<xsd:schema
  xmlns:xsl="http://www.w3.org/2001/XMLSchema">
```

Właściwości węzła drzewa utworzonego dla reprezentacji tego elementu będą następujące:

- pełna nazwa = `xsd:schema`,
- identyfikator URI przestrzeni nazw = `http://www.w3.org/2001/XMLSchema`,
- nazwa lokalna = `schema`.



Jeśli parser nie korzysta z przestrzeni nazw, to metody `getLocalName` i `getNamespaceURI` zwracają wartość null.

#### org.w3c.dom.Node 1.4

- `String getLocalName()`  
zwraca lokalną nazwę (bez prefiku) lub wartość null, gdy parser nie korzysta z przestrzeni nazw.
- `String getNameSpaceURI()`  
zwraca identyfikator URI przestrzeni nazw lub wartość null, jeśli węzeł nie należy do żadnej przestrzeni nazw lub parser nie korzysta z przestrzeni nazw.

#### javax.xml.parsers.DocumentBuilderFactory 1.4

- `boolean isNamespaceAware()`
- `void setNamespaceAware(boolean value)`  
sprawdzają lub nadają wartość właściwości fabryki polegającej na tym, że tworzone przez nią parsey korzystają z przestrzeni nazw.

## 2.6. Parsery strumieniowe

Parser DOM wczytuje cały dokument XML i prezentuje jego strukturę w postaci drzewa. Rozwiążanie takie sprawdza się w większości typowych aplikacji. Może jednak okazać się mało efektywne, jeśli dokument jest sporych rozmiarów lub algorytm jego przetwarzania jest

na tyle prosty, że wymaga jedynie analizy elementów dokumentu „w locie” i nie potrzebuje do tego informacji o pełnej strukturze dokumentu. W takich przypadkach lepszym rozwiązaniem będzie wykorzystanie parsera strumieniowego.

W kolejnych podrozdziałach omówimy parsery strumieniowe dostępne w bibliotekach języka Java: parser SAX zawiadamiający o zdarzeniach za pomocą wywołań zwrotnych oraz nowszy parser StAX (wprowadzony w Java SE 6) udostępniający iterator do przeglądania zdarzeń parsowania. To ostatnie rozwiązanie zwykle okazuje się w praktyce nieco wygodniejsze.

## 2.6.1. Wykorzystanie parsera SAX

Parser SAX generuje zdarzenia w trakcie czytania dokumentu XML i nie tworzy żadnej reprezentacji dokumentu. Wykreowanie tej ostatniej zależy jedynie od wybranego sposobu obsługi zdarzeń parsera. W rzeczywistości parser DOM posługuje się właśnie parserem SAX i tworzy drzewo reprezentacji dokumentu na podstawie zdarzeń generowanych przez parser SAX.

Korzystając z parsera SAX, musimy dostarczyć obiekt definiujący akcje dla różnych zdarzeń generowanych przez parser. Interfejs `ContentHandler` definiuje metody wywoływanie zwrotnie przez parser. Do najważniejszych z nich należą:

- `startElement/endElement` wywoływanie, gdy parser napotka znacznik początkowy lub końcowy elementu,
- `characters` wywoływanie, gdy parser napotka dane znakowe,
- `startDocument/endDocument` wywoływanie tylko raz na początku i na końcu przetwarzania dokumentu.

Na przykład parsując poniższy fragment dokumentu:

```
<font>
    <name>Helvetica</name>
    <size units="pt">36</size>
</font>
```

parser wywoła zwrotnie następujące metody:

- 1.** `startElement` dla elementu o nazwie `font`.
- 2.** `startElement` dla elementu o nazwie `name`.
- 3.** `characters` dla danych `Helvetica`.
- 4.** `endElement` dla elementu o nazwie `name`.
- 5.** `startElement` dla elementu o nazwie `size` i atrybutu `units="pt"`.
- 6.** `characters` dla danych `36`.
- 7.** `endElement` dla elementu o nazwie `size`.
- 8.** `endElement` dla elementu o nazwie `font`.

Dostarczony przez nas obiekt obsługi zdarzeń powinien zastąpić te metody własną implementacją, która wykona odpowiednie akcje podczas parsowania dokumentu. Przykładowy program, który zamieszczamy na końcu tego podrozdziału, wyświetla wszystkie hiperłącza

postaci <a href=". . ."> zawarte w pliku HTML. W tym przypadku obiekt obsługi zdarzeń zastępuje jedynie metodę startElement i sprawdza, czy dany element posiada nazwę a i atrybut o nazwie href. Taki sposób działania może być na przykład wykorzystywany przez program analizujący powiązania między stronami internetowymi.



Niestety większość stron w języku HTML różni się na tyle istotnie od języka XML, że nasz przykładowy program nie będzie w stanie ich parsować. Jednak jak już wcześniej wspomnieliśmy, konsorcjum W3C (*World Wide Web Consortium*) zaleca do tworzenia stron internetowych użycie języka XHTML, nowego dialekta HTML, który jest obsługiwany przez najnowsze wersje przeglądarek i także zgodny z językiem XML. Ponieważ strony internetowe konsorcjum W3C utworzono właśnie przy użyciu języka XHTML, to możemy wykorzystać je do przetestowania naszego programu. Jeśli uruchomimy go na przykład w następujący sposób:

```
java SAXTest http://www.w3c.org/MarkUp
```

to zobaczymy listę wszystkich hiperłączy znajdujących się na tej stronie.

Nasz przykładowy program stanowi bardzo dobrą ilustrację sytuacji, w której wykorzystujemy parser SAX. Poszukując elementu o nazwie a, nie musimy analizować jego kontekstu, ani tworzyć struktury drzewiastej dokumentu.

Parser SAX uzyskujemy w następujący sposób:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
```

Możemy teraz przetwarzać dokument za jego pomocą:

```
parser.parse(source, handler);
```

Parametr source może być plikiem, adresem URL bądź strumieniem wejściowym. Parametr handler musi należeć do klasy pochodnej klasy DefaultHandler. Definiuje ona „puste” wersje metod należących do czterech interfejsów:

```
ContentHandler
DTDHandler
EntityResolver
ErrorHandler
```

W przykładowym programie zastępujemy jedynie metodę startElement interfejsu ContentHandler, aby wykrywać wystąpienie elementu o nazwie a i atrybutu href:

```
DefaultHandler handler = new
DefaultHandler()
{
    public void startElement(String namespaceURI,
        String lname, String qname, Attributes attrs)
        throws SAXException
    {
        if (lname.equalsIgnoreCase("a") && attrs != null)
        {
            for (int i = 0; i < attrs.getLength(); i++)
            {
                String aname = attrs.getLocalName(i);
                if (aname.equalsIgnoreCase("href"))
                    System.out.println(attrs.getValue(i));
            }
        }
    }
}
```

} ; }

Metoda startElement posiada trzy parametry opisujące nazwę elementu. Parametr qname przekazuje jej pełną nazwę elementu w postaci prefix:localname. Jeśli parser korzysta z przestrzeni nazw, to parametry namespaceURI i lname przekazują identyfikator przestrzeni nazw i lokalną nazwę elementu.

Podobnie jak w przypadku parsera DOM, także dla parsera SAX wykorzystanie przestrzeni nazw jest domyślnie wyłączone. Możemy aktywować je za pomocą metody setNamespaceAware fabryki SAXParserFactory:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setNamespaceAware(true);
SAXParser saxParser = factory.newSAXParser();
```

W tym programie musimy poradzić sobie z jeszcze jednym typowym problemem. Plik XHTML rozpoczyna się od znacznika zawierającego referencję definicji DTD i oczywiście parser będzie chciał ją załadować. Co zrozumiałe, W3C nie ma zamieru obsługiwać miliardów żądań dotyczących plików takich jak [www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd](http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd). Początkowo żądania dotyczące tych plików były odrzucane, a obecnie są obsługiwane w żółwim tempie. Jeśli sprawdzenie poprawności dokumentu nie jest potrzebne, wystarczy użyć następującego wywołania:

```
factory.setFeature("http://apache.org/xml/features/nonvalidating/load-external-dtd",  
false);
```

Listing 2.8 prezentuje kod źródłowy przykładowego programu. W dalszej części rozdziału pokażemy jeszcze inny, interesujący przykład wykorzystania parsera SAX. Jednym ze sposobów przekształcenia pliku na format XML jest wykorzystanie zdarzeń generowanych przez parser SAX.Więcej informacji na ten temat zawiera podrozdział 2.8. „Przekształcenia XML”.

**Listing 2.8.** *sax/SAXTest.java*

```
package sax;

import java.io.*;
import java.net.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

/**
 * Program demonstrujący wykorzystanie parsera SAX.
 * Wyświetla wszystkie hiperłącza umieszczone na stronie XHTML.
 * Uruchomienie programu: java sax.SAXTest url
 * @version 1.00 2001-09-29
 * @author Cay Horstmann
 */
public class SAXTest
{
    public static void main(String[] args) throws Exception
    {
        String url;
```

```
if (args.length == 0)
{
    url = "http://www.w3c.org";
    System.out.println("Using " + url);
}
else url = args[0];

DefaultHandler handler = new DefaultHandler()
{
    public void startElement(String namespaceURI, String lname, String qname,
                             Attributes attrs)
    {
        if (lname.equals("a") && attrs != null)
        {
            for (int i = 0; i < attrs.getLength(); i++)
            {
                String aname = attrs.getLocalName(i);
                if (aname.equals("href")) System.out.println(attrs.getValue(i));
            }
        }
    }
};

SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setNamespaceAware(true);
factory.setFeature("http://apache.org/xml/features/nonvalidating/
➥load-external-dtd", false);
SAXParser saxParser = factory.newSAXParser();
InputStream in = new URL(url).openStream();
saxParser.parse(in, handler);
}
}
```

---

**API** javax.xml.parsers.SAXParserFactory 1.4

- static SAXParserFactory newInstance()  
zwraca instancję klasy SAXParserFactory.
- SAXParser newSAXParser()  
zwraca instancję klasy SAXParser.
- boolean isNamespaceAware()
- void setNamespaceAware(boolean value)  
sprawdzają lub nadają wartość właściwości fabryki polegającej na tym, że tworzone przez nią parsery korzystają z przestrzeni nazw.
- boolean isValidating()
- void setValidating(boolean value)  
sprawdzają lub ustawiają właściwość fabryki polegającą na kontroli poprawności dokumentów. Jeśli właściwość ta posiada wartość true, to parsery pobierane z fabryki sprawdzają poprawność struktury dokumentów XML.

**API javax.xml.parsers.SAXParser 1.4**

- void parse(File f, DefaultHandler handler)
- void parse(String url, DefaultHandler handler)
- void parse(InputStream in, DefaultHandler handler)

parsują dokument XML pobrany z podanego pliku, adresu URL bądź strumienia wejściowego i przekazują zdarzenia parsowania obiektowi handler.

**API org.xml.sax.ContentHandler 1.4**

- void startDocument()
- void endDocument()

Metody wywoływane na początku i końcu przetwarzania dokumentu.

- void startElement(String uri, String lname, String qname, Attributes attr)
- void endElement(String uri, String lname, String qname)

Metody wywoływane po napotkaniu znacznika początkowego i końcowego elementu.

*Parametry:*   uri       identyfikator URI przestrzeni nazw (jeśli parser wykorzystuje przestrzeń nazw),  
                  lname      lokalna nazwa elementu bez prefiksu (jeśli parser wykorzystuje przestrzeń nazw),  
                  qname      nazwa elementu (jeśli parser nie wykorzystuje przestrzeni nazw) lub pełna nazwa elementu poprzedzona prefiksem.

- void characters(char[] data, int start, int length)

Metoda wywoływana, gdy parser napotka dane znakowe.

*Parametry:*   data      tablica danych znakowych,  
                  start     indeks pierwszego znaku w tablicy, który należy do raportowanych danych znakowych,  
                  length    długość danych znakowych.

**API org.xml.sax.Attributes 1.4**

- int getLength()

zwraca liczbę atrybutów w danej kolekcji.

- String getLocalName(int index)

zwraca lokalną nazwę (bez prefiksu) dla atrybutu o podanym indeksie lub pusty łańcuch, gdy parser nie wykorzystuje przestrzeni nazw.

- `String getURI(int index)`  
zwraca identyfikator URI przestrzeni nazw dla atrybutu o podanym indeksie lub pusty łańcuch, gdy węzeł atrybutu nie należy do żadnej przestrzeni nazw lub parser nie wykorzystuje przestrzeni nazw.
- `String getQName(int index)`  
zwraca pełną nazwę (z prefiksem) dla atrybutu o podanym indeksie lub pusty łańcuch, jeśli parser nie przekazał pełnej nazwy.
- `String getValue(int index)`
- `String getValue(String qname)`
- `String getValue(String uri, String lname)`  
zwracają wartość atrybutu o podanym indeksie, pełnej nazwie bądź określonego przez identyfikator przestrzeni nazw i nazwę lokalną. Zwracają wartość null, jeśli atrybut taki nie istnieje.

## 2.6.2. Wykorzystanie parsera StAX

W przypadku parsera StAX nie musimy instalować własnej obsługi zdarzeń parsowania, lecz przeglądamy zdarzenia parsowania za pomocą poniższej pętli:

```
InputStream in = url.openStream();
XMLInputFactory factory = XMLInputFactory.newInstance();
XMLStreamReader parser = factory.createXMLStreamReader(in);
while (parser.hasNext())
{
    int event = parser.next();
    wywołanie metod parsera udostępniających szczegółowe zdarzenia
}
```

Na przykład parsując poniższy fragment:

```
<font>
    <name>Helvetica</name>
    <size units="pt">36</size>
</font>
```

parser StAX udostępni nam następujące zdarzenia:

- 1. START\_ELEMENT**, nazwa elementu: font;
- 2. CHARACTERS**, zawartość: odstęp;
- 3. START\_ELEMENT**, nazwa elementu: name;
- 4. CHARACTERS**, zawartość: Helvetica;
- 5. END\_ELEMENT**, nazwa elementu: name;
- 6. CHARACTERS**, zawartość: odstęp;
- 7. START\_ELEMENT**, nazwa elementu: size;

- 8.** CHARACTERS, zawartość: 36;
- 9.** END\_ELEMENT, nazwa elementu: size;
- 10.** CHARACTERS, zawartość: odstęp;
- 11.** END\_ELEMENT, nazwa elementu: font.

Aby przeanalizować wartości atrybutu, wywołujemy odpowiednią metodę klasy XMLStreamReader. Na przykład wywołanie

```
String units = parser.getAttributeValue(null, "units");
```

zwraca wartość atrybutu units bieżącego elementu.

Domyślnie obsługa przestrzeni nazw jest tym razem włączona. Możemy ją wyłączyć, modyfikując odpowiednią właściwość fabryki:

```
XMLInputFactory factory = XMLInputFactory.newInstance();
factory.setProperty(XMLInputFactory.IS_NAMESPACE_AWARE, false);
```

Na listingu 2.9 przedstawiliśmy wersję programu z listingu 2.8 zaimplementowaną przy użyciu parsera StAX. Jak łatwo zauważyc, wersja ta jest prostsza, ponieważ nie zawiera obsługi zdarzeń.

#### **Listing 2.9.** stax/StAXTest.java

```
package stax;

import java.io.*;
import java.net.*;
import javax.xml.stream.*;

/**
 * Program demonstrujący wykorzystanie parsera StAX.
 * Wyświetla wszystkie hiperłącza umieszczone na stronie XHTML.
 * Uruchomienie programu: java stax.StAXTest url
 * @author Cay Horstmann
 * @version 1.0 2007-06-23
 */
public class StAXTest
{
    public static void main(String[] args) throws Exception
    {
        String urlString;
        if (args.length == 0)
        {
            urlString = "http://www.w3c.org";
            System.out.println("Using " + urlString);
        }
        else urlString = args[0];
        URL url = new URL(urlString);
        InputStream in = url.openStream();
        XMLInputFactory factory = XMLInputFactory.newInstance();
        XMLStreamReader parser = factory.createXMLStreamReader(in);
        while (parser.hasNext())
        {
            int event = parser.next();
```

```
if (event == XMLStreamConstants.START_ELEMENT)
{
    if (parser.getLocalName().equals("a"))
    {
        String href = parser.getAttributeValue(null, "href");
        if (href != null)
            System.out.println(href);
    }
}
}
```

API javax.xml.stream.XMLInputFactory 6

- static XMLInputFactory newInstance()  
zwraca instancję klasy XMLInputFactory.
  - void setProperty(String name, Object value)  
nadaje wartość wybranej właściwości fabryki lub wyrzuca wyjątek  
`IllegalArgumentException`, jeśli właściwość nie jest obsługiwana lub nie może  
otrzymać podanej wartości. Implementacja Java obsługuje następujące właściwości  
typu Boolean:
    - "`javax.xml.stream.isValidating`"  
Gdy ma wartość `false` (domyślnie), poprawność dokumentu nie jest  
sprawdzana. Właściwość ta nie jest wymagana przez specyfikację.
    - "`javax.xml.stream.isNamespaceAware`"  
Gdy ma wartość `true` (domyślnie), przestrzenie nazw są obsługiwane.  
Właściwość ta nie jest wymagana przez specyfikację.
    - "`javax.xml.stream.isCoalescing`"  
Gdy ma wartość `false` (domyślnie), sąsiednie dane znakowe nie są scalane.
    - "`javax.xml.stream.isReplacingEntityReferences`"  
Gdy ma wartość `true` (domyślnie), encje znakowe zostają zastąpione  
rzeczywistymi znakami i są raportowane przez parser jako znaki.
    - "`javax.xml.stream.isSupportingExternalEntities`"  
Gdy ma wartość `true` (domyślnie), obsługiwane są także encje zewnętrzne.  
Specyfikacja nie określa wartości domyślnej dla tej właściwości.
    - "`javax.xml.stream.supportDTD`"  
Gdy ma wartość `true` (domyślnie), DTD nie są raportowane jako zdarzenia  
parsowania.
  - `XMLStreamReader createXMLStreamReader(InputStream in)`
  - `XMLStreamReader createXMLStreamReader(InputStream in, String characterEncoding)`

- `XMLStreamReader createXMLStreamReader(Reader in)`
- `XMLStreamReader createXMLStreamReader(Source in)`

tworzy parser wczytujący dane XML z podanego strumienia, obiektu klasy Reader lub źródła JAXP.

### `javax.xml.stream.XMLStreamReader` 6

- `boolean hasNext()`

zwraca wartość true, jeśli istnieje kolejne zdarzenie parsowania.

- `int next()`

powoduje, że stanem parsera staje się następne zdarzenie parsowania, i zwraca jedną z następujących stałych: START\_ELEMENT, END\_ELEMENT, CHARACTERS, START\_DOCUMENT, END\_DOCUMENT, CDATA, COMMENT, SPACE (ignorowany odstęp), PROCESSING\_INSTRUCTION, ENTITY\_REFERENCE, DTD.

- `boolean isStartElement()`

- `boolean isEndElement()`

- `boolean isCharacters()`

- `boolean isWhiteSpace()`

zwraca wartość true, jeśli bieżące zdarzenie parsowania dotyczy odpowiednio elementu początkowego, elementu końcowego, danych znakowych lub odstępu.

- `QName getName()`

- `String getLocalName()`

pobiera nazwę elementu dla zdarzenia START\_ELEMENT lub END\_ELEMENT.

- `String getText()`

zwraca łańcuch znaków dla zdarzenia CHARACTERS, COMMENT lub CDATA, wartość zastępczą dla zdarzenia ENTITY\_REFERENCE lub wewnętrzny podzbiór DTD.

- `int getAttributeCount()`

- `QName getAttributeName(int index)`

- `String getAttributeLocalName(int index)`

- `String getAttributeValue(int index)`

zwraca liczbę atrybutów oraz ich nazwy i wartości, przy założeniu, że bieżącym zdarzeniem parsowania jest START\_ELEMENT.

- `String getAttributeValue(String namespaceURI, String name)`

zwraca wartość atrybutu o podanej nazwie, przy założeniu, że bieżącym zdarzeniem parsowania jest START\_ELEMENT. Jeśli namespaceURI jest równy null, to przestrzeń nazw nie jest sprawdzana.

## 2.7. Tworzenie dokumentów XML

Ponieważ umiemy już tworzyć programy czytające dokumenty XML, to zajmiemy się teraz procesem tworzenia takich dokumentów. Plik XML możemy oczywiście utworzyć za pomocą sekwencji wywołań metody print, zapisując w ten sposób elementy, atrybuty i teksty dokumentu. Nie jest to jednak najlepsze rozwiązanie. Wymaga sporego nakładu pracy, podczas której łatwo popełnić błąd polegający na przykład na opuszczeniu znaków specjalnych (takich jak < czy >) podczas zapisu wartości atrybutów bądź tekstów.

Doskonalsze rozwiązanie problemu tworzenia dokumentów XML polegać będzie na utworzeniu drzewa DOM przyszłego dokumentu, a następnie zapisie jego zawartości do pliku.

### 2.7.1. Dokumenty bez przestrzeni nazw

Tworzenie drzewa DOM rozpoczynamy od uzyskania pustego dokumentu za pomocą metody newDocument klasy DocumentBuilder.

```
Document doc = builder.newDocument();
```

Do tworzenia elementów dokumentu wykorzystujemy metodę createElement klasy Document.

```
Element rootElement = doc.createElement(rootName);
Element childElement = doc.createElement(childName);
```

Natomiast metoda createTextNode posłuży do utworzenia węzłów tekstowych:

```
text textNode = doc.createTextNode(textContents);
```

Element, który ma się znaleźć w korzeniu drzewa, dodajemy do obiektu dokumentu, a węzły podzielone do odpowiednich węzłów nadzialeń:

```
doc.appendChild(rootElement);
rootElement.appendChild(childElement);
childElement.appendChild(textNode);
```

Podczas budowy drzewa DOM możemy także określić atrybuty elementów, korzystając z metody setAttribute klasy Element:

```
rootElement.setAttribute(name, value);
```

### 2.7.2. Dokumenty z przestrzenią nazw

Jeśli używamy przestrzeni nazw, procedura tworzenia dokumentu jest nieco inna.

Najpierw oczywiście konfigurujemy fabrykę, aby korzystała z przestrzeni nazw, i pobieramy z niej obiekt DocumentBuilder:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setNamespaceAware(true);
builder = factory.newDocumentBuilder();
```

Następnie używamy metody createElementNS zamiast metody createElement do tworzenia węzłów:

```
String namespace = "http://www.w3.org/2000/svg";
Element rootElement = doc.createElementNS(namespace, "svg");
```

Jeśli węzeł jest poprzedzony prefiksem, to wszystkie niezbędne atrybuty poprzedzone prefiksem xmlns są tworzone automatycznie. Na przykład jeśli tworzymy SVG w pliku XHTML, wystarczy następująca instrukcja:

```
Element svgElement = doc.createElement(namespace, "svg:svg")
```

Podczas zapisu element ten otrzyma następującą postać:

```
<svg:svg xmlns:svg="http://www.w3.org/2000/svg">
```

Jeśli potrzebujemy nadać elementowi atrybuty, których nazwy należą do przestrzeni nazw, używamy metody setAttributeNS klasy Element:

```
rootElement.setAttributeNS(namespace, qualifiedName, value);
```

## 2.7.3. Zapisywanie dokumentu

Co ciekawe, interfejs programowy DOM nie udostępnia dotąd sposobu zapisu drzewa DOM za pomocą strumienia wyjściowego. Aby obejść to ograniczenie, skorzystamy z interfejsu programowego XSLT (*XML Style Sheet Transformation*). Więcej informacji na temat interfejsu XSLT zamieszczono w podrozdziale 2.8., „Przekształcenia XSL”. Na razie ograniczymy się tylko do wykorzystania go do zapisu pliku XML.

Zastosujemy tożsamościowe przekształcenie dokumentu i wykorzystamy jego wynik. Aby zapisać także węzeł DOCTYPE, musimy określić identyfikatory SYSTEM i PUBLIC jako właściwości zapisu.

```
// tworzy przekształcenie tożsamościowe
Transformer t = TransformerFactory
    .newInstance().newTransformer();
// określa właściwości zapisu, aby umieścić węzeł DOCTYPE
t.setOutputProperty(OutputKeys.DOCUMENT_TYPE_NAME, systemIdentifier);
t.setOutputProperty(OutputKeys.DOCUMENT_PUBLIC_IDENTIFIER, publicIdentifier);
// konfiguruje wcięcia
t.setOutputProperty(OutputKeys.INDENT, "yes");
t.setOutputProperty(OutputKeys.METHOD, "xml");
t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
// wykonuje przekształcenie tożsamościowe
// i wysyła rezultat do pliku
t.transform(new DOMSource(doc),
    new StreamResult(new FileOutputStream(f)));
```

Inne rozwiązanie polega na wykorzystaniu interfejsu LSSerializer. Aby uzyskać odpowiedni obiekt, posłużymy się poniższą sekwencją wywołań:

```
DOMImplementation impl = doc.getImplementation();
DOMImplementationLS implLS = (DOMImplementationLS) impl.getFeature("LS", "3.0");
LSSerializer ser = implLS.createLSSerializer();
```

Jeśli plik ma zawierać odstępy i znaki nowego wiersza, należy skonfigurować poniższy znacznik:

```
ser.getDomConfig().setParameter("format-pretty-print", true);
```

Teraz łatwo możemy już przekształcić dokument w łańcuch znaków:

```
String str = ser.writeToString(doc);
```

Jeśli chcemy zapisać go wprost do pliku, potrzebujemy obiektu LSOutput:

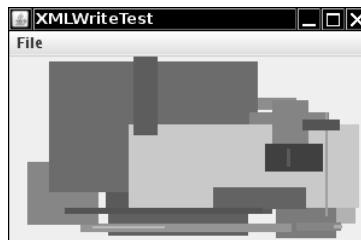
```
LSOutput out = implLS.createLSOutput();
out.setEncoding("UTF-8");
out.setByteStream(Files.newOutputStream(path));
ser.write(doc, out);
```

## 2.7.4. Przykład: tworzenie pliku SVG

Listing 2.10 prezentuje kod źródłowy typowego programu zapisującego plik XML. Program ten rysuje sekwencję prostokątów o losowych rozmiarach i kolorach (patrz rysunek 2.6). Do zapisu uzyskanego w ten sposób obrazu wykorzystujemy format SVG (*Scalable Vector Graphics*). Format SVG stworzono w języku XML do zapisu złożonej grafiki wektorowej w sposób niezależny od platformy. Więcej informacji na temat formatu SVG można odnaleźć pod adresem <http://www.w3c.org/Graphics/SVG>. Do przeglądania plików formatu SVG możemy wykorzystać na przykład każdą nowoczesną przeglądarkę.

**Rysunek 2.6.**

Program  
XMLWriteTest  
w działaniu



**Listing 2.10.** write/XMLWriteTest.java

```
package write;

import java.awt.*;
import javax.swing.*;

/**
 * Program demonstrujący sposób tworzenia dokumentu XML.
 * Zapisuje grafikę w pliku formatu SVG
 * @version 1.11 2012-01-26
 * @author Cay Horstmann
 */
public class XMLWriteTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
```

```
    {
        public void run()
        {
            JFrame frame = new XMLWriteFrame();
            frame.setTitle("XMLWriteTest");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
        }
    );
}
```

Nie będziemy omawiać tutaj szczegółów formatu SVG. Dla naszych potrzeb wystarczy jedyne znajomość sposobu zapisu zbioru kolorowych prostokątów. Poniżej prezentujemy fragment pliku w formacie SVG.

Jak łatwo zauważyc, każdy z prostokątów opisany jest za pomocą elementu rect. Położenie, szerokość, wysokość i kolor wypełnienia są atrybutami tego elementu. Kolor wypełnienia jest wartością RGB wyrażoną szesnastkowo.



 Jak pokazuje powyższy przykład, format SVG wykorzystuje intensywnie atrybuty elementów. Niektóre z tych atrybutów są dość skomplikowane. Poniższy przykład zawiera opis ścieżki:

```
<path d="M 100 100 L 300 100 L 200 300 z">
```

Znak M oznacza polecenie przesunięcia kurSORA, L — narysowania linii, a z —zamknięcia ścieżki. Świadczy to jedynie o tym, że projektanci formatu SVG nie mieli zbyt dużego doświadczenia w korzystaniu z elementów dokumentów XML. Jeśli będziemy tworzyć własne formaty w języku XML, to nie zalecamy naśladowictwa i doradzamy wykorzystanie elementów zamiast złożonych atrybutów.



javax.xml.parsers.DocumentBuilder 1.4

- Document newDocument()  
zwraca pusty dokument.



*org.w3c.dom.Document* 1.4

- Element createElement(String name)
  - Element createElementNS(String uri, String qname)  
tworzy element o podanej nazwie.

- `Text createTextNode(String data)`  
tworzy węzeł tekstowy zawierający podany tekst.

**API org.w3c.dom.Node 1.4**

- `Node appendChild(Node child)`  
dołącza węzeł do listy węzłów podrzędnych danego węzła. Zwraca dołączony węzeł.

**API org.w3c.dom.Element 1.4**

- `void setAttribute(String name, String value)`
- `void setAttributeNS(String uri, String qname, String value)`  
nadaje podaną wartość atrybutowi o podanej nazwie.  
 Parametry:  
 uri identyfikator URI przestrzeni nazw lub wartość null,  
 qname pełna nazwa atrybutu; jeśli poprzedzona jest prefiksem,  
 to parametr uri musi mieć wartość null,  
 value wartość atrybutu.

**API javax.xml.transform.TransformerFactory 1.4**

- `static TransformerFactory newInstance()`  
zwraca instancję fabryki klasy TransformerFactory.
- `Transformer newTransformer()`  
zwraca instancję klasy Transformer reprezentującą przekształcenie tożsamościowe.

**API javax.xml.transform.Transformer 1.4**

- `void setOutputProperty(String name, String value)`  
określa właściwość zapisu. Listę właściwości odnaleźć można na stronie <http://www.w3.org/TR/xslt#output>. Do najbardziej przydatnych należą:  
 doctype-public identyfikator publiczny używany w deklaracji DOCTYPE,  
 doctype-system identyfikator systemowy używany w deklaracji DOCTYPE,  
 indent określa, czy mają być stosowane znaki kontrolne, wartość yes lub no,  
 method "xml", "html", "text" lub inny, własny łańcuch.
- `void transform(Source from, Result to)`  
wykonuje przekształcenie dokumentu XML.

**API javax.xml.transform.dom.DOMSource 1.4**

- DOMSource(Node n)

tworzy źródło na podstawie podanego węzła n. Zwykle n jest węzłem reprezentującym dokument.

**API javax.xml.transform.stream.StreamResult 1.4**

- StreamResult(File f)
- StreamResult(OutputStream out)
- StreamResult(Writer out)
- StreamResult(String systemID)

tworzą obiekt klasy StreamResult na podstawie pliku, strumienia, obiektu klasy Writer lub identyfikatora systemowego (zwykle w postaci względnego lub bezwzględnego adresu URL).

## 2.7.5. Tworzenie dokumentu XML za pomocą parsera StAX

W poprzednim podrozdziale pokazaliśmy, jak stworzyć i zapisać dokument XML, używając w tym celu drzewa DOM. Jeśli jednak drzewo DOM nie jest nam potrzebne z innych powodów, to jego wykorzystanie do tworzenia dokumentu XML nie jest najefektywniejszym rozwiązaniem.

Interfejs programowy parsera StAX umożliwia nam bezpośredni zapis drzewa XML. W tym celu musimy utworzyć obiekt XMLStreamWriter dla strumienia OutputStream:

```
XMLOutputFactory factory = XMLOutputFactory.newInstance();
XMLStreamWriter writer = factory.createXMLStreamWriter(out);
```

Aby utworzyć nagłówek XML używamy poniższego wywołania:

```
writer.writeStartDocument()
```

Następnie wywołujemy:

```
writer.writeStartElement(name);
```

Dla atrybutów stosujemy następujące wywołania:

```
writer.writeAttribute(name, value);
```

Elementy podrzędne dodajemy za pomocą ponownych wywołań writeStartElement, a dane tekstowe za pomocą poniższego wywołania:

```
writer.writeCharacters(text);
```

Po zapisaniu wszystkich elementów podrzędnych wywołujemy

```
writer.writeEndElement();
```

co spowoduje zamknięcie bieżącego elementu.

Aby zapisać element, który nie posiada elementów podrzędnych (taki jak na przykład <img .../>), używamy wywołania:

```
writer.writeEmptyElement(name);
```

Aby zakończyć dokument, wywołujemy

```
writer.writeEndDocument();
```

co spowoduje zamknięcie wszystkich otwartych elementów.

Podobnie jak w podejściu DOM/XSLT, nie musimy martwić się specjalnymi sekwencjami znaków w wartościach atrybutów i danych znakowych. Jednak tym razem musimy dodatkowo uważać, aby nie stworzyć niepoprawnego dokumentu XML, na przykład zawierającego wiele węzłów korzenia. Aktualnie dostępna implementacja parsera StAX nie umożliwia tworzenia wcięć w dokumencie XML.

Program przedstawiony na listingu 2.10 pokazuje sposób implementacji obu omówionych podejść do tworzenia dokumentów XML. Listingi 2.11 i 2.12 przedstawiają klasy ramki i komponentu służących do rysowania prostokątów.

---

**Listing 2.11.** *write/XMLWriteFrame.java*

```
package write;

import java.awt.event.*;
import java.beans.*;
import java.io.*;
import java.nio.file.*;
import javax.swing.*;
import javax.xml.stream.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import org.w3c.dom.*;

/**
 * Ramka zawierająca panel prezentacji grafiki.
 */
public class XMLWriteFrame extends JFrame
{
    private RectangleComponent comp;
    private JFileChooser chooser;

    public XMLWriteFrame()
    {
        chooser = new JFileChooser();

        // dodaje komponent do ramki
        comp = new RectangleComponent();
        add(comp);

        // tworzy menu
        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);

        JMenu menu = new JMenu("File");

```

```

menuBar.add(menu);

JMenuItem newItem = new JMenuItem("New");
menu.add(newItem);
newItem.addActionListener(EventHandler.create(ActionListener.class, comp,
    "newDrawing"));

JMenuItem saveItem = new JMenuItem("Save with DOM/XSLT");
menu.add(saveItem);
saveItem.addActionListener(EventHandler.create(ActionListener.class, this,
    "saveDocument"));

JMenuItem saveStAXItem = new JMenuItem("Save with StAX");
menu.add(saveStAXItem);
saveStAXItem.addActionListener(EventHandler.create(ActionListener.class, this,
    "saveStAX"));

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

/**
 * Zapisuje grafikę w formacie SVG, używając DOM/XSLT.
 */
public void saveDocument() throws TransformerException, IOException
{
    if (chooser.showSaveDialog(this) != JFileChooser.APPROVE_OPTION) return;
    File file = chooser.getSelectedFile();
    Document doc = comp.buildDocument();
    Transformer t = TransformerFactory.newInstance().newTransformer();
    t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM,
        "http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd");
    t.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC, "-//W3C//DTD SVG 20000802//EN");
    t.setOutputProperty(OutputKeys.INDENT, "yes");
    t.setOutputProperty(OutputKeys.METHOD, "xml");
    t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
    t.transform(new DOMSource(doc), new
    StreamResult(Files.newOutputStream(file.toPath())));
}

/**
 * Zapisuje grafikę w formacie SVG, używając StAX.
 */
public void saveStAX() throws IOException, XMLStreamException
{
    if (chooser.showSaveDialog(this) != JFileChooser.APPROVE_OPTION) return;
    File file = chooser.getSelectedFile();
    XMLOutputFactory factory = XMLOutputFactory.newInstance();
    XMLStreamWriter writer =
        factory.createXMLStreamWriter(Files.newOutputStream(file.toPath()));
}

```

```
        try
        {
            comp.writeDocument(writer);
        }
        finally
        {
            writer.close(); // nie implementuje AutoCloseable
        }
    }
}
```

---

**Listing 2.12.** write/RectangleComponent.java

```
package write;

import java.awt.*;
import java.awt.geom.*;
import java.util.*;
import javax.swing.*;
import javax.xml.parsers.*;
import javax.xml.stream.*;
import org.w3c.dom.*;

/**
 * Komponent rysujący sekwencję kolorowych prostokątów.
 */
public class RectangleComponent extends JComponent
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;

    private java.util.List<Rectangle2D> rects;
    private java.util.List<Color> colors;
    private Random generator;
    private DocumentBuilder builder;

    public RectangleComponent()
    {
        rects = new ArrayList<>();
        colors = new ArrayList<>();
        generator = new Random();

        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        factory.setNamespaceAware(true);
        try
        {
            builder = factory.newDocumentBuilder();
        }
        catch (ParserConfigurationException e)
        {
            e.printStackTrace();
        }
    }

    /**
     * Tworzy nowy, losowy rysunek.
     */
```

```

public void newDrawing()
{
    int n = 10 + generator.nextInt(20);
    rects.clear();
    colors.clear();
    for (int i = 1; i <= n; i++)
    {
        int x = generator.nextInt(getWidth());
        int y = generator.nextInt(getHeight());
        int width = generator.nextInt(getWidth() - x);
        int height = generator.nextInt(getHeight() - y);
        rects.add(new Rectangle(x, y, width, height));
        int r = generator.nextInt(256);
        int g = generator.nextInt(256);
        int b = generator.nextInt(256);
        colors.add(new Color(r, g, b));
    }
    repaint();
}

public void paintComponent(Graphics g)
{
    if (rects.size() == 0) newDrawing();
    Graphics2D g2 = (Graphics2D) g;

    //rysuje wszystkie prostokąty
    for (int i = 0; i < rects.size(); i++)
    {
        g2.setPaint(colors.get(i));
        g2.fill(rects.get(i));
    }
}

/**
 * Tworzy dokument SVG dla bieżącego rysunku.
 * @return drzewo DOM dokumentu SVG
 */
public Document buildDocument()
{
    String namespace = "http://www.w3.org/2000/svg";
    Document doc = builder.newDocument();
    Element svgElement = doc.createElementNS(namespace, "svg");
    doc.appendChild(svgElement);
    svgElement.setAttribute("width", "" + getWidth());
    svgElement.setAttribute("height", "" + getHeight());
    for (int i = 0; i < rects.size(); i++)
    {
        Color c = colors.get(i);
        Rectangle2D r = rects.get(i);
        Element rectElement = doc.createElementNS(namespace, "rect");
        rectElement.setAttribute("x", "" + r.getX());
        rectElement.setAttribute("y", "" + r.getY());
        rectElement.setAttribute("width", "" + r.getWidth());
        rectElement.setAttribute("height", "" + r.getHeight());
        rectElement.setAttribute("fill", colorToString(c));
        svgElement.appendChild(rectElement);
    }
    return doc;
}

```

```

    }

    /**
     * Zapisuje dokument SVG dla bieżącego rysunku.
     * @param writer dokument docelowy
     */
    public void writeDocument(XMLStreamWriter writer) throws XMLStreamException
    {
        writer.writeStartDocument();
        writer.writeDTD("<!DOCTYPE svg PUBLIC \"-//W3C//DTD SVG 20000802//EN\" "
                + "\n\"http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd\">");

        writer.writeStartElement("svg");
        writer.writeDefaultNamespace("http://www.w3.org/2000/svg");
        writer.writeAttribute("width", " " + getWidth());
        writer.writeAttribute("height", " " + getHeight());
        for (int i = 0; i < rects.size(); i++)
        {
            Color c = colors.get(i);
            Rectangle2D r = rects.get(i);
            writer.writeEmptyElement("rect");
            writer.writeAttribute("x", " " + r.getX());
            writer.writeAttribute("y", " " + r.getY());
            writer.writeAttribute("width", " " + r.getWidth());
            writer.writeAttribute("height", " " + r.getHeight());
            writer.writeAttribute("fill", colorToString(c));
        }
        writer.writeEndDocument(); //zamyka element svg
    }

    /**
     * Zamienia obiekt klasy Color na wartość szesnastkową.
     * @param c obiekt klasy Color
     * @returnłańcuch postaci #rrggb
     */
    private static String colorToString(Color c)
    {
        StringBuffer buffer = new StringBuffer();
        buffer.append(Integer.toHexString(c.getRGB() & 0xFFFFFFFF));
        while (buffer.length() < 6)
            buffer.insert(0, '0');
        buffer.insert(0, '#');
        return buffer.toString();
    }

    public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH,
        DEFAULT_HEIGHT); }
}

```

### API javax.xml.stream.XMLOutputFactory 6

- static XMLOutputFactory newInstance()
   
zwraca instancję klasy XMLOutputFactory.
- XMLStreamWriter createXMLStreamWriter(OutputStream in)

- `XMLStreamWriter createXMLStreamWriter(OutputStream in, String characterEncoding)`
  - `XMLStreamWriter createXMLStreamWriter(Writer in)`
  - `XMLStreamWriter createXMLStreamWriter(Result in)`
- tworzy obiekt zapisujący XML dla podanego strumienia, obiektu zapisującego lub wyniku działania JAXP.

#### **javax.xml.stream.XMLStreamWriter** 6

- `void writeStartDocument()`
  - `void writeStartDocument(String xmlVersion)`
  - `void writeStartDocument(String encoding, String xmlVersion)`
- zapisuje instrukcje przetwarzania XML na początku dokumentu. Zwróćmy uwagę, że parametr `encoding` jest używany tylko przy zapisie atrybutów. Nie określa on zatem sposobu kodowania znaków dla całego dokumentu.
- `void setDefaultNamespace(String namespaceURI)`
  - `void setPrefix(String prefix, String namespaceURI)`
- określa domyślną przestrzeń nazw lub przestrzeń związaną z podanym prefiksem. Zasięg deklaracji przestrzeni nazw obejmuje bieżący element. Natomiast gdy żaden element nie został jeszcze zapisany, to deklaracja dotyczy korzenia dokumentu.
- `void writeStartElement(String localName)`
  - `void writeStartElement(String namespaceURI, String localName)`
- zapisuje znacznik początkowy, zastępując `namespaceURI` odpowiednim prefiksem.
- `void writeEndElement()`
- zamyka bieżący element.
- `void writeEndDocument()`
- zamyka wszystkie otwarte elementy.
- `void writeEmptyElement(String localName)`
  - `void writeEmptyElement(String namespaceURI, String localName)`
- zapisuje znacznik, który nie wymaga osobnego zamknięcia, zastępując `namespaceURI` odpowiednim prefiksem.
- `void writeAttribute(String localName, String value)`
  - `void writeAttribute(String namespaceURI, String localName, String value)`
- zapisuje atrybut bieżącego elementu, zastępując `namespaceURI` odpowiednim prefiksem.
- `void writeCharacters(String text)`
- zapisuje dane znakowe.

- void writeCData(String text)  
zapisuje blok CDATA.
- void writeDTD(String dtd)  
zapisuje łańcuch dtd, zakładając, że zawiera on deklarację DOCTYPE.
- void writeComment(String comment)  
zapisuje komentarz.
- void close()  
zamyka obiekt zapisujący.

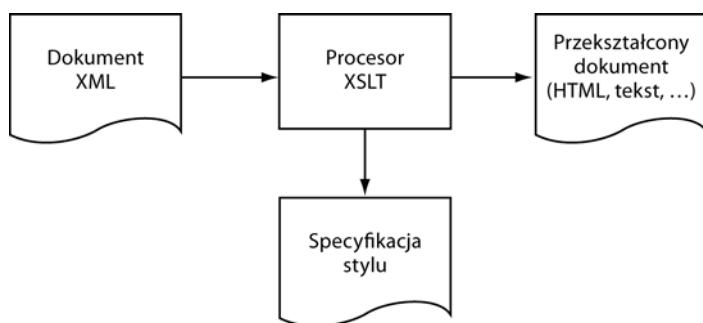
## 2.8. Przekształcenia XSL

Mechanizm przekształceń XSL umożliwia określenie reguł przetwarzania dokumentów XML na inne formaty, na przykład zwykły format tekstowy czy XHTML. Mechanizm ten jest stosowany do przekształcania danych pomiędzy różnymi formatami danych XML wykorzystywanych przez aplikacje lub do przekształcania dokumentów XML na postać czytelną dla użytkownika aplikacji.

Zadaniem specyfikacji stylu XSL zawierającej szablon przekształcenia jest opis sposobu konwersji dokumentów XML na inny format. Procesor XSLT wczytuje dokument XML i specyfikację stylu i na ich podstawie tworzy wyjściowy dokument XML (patrz rysunek 2.7).

**Rysunek 2.7.**

Zastosowanie przekształceń XSL



A oto typowy przykład. Założmy, że chcemy przekształcić dokument XML zawierający informacje o pracownikach na dokument HTML. Wejściowy dokument XML będzie miał następującą postać.

```

<staff>
  <employee>
    <name>Carl Cracker</name>
    <salary>75000</salary>
    <hiredate year="1987" month="12" day="15"/>
  </employee>
  <employee>
    <name>Harry Hacker</name>
    <salary>50000</salary>
  
```

```

<hiredate year="1989" month="10" day="1"/>
</employee>
<employee>
  <name>Tony Tester</name>
  <salary>40000</salary>
  <hiredate year="1990" month="3" day="15"/>
</employee>
</staff>
```

Informację tę będziemy chcieli zaprezentować w postaci poniższej tabeli HTML:

```


|              |           |            |
|--------------|-----------|------------|
| Carl Cracker | \$75000.0 | 1987-12-15 |
| Harry Hacker | \$50000.0 | 1989-10-1  |
| Tony Tester  | \$40000.0 | 1990-3-15  |


```

Specyfikacja XSLT jest dość złożona i poświecono jej już wiele książek. Poprzestaniemy więc tylko na przedstawieniu reprezentatywnego przykładu.Więcej informacji na ten temat odnajdziemy na przykład w książce *Essential XML* autorstwa Dona Boxa i innych. Specyfikacja XSLT dostępna jest pod adresem <http://www.w3.org/TR/xslt>.

Specyfikacja stylu zawierająca szablon przekształcenia posiada poniższą formę:

```

<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:output method="html"/>
  szablon1
  szablon2
  .
  .
  .
</xsl:stylesheet>
```

W naszym przykładzie element `xsl:output` określa metodę przekształcenia jako HTML. Inne możliwości to `xml` i `text`.

Poniżej prezentujemy typowy szablon przekształcenia:

```

<xsl:template match="/staff/employee">
  <tr><xsl:apply-templates/></tr>
</xsl:template>
```

Wartość atrybutu `match` jest wyrażeniem XPath. Szablon mówi: kiedykolwiek napotkasz węzeł należący do zbioru XPath/staff/employee wykonaj, co następuje:

- 1.** Wyślij łańcuch `<tr>`.
- 2.** Zastosuj szablony do elementów podrzędnych.
- 3.** Wyślij łańcuch `</tr>` po zakończeniu przetwarzania elementów podrzędnych.

Innymi słowy szablon ten generuje znaczniki wierszy tabeli HTML wokół każdego rekordu opisującego pracownika.

Procesor XSLT rozpoczyna przetwarzanie od elementu znajdującego się w korzeniu. Jeśli węzeł pasuje do jednego z szablonów, to XSLT stosuje go. (Jeśli pasuje do wielu szablonów, to wybiera najlepszy z nich, posługując się kryteriami opisanymi szczegółowo w specyfikacji). Jeśli nie pasuje do żadnego z szablonów, to wykonywana jest domyślna akcja. W przypadku węzłów tekstowych polega ona na umieszczeniu na wyjściu ich zawartości. W przypadku elementów podejmowane jest przetwarzanie ich elementów podrzędnych.

Poniżej prezentujemy szablon służący do przetwarzania węzłów name w dokumencie opisującym pracowników:

```
<xsl:template match="/staff/employee/name">
    <td><xsl:apply-templates/></td>
</xsl:template>
```

Tworzy on parę znaczników <td>...</td> i zleca procesorowi rekurencyjne przetwarzanie elementów podrzędnych elementu name. Dla elementu name istnieje jedynie pojedynczy, podrzędny węzeł tekstowy. Przetwarzając go, procesor umieszcza jego tekst na wyjściu (zakładając, że nie istnieje odpowiedni szablon opisujący inną operację).

Skopiowanie wartości atrybutów na wyjście procesora jest nieco bardziej skomplikowane. Oto jego przykład:

```
<xsl:template match="/staff/employee/hiredate">
    <td><xsl:value-of select="@year"/>-<xsl:value-of
        select="@month"/>-<xsl:value-of select="@day"/></td>
</xsl:template>
```

Szablon ten powoduje, że podczas przetwarzania węzła hiredate zostaną wysłane na wyjście kolejno:

1. łańcuch <td>,
2. wartość atrybutu year,
3. znak łącznika,
4. wartość atrybutu month,
5. znak łącznika,
6. wartość atrybutu day,
7. łańcuch </td>.

Wyrażenie `xsl:value-of` wyznacza łańcuch dla zbioru węzłów. Zbiór ten określony jest przez wartość XPath atrybutu `select`. W tym przypadku ścieżka podana jest względem przetwarzanego właśnie węzła. Zbiór węzłów przetwarzany jest na łańcuch będący konkatenacją łańcuchów dla wszystkich węzłów zbioru. łańcuchem węzła atrybutu jest jego wartość. łańcuchem węzła tekstopowego jest jego zawartość. łańcuchem elementu jest konkatenacja łańcuchów jego elementów podrzędnych (niebędących jednak atrybutami).

Listing 2.13 prezentuje szablon przekształceń służący do zamiany pliku XML zawierającego rekordy opisujące pracowników na tabelę HTML.

**Listing 2.13.** transform/makehtml.xsl

---

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">

    <xsl:output method="html"/>

    <xsl:template match="/staff">
        <table border="1"><xsl:apply-templates/></table>
    </xsl:template>

    <xsl:template match="/staff/employee">
        <tr><xsl:apply-templates/></tr>
    </xsl:template>

    <xsl:template match="/staff/employee/name">
        <td><xsl:apply-templates/></td>
    </xsl:template>

    <xsl:template match="/staff/employee/salary">
        <td>$<xsl:apply-templates/></td>
    </xsl:template>

    <xsl:template match="/staff/employee/hiredate">
        <td><xsl:value-of select="@year"/>-<xsl:value-of
            select="@month"/>-<xsl:value-of select="@day"/></td>
    </xsl:template>

</xsl:stylesheet>
```

---

Listing 2.14 prezentuje inny zestaw przekształceń. Działają one na tym samym dokumencie XML, ale ich rezultatem jest plik właściwości o poniższej postaci:

```
employee.1.name=Carl Cracker
employee.1.salary=75000.0
employee.1.hiredate=1987-12-15
employee.2.name=Harry Hacker
employee.2.salary=50000.0
employee.2.hiredate=1989-10-1
employee.3.name=Tony Tester
employee.3.salary=40000.0
employee.3.hiredate=1990-3-15
```

**Listing 2.14.** transform/makeprop.xsl

---

```
<?xml version="1.0"?>

<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
```

---

```
<xsl:output method="text"/>

<xsl:template match="/staff/employee">
employee.<xsl:value-of select="position()" />.name=<xsl:value-of
  >select="name/text()" />
employee.<xsl:value-of select="position()" />.salary=<xsl:value-of
  >select="salary/text()" />
employee.<xsl:value-of select="position()" />.hiredate=<xsl:value-of
  >select="hiredate/@year" />-<xsl:value-of select="hiredate/@month" />-<xsl:value-of
  >select="hiredate/@day" />
</xsl:template>

</xsl:stylesheet>
```

---

W tym przypadku korzystamy w szablonach z funkcji `position()`, która zwraca pozycję węzła podzielnego z punktu widzenia jego węzła nadrzędnego. Zmieniając jedynie plik specyfikacji stylu, uzyskujemy zupełnie inny rezultat. Dzięki temu możemy bezpiecznie korzystać z opisu danych w języku XML, nawet jeśli okaże się, że pewna aplikacja oczekuje danych w zupełnie innym formacie. Wykorzystanie XSLT umożliwi bowiem wygenerowanie danych w żądanym formacie.

Przekształcenia XSL w języku Java wykonuje się niezwykle łatwo. Dla danej specyfikacji stylu tworzy się fabrykę przekształcania, z której następnie pobiera się obiekt reprezentujący przekształcenie i zleca mu jego wykonanie.

```
File styleSheet = new File(filename);
StreamSource styleSource = new StreamSource(styleSheet);
Transformer t = TransformerFactory.newInstance().newTransformer(styleSource);
t.transform(source, result);
```

Parametrami metody `transform` są obiekty implementujące interfejsy `Source` i `Result`. Dostępne są cztery implementacje interfejsu `Source`:

```
DOMSource
SAXSource
StAXSource
StreamSource
```

Obiekt klasy `StreamSource` możemy utworzyć z pliku, strumienia, obiektu klasy `Reader`, adresu URL, a obiekt klasy `DOMSource` — z węzła drzewa DOM. Na przykład w poprzednim podrozdziale wywołaliśmy przekształcenie tożsamościowe w następujący sposób:

```
t.transform(new DOMSource(doc), result)
```

Tym razem jednak wykonamy przekształcenie w ciekawszy sposób. Zamiast rozpoczynać działanie od istniejącego pliku XML, utworzymy obiekt odczytu pliku za pomocą parsera SAX, który wykreuje iluzję parsowania prawdziwego pliku XML, generując odpowiednie zdarzenia parsera SAX. W rzeczywistości obiekt odczytu będzie czytał zwykły plik tekstowy przedstawiony już w 1. rozdziale książki. Plik ten będzie wyglądać następująco:

```
Carl Cracker|75000.0|1987|12|15
Harry Hacker|50000.0|1989|10|1
Tony Tester|40000.0|1990|3|15
```

Podczas przetwarzania jego zawartości obiekt odczytu będzie generował zdarzenia parsera SAX. Poniżej przedstawiamy fragment implementacji metody parse klasy EmployeeReader implementującej interfejs XMLReader.

```
AttributesImpl attributes = new AttributesImpl();
handler.startDocument();
handler.startElement("", "staff", "staff", attributes);
while ((line = in.readLine()) != null)
{
    handler.startElement("", "employee", "employee", attributes);
    StringTokenizer t = new StringTokenizer(line, "|");
    handler.startElement("", "name", "name", attributes);
    String s = t.nextToken();
    handler.characters(s.toCharArray(), 0, s.length());
    handler.endElement("", "name", "name");
    .
    .
    .
    handler.endElement("", "employee", "employee");
}
handler.endElement("", rootElement, rootElement);
handler.endDocument();
```

Obiekt klasy SAXSource utworzymy z obiektu odczytu dokumentu XML:

```
t.transform(new SAXSource(new EmployeeReader(),
    new InputSource(new FileInputStream(filename))), result);
```

Sztuczka ta pozwala stosunkowo łatwo zamienić dowolny plik na format XML. Oczywiście w praktyce taką potrzebą nie będzie zdarzać się zbyt często, bo większość aplikacji będzie posiadać już jakieś dane w formacie XML. Wtedy metodę transform wywołamy, jak poniżej:

```
t.transform(new StreamSource(file), result);
```

Wynik przekształcenia jest obiektem klasy implementującej interfejs Result. Biblioteka języka Java udostępnia trzy takie klasy:

```
DOMResult
SAXResult
StreamResult
```

Aby wynik przekształcenia miał postać drzewa DOM, wykorzystamy obiekt klasy Document →Builder do utworzenia nowego węzła i obudujemy go obiektem klasy DOMResult:

```
Document doc = builder.newDocument();
t.transform(source, new DOMResult(doc));
```

Aby zapisać wynik przekształcenia w pliku, wykorzystamy obiekt klasy StreamResult:

```
t.transform(source, new StreamResult(file));
```

Listing 2.15 zawiera kompletny kod źródłowy programu.

#### **Listing 2.15.** transform/TransformTest.java

```
package transform;

import java.io.*;
import java.nio.file.*;
```

```
import java.util.*;
import javax.xml.transform.*;
import javax.xml.transform.sax.*;
import javax.xml.transform.stream.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

/**
 * Program demonstrujący wykorzystanie przekształceń XSL na przykładzie
 * zbioru informacji o pracowników. Informacja ta umieszczona jest
 * w pliku tekstowym employee.dat i przekształcana na format XML.
 * Uruchamiając program, należy podać specyfikację stylu, na przykład:
 * java transform.TransformTest transform/makeprop.xsl
 * @version 1.02 2012-06-04
 * @author Cay Horstmann
 */
public class TransformTest
{
    public static void main(String[] args) throws Exception
    {
        Path path;
        if (args.length > 0) path = Paths.get(args[0]);
        else path = Paths.get("transform", "makehtml.xsl");
        try (InputStream styleIn = Files.newInputStream(path))
        {
            StreamSource styleSource = new StreamSource(styleIn);

            Transformer t = TransformerFactory.newInstance().newTransformer(styleSource);
            t.setOutputProperty(OutputKeys.INDENT, "yes");
            t.setOutputProperty(OutputKeys.METHOD, "xml");
            t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");

            try (InputStream docIn = Files.newInputStream(Paths.get("transform",
                "employee.dat")))
            {
                t.transform(new SAXSource(new EmployeeReader(), new InputSource(docIn)),
                    new StreamResult(System.out));
            }
        }
    }

    /**
     * Klasa odczytu pliku tekstopowego employee.dat. Generuje
     * zdarzenia parsera SAX jak podczas odczytu
     * pliku w formacie XML.
     */
    class EmployeeReader implements XMLReader
    {
        private ContentHandler handler;

        public void parse(InputSource source) throws IOException, SAXException
        {
            InputStream stream = source.getByteStream();
            BufferedReader in = new BufferedReader(new InputStreamReader(stream));
            String rootElement = "staff";
            AttributesImpl atts = new AttributesImpl();
```

```
if (handler == null) throw new SAXException("No content handler");

handler.startDocument();
handler.startElement("", rootElement, rootElement, atts);
String line;
while ((line = in.readLine()) != null)
{
    handler.startElement("", "employee", "employee", atts);
    StringTokenizer t = new StringTokenizer(line, "|");

    handler.startElement("", "name", "name", atts);
    String s = t.nextToken();
    handler.characters(s.toCharArray(), 0, s.length());
    handler.endElement("", "name", "name");

    handler.startElement("", "salary", "salary", atts);
    s = t.nextToken();
    handler.characters(s.toCharArray(), 0, s.length());
    handler.endElement("", "salary", "salary");

    atts.addAttribute("", "year", "year", "CDATA", t.nextToken());
    atts.addAttribute("", "month", "month", "CDATA", t.nextToken());
    atts.addAttribute("", "day", "day", "CDATA", t.nextToken());
    handler.startElement("", "hiredate", "hiredate", atts);
    handler.endElement("", "hiredate", "hiredate");
    atts.clear();

    handler.endElement("", "employee", "employee");
}

handler.endElement("", rootElement, rootElement);
handler.endDocument();
}

public void setContentHandler(ContentHandler newValue)
{
    handler = newValue;
}

public ContentHandler getContentHandler()
{
    return handler;
}

// poniższe metody mają "pustą" implementację
public void parse(String systemId) throws IOException, SAXException
{
}

public void setErrorHandler(ErrorHandler handler)
{
}

public ErrorHandler getErrorHandler()
{
    return null;
}
```

```
public void setDTDHandler(DTDHandler handler)
{
}

public DTDHandler getDTDHandler()
{
    return null;
}

public void setEntityResolver(EntityResolver resolver)
{
}

public EntityResolver getEntityResolver()
{
    return null;
}

public void setProperty(String name, Object value)
{
}

public Object getProperty(String name)
{
    return null;
}

public void setFeature(String name, boolean value)
{
}

public boolean getFeature(String name)
{
    return false;
}
```

---

**API** javax.xml.transform.TransformerFactory 1.4

- Transformer newTransformer(Source from)  
zwraca instancję klasy Transformer, która odczytuje specyfikację stylu z podanego źródła.

**API** javax.xml.transform.stream.StreamSource 1.4

- StreamSource(File f)
- StreamSource(InputStream in)
- StreamSource(Reader in)
- StreamSource(String systemID)

Tworzą obiekt klasy StreamSource na podstawie pliku, strumienia, obiektu klasy Reader lub identyfikatora systemowego (zwykle w postaci względnego lub bezwzględnego adresu URL).

**API javax.xml.transform.sax.SAXSource 1.4**

- SAXSource(XMLReader reader, InputSource source)

tworzy obiekt klasy SAXSource z obiektu klasy InputSource i wykorzystuje obiekt typu XMLReader do parsowania wejścia.

**API org.xml.sax.XMLReader 1.4**

- void setContentHandler(ContentHandler handler)  
instaluje obiekt obsługi zdarzeń generowanych podczas parsowania wejścia.
- void parse(InputSource source)  
parsuje dane wejściowe pochodzące z podanego źródła i wysyła zdarzenia parsowania do obiektu obsługi typu ContentHandler.

**API javax.xml.transform.dom.DOMResult 1.4**

- DOMResult(Node n)  
tworzy obiekt klasy DOMResult na podstawie węzła n, który zwykle reprezentuje dokument.

**API org.xml.sax.helpers.AttributesImpl 1.4**

- void addAttribute(String uri, String lname, String qname, String type, String value)  
umieszcza atrybut w danej kolekcji atrybutów.  
*Parametry:*   uri       identyfikator URI przestrzeni nazw,  
                  lname      nazwa lokalna (bez prefiku),  
                  qname     pełna nazwa z prefiksem  
                  type       typ atrybutu określony jako "CDATA", "ID", "IDREF",  
                          "IDREFS", "NMTOKEN", "NMTOKENS", "ENTITY", "ENTITIES"  
                          lub "NOTATION",  
                  value      wartość atrybutu.
- void clear()  
usuwa wszystkie atrybuty z danej kolekcji atrybutów.

Przykład ten kończy omówienie wykorzystania języka XML w programach tworzonych w języku Java. Mamy nadzieję, że w rozdziale tym udało nam się pokazać zalety technologii XML, szczególnie wynikające z automatyzacji parsowania i weryfikacji dokumentów, a także mechanizmu przekształceń. Będziesz mógł, Czytelniku, skorzystać z tych zalet w praktyce, pod warunkiem jednak, że prawidłowo zaprojektujesz własne formaty dokumentów XML. Powinny być one wystarczająco złożone, by opisać wszystkie bieżące i przyszłe potrzeby aplikacji, nie zmieniać się istotnie z upływem czasu i uzyskać akceptację użytkowników i ich partnerów biznesowych. W praktyce osiągnięcie tych celów okazuje się trudniejsze niż posługiwanie się parserami, definicjami typów dokumentów DTD czy przekształceniemi XSL.

W kolejnym rozdziale omówimy programowanie aplikacji sieciowych na platformie Java, rozpoczynając od podstawowych zagadnień programowania obsługi gniazd sieciowych, a później przechodząc do protokołów wyższych warstw związanych z aplikacjami poczty elektronicznej i witrynami WWW.

# 3

## Programowanie aplikacji sieciowych

W tym rozdziale:

- Połączenia z serwerem.
- Implementacja serwerów.
- Przerywanie działania gniazd sieciowych.
- Połączenia wykorzystujące URL.
- Wysyłanie poczty elektronicznej.

Rozdział ten rozpoczniemy od krótkiego opisu aplikacji sieciowych. Następnie przejdziemy do omówienia sposobu pisania programów w języku Java, które łączą się z usługami sieciowymi. Przedstawimy sposoby pobierania informacji z serwera Web i wysyłania poczty elektronicznej przez program w języku Java.

### 3.1. Połączenia z serwerem

Zanim napiszemy pierwszy program działający w sieci, zapoznajmy się najpierw z doskonałym narzędziem uruchomieniowym programów sieciowych, jakim jest terminal sieciowy telnet. Większość systemów operacyjnych, takich jak Unix czy Windows, zawiera implementację telnetu. Program uruchamiamy, wpisując polecenie `telnet`.



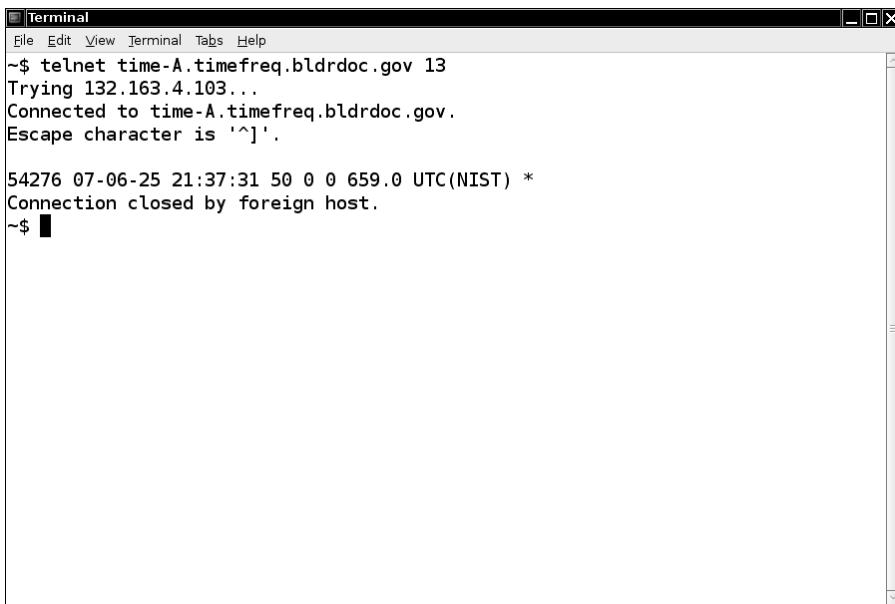
W systemie Windows Vista telnet jest zainstalowany, ale domyślnie nie jest aktywny. Aby uczynić go aktywnym, należy przejść do *Panelu sterowania*, wybrać kartę *Programy*, kliknąć opcję *Włącz lub wyłącz funkcje systemu Windows*, a następnie zaznaczyć pole wyboru *Klient programu telnet*. Zapora ogniodziału systemu Windows blokuje kilka portów sieciowych, które będziemy wykorzystywać w tym rozdziale; ich odblokowanie może wymagać uprawnień administratora.

Telnet często wykorzystuje się do łączenia z innymi komputerami w sieci, ale umożliwia on także komunikację z innymi usługami hostów. Wprowadźmy na przykład poniższą komendę:

```
telnet time-A.timefreq.bldrdoc.gov 13
```

Jak pokazano na rysunku 3.1, w efekcie powinniśmy uzyskać odpowiedź zbliżoną do poniższej:

```
54276 07-06-25 21:37:31 50 0 0 659.0 UTC(NIST) *
```

A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The main area displays the following text:

```
~$ telnet time-A.timefreq.bldrdoc.gov 13
Trying 132.163.4.103...
Connected to time-A.timefreq.bldrdoc.gov.
Escape character is '^]'.

54276 07-06-25 21:37:31 50 0 0 659.0 UTC(NIST) *
Connection closed by foreign host.
~$ █
```

The window has scroll bars on the right and bottom.

Rysunek 3.1. Odpowiedź uzyskana od usługi zegara atomowego

Co to oznacza? Połączymy się z usługą informującą o aktualnym czasie, która działa na większości hostów pracujących z systemem Unix. W naszym przykładzie usługa ta działa na serwerze w Narodowym Instytucie Standardów i Technologii w Boulder, stan Colorado i podaje wskazania cezowego zegara atomowego. (Oczywiście uzyskany czas nie jest dokładny ze względu na opóźnienia związane z transmisją w sieci).

Usługa informująca o czasie dostępna jest zawsze za pośrednictwem portu 13.

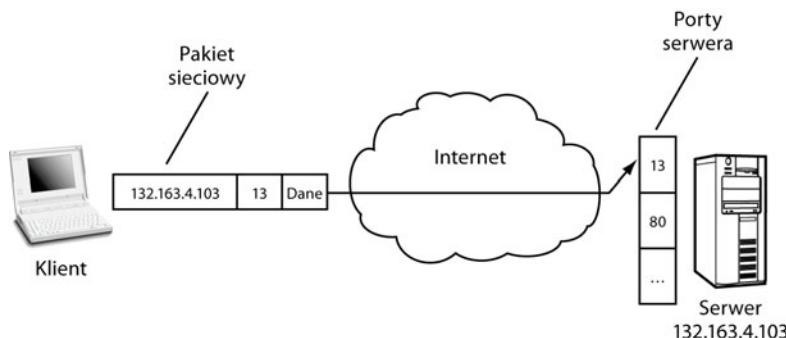


Port nie oznacza w tym przypadku fizycznego urządzenia, ale stanowi abstrakcję występującą w procesie komunikacji pomiędzy klientem i serwerem (patrz rysunek 3.2).

Oprogramowanie działa na serwerze, oczekując na próby komunikacji z portem 13. Jeśli więc system operacyjny odbierze pakiet zawierający żądanie połączenia z portem 13, to aktywuje on nasłuchujący proces i utworzy połączenie. Połączenie to pozostaje aktywne do momentu rozłączenia przez którąś ze stron.

**Rysunek 3.2.**

Klient łączący się z portem serwera



Kiedy inicjujemy sesję telnetu z portem 13 maszyny time-A.timefreq.bldrdoc.gov, ciąg znaków zawierający jego nazwę zostaje przekształcony na odpowiadający mu adres protokołu IP: 132.163.4.103. Do komputera o tym adresie wysłane zostaje żądanie połączenia z portem 13. Kiedy zostanie ono ustanowione, to serwer wysyła linię danych i zamknie połączenie. Oczywiście w ogólnym przypadku dialog między klientem i serwerem jest dużo bardziej złożony, zanim połączenie zostanie zamknięte.

Poniżej demonstrujemy kolejny eksperyment, tym razem bardziej interesujący. Wywołajmy najpierw:

```
telnet horstmann.com 80
```

a następnie uważnie wpiszmy poniższy tekst:

```
GET / HTTP/1.0
Host: horstmann.com
pusty wiersz
```

Pusty wiersz oznacza w praktyce konieczność dwukrotnego naciśnięcia klawisza *Enter*.

Rysunek 3.3 pokazuje uzyskaną odpowiedź. Wygląda ona znajomo — uzyskaliśmy tekst głównej strony witryny autora tej książki w języku HTML.

Dokładnie samo połączenie wykorzystuje przeglądarka internetowa, ładowając stronę. Jedyna różnica polega na tym, że przeglądarka wyświetla kod HTML, wykorzystując różne czcionki.



Nazwa hosta jest wymagana, gdy łączymy się z serwerem obsługującym wiele domen o tym samym adresie IP. Jeśli serwer obsługuje jedną domenę, nazwę hosta możemy pominąć.

Nasz pierwszy program sieciowy zamieszczony w listingu 3.1 wykona dokładnie to samo działanie — połączy się do portu i wyświetli uzyskaną odpowiedź.

**Listing 3.1. socket/SocketTest.java**

```
package socket;

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

```

Terminal
Fichier Édition Affichage Rechercher Terminal Aide
~$ telnet horstmann.com 80
Trying 67.210.118.65...
Connected to horstmann.com.
Escape character is '^]'.
GET / HTTP/1.1
Host: horstmann.com

HTTP/1.1 200 OK
Date: Sun, 26 Jun 2011 01:05:27 GMT
Server: Apache/1.3.42 (Unix) Sun-ONE-ASP/4.0.2 mod_fastcgi/2.4.6 mod_log_bytes/1
.2 mod_bwlimited/1.4 mod_auth_passthrough/1.8 FrontPage/5.0.2.2635 mod_ssl/2.8.3
1 OpenSSL/0.9.7a
Last-Modified: Sun, 22 May 2011 08:11:15 GMT
ETag: "305e572-189f-4dd8c523"
Accept-Ranges: bytes
Content-Length: 6303
Content-Type: text/html

<?xml version="1.0" encoding="us-ascii"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Cay Horstmann's Home Page</title>

```

**Rysunek 3.3.** Połączenie telnetem do portu HTTP

```

/**
 * Program łączący się z zegarem atomowym
 * w Boulder, stan Colorado, i wyświetlający uzyskany czas.
 *
 * @version 1.20 2004-08-03
 * @author Cay Horstmann
 */
public class SocketTest
{
    public static void main(String[] args) throws IOException
    {
        try (Socket s = new Socket("time-A.timefreq.bldrdoc.gov", 13))
        {
            InputStream inStream = s.getInputStream();
            Scanner in = new Scanner(inStream);

            while (in.hasNextLine())
            {
                String line = in.nextLine();
                System.out.println(line);
            }
        }
    }
}

```

Kluczowe wiersze powyższego programu wyglądają następująco:

```

Socket s = new Socket("time-A.timefreq.bldrdoc.gov", 13);
InputStream inStream = s.getInputStream();

```

Pierwszy wiersz powoduje otwarcie *gniazdka*, które stanowi abstrakcję umożliwiającą komunikację programu przy użyciu sieci. Konstruktorowi gniazdka przekazujemy adres zdalnej maszyny oraz numer portu. Jeśli połączenie się nie uda, wyrzucony zostanie wyjątek `UnknownHostException`. Jeśli wystąpi problem innego rodzaju, to wyrzucony zostanie wyjątek `IOException`. Ponieważ klasa wyjątku `UnknownHostException` jest klasą pochodną klasy `IOException`, to w naszym prostym programie zajmujemy się jedynie wyjątkami klasy bazowej.

Gdy gniazdko zostanie otwarte, metoda `getInputStream` klasy `java.net.Socket` zwróci obiekt `InputStream`, który możemy wykorzystywać jak każdy inny strumień. Po uzyskaniu strumienia program wysyła każdy wczytany wiersz do standardowego wyjścia. Odbiera się to tak długo, aż strumień nie zostanie zamknięty, a połączenie zakończone.

Program taki może działać jedynie dla najprostszych usług sieciowych. W przypadku innych usług także klient wysyła pewne dane serwerowi, który nie musi zakończyć połączenia po wysłaniu odpowiedzi. W dalszej części tego rozdziału pokażemy implementację takiego działania na kilku przykładach.

Klasa `Socket` jest wyjątkowo łatwa w użyciu, ponieważ ukrywa przed nami skomplikowany proces tworzenia połączenia i przesyłania danych. Pakiet `java.net` udostępnia programistom interfejs, podobny jak w przypadku pracy z plikami.



W książce tej prezentujemy jedynie przykłady wykorzystania protokołu TCP (*Transmission Control Protocol*). Zestawia on niezawodne połączenie pomiędzy dwoma komputerami. Platforma Java umożliwia także użycie protokołu UDP (*User Datagram Protocol*), który pozwala na przesyłanie pakietów (*datagramów*) bez dodatkowego kosztu związanego z organizacją połączeń TCP. Wadą UDP jest to, że pakiety dostarczane są w dowolnej kolejności, a nawet mogą zostać utracone. Zadaniem odbiorcy jest uporządkowanie pakietów bądź zażądanie retransmisji utraconego pakietu. Protokół UDP jest szczególnie odpowiedni dla aplikacji, które mogą tolerować utracone pakiety, takie jak na przykład transmisja dźwięku bądź obrazu.

#### `java.net.Socket` 1.0

- `Socket(String host, int port)`  
tworzy gniazdo i łączy je z portem hosta zdalnego.
- `InputStream getInputStream()`
- `OutputStream getOutputStream()`  
zwraca strumienie umożliwiające odczyt i zapis danych do gniazda.

### 3.1.1. Limity czasu gniazd

Odczyt z gniazd blokuje wykonanie programu do momentu gdy odczytywane dane pojawią się w gnieździe. Jeśli nie można nawiązać połączenia z hostem, to aplikacja pozostanie забlokowana przez długi czas i pozostawać będzie na ląsce systemu operacyjnego, który w końcu zdecyduje o przekroczeniu limitu czasu operacji.

Dlatego też programista aplikacji sam powinien zdecydować o limitach czasowych związanych z komunikacją w sieci. Metoda setSoTimeout umożliwia określenie limitu czasu gniazda w milisekundach.

```
Socket s = new Socket( . . . );
s.setSoTimeout(10000); // limit czasu 10 sekund
```

Gdy określmy w ten sposób limit czasu gniazda, to wszystkie kolejne operacje odczytu i zapisu danych wyrzucać będą wyjątek SocketTimeoutException, jeśli nie zostaną wykonane przed upłynięciem limitu czasu. Możemy obsłużyć ten wyjątek i zareagować w ten sposób na sytuację związaną z wyczerpaniem limitu czasu.

```
try
{
    InputStream in = s.getInputStream();
    // odczyt ze strumienia in
    .
}
catch (InterruptedIOException exception)
{
    reakcja na upłynięcie limitu czasu
}
```

Istnieje jeszcze jedna sytuacja związana z określeniem limitu czasu. Konstruktor

```
Socket(String host, int port)
```

powoduje zablokowanie wykonywania programu do momentu nawiązania połączenia.

Możemy poradzić sobie z tym problemem, tworząc najpierw gniazdo niepołączone, a następnie zainicjować połączenie i określić limit czasu tej operacji:

```
Socket s = new Socket();
s.connect(new InetSocketAddress(host, port), timeout);
```

W podrozdziale 3.3, „Przerywanie działania gniazd sieciowych”, omówimy, w jaki sposób użytkownik aplikacji może przerwać połączenie gniazda w dowolnym momencie.

### java.net.Socket 1.0

- **Socket() 1.1**  
tworzy gniazdo, nie próbując go połączyć.
- **void connect(SocketAddress address) 1.4**  
Łączy gniazdo z podanym adresem.
- **void connect(SocketAddress address, int timeoutInMilliseconds) 1.4**  
Łączy gniazdo z podanym adresem w określonym limicie czasu.
- **void setSoTimeout(int timeout) 1.1**  
określa limit czasu blokowania wątku przez operacje odczytu danych z gniazda.  
Jeśli limit czasu zostanie wyczerpany, to wyrzucany jest wyjątek  
InterruptedException.

- boolean isConnected() **1.4**  
zwraca wartość true, jeśli gniazdo jest połączone.
- boolean isClosed() **1.4**  
zwraca wartość true, jeśli gniazdo jest zamknięte.

### 3.1.2. Adresy internetowe

Pisząc program, na ogół nie musimy specjalnie zajmować się adresami internetowymi — zapisanymi w postaci czterech bajtów (lub szesnastu w wersji IPv6) — na przykład 132.163.4.102. Jeśli musimy dokonać konwersji pomiędzy nazwami hostów a adresami internetowymi, to możemy wykorzystać klasę InetAddress.

Pakiet java.net obsługuje adresy internetowe IPv6, pod warunkiem że są one obsługiwane przez system operacyjny.

Metoda statyczna getByName zwraca obiekt klasy InetAddress dla danego hosta. Na przykład

```
InetAddress address  
= InetAddress.getByName("time-A.timefreq.bldrdoc.gov");
```

zwraca obiekt klasy InetAddress zawierający sekwencję czterech bajtów 132.163.4.104. Możemy uzyskać do nich dostęp, stosując metodę getAddress.

```
byte[] addressBytes = address.getAddress();
```

Niekłtórym hostom obsługującym ruch o dużym natężeniu odpowiada wiele adresów internetowych, umożliwiając w ten sposób rozłożenie obciążenia. Na przykład gdy pisaliśmy tę książkę, nazwa hosta google.com odpowiadała dwunastu różnym adresom internetowym. Przy próbie dostępu do hosta wybierany jest losowo jeden z adresów. Wszystkie adresy możemy uzyskać za pomocą metody getAllByName.

```
InetAddress[] addresses = InetAddress.getAllByName(host);
```

Czasami musimy uzyskać także adres lokalnego hosta. Jeśli zapytamy po prostu o adres maszyny localhost, to zawsze otrzymamy adres lokalnej pętli zwrotnej 127.0.0.1, który nie może być używany do łączenia się z naszym komputerem. Aby uzyskać właściwy adres lokalnego hosta, powinniśmy skorzystać z metody statycznej getLocalHost.

```
InetAddress address  
= InetAddress.getLocalHost();
```

Listing 3.2 zawiera tekst źródłowy prostego programu, który podaje adres internetowy lokalnego hosta, jeśli zostanie uruchomiony bez parametru. Jeśli natomiast, uruchamiając program, podamy mu jako parametr nazwę pewnego hosta, to program pokaże wszystkie jego adresy internetowe, na przykład:

```
java InetAddress/InetAddressTest www.horstmann.com
```

**Listing 3.2.** *inetAddress/InetAddressTest.java*

```

package inetAddress;

import java.io.*;
import java.net.*;

/**
 * Program demonstrujący zastosowanie klasy InetAddress.
 * W wierszu polecień przekazujemy mu jako parametr nazwę hosta.
 * W przeciwnym razie uzyskamy adres lokalnego hosta.
 * @version 1.02 2012-06-05
 * @author Cay Horstmann
 */
public class InetAddressTest
{
    public static void main(String[] args) throws IOException
    {
        if (args.length > 0)
        {
            String host = args[0];
            InetAddress[] addresses = InetAddress.getAllByName(host);
            for (InetAddress a : addresses)
                System.out.println(a);
        }
        else
        {
            InetAddress localHostAddress = InetAddress.getLocalHost();
            System.out.println(localHostAddress);
        }
    }
}

```

**API** `java.net.InetAddress 1.0`

- `static InetAddress getByName(String host)`  
tworzy obiekt klasy InetAddress lub tablicę wszystkich adresów internetowych danego hosta.
- `static InetAddress[] getAllByName(String host)`  
tworzy obiekt klasy InetAddress dla lokalnego hosta.
- `static InetAddress getLocalHost()`  
zwraca tablicę bajtów zawierającą adres internetowy w postaci numerycznej.
- `byte[] getAddress()`  
zwraca łańcuch znaków zawierający reprezentację adresu internetowego w postaci numerycznej, na przykład "132.163.4.102".
- `String getHostAddress()`  
zwraca nazwę hosta.
- `String getHostName()`

## 3.2. Implementacja serwerów

Ponieważ zaimplementowaliśmy już właściwie prostego klienta odbierającego informację z sieci, to nadeszła pora napisania prostego programu serwera wysyłającego taką informację. Program serwera oczekuje na klientów łączących się do jego portu. Dla potrzeb naszego przykładu wybierzemy port 8189, który nie jest zarezerwowany przez żadną za standardowych usług. Do utworzenia gniazdko wykorzystamy klasę ServerSocket. Poniższy wiersz programu

```
ServerSocket s = new ServerSocket(8189);
```

tworzy serwer monitorujący port 8189. Natomiast polecenie

```
Socket incoming = s.accept();
```

powoduje, że program oczekuje, aż klient przyłączy się do portu. Jeśli jakiś klient przyłączy się do portu, przesyłając odpowiednie żądanie przez sieć, to metoda ta zwróci obiekt klasy Socket reprezentujący utworzone połączenie. Obiekt ten możemy wykorzystać do uzyskania strumienia wejściowego i wyjściowego, tak jak w przykładzie poniżej:

```
InputStream inStream = incoming.getInputStream();
OutputStream outStream = incoming.getOutputStream();
```

Wszystko, co serwer wyśle do strumienia wyjściowego, pojawi się w strumieniu wejściowym klienta, a wszystko, co klient umieści w swoim strumieniu wyjściowym, zostanie przeczytane przez serwer ze strumienia wejściowego.

Ponieważ we wszystkich przykładach w tym rozdziale przesyłać przez sieć będziemy wyłącznie różne teksty, to strumienie wejściowe i wyjściowe konwertujemy odpowiednio do obiektów klasy Scanner i PrintWriter.

```
Scanner in = new Scanner(inStream);
PrintWriter out = new PrintWriter(outStream, true /*autoFlush */);
```

Wyślijmy więc klientowi na początek tekst pozdrowienia:

```
out.println("Hello! Enter BYE to exit." );
```

Jeśli połączymy się teraz do portu 8189 i tym samym naszego programu serwera, to zobaczymy tekst pozdrowienia w oknie terminala.

W przykładowym programie serwera będziemy czytać dane przesypane przez klienta i wysyłać każdą ich linię z powrotem, uzyskując efekt echo. Oczywiście prawdziwy program serwera wysyałby do klienta odpowiedź zależną od przesyłanych przez niego danych.

```
String line = in.nextLine();
out.println("Echo: " + line);
if (line.trim().equals("BYE")) done = true;
```

Na końcu zamknijemy wykorzystywane w programie gniazdko.

```
incoming.close();
```

I to wszystko. Każdy program serwera, na przykład serwer HTTP, wykonuje następującą pętlę:

1. pobiera informację wyslaną przez klienta ze strumienia wejściowego, na przykład zawierającą żądanie przesłania pewnej informacji,
2. dekoduje żądanie klienta
3. uzyskuje w pewien określony sposób informację żadaną przez klienta,
4. wysyła ją klientowi za pomocą strumienia wyjściowego.

Listing 3.3 zawiera kompletny program serwera.

---

**Listing 3.3. server/EchoServer.java**

---

```
package server;

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

/**
 * Program implementujący prosty serwer
 * nasłuchujący na porcie 8189
 * i wysyłający echo informacji otrzymanej od klienta.
 * @version 1.21 2012-05-19
 * @author Cay Horstmann
 */
public class EchoServer
{
    public static void main(String[] args) throws IOException
    {
        // tworzy gniazdo serwera
        try (ServerSocket s = new ServerSocket(8189))
        {
            // oczekuje na połączenie z klientem
            try (Socket incoming = s.accept())
            {
                InputStream inStream = incoming.getInputStream();
                OutputStream outStream = incoming.getOutputStream();

                try (Scanner in = new Scanner(inStream))
                {
                    PrintWriter out = new PrintWriter(outStream, true /* autoFlush */);

                    out.println("Hello! Enter BYE to exit.");

                    // wysyła echo informacji otrzymanej od klienta
                    boolean done = false;
                    while (!done && in.hasNextLine())
                    {
                        String line = in.nextLine();
                        out.println("Echo: " + line);
                        if (line.trim().equals("BYE")) done = true;
                    }
                }
            }
        }
    }
}
```

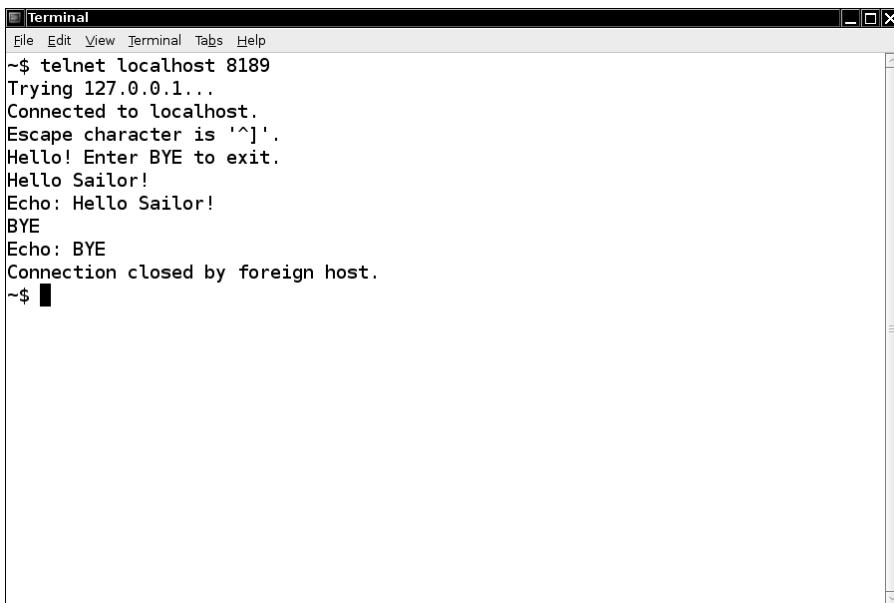
---

Aby przetestować jego działanie, należy go skompilować i uruchomić. Następnie, korzystając z telnetu, połączyć się z serwerem o adresie 127.0.0.1 port 8189.

Jeśli komputer podłączony jest bezpośrednio do Internetu, to serwer będzie dostępny dla każdego, kto zna adres IP komputera oraz numer portu używanego przez serwer.

Jeśli połączymy się do portu 8189, otrzymamy wiadomość pokazaną na rysunku 3.4:

Hello! Enter BYE to exit.



```
Terminal
File Edit View Terminal Tabs Help
~$ telnet localhost 8189
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^J'.
Hello! Enter BYE to exit.
Hello Sailor!
Echo: Hello Sailor!
BYE
Echo: BYE
Connection closed by foreign host.
~$
```

**Rysunek 3.4.** Połączenie z serwerem echo

Następnie możemy wpisywać dowolny tekst, otrzymując jego echo na ekranie. Dopiero wpisanie tekstu BYE (koniecznie dużymi literami) spowoduje rozłączenie połączenia, a program serwera zakończy działanie.

- |     |                           |
|-----|---------------------------|
| API | java.net.ServerSocket 1.0 |
|-----|---------------------------|
- `ServerSocket(int port) throws IOException`  
tworzy gniazdko serwera monitorującego port.
  - `Socket accept()`  
oczekuje na połączenie. Metoda ta blokuje wykonanie wątku do momentu utworzenia połączenia. Zwraca obiekt `Socket`, za pomocą którego program może komunikować się z klientem, który właśnie się połączył.
  - `void close()`  
zamyka gniazdko serwera.

### 3.2.1. Obsługa wielu klientów

Nasz przykładowy serwer ma jedną poważną wadę. Zwykle serwer działa w sieci non stop i umożliwia korzystanie ze swoich usług przez Internet wielu klientom równocześnie. Jeśli program serwera potrafi obsłużyć w danym momencie tylko jednego klienta, to pojedynczy klient może zmonopolizować dostęp do usługi, gdy będzie połączony przez dłuższy czas. Powinniśmy więc zapewnić możliwość równoczesnego dostępu do naszego serwera wielu klientom. W tym celu wykorzystamy wątki.

Za każdym razem gdy utworzone zostanie nowe połączenie, czyli wykonanie metody `accept` zakończy się utworzeniem kolejnego gniazdkła, uruchomimy nowy wątek dla obsługi połączenia z nowym klientem. Natomiast program główny powróci do stanu oczekiwania na nowe połączenia. W tym celu główna pętla programu serwera powinna wyglądać następująco:

```
while(true)
{
    Socket incoming = s.accept();
    Runnable r = ThreadedEchoHandler(incoming);

    Thread t = new Thread(r);
    t.start();
}
```

Klasa `ThreadedEchoHandler` implementuje interfejs `Runnable` i zawiera wewnętrz metody run pętle, w której odbywa się komunikacja z klientem.

```
class ThreadedEchoHandler implements Runnable
{
    ...
    public void run()
    {
        try
        {
            InputStream inStream = incoming.getInputStream();
            OutputStream outStream = incoming.getOutputStream();
            przetwarzanie danych wejściowych i wysyłanie odpowiedzi
            incoming.close();
        }
        catch (IOException e)
        {
            obsługa wyjątku
        }
    }
}
```

Ponieważ każde nowe połączenie powoduje uruchomienie nowego wątku, serwer może obsługiwać jednocześnie wielu klientów. Możemy to łatwo sprawdzić.

- 1 Kompilujemy i uruchamiamy serwer (listing 3.4).

---

**Listing 3.4.** *threaded/ThreadedEchoServer.java*

---

```
package threaded;

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

```

import java.util.*;

/**
 * Program implementujący wielowątkowy serwer
 * nasłuchujący na porcie 8189
 * i wysyłający echo informacji otrzymanej od klientów.
 * @author Cay Horstmann
 * @version 1.21 2012-06-04
 */
public class ThreadedEchoServer
{
    public static void main(String[] args )
    {
        try
        {
            int i = 1;
            ServerSocket s = new ServerSocket(8189);

            while (true)
            {
                Socket incoming = s.accept();
                System.out.println("Spawning " + i);
                Runnable r = new ThreadedEchoHandler(incoming);
                Thread t = new Thread(r);
                t.start();
                i++;
            }
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}

/**
 * Klasa obsługująca komunikację z pojedynczym klientem.
 */
class ThreadedEchoHandler implements Runnable
{
    private Socket incoming;

    /**
     * Tworzy obiekt obsługi.
     * @param i gniazdo wejściowe
     */
    public ThreadedEchoHandler(Socket i)
    {
        incoming = i;
    }

    public void run()
    {
        try
        {
            try
            {
                InputStream inStream = incoming.getInputStream();

```

```
OutputStream outStream = incoming.getOutputStream();

Scanner in = new Scanner(inStream);
PrintWriter out = new PrintWriter(outStream, true /* autoFlush */);

out.println( "Hello! Enter BYE to exit." );

// wyświetla echo
boolean done = false;
while (!done && in.hasNextLine())
{
    String line = in.nextLine();
    out.println("Echo: " + line);
    if (line.trim().equals("BYE"))
        done = true;
}
finally
{
    incoming.close();
}
}
catch (IOException e)
{
    e.printStackTrace();
}
}
```

- Otwieramy kilka okien programu telnet, co pokazano na rysunku 3.5.
  - Przełączamy się pomiędzy kolejnymi oknami i wpisujemy polecenia w każdym z nich. Zauważmy, że wszystkie okna mogą komunikować się równocześnie z serwerem.
  - Kończymy działanie serwera przez wybranie kombinacji klawiszy *Ctrl+C*.

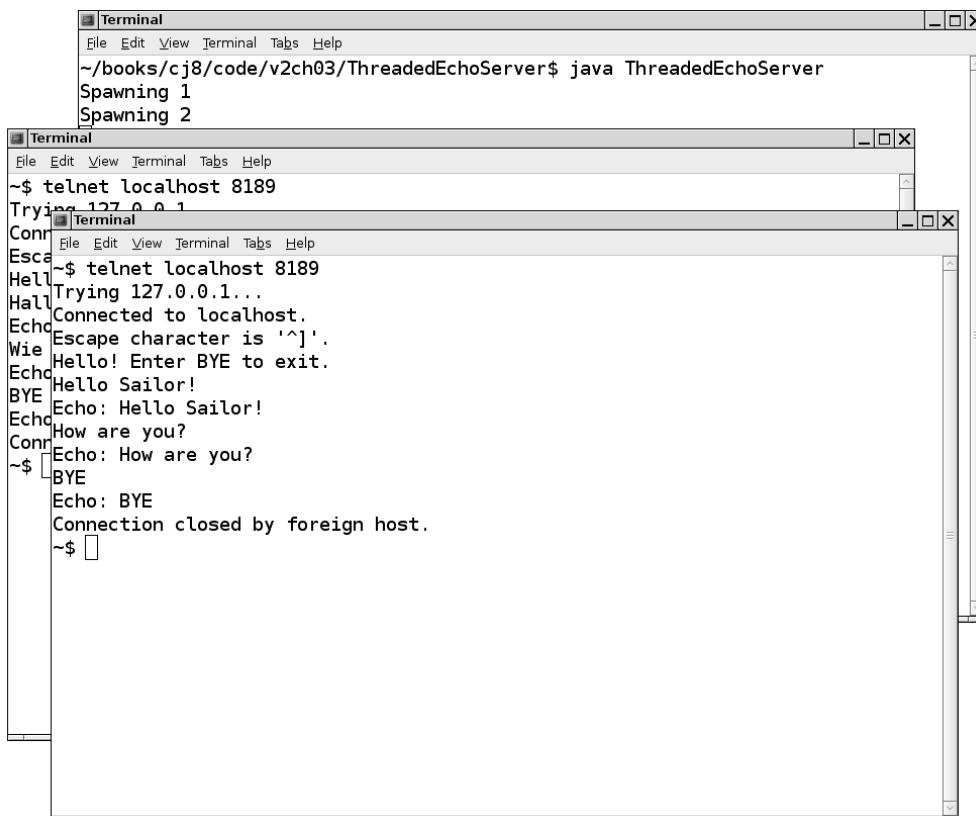


 Przedstawiony program serwera uruchamia osobny wątek dla obsługi każdego połączenia. Rozwiązanie takie nie jest wskazane w przypadku implementacji serwerów, które mają charakteryzować się wysoką efektywnością. Lepszą wydajność serwera można osiągnąć, używając możliwości oferowanych przez pakiet `java.nio`. Więcej informacji na ten temat znajduje się na stronie <http://www.ibm.com/developerworks/java/library/j-Javaio>.

### **3.2.2. Połączenia częściowo zamknięte**

*Połączenia częściowo zamknięte* umożliwiają zamknięcie strumienia wyjściowego gniazda i jednocześnie pozostawienie otwartego strumienia wejściowego.

Przykładem zastosowania może być sytuacja, w której wysyłamy do serwera dane, ale nie znamy ich ostatecznego rozmiaru. W przypadku pliku zamknęlibyśmy go po zapisaniu wszystkich danych. Jeśli jednak w podobny sposób zamknimy gniazdo, to połączenie z serwerem zostanie natychmiast zakończone i nie będziemy mogli odczytać odpowiedzi serwera.



**Rysunek 3.5.** Równoczesny dostęp do serwera echo wykorzystującego wątki

Zastosowanie połączenia częściowo zamkniętego rozwiązuje ten problem. Możemy wtedy zamknąć strumień wyjściowy gniazda, sygnalizując serwerowi koniec danych, ale zachować otwarty strumień wejściowy.

Program klienta działa wtedy następująco:

```
Socket socket = new Socket(host, port);
Scanner in = new Scanner(socket.getInputStream());
PrintWriter writer = new PrintWriter(
    socket.getOutputStream());
// wysyła żądane dane
writer.print( . . . );
writer.flush();
socket.shutdownOutput();
// gniazdo jest teraz częściowo zamknięte
// czyta odpowiedź serwera
while (in.hasNextLine() != null) { String line = in.nextLine(); . . . }
socket.close();
```

Program serwera czyta dane tak długo, aż jego strumień wejściowy się nie skończy. Dopiero wtedy wysyła klientowi odpowiedź.

Oczywiście ten sposób działania (protokół) jest przydatny jedynie w przypadku usług związanych z jednorazowym wysłaniem danych (na przykład HTTP), gdzie klient łączy się, wysyła żądanie, odbiera odpowiedź i się rozłącza.

#### **java.net.Socket 1.0**

- **void shutdownOutput() 1.3**  
sygnalizuje koniec strumienia wyjściowego.
- **void shutdownInput() 1.3**  
sygnalizuje koniec strumienia wejściowego.
- **boolean isOutputShutdown() 1.4**  
zwraca wartość true, jeśli strumień wyjściowy został zamknięty.
- **boolean isInputShutdown() 1.4**  
zwraca wartość true, jeśli strumień wejściowy został zamknięty.

## 3.3. Przerywanie działania gniazd sieciowych

Próba nawiązania połączenia za pomocą gniazda powoduje zablokowanie wątku do momentu, gdy połączenie zostanie nawiązane lub upłynie limit czasu. Podobnie w przypadku odczytu lub zapisu danych do gniazda wątek zostaje zablokowany, dopóki operacja nie zostanie pomyślnie wykonana lub nie upłynie limit czasu.

Aplikacje interaktywne powinny oferować użytkownikom możliwość przerwania niedziałającego połączenia. Jednak wątek zablokowany przez gniazdo nie może zostać przerwany przez wywołanie metody `interrupt()`.

Aby przerwać działanie gniazda, musimy użyć klasy `SocketChannel` udostępnianej przez pakiet `java.nio`. Kanał klasy `SocketChannel` tworzymy w następujący sposób:

```
SocketChannel channel = SocketChannel.open(new InetSocketAddress(host, port));
```

Z kanałem nie są związane strumienie. Zamiast nich posiada on metody `read` i `write` używające obiektów klasy `Buffer`. (Więcej informacji o buforach NIO zamieściliśmy w rozdziale 1.). Metody te zostały zadeklarowane w interfejsach `ReadableByteChannel` i `WritableByteChannel`.

Jeśli nie chcemy posługiwać się buforami, to do odczytu danych z kanału `SocketChannel` możemy użyć obiektu klasy `Scanner`, która posiada konstruktor o parametrze typu `ReadableByteChannel`:

```
Scanner in = new Scanner(channel);
```

Aby przekształcić kanał w strumień wyjściowy, używamy metody statycznej `Channel.newOutputStream()`:

```
OutputStream outStream = Channel.newOutputStream(channel);
```

I to wszystko. Gdy wątek zostanie przerwany podczas operacji nawiązywania połączenia, odczytu lub zapisu, to operacja ta nie zablokuje jego wykonania, lecz zostanie zakończona i wygeneruje wyjątek.

Program przedstawiony na listingu 3.5 ilustruje różnicę w zachowaniu gniazd blokujących i takich, których działanie można przerwać. Serwer wysyła strumień liczb do klienta, ale po dziesiątej z nich zawiesza dalsze działanie. Klikając przycisk *Interruptible* lub *Blocking*, uruchamiamy wątek, który nawiązuje połączenie z serwerem i wyświetla odebrane liczby. Jeśli w czasie wyświetlania tych liczb wybierzemy przycisk *Cancel*, to przerwiemy wątek niezależnie od tego, za pomocą którego przycisku został uruchomiony.

**Listing 3.5.** *interruptible/InterruptibleSocketTest.java*

```
package interruptible;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.net.*;
import java.io.*;
import java.nio.channels.*;
import javax.swing.*;

/**
 * Program prezentujący sposób przerwania działania kanału gniazda.
 * @author Cay Horstmann
 * @version 1.03 2012-06-04
 */
public class InterruptibleSocketTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new InterruptibleSocketFrame();
                frame.setTitle("InterruptibleSocketTest");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

class InterruptibleSocketFrame extends JFrame
{
    public static final int TEXT_ROWS = 20;
    public static final int TEXT_COLUMNS = 60;

    private Scanner in;
    private JButton interruptibleButton;
    private JButton blockingButton;
    private JButton cancelButton;
    private JTextArea messages;
    private TestServer server;
```

```
private Thread connectThread;

public InterruptibleSocketFrame()
{
    JPanel northPanel = new JPanel();
    add(northPanel, BorderLayout.NORTH);

    messages = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
    add(new JScrollPane(messages));

    interruptibleButton = new JButton("Interruptible");
    blockingButton = new JButton("Blocking");

    northPanel.add(interruptibleButton);
    northPanel.add(blockingButton);

    interruptibleButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            interruptibleButton.setEnabled(false);
            blockingButton.setEnabled(false);
            cancelButton.setEnabled(true);
            connectThread = new Thread(new Runnable()
            {
                public void run()
                {
                    try
                    {
                        connectInterruptibly();
                    }
                    catch (IOException e)
                    {
                        messages.append("\nInterruptibleSocketTest.connectInterruptibly: " + e);
                    }
                }
            });
            connectThread.start();
        }
    });
    blockingButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            interruptibleButton.setEnabled(false);
            blockingButton.setEnabled(false);
            cancelButton.setEnabled(true);
            connectThread = new Thread(new Runnable()
            {
                public void run()
                {
                    try
                    {
                        connectBlocking();
                    }
                }
            });
        }
    });
}
```

```

        catch (IOException e)
        {

messages.append("\nInterruptibleSocketTest.connectBlocking: " + e);
                }
            }
        });
connectThread.start();
    }
});

cancelButton = new JButton("Cancel");
cancelButton.setEnabled(false);
northPanel.add(cancelButton);
cancelButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        connectThread.interrupt();
        cancelButton.setEnabled(false);
    }
});
server = new TestServer();
new Thread(server).start();
pack();
}

/**
 * Łączy się z serwerem testowym, używając przerywalnych operacji wejścia-wyjścia.
 */
public void connectInterruptibly() throws IOException
{
    messages.append("Interruptible:\n");
try (SocketChannel channel = SocketChannel.open(new InetSocketAddress("localhost",
    ↴8189)))
{
    in = new Scanner(channel);
    while (!Thread.currentThread().isInterrupted())
    {
        messages.append("Reading ");
        if (in.hasNextLine())
        {
            String line = in.nextLine();
            messages.append(line);
            messages.append("\n");
        }
    }
}
finally
{
    EventQueue.invokeLater(new Runnable()
    {
        public void run()
        {
            messages.append("Channel closed\n");
            interruptibleButton.setEnabled(true);
            blockingButton.setEnabled(true);
        }
    });
}
}

```

```
        });
    }
}

/**
 * Łączy się z serwerem testowym, używając blokujących operacji wejścia-wyjścia
 */
public void connectBlocking() throws IOException
{
    messages.append("Blocking:\n");
    try (Socket sock = new Socket("localhost", 8189))
    {
        in = new Scanner(sock.getInputStream());
        while (!Thread.currentThread().isInterrupted())
        {
            messages.append("Reading ");
            if (in.hasNextLine())
            {
                String line = in.nextLine();
                messages.append(line);
                messages.append("\n");
            }
        }
    }
    finally
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                messages.append("Socket closed\n");
                interruptibleButton.setEnabled(true);
                blockingButton.setEnabled(true);
            }
        });
    }
}

/**
 * Serwer wielowątkowy nasłuchujący na porcie 8189 i wysyłający klientom wartości
 * liczbowe.
 * Po 10 liczbach symuluje "zawieszenie" działania.
 */
class TestServer implements Runnable
{
    public void run()
    {
        try
        {
            ServerSocket s = new ServerSocket(8189);

            while (true)
            {
                Socket incoming = s.accept();
                Runnable r = new TestServerHandler(incoming);
                Thread t = new Thread(r);
                t.start();
            }
        }
    }
}
```

```

        }
        catch (IOException e)
        {
            messages.append("\nTestServer.run: " + e);
        }
    }

/**
 * Klasa obsługująca połączenie z pojedynczym klientem.
 */
class TestServerHandler implements Runnable
{
    private Socket incoming;
    private int counter;

    /**
     * Tworzy obiekt obsługi.
     * @param i gniazdko połączenia
     */
    public TestServerHandler(Socket i)
    {
        incoming = i;
    }

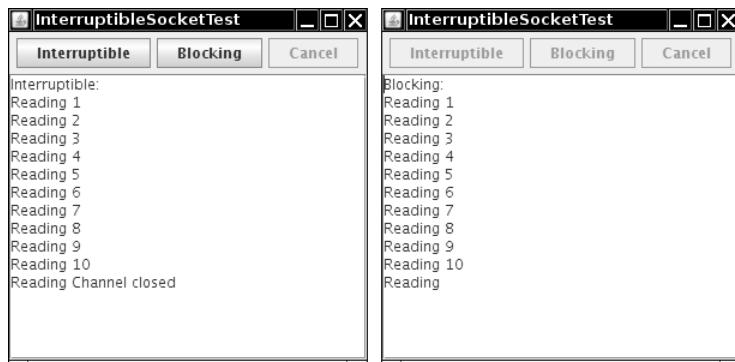
    public void run()
    {
        try
        {
            try
            {
                OutputStream outStream = incoming.getOutputStream();
                PrintWriter out = new PrintWriter(outStream, true /* autoFlush */);
                while (counter < 100)
                {
                    counter++;
                    if (counter <= 10) out.println(counter);
                    Thread.sleep(100);
                }
            }
            finally
            {
                incoming.close();
                messages.append("Closing server\n");
            }
        }
        catch (Exception e)
        {
            messages.append("\nTestServerHandler.run: " + e);
        }
    }
}

```

Jednak po odebraniu dziesięciu liczb możemy jedynie przerwać wątek uruchomiony za pomocą przycisku *Interruptible*. Wątek uruchomiony przyciskiem *Blocking* będzie pozostawać zablokowany do momentu, w którym serwer zamknie połączenie (patrz rysunek 3.6).

**Rysunek 3.6.**

Przerwanie połączenia



#### **API** java.net.InetSocketAddress 1.4

- `InetSocketAddress(String hostname, int port)`  
tworzy obiekt złożony z adresu hosta i jego portu, zamieniając przy tym nazwę hosta na jego adres. Jeśli operacja zamiany nie powiedzie się, to właściwość obiektu o nazwie `unresolved` posiadać będzie wartość `true`.
- `boolean isUnresolved()`  
zwraca wartość `true`, jeśli zamiana nazwy hosta na adres nie powiodła się.

#### **API** java.nio.channels.SocketChannel 1.4

- `static SocketChannel open(SocketAddress address)`  
otwiera gniazdo sieciowe i łączy je ze zdalnym adresem.

#### **API** java.nio.channels.Channels 1.4

- `static InputStream newInputStream(ReadableByteChannel channel)`  
tworzy strumień wejściowy służący do odczytu z danego kanału.
- `static OutputStream newOutputStream(WritableByteChannel channel)`  
tworzy strumień wyjściowy służący do zapisu w danym kanale.

## 3.4. Połączenia wykorzystujące URL

Efektywne programowanie aplikacji Java korzystających z dostępu do serwerów WWW wymaga wyższego poziomu abstrakcji niż nawiązywanie połączeń za pomocą gniazd sieciowych i wysyłanie poleceń protokołu HTTP. W pozostałej części tego rozdziału skoncentrujemy się na omówieniu klas dostarczanych w standardowej edycji platformy Java, które służą właśnie do tego celu.

### 3.4.1. URL i URI

Klasy URL oraz URLConnection hermetyzują złożony proces uzyskiwania informacji ze zdalnej lokalizacji. URL specyfikujemy w następujący sposób.

```
URL url = new URL(urlString);
```

Jeśli chcemy po prostu pobrać zawartość pewnego zasobu w sieci, to wykorzystać możemy metodę openStream klasy URL. Metoda ta udostępnia obiekt klasy InputStream. Zwykle używamy go do skonstruowania obiektu Scanner:

```
InputStream inStream = url.openStream();
Scanner in = new Scanner(inStream);
```

Pakiet java.net umożliwia rozróżnianie adresów URL (ang. *uniform resource locator*) oraz URI (ang. *uniform resource identifiers*).

URI jest konstrukcją czysto składniową, która specyfikuje różne części łańcucha określającego położenie zasobu w sieci. URL jest więc specjalnym rodzajem URI, wystarczającym do *zlokalizowania* zasobu. Inne rodzaje URI, na przykład

<mailto:cay@horstmann.com>

nie umożliwiają zlokalizowania zasobu w sieci, ponieważ z takim identyfikatorem URI nie są związane żadne dane. Ten rodzaj URI dla odróżnienia nazywamy więc URN (ang. *uniform resource name*).

W bibliotece języka Java klasa URI nie posiada metod dostępu do zasobu określonego przez ten identyfikator — podlega ona wyłącznie parsowaniu. Natomiast klasa URL może otworzyć strumień do zasobu. Z tego też powodu klasa URL działa jedynie z identyfikatorami, o których wie, w jaki sposób należy je obsługiwać, na przykład http:, https:, ftp:, file: (lokalny system plików) i jar: (pliki JAR).

Aby przekonać się, do czego potrzebna jest klasa URI, pokazujemy poniżej, jak skomplikowane potrafią być te identyfikatory.

```
http://maps.yahoo.com/py/maps.py?csz=Cupertino+CA
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

Specyfikacja URI określa sposób, w jaki zbudowane są takie identyfikatory. URI posiada następującą składnię:

`[scheme:]schemeSpecificPart[#fragment]`

gdzie [...] oznacza opcjonalną część identyfikatora, a znaki : oraz # są jego częściami.

Jeśli identyfikator zawiera część `scheme:`, to taki URI nazywamy *absolutnym*. W przeciwnym razie identyfikator jest *względny*.

Absolutny URI nazywamy *nieprzenikalnym*, jeśli jego część `schemeSpecificPart` nieaczyna się od znaku /, jak w poniższym przykładzie.

<mailto:cay@horstmann.com>

Wszystkie absolutne URI, które są przenikalne, oraz wszystkie URI względne są *hierarchiczne*. Należą do nich poniższe przykłady:

```
http://java.sun.com/index.html  
.../java/net/Socket.html#Socket()
```

Część *schemeSpecificPart* należąca do hierarchicznego URI posiada następującą strukturę:

```
[//authority][path][?query]
```

gdzie [...] , podobnie jak poprzednio oznacza część opcjonalną.

W przypadku URI dotyczących serwerów część *authority* posiada postać

```
[user-info@]host[:port]
```

gdzie *port* musi być liczbą całkowitą.

Dokument RFC 2396 standardyzujący postać URI specyfikuje także mechanizm wykorzystujący rejestr, gdzie część *authority* posiada inny format. Nie jest on jednak powszechnie wykorzystywany.

Jednym z zadań klasy URI jest parsowanie identyfikatora i jego rozkład na komponenty, które możemy pobrać, korzystając z następujących metod

```
getScheme  
getSchemeSpecificPart  
getAuthority  
getUserInfo  
getHost  
getPort  
getPath  
getQuery  
getFragment
```

Kolejnym zadaniem klasy URI jest obsługa identyfikatorów absolutnych i względnych. Jeśli dysponujemy absolutnym URI, na przykład

```
http://docs.mycompany.com/api/java/net/ServerSocket.html
```

oraz poniższym URI względnym

```
.../java/net/Socket.html#Socket()
```

to możemy połączyć je w jeden URI absolutny postaci

```
http://docs.mycompany.com/api/java/net/Socket.html#Socket()
```

Proces ten nazywamy *rozwiązywaniem* względnego identyfikatora URI.

Proces odwrotny do niego nosi natomiast nazwę *relatywizacji*. Na przykład założymy, że posiadamy następujący wyjściowy URI

```
http://docs.mycompany.com/api
```

oraz URI

```
http://docs.mycompany.com/api/java/lang/String.html
```

to identyfikator wzajemny powstały na skutek relatywizacji będzie miał postać

```
java/lang/String.html
```

Klasa URI dostarcza metod implementujących oba te procesy:

```
relative = base.relativize(combined);
combined = base.resolve(relative);
```

### 3.4.2. Zastosowanie klasy URLConnection do pobierania informacji

Jeśli potrzebujemy więcej informacji dotyczących pewnego zasobu w sieci, to powinniśmy wykorzystać klasę URLConnection, która umożliwia kontrolę nad zasobem w większym stopniu niż klasa URL.

Korzystając z klasy URLConnection, musimy przestrzegać następującego trybu postępowania.

- 1 Wywołujemy metodę openConnection klasy URL, aby uzyskać obiekt klasy URLConnection:

```
URLConnection connection = url.openConnection();
```

- 2 Skonfigurować właściwości żądania za pomocą metod:

```
setDoInput
setDoOutput
setIfModifiedSince
setUseCaches
setAllowUserInteraction
setRequestProperty
setConnectTimeout
setReadTimeout
```

Metody te omówimy w dalszej części tego podrozdziału i opiszemy dokładnie na jego końcu.

- 3 Łączymy się do zdalnego zasobu, wykorzystując metodę connect

```
connection.connect();
```

Oprócz utworzenia gniazdka połączenia do serwera metoda ta pobiera z serwera także *nagłówek*.

- 4 Po uzyskaniu połączenia z serwerem możemy sprawdzić informacje zawarte w nagłówku. Wyliczeniu pól nagłówka służą dwie metody, getHeaderFieldKey oraz getHeaderField. Istnieje również metoda getHeaderFields zwracająca obiekt klasy Map zawierający pola nagłówka. Poniższe metody pomogą w uzyskaniu informacji ze standardowych pól nagłówka.

```
getContentType
getContentLength
getContentEncoding
getDate
getExpiration
getLastModified
```

5. Na końcu możemy uzyskać wreszcie dostęp do danych zasobu. Metoda `getInputStream` zwraca strumień wejściowy umożliwiający odczytanie tej informacji. (Jest to ten sam strumień wejściowy, który zwraca metoda `openStream` klasy `URL`). Istnieje także metoda `getObject`, która okazuje się jednak mało pożyteczna. Obiekty zawracane dla standardowych typów zasobów, takich jak na przykład `text/plain` czy `image/gif`, wymagają wykorzystania klas hierarchii `com.sun`. Można także zarejestrować własne klasy obsługi, ale techniki tej nie będziemy dalej omawiać.



Niektórzy programiści tkwią w błędnym przekonaniu, że metody `getInputStream` oraz `getOutputStream` działają w przypadku klasy `URLConnection` tak samo jak w przypadku klasy `Socket`. Nie jest to do końca prawdą. Klasa `URLConnection` ukrywa przed programistą wiele dodatkowych działań związanych głównie z przetwarzaniem nagłówków żądań i odpowiedzi. Dlatego też ważne jest, by stosować się do procedury korzystania z klasy `URLConnection` przedstawionej powyżej.

Przyjrzyjmy się teraz bliżej niektórym metodom klasy `URLConnection`. Szereg metod służy do określenia właściwości połączenia do serwera, zanim jeszcze zostanie ono zestawione. Najważniejszymi z nich są `setDoInput` oraz `setDoOutput`. Domyslnie połączenie udostępnia strumień wejściowy do odczytu danych z serwera, ale nie strumień wyjściowy do zapisu. Jeśli potrzebujemy strumienia wyjściowego do umieszczania danych na serwerze, to musimy dodatkowo wywołać

```
connection.setDoOutput(true);
```

Z pomocą metod klasy `URLConnection` możemy także określić niektóre z nagłówków żądań. Są one wysyłane razem z komendami żądań do serwera. Poniżej podajemy przykład takiego nagłówka:

```
GET www.server.com/index.html HTTP/1.0
Referer: http://www.somewhere.com/links.html
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.4)
Host: www.server.com
Accept: text/html, image/gif, image/jpeg, image/png, /*
Accept-Language: en
Accept-Charset: iso-8859-1.*.utf-8
Cookie: orangemilano=1922188887821987
```

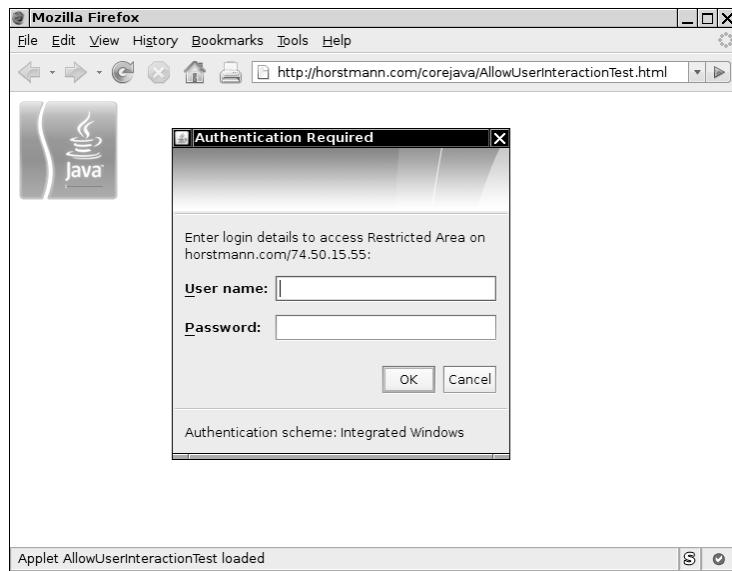
Metoda `setModifiedSince` wskazuje połączeniu, że jesteśmy zainteresowani jedynie danymi, które zostały zmodyfikowane od pewnej daty.

Metody `setUseCaches` oraz `setAllowUserInteraction` są stosowane jedynie przez aplenty. Metoda `setUseCaches` wskazuje przeglądarkę, aby sprawdziła najpierw bufor podręczny. Metoda `setAllowUserInteraction` pozwala przeglądarkę otworzyć okno dialogowe umożliwiające użytkownikowi wprowadzenie jego nazwy i hasła (patrz rysunek 3.7).

Metodę `setRequestProperty` możemy wykorzystać do skonfigurowania dowolnej pary nazwa-wartość, która jest istotna z punktu widzenia danego protokołu. Dokument RFC 2616 zawiera opis formatu nagłówków żądań protokołu HTTP. Niektóre z opisanych parametrów nie są wystarczająco udokumentowane. Na przykład chcąc uzyskać dostęp do strony internetowej chronionej hasłem, musimy kolejno wykonać następujące kroki.

**Rysunek 3.7.**

Okno dialogowe hasła dostępu do zasobu w sieci



- 1 Łączymy w jeden łańcuch nazwę użytkownika, znak dwukropka i hasło.

```
String input = username + ":" + password;
```

- 2 Uzyskujemy kodowanie powyższego ciągu zgodnie z regułą Base64 (koduje ona sekwencję bajtów za pomocą kodów ASCII posiadających interpretację graficzną).

```
String encoding = base64Encode(input);
```

- 3 Wywołujemy metodę `setRequestProperty` dla nazwy "Authorization" i wartości "Basic " + encoding:

```
connection.setRequestProperty("Authorization", "Basic " + encoding);
```



Powyżej przedstawiliśmy sposób dostępu do strony internetowej zabezpieczonej hasłem. Aby uzyskać dostęp do pliku zabezpieczonego hasłem znajdującego się na serwerze FTP, stosujemy zupełnie inną metodę. Tworzymy po prostu URL postaci

```
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

Po wywołaniu metody `connect` możemy sprawdzić informację zawartą w nagłówku odpowiedzi. Najpierw pokażemy jednak, w jaki sposób dokonać wyliczenia wszystkich pól nagłówka odpowiedzi. Niestety projektanci klasy postanowili w tym przypadku pokazać swój indywidualizm i wprowadzili jeszcze jeden protokół iteracji. Wywołanie

```
String key = connection.getHeaderFieldKey(n);
```

pobiera n-ty klucz nagłówka odpowiedzi, gdzie n zaczyna się od 1! W przypadku gdy n wynosi 0 lub jest większe od liczby pól nagłówka, metoda zwraca wartość null. Nie istnieje metoda, która zwróciłaby liczbę pól nagłówka. Musimy więc wywoływać metodę `getHeaderFieldKey` tak długo, aż otrzymamy wartość null. Wywołanie

```
String value = connection.getHeaderField(n);
```

zwraca natomiast n-tą wartość.

Metoda `getHeaderFields` zwraca obiekt klasy `Map` zawierający mapę pól nagłówka odpowiedzi:

```
Map<String, List<String>> headerFields = connection.getHeaderFields();
```

Poniżej przedstawiamy zbiór pól nagłówka odpowiedzi na typowe żądanie protokołu HTTP.

```
Date: Wed, 27 Aug 2008 00:15:48 GMT
Server: Apache/2.2.2 (Unix)
Last-Modified: Sun, 22 Jun 2008 20:53:38 GMT
Accept-Ranges: bytes
Content-Length: 4813
Connection: close
Content-Type: text/html
```

Dla wygody programistów udostępniono sześć metod zwracających wartości najczęściej spotykanych pól i przekształcających je do typów numerycznych, tam gdzie jest to uzasadnione (tabela 3.1). Metody zwracające wartość typu `long` podają liczbę sekund, które upłynęły od północy 1 stycznia 1970 roku czasu GMT.

**Tabela 3.1.** Metody zwracające wartości pól nagłówków odpowiedzi

Nazwa klucza	Nazwa metody	Typ zwracanej wartości
Date	getDate	long
Expires	getExpiration	long
Last-Modified	getLastModified	long
Content-Length	getContentLength	int
Content-Type	getContentType	String
Content-Encoding	getContentEncoding	String

Program, którego tekst źródłowy zawiera listing 3.6, pozwala eksperymentować z połączeniami wykorzystującymi URL. Jako jego parametry możemy podać URL oraz, opcjonalnie, nazwę użytkownika i hasło, jak poniżej:

```
java urlConnection.URLConnectionTest http://www.yourserver.com user password
```



Klasa `javax.mail.internet.MimeUtility`, którą znajdziemy w standardowym pakiecie rozszerzeń JavaMail, także udostępnia metodę kodowania Base64. JDK zawiera klasę `java.util.prefs.Base64`, która jednak nie jest publicznie dostępna i wobec tego nie możemy jej wykorzystać w naszym kodzie.

**Listing 3.6.** urlConnection/URLConnectionTest.java

```
package urlConnection;

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

```

/*
 * Program łączący się z lokalizacją określoną przez URL
 * i wyświetlający dane nagłówka odpowiedzi oraz pierwszych
 * 10 wierszy danych żądanego zasobu.
 *
 * Parametrem programu jest adres URL oraz, opcjonalnie,
 * nazwa i hasło służące do uwierzytelnienia użytkownika
 * w protokole HTTP.
 * @version 1.11 2007-06-26
 * @author Cay Horstmann
 */
public class URLConnectionTest
{
    public static void main(String[] args)
    {
        try
        {
            String urlName;
            if (args.length > 0) urlName = args[0];
            else urlName = "http://java.sun.com";

            URL url = new URL(urlName);
            URLConnection connection = url.openConnection();

            // wykorzystuje nazwę użytkownika i hasło, jeśli podane w wierszu poleceń
            if (args.length > 2)
            {
                String username = args[1];
                String password = args[2];
                String input = username + ":" + password;
                String encoding = base64Encode(input);
                connection.setRequestProperty("Authorization", "Basic " + encoding);
            }

            connection.connect();

            // wyświetla pola nagłówka
            Map<String, List<String>> headers = connection.getHeaderFields();
            for (Map.Entry<String, List<String>> entry : headers.entrySet())
            {
                String key = entry.getKey();
                for (String value : entry.getValue())
                    System.out.println(key + ": " + value);
            }

            // wyświetla rezultaty działania metod dostępu do wartości
            System.out.println("-----");
            System.out.println("getContentType: " + connection.getContentType());
            System.out.println("getContentLength: " + connection.getContentLength());
            System.out.println("getContentEncoding: " + connection.getContentEncoding());
            System.out.println("getDate: " + connection.getDate());
            System.out.println("getExpiration: " + connection.getExpiration());
            System.out.println("getLastModified: " + connection.getLastModified());
            System.out.println("-----");

            Scanner in = new Scanner(connection.getInputStream());

            // wyświetla pierwszych 10 wierszy danych zasobu

```

```

        for (int n = 1; in.hasNextLine() && n <= 10; n++)
            System.out.println(in.nextLine());
        if (in.hasNextLine()) System.out.println("...:");
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}

/**
 * Wyznacza kodowanie Base64 łańcucha znaków
 * @param s łańcuch znaków
 * @return łańcuch zakodowany za pomocą Base64
 */
public static String base64Encode(String s)
{
    ByteArrayOutputStream b0ut = new ByteArrayOutputStream();
    Base64OutputStream out = new Base64OutputStream(b0ut);
    try
    {
        out.write(s.getBytes());
        out.flush();
    }
    catch (IOException e)
    {
    }
    return b0ut.toString();
}

/**
 * Filtr strumienia przekształcający strumień bajtów na kod Base64.
 *
 * Base64 koduje 3 bajty za pomocą 4 znaków.
 * |11111122|22223333|33444444|
 * Każde 6 bitów kodowane jest według mapy Base64.
 * Jeśli liczba bajtów nie jest wielokrotnością 3, to ostatnia
 * grupa 4 znaków uzupełniana jest za pomocą jednego lub dwu
 * znaków =. Każdy wiersz zawiera co najwyżej 76 znaków.
 */
class Base64OutputStream extends FilterOutputStream
{
    private static char[] toBase64 = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I',
        'J', 'K', 'L',
        'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a',
        'b', 'c', 'd',
        'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's',
        't', 'u', 'v',
        'w', 'x', 'y', 'z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+',
        '/' };

    private int col = 0;
    private int i = 0;
    private int[] inbuf = new int[3];

    /**
     * Tworzy filtr strumienia.
     * @param out filtrowany strumień
     */
}

```

```

public Base64OutputStream(OutputStream out)
{
    super(out);
}

public void write(int c) throws IOException
{
    inbuf[i] = c;
    i++;
    if (i == 3)
    {
        if (col >= 76)
        {
            super.write('\n');
            col = 0;
        }
        super.write(toBase64[((inbuf[0] & 0xFC) >> 2)]);
        super.write(toBase64[((inbuf[0] & 0x03) << 4) | ((inbuf[1] & 0xF0) >> 4)]);
        super.write(toBase64[((inbuf[1] & 0x0F) << 2) | ((inbuf[2] & 0xC0) >> 6)]);
        super.write(toBase64[inbuf[2] & 0x3F]);
        col += 4;
        i = 0;
    }
}

public void flush() throws IOException
{
    if (i > 0 && col >= 76)
    {
        super.write('\n');
        col = 0;
    }
    if (i == 1)
    {
        super.write(toBase64[((inbuf[0] & 0xFC) >> 2)]);
        super.write(toBase64[((inbuf[0] & 0x03) << 4)]);
        super.write('\'');
        super.write('\'');
    }
    else if (i == 2)
    {
        super.write(toBase64[((inbuf[0] & 0xFC) >> 2)]);
        super.write(toBase64[((inbuf[0] & 0x03) << 4) | ((inbuf[1] & 0xF0) >> 4)]);
        super.write(toBase64[((inbuf[1] & 0x0F) << 2)]);
        super.write('\'');
    }
}

```

Program ten pokazuje:

- wszystkie klucze i wartości nagłówka,
- wartości zwracane przez metody wymienione w tabeli 3.1,
- pierwszych dziesięć wierszy żądanego zasobu.

Program jest dość nieskomplikowany z wyjątkiem obliczeń kodowania Base64. Istnieje nieudokumentowana klasa sun.misc.Base64Encoder, którą można wykorzystać zamiast zdefiniowanej w przykładzie. W tym celu wystarczy zastąpić wywołanie base64Encode za pomocą

```
String encoding  
= new sun.misc.BASE64Encoder().encode(input.getBytes());
```

Nie chcąc wykorzystywać nieudokumentowanych klas woleliśmy zdefiniować własną.

**API java.net.URL 1.0**

- `InputStream openStream()`  
otwiera strumień wejściowy umożliwiający odczyt danych zasobu.
- `URLConnection openConnection()`  
zwraca obiekt klasy `URLConnection` zarządzający połączeniem do zasobu.

**API java.netURLConnection 1.0**

- `void setDoInput(boolean doInput)`
- `boolean getDoInput()`  
jeśli `doInput` posiada wartość `true`, to użytkownik może pobierać dane z `URLConnection`.
- `void setDoOutput(boolean doOutput)`
- `boolean getDoOutput()`  
jeśli `doOutput` posiada wartość `true`, to użytkownik może wysyłać dane przez `URLConnection`.
- `void setIfModifiedSince(long time)`
- `long getIfModifiedSince()`  
właściwość `ifModifiedSince` konfiguruje `URLConnection` tak, by pobierało jedynie dane zmodyfikowane po określonym czasie. Czas ten określony jest w milisekundach, które upłynęły od północy GMT 1 stycznia 1970.
- `void setUseCaches(boolean useCaches)`
- `boolean getUseCaches()`  
właściwość `useCaches` posiada wartość `true`, to dane pobierane będą z lokalnego bufora. Zwróćmy uwagę, że `URLConnection` nie utrzymuje takiego bufora, a dostarczany jest on z zewnątrz, na przykład przez program przeglądarki.
- `void setAllowUserInteraction(boolean allowUserInteraction)`
- `boolean getAllowUserInteraction()`  
jeśli właściwość `allowUserInteraction` posiada wartość `true`, to użytkownik może zostać zapytany przez przeglądarkę o hasło.

- void setConnectTimeout(int timeout) **5.0**

- int getConnectTimeout() **5.0**

określa limit czasu dla nawiązania połączenia (w milisekundach). Jeśli czas ten upłynie przed nawiązaniem połączenia, to metoda read strumienia wejściowego wygeneruje wyjątek `SocketTimeoutException`.

- void setReadTimeout(int timeout) **5.0**

- int getConnectTimeout() **5.0**

określa limit czasu dla odczytu danych (w milisekundach). Jeśli czas ten upłynie przed odczytaniem danych, to metoda `connect` wygeneruje wyjątek `SocketTimeoutException`.

- void setRequestProperty(String key, String value)

umożliwia nadanie wartości polu nagłówka.

- Map<String, List<String>> getRequestProperties() **1.4**

zwraca mapę par klucz-wartość nagłówka żądania. Wszystkie wartości dla tego samego klucza zostają umieszczone na liście.

- void connect()

łączy się do zdalnego zasobu i pobiera nagłówek odpowiedzi.

- Map<String, List<String>> getHeaderFields() **1.4**

zwraca mapę par klucz-wartość pól nagłówka odpowiedzi. Wszystkie wartości dla tego samego klucza zostają umieszczone na liście.

- String getHeaderFieldKey(int n)

pobiera klucz n-tego pola nagłówka odpowiedzi lub zwraca wartość null, jeśli n jest mniejsze, lub równe 0, lub większe od liczby pól nagłówka odpowiedzi.

- String getHeaderField (int n)

pobiera wartość n-tego pola nagłówka odpowiedzi lub zwraca wartość null, jeśli n jest mniejsze, lub równe 0, lub większe od liczby pól nagłówka odpowiedzi.

- int getContentLength() zwraca wielkość zawartości, jeśli jest dostępna bądź wartość -1, jeśli jest nieznana.

- String getContentType

zwraca typ zawartości, na przykład `text/plain` bądź `image/gif`.

- String getContentEncoding()

zwraca sposób kodowania zawartości, na przykład `gzip`. Wartość ta nie jest najczęściej określona, ponieważ domyślne kodowanie `identity` nie jest podawane w nagłówku `Content-Encoding`.

- long getDate()

- long getExpiration()

- `long getLastModified()`

Metody te pobierają czas utworzenia, przeterminowania oraz ostatniej modyfikacji zasobu. Czas ten określony jest w milisekundach, które uplynęły od północy GMT 1 stycznia 1970.

- `InputStream openInputStream()`
- `OutputStream openOutputStream()`

Metody te zwracają strumień do odczytu bądź zapisu zasobu.

- `Object getContent()`

wybiera odpowiedni obiekt obsługi zawartości pozwalający odczytać dane zasobu i utworzyć z nich obiekt. Metoda ta nie jest przydatna w przypadku odczytu standardowych typów, takich jak `text/plain` lub `image/gif`, jeśli nie zainstalujemy własnych obiektów obsługi zawartości.

### 3.4.3. Wysyłanie danych do formularzy

W poprzednim podrozdziale pokazaliśmy sposób odczytu danych z serwera stron internetowych. Teraz natomiast zaprezentujemy możliwość wysyłania danych do serwera Web i uruchamianych przez niego programów.

Aby wysłać informację do serwera stron internetowych za pomocą przeglądarki, użytkownik wypełnia *formularz* na przykład, taki jak pokazano na rysunku 3.8.

Po wybraniu przycisku *Submit* zawartość pól tekstowych formularza oraz ustawienia innych kontrolek formularza (przyciski wyboru, listy wyboru etc.) wysyłane są do serwera Web. Dane te zostają następnie przetworzone przez program wywołany przez serwer.

Istnieje wiele technologii umożliwiających serwerom Web wykonywanie programów. Do najbardziej znanych należą serwlety Java, JavaServer Faces, Microsoft ASP (*Active Server Pages*) i skrypty CGI (*Common Gateway Interface*). Dla uproszczenia będziemy dalej używać określenia *skrypt* dla dowolnej z tych technologii.

Skrypt uruchomiony przez serwer przetwarza dane formularza i tworzy nową stronę w języku HTML, którą serwer przesyła z powrotem do przeglądarki. Sekwencję tę pokazuje rysunek 3.9. Strona odpowiedzi zawierać może nową informację (na przykład wynik wyszukiwania) bądź jedynie potwierdzenie. Strona odpowiedzi zostaje wyświetlona przez przeglądarkę.

W tej książce nie będziemy zajmować się implementacją skryptów wykonywanych przez serwer Web. Interesować będzie nas jedynie tworzenie programów klienckich współpracujących z istniejącymi skryptami serwera.

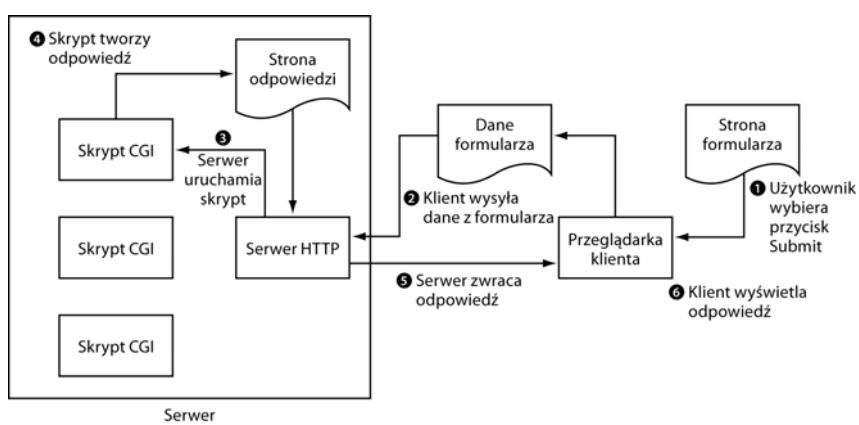
Wysyłając dane do serwera, nie musimy wiedzieć, czy będą one przetwarzane przez skrypt CGI, serwlet czy jeszcze inne rozwiązanie działające na serwerze. Dane przesyłane są bowiem w standardowym formacie do serwera, który następnie zajmuje się przekazaniem ich jednemu z programów, który przygotuje odpowiedź.

The screenshot shows a Mozilla Firefox browser window displaying the "World Population Prospects: The 2006 Revision Population Database". The main content area is titled "Panel 1: Basic data". On the left, a sidebar lists navigation options: Panel 1 (Basic data), Panel 2 (Detailed data), Country profile, Assumptions, Definition of regions, Sources, and Glossary. The copyright notice "Copyright © United Nations, 2007" is at the bottom of the sidebar.

The main panel contains two dropdown menus: "Select Variables (up to 5)" and "Select Country/Region (up to 5)". The "Select Variables" menu includes options like Population, Population density, Percentage urban, and Percentage rural. The "Select Country/Region" menu lists various countries and regions, with "Kenya" currently selected. Below these are dropdowns for "Select Variant" (Medium variant), "Select Start Year" (1950), "Select End Year" (2050), and a "Display" button, along with a link to "Download as .CSV File". At the bottom of the main panel are links to "Basic Data", "Detailed Data", "Country profile", "Assumptions", "Definition of regions", "Sources", and "Glossary". A footer note states "This website is last updated on 20-Sept-2007". The status bar at the bottom of the browser window shows "Done".

**Rysunek 3.8.** Formularz HTML

**Rysunek 3.9.**  
Przepływ danych podczas wykonywania skryptu CGI



Istnieją dwa sposoby wysyłania informacji do serwera: GET oraz POST.

Metoda GET polega na dołączeniu parametrów na końcu łańcucha URL. URL przybiera wtedy postać:

```
http://host/skrypt?parametry
```

Każdy z parametrów powyższego łańcucha URL ma postać *nazwa=wartość*. Parametry są oddzielone od siebie znakiem &. Wartości parametrów muszą być zakodowane zgodnie ze schematem *kodowania URL*, w którym obowiązują następujące reguły:

- Wszystkie znaki od A do Z, od a do z, od 0 do 9 oraz . - \* \_ pozostają bez zmian.
- Każdy odstęp zostaje zastąpiony znakiem +.
- Wszystkie pozostałe znaki zostają zakodowane za pomocą kodu UTF-8 i każdy z nich zostaje zamieniony na dwucyfrową liczbę szesnastkową poprzedzoną znakiem %.

Na przykład: jeśli zechcemy przesłać *New York, NY*, to zakodujemy ją jako *New+York%2c+NY*, ponieważ kod szesnastkowy 2c (dziesięćte 44) reprezentuje znak przecinka.

Kodowanie to zapobiega przetwarzaniu odstępów i znaków specjalnych przez programy pośredniczące.

Gdy pisaliśmy tę książkę, Yahoo wykorzystywało na przykład skrypt *py/maps.py* umieszczony na serwerze *maps.yahoo.com*. Wymagał on podania dwu parametrów *addr* oraz *csz*. Aby uzyskać mapę okolic adresu 1 Market Street, San Francisco, stan Kalifornia, należy utworzyć następujący URL:

```
http://maps.yahoo.com/py/maps.py?addr=1+Market+Street&csz=San+Francisco+CA
```

Metoda GET jest więc bardzo prosta, jednak posiada ograniczenie, które sprawia, że jest rzadko stosowana. Większość przeglądarek wprowadza bowiem ograniczenie liczby znaków, które można przesyłać metodą GET.

Metoda POST nie dodaje natomiast parametrów do URL. Wykorzystuje ona strumień wyjściowy *URLConnection*, do którego zapisuje pary nazwa-wartość. Pary te nadal muszą być poddane kodowaniu URL i rozdzielone znakiem &.

Przyjrzyjmy się temu procesowi nieco dokładniej. Aby wysłać dane, musimy najpierw utworzyć *URLConnection*.

```
URL url = new URL("http://host/script");
URLConnection connection = url.openConnection();
```

Następnie wywołujemy metodę *setDoOutput*, aby umożliwić wysyłanie danych.

```
connection.setDoOutput(true);
```

Metoda *getOutputStream* pozwoli uzyskać strumień, który wykorzystamy do wysyłania danych. Jeśli będą to tylko dane tekstowe, to wygodnie będzie opakować ten strumień za pomocą obiektu klasy *PrintWriter*.

```
PrintWriter out
= new PrintWriter(connection.getOutputStream());
```

Możemy już teraz wysłać dane do serwera:

```
out.print(name1 + "=" + URLEncoder.encode(value1, "UTF-8") + "&");  
out.print(name2 + "=" + URLEncoder.encode(value2, "UTF-8")));
```

Następnie musimy zamknąć strumień.

```
out.close();
```

Na końcu wywołujemy metodę getInputStream i odczytujemy odpowiedź serwera.

Prześledźmy działanie tego sposobu na przykładzie. Strona [http://esa.un.org/unpd/wpp/unpp/panel\\_population.htm](http://esa.un.org/unpd/wpp/unpp/panel_population.htm) zawiera formularz służący zbieraniu danych o populacji (patrz rysunek 3.8). Jeśli spojrzymy w jej tekst źródłowy, to odnajdziemy poniższą etykietę języka HTML:

```
<form method="post" onSubmit="return checksubmit(. . .)" . . .>
```

W kodzie JavaScript funkcji checksubmit odnajdziemy poniższy wiersz:

```
document.menuForm.action ="p2k0data_script.asp";
```

Informuje on nas o nazwie skryptu przetwarzającego akcję POST.

Musimy odnaleźć jeszcze nazwy pól oczekiwanych przez skrypt. Przyjrzyjmy się w tym celu elementom formularza. Każdy z nich posiada atrybut name, na przykład

```
<select name="Variable">  
  <option value="12;">Population</option>  
  pozostałe opcje . . .  
</select>
```

Nazwą tego pola jest Variable. Określa ono typ tabeli populacji. Jeśli wybierzemy wartość "12;", to otrzymamy tabelę całkowitej populacji. Dalej znajdziemy pole o nazwie Location, którego wartość równa 900 określa cały świat, a na przykład 404 dotyczy Kenii.

Istnieje jeszcze kilka innych pól, których wartości należy odpowiednio skonfigurować. Aby uzyskać prognozowane dane o populacji Kenii w latach od 1950 do 2050, utworzymy następujący łańcuch:

```
Panel=1&Variable=12%3b&Location=404&Varient=2&StartYear=1950&EndYear=2050;
```

Wyślemy go do URL

<http://esa.un.org/unpd/wpp/unpp/p2k0data.asp>

Skrypt wyśle następującą odpowiedź:

```
"Country","Variable","Variant","Year","Value"  
"Kenya","Population (thousands)","Medium variant","1950",6077  
"Kenya","Population (thousands)","Medium variant","1955",6984  
"Kenya","Population (thousands)","Medium variant","1960",8115  
"Kenya","Population (thousands)","Medium variant","1965",9524  
...  
...
```

Jak widzimy, skrypt utworzył w tym przypadku odpowiedź danych oddzielonych znakami przecinka. Skrypt ten nie tworzy skomplikowanej tabelki w języku HTML i właśnie dla tego wybraliśmy go jako przykład. Bez konieczności analizowania złożonej strony w języku HTML możemy łatwo zobaczyć rezultat jego działania.

Program zamieszczony w listingu 3.7 wysyła dane metodą za pomocą POST do dowolnego skryptu. Dane te należy umieścić w pliku *.properties* w następujący sposób:

```
url=http://esa.un.org/unpd/wpp/unpp/p2k0data_script.asp
Panel=1
Variable=12
Location=404
Varient=2
StartYear=1950
EndYear=2050
```

Program usunie pozycję url i przekaże wszystkie pozostałe metodzie doPost.

---

**Listing 3.7. post/PostTest.java**

---

```
package post;

import java.io.*;
import java.net.*;
import java.nio.file.*;
import java.util.*;

/**
 * Program używający klasy URLConnection do wysłania żądania POST.
 * @version 1.30 2012-06-04
 * @author Cay Horstmann
 */
public class PostTest
{
    public static void main(String[] args) throws IOException
    {
        Properties props = new Properties();
        try (InputStream in = Files.newInputStream(Paths.get(args[0])))
        {
            props.load(in);
        }
        String url = props.remove("url").toString();
        String result = doPost(url, props);
        System.out.println(result);
    }

    public static String doPost(String urlString, Map<Object, Object> nameValuePairs)
        throws IOException
    {
        URL url = new URL(urlString);
       URLConnection connection = url.openConnection();
        connection.setDoOutput(true);

        try (PrintWriter out = new PrintWriter(connection.getOutputStream()))
        {
            boolean first = true;
            for (Map.Entry<Object, Object> pair : nameValuePairs.entrySet())
            {
                if (first) first = false;
                else out.print('&');

                String name = pair.getKey().toString();
                String value = pair.getValue().toString();
                out.print(name + "=" + value);
            }
        }
    }
}
```

```

        out.print(name);
        out.print('=');
        out.print(URLEncoder.encode(value, "UTF-8"));
    }
}

StringBuilder response = new StringBuilder();
try (Scanner in = new Scanner(connection.getInputStream()))
{
    while (in.hasNextLine())
    {
        response.append(in.nextLine());
        response.append("\n");
    }
}
catch (IOException e)
{
    if (!(connection instanceof HttpURLConnection)) throw e;
    InputStream err = ((HttpURLConnection) connection).getErrorStream();
    if (err == null) throw e;
    Scanner in = new Scanner(err);
    response.append(in.nextLine());
    response.append("\n");
}

return response.toString();
}
}

```

Metoda `doPost` nawiązuje najpierw połaczenie, wywołując metodę `setDoOutput(true)`, po czym otwiera strumień wyjściowy. Następnie dokonuje wyliczenia nazw i wartości obiektu Map. Dla każdej z nich wysyła kolejno name, znak =, value i znak separatora &:

```

out.print(name);
out.print('=');
out.print(URLEncoder.encode(value, "UTF-8"));
if (istnieje więcej par) out.print('&');

```

Na końcu odczytuje odpowiedź serwera.

Z odczytaniem odpowiedzi serwera związana jest pewna sztuczka. Jeśli podczas próby wykonania skryptu wystąpi błąd, to wywołanie `connection.getInputStream()` wyrzuci wyjątek `FileNotFoundException`. Jednak serwer wyśle do przeglądarki stronę zawierającą informację o błędzie (na przykład „Error 404 — page not found”). Aby przejąć stronę zawierającą informację o błędzie, rzutujemy obiekt klasy `URLConnection` do klasy `HttpURLConnection` i wywołujemy jej metodę `getErrorStream`:

```

InputStream err
= ((HttpURLConnection)connection).getErrorStream();

```

Raczej z czystej ciekawości niż z powodu praktycznej przydatności możemy zastanawiać się, jaką informację oprócz naszych danych przesyła jeszcze do serwera klasa `URLConnection`.

Obiekt klasy `URLConnection` wysyła najpierw do serwera nagłówki żądania. Gdy wysyłamy dane formularza, to nagłówek musi zawierać:

`Content-Type: application/x-www-form-urlencoded`

Musimy również określić długość danych, na przykład:

Content-Length: 124

Pusty wiersz oznacza koniec nagłówka. Następnie wysyłane są dane. Serwer stron internetowych usuwa nagłówki i przekazuje dane do skryptu.

Zwróćmy uwagę, że obiekt klasy `URLConnection` musi najpierw zgromadzić wszystkie wysyłane przez nas dane w buforze, aby określić łączną długość zawartości.

Technika wykorzystywana przez nasz program jest przydatna zawsze, gdy zachodzi potrzeba uzyskania informacji z istniejącego już serwera stron internetowych. Wystarczy odnaleźć parametry, które należy przesłać (zwykle przeglądając w tym celu tekst źródłowy w języku HTML strony internetowej umożliwiającej wykonanie analogicznego zapytania), a następnie usunąć etykiety HTML i inne zbędne informacje z otrzymanej odpowiedzi.

#### API `java.net.HttpURLConnection 1.0`

- `InputStream getErrorStream()`  
zwraca strumień, za pomocą którego można odczytać informacje o błędach serwera internetowego.

#### API `java.net.URLEncoder 1.0`

- `static String encode(String s, String encoding) 1.4`  
zwraca łańcuch znaków s zakodowany jako URL zgodnie z podanym schematem kodowania. (Zaleca się schemat kodowania "UTF-8"). Kodowanie URL pozostawia niezmienione następujące znaki: 'A'-'Z', 'a'-'z', '0'-'9', '-', '\_', '.' oraz '~'. Odstęp jest kodowany za pomocą znaku '+', a wszystkie inne znaki przy użyciu łańcucha "%XY", gdzie 0xXY stanowi mniej znaczący bajt kodu znaku.

#### API `java.net.URLDecoder 1.2`

- `static String decode(String s, String encoding) 1.4`  
zwraca łańcuch znaków, który został uprzednio poddany kodowaniu URL.

## 3.5. Wysyłanie poczty elektronicznej

W przeszłości implementacja programu wysyłającego pocztę elektroniczną była bardzo prosta. Wystarczyło połączyć się do portu usługi SMTP, który posiada numer 25. SMTP (*Simple Mail Transport Protocol*) opisuje format wiadomości poczty elektronicznej. Możemy połączyć się z dowolnym serwerem usługi SMTP. Po połączeniu się z serwerem należy wysłać nagłówki wiadomości w formacie określonym przez SMTP a następnie treść wiadomości.

Poniżej przedstawiamy szczegóły tych operacji.

**1.** Otwieramy gniazdko połączenia do serwera.

```
Socket s = new Socket("mail.yourserver.com", 25); //port 25 usługa SMTP
PrintWriter out = new PrintWriter(s.getOutputStream());
```

**2.** Do strumienia wysyłamy następujące informacje:

```
HELO nazwa lokalnego hosta
MAIL FROM: adres pocztowy nadawcy
RCPT TO: adres pocztowy adresata
DATA
Subject: temat
(pusty wiersz)
treść wiadomości (dowolna liczba wierszy)

QUIT
```

Specyfikacja SMTP (RFC 812) wymaga, aby każda linia była zakończona znakiem \r, po którym następuje znak \n.

Dawniej serwery pocztowe przyjmowały wiadomości od każdego nadawcy. Dzisiaj, gdy w Internecie tłoczono od wiadomości zawierających informacje reklamowe, zdecydowana większość serwerów akceptuje jedynie wiadomości wysyłane przez określonych użytkowników, z określonych domen lub maszyn o adresach IP z pewnego zakresu. Ich uwierzytelnianie odbywa się zwykle przy użyciu bezpiecznych, szyfrowanych połączeń.

Implementacja uwierzytelniania byłaby dość pracochłonnym zadaniem. Dlatego poprzestaniemy na zaprezentowaniu wykorzystania interfejsu JavaMail do wysyłania wiadomości poczty elektronicznej przez programy w języku Java.

JavaMail należy załadować ze strony [www.oracle.com/technetwork/java/javamail](http://www.oracle.com/technetwork/java/javamail) i rozpakować na dysku lokalnym.

Aby korzystać z JavaMail, należy skonfigurować kilka właściwości zależnych od używanego serwera poczty elektronicznej. W przypadku poczty GMail konfiguracja ta będzie wyglądać następująco:

```
mail.transport.protocol=smtpls
mail.smtps.auth=true
mail.smtps.host=smtp.gmail.com
mail.smtps.user=cayhorstmann@gmail.com
```

Nasz przykładowy program wczyta ją z pliku właściwości.

Ze względów bezpieczeństwa nie należy umieszczać hasła w pliku właściwości, lecz prosić o nie podczas działania programu.

Po wczytaniu właściwości ustanawiamy sesję poczty elektronicznej w następujący sposób:

```
Session mailSession = Session.getDefaultInstance(props);
```

A następnie tworzymy wiadomość zawierającą nadawcę, odbiorcę, temat i treść:

```
MimeMessage message = new MimeMessage(mailSession);
message.setFrom(new InternetAddress(from));
message.addRecipient(RecipientType.TO, new InternetAddress(to));
```

```
message.setSubject(subject);
message.setText(builder.toString());
```

Teraz pozostaje nam tylko ją wysłać:

```
Transport tr = mailSession.getTransport();
tr.connect(null, password);
tr.sendMessage(message, message.getAllRecipients());
tr.close();
```

Program przedstawiony na listingu 3.8 wczytuje wiadomość z pliku tekstowego o następującym formacie:

```
Nadawca
Odbiorca
Temat
Treść wiadomości (dowolna liczba wierszy)
```

---

**Listing 3.8.** *mail/MailTest.java*

```
package mail;

import java.io.*;
import java.nio.charset.*;
import java.nio.file.*;
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.mail.internet.MimeMessage.RecipientType;

/**
 * Program ilustrujący sposób wykorzystania JavaMail
 * do wysyłania wiadomości poczty elektronicznej.
 * @author Cay Horstmann
 * @version 1.00 2012-06-04
 */
public class MailTest
{
    public static void main(String[] args) throws MessagingException, IOException
    {
        Properties props = new Properties();
        try (InputStream in = Files.newInputStream(Paths.get("mail",
            "mail.properties")))
        {
            props.load(in);
        }
        List<String> lines = Files.readAllLines(Paths.get(args[0]),
            Charset.forName("UTF-8"));

        String from = lines.get(0);
        String to = lines.get(1);
        String subject = lines.get(2);

        StringBuilder builder = new StringBuilder();
        for (int i = 3; i < lines.size(); i++)
        {
            builder.append(lines.get(i));
            builder.append("\n");
        }
```

```
}

Console console = System.console();
String password = new String(console.readPassword("Password: "));

Session mailSession = Session.getDefaultInstance(props);
// mailSession.setDebug(true);
MimeMessage message = new MimeMessage(mailSession);
message.setFrom(new InternetAddress(from));
message.addRecipient(RecipientType.TO, new InternetAddress(to));
message.setSubject(subject);
message.setText(builder.toString());
Transport tr = mailSession.getTransport();
try
{
    tr.connect(null, password);
    tr.sendMessage(message, message.getAllRecipients());
}
finally
{
    tr.close();
}
}
```

Program ten należy uruchomić w poniższy sposób:

```
java -classpath .:path/to/mail.jar path/to/message.txt
```

gdzie *mail.jar* jest plikiem JAR interfejsu programowego JavaMail. (Użytkownicy systemu Windows powinni pamiętać o użyciu średnika zamiast dwukropka poprzedzającego ścieżkę dostępu).

Gdy powstawała ta książka, GMail nie sprawdzał wiarygodności wprowadzonych informacji — można było podać dowolnego nadawcę. (Pamiętaj o tym, gdy dostaniesz pocztą elektroniczną zaproszenie od prezydenta USA na kolację w Białym Domu).



Jeśli połączenie z serwerem poczty nie działa, pomocne może okazać się poniższe wywołanie:

```
mailSession.setDebug(true);
```

Również dokumentacja JavaMail zawiera szereg wskazówek przydatnych do zidentyfikowania problemu.

W tym rozdziale omówiliśmy sposób implementacji serwerów sieciowych i ich klientów w języku Java. Przedstawiliśmy również sposoby pobierania informacji z serwerów WWW. Następny rozdział poświęciliśmy programowaniu aplikacji baz danych. Pokażemy w nim sposób, w jaki programy platformy Java mogą komunikować się z relacyjnymi bazami danych przy użyciu interfejsu programowego JDBC. Zamieścimy również krótkie wprowadzenie do hierarchicznych baz danych (takich jak na przykład katalogi LDAP) oraz interfejsu programowego JNDI.



# 4

## Programowanie baz danych: JDBC

W tym rozdziale:

- Architektura JDBC.
- Język SQL.
- Instalacja JDBC.
- Wykonywanie poleceń języka SQL.
- Wykonywanie zapytań.
- Przewijalne i aktualizowalne zbiory wyników zapytań.
- Zbiory rekordów.
- Metadane.
- Transakcje.
- Zarządzanie połączniami w zaawansowanych aplikacjach.

Latem 1996 roku firma Sun udostępniła pierwszą wersję pakietu Java Database Connectivity (JDBC). Pakiet ten umożliwia programistom łączenie się z bazami danych oraz wykonywanie zapytań i aktualizację danych w języku SQL (*Structured Query Language*), który jest standaryzowanym językiem dostępu do baz danych. Od tego czasu JDBC stało się jednym z najczęściej używanych interfejsów programowych oferowanych przez biblioteki języka Java.

Pakiet JDBC był aktualizowany kilka razy. Druga wersja pakietu JDBC pojawiła się w 1998 roku razem z Java SE 1.2. Wersję JDBC 3 dołączono do edycji Java SE 1.4 i 5.0. Gdy pisaliśmy tę książkę, dostępna była już wersja JDBC 4.1, którą dołączono do Java SE 7.

W rozdziale tym wyjaśniamy ideę działania pakietu JDBC i wprowadzamy temat języka SQL lub odświeżamy wiedzę użytkownika w tym zakresie. Dostarczamy następnie takiej ilości szczegółów i przykładów, by można było zacząć samodzielnie wykorzystywać możliwości JDBC w typowych zastosowaniach.



Według firmy Oracle nazwa JDBC stanowi zastrzeżony znak handlowy i nie jest tylko prostym skrótem pochodzącym od nazwy Java Database Connectivity. Nazwa ta jest celowo podobna do ODBC, która wprowadzona została przez firmę Microsoft dla oznaczenia pierwszego interfejsu programowego umożliwiającego dostęp do baz danych zgodnych z językiem SQL.

## 4.1. Architektura JDBC

Od samego początku projektanci technologii Java zdawali sobie sprawę z tkwiącego w niej potencjału, który można było wykorzystać do pracy z bazami danych. W 1995 roku rozpoczęto więc prace zmierzające do rozszerzenia standardowej biblioteki języka Java o dostęp do baz danych w języku SQL. Pierwszym zamierzeniem było utworzenie takiego rozszerzenia platformy Java, które umożliwiałoby dostęp do dowolnej bazy danych wyłącznie przez zastosowanie „czystego” języka Java. Wkrótce zrozumiano jednak, że jest to zadanie niewykonalne z powodu istnienia zbyt wielu systemów baz danych wykorzystujących różne sposoby dostępu. Co więcej, producenci baz danych sprzyjali tworzeniu standardowego sposobu dostępu do baz danych w języku Java jedynie pod warunkiem, że Java wykorzysta w tym celu właśnie ich specyficzne rozwiązanie.

W końcu producenci baz danych i narzędzi do tworzenia aplikacji zgodzili się, że najlepszym rozwiązaniem umożliwiającym dostęp do baz danych w języku SQL będzie interfejs programowy w języku Java razem z programem zarządzającym sterownikami, które dostarczane będą przez producentów do poszczególnych systemów baz danych. Powstać więc musiał również prosty mechanizm służący do rejestrowania sterowników w programie zarządzającym.

Rozwiązanie to naśladowało udany model dostępu do baz danych ODBC opracowany przez firmę Microsoft, który udostępniał interfejs dostępu do baz danych w języku C. JDBC i ODBC korzystają z tej samej koncepcji, według której programy używające interfejsu programowego komunikują się z programem zarządzającym, który z kolei wykorzystuje odpowiedni zarejestrowany sterownik do dostępu do bazy danych.

W praktyce oznacza to, że JDBC API jest interfejsem programowym wystarczającym dla większości programistów tworzących aplikacje baz danych.

### 4.1.1. Typy sterowników JDBC

Specyfikacja JDBC wyróżnia 4 typy sterowników.

- *Typ 1* tłumaczy JDBC na ODBC i wykorzystuje wobec tego sterownik ODBC do komunikacji z bazą danych. Wczesne wersje Javy dostarczały jeden sterownik tego typu — *mostek JDBC/ODBC*. Wymaga on właściwego zainstalowania i skonfigurowania sterownika ODBC. Udostępniono go jednak w celach testowych, a nie do stosowania w aplikacjach użytkowych. Obecnie dostępnych jest szereg innych, lepszych sterowników, dlatego też odradzamy wykorzystanie mostka.

- *Typ 2* napisany jest częściowo w języku Java, a częściowo w kodzie macierzystym danej platformy i komunikuje się z bazą danych, używając interfejsu programowego klienta danego systemu baz danych. Wykorzystując taki sterownik, musimy zainstalować więc oprócz biblioteki języka Java także pewien fragment oprogramowania specyficznego dla danej platformy.
- *Typ 3* napisany jest wyłącznie w języku Java i korzysta z protokołu komunikacji niezależnego od systemu bazy danych. Komunikacja ta odbywa się z komponentem serwera, który tłumaczy żądania sterownika na specyficzny protokół określonego systemu bazy danych. Dzięki temu oprogramowanie po stronie klienta jest niezależne od systemu bazy danych, co ułatwia jego wykorzystanie.
- *Typ 4* napisany jest wyłącznie w języku Java i tłumaczy żądania JDBC na specyficzny protokół danego systemu bazy danych.

Większość producentów systemów baz danych dostarcza sterowniki typu 3. lub 4. Niezależne firmy specjalizują się natomiast w tworzeniu sterowników lepiej spełniających standardy, dostępnych dla większej liczby platform, o poprawionej efektywności a czasami nawet większej niezawodności niż oryginalne sterowniki dostarczane przez producentów baz danych.

Podsumowując, niżej wyliczamy zasadnicze cele, dla których opracowano JDBC.

- Umożliwienie programistom tworzenia aplikacji w języku Java korzystających z dostępu do dowolnych typów baz danych za pomocą języka SQL — lub nawet specjalizowanych rozszerzeń SQL — przy zachowaniu konwencji obowiązujących w języku Java.
- Wykorzystanie zoptymalizowanych sterowników niskiego poziomu tworzonych przez producentów baz danych oraz producentów narzędzi służących tworzeniu aplikacji baz danych.



Zastanawiać mogą powody, dla których nie wykorzystano w języku Java po prostu standardu ODBC. Na konferencji JavaOne w 1996 roku podano następujące przyczyny takiej decyzji.

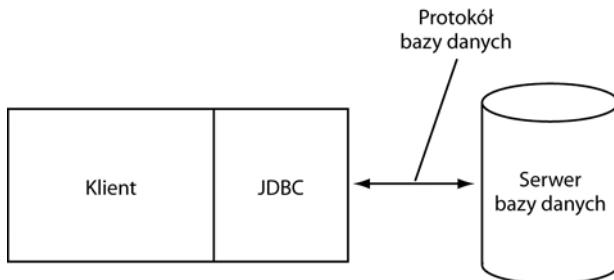
- Nauka posługiwania się standardem ODBC nie jest łatwa.
- ODBC dysponuje tylko kilkoma komendami, z których każda posiada wiele złożonych opcji. Nie jest to zgodne ze stylem programowania w języku Java, który polega na tworzeniu wielu prostych metod o intuicyjnym zastosowaniu.
- ODBC stosuje w szerokim zakresie wskaźniki typu `void *` oraz inne typowe cechy programów w języku C, których użycie w języku Java nie jest naturalne.
- Rozwiązania oparte o ODBC charakteryzują się niższym poziomem bezpieczeństwa i stwarzają większe trudności we wdrożeniu niż rozwiązania w języku Java.

## 4.1.2. Typowe zastosowania JDBC

Tradycyjny model klient-serwer dzieli aplikację na klienta wyposażonego w bogaty interfejs użytkownika i serwer bazy danych (patrz rysunek 4.1). W takim modelu sterownik JDBC znajduje się po stronie klienta.

**Rysunek 4.1.**

Aplikacja  
klient-serwer



Obecnie jednak aplikacje o architekturze klient-serwer ustępują miejsca rozwiązaniom o architekturze trójwarstwowej, a nawet *n*-warstwowej. W modelu trójwarstwowym klient nie zajmuje się dostępem do bazy danych, a jedynie komunikuje się z warstwą pośrednią na serwerze, która z kolei komunikuje się z bazą danych. Model taki posiada szereg zalet. Zapewnia oddzielenie *prezentacji* (przez klienta) od *logiki biznesowej* (w warstwie pośredniej) i danych (w bazie danych). Dzięki temu możliwe jest wykorzystanie tych samych danych i tych samych reguł biznesowych przez wielu różnych klientów, takich jak aplikacja w języku Java, applet czy formularz na stronie internetowej.

Komunikacja pomiędzy klientem a warstwą pośrednią odbywać się może przez HTTP (kiedy klientem jest przeglądarka internetowa), RMI (kiedy klientem jest aplikacja bądź applet — patrz rozdział 11.) lub przy użyciu jeszcze innego mechanizmu. JDBC zarządza wtedy komunikacją między warstwą pośrednią a znajdującą się za nią bazą danych. Rysunek 4.2 przedstawia podstawowy schemat takiej architektury. Istnieje też wiele odmian tego modelu. W szczególności Java Enterprise Edition definiuje strukturę *serwerów aplikacji*, które zarządzają modułami kodu zwartymi *Enterprise JavaBeans* i dostarcza dla nich szeregu usług związanych z równoważeniem obciążenia, buforowaniem żądań, bezpieczeństwem i mapowaniem obiektów na relacyjny schemat danych. W architekturze tej JDBC nadal odgrywa istotną rolę związaną z obsługą złożonych zapytań do baz danych. (Więcej informacji o Enterprise Edition na stronie <http://www.oracle.com/technetwork/java/javaee/overview>).

**Rysunek 4.2.**

Aplikacja  
o architekturze  
trójwarstwowej



JDBC mogą używać również aplenty i aplikacje Web Start, ale nie jest to najlepsze rozwiązanie. Domyslnie menedżer bezpieczeństwa zezwala appletowi jedynie na połączenie z bazą danych znajdującą się na tym samym serwerze, z którego został załadowany applet. Oznacza to, że serwer WWW i serwer bazy danych (lub komponent przekaźnika zgodny z 3. typem sterownika JDBC) muszą działać na tej samej maszynie, co nie jest zbyt częstą konfiguracją. Aby uniknąć tego ograniczenia, należy podpisać kod appletu.

## 4.2. Język SQL

JDBC stanowi interfejs języka SQL, który jest standardowym sposobem dostępu do wszystkich współczesnych relacyjnych baz danych. Osobiste bazy danych często posiadają interfejs użytkownika umożliwiający bezpośrednią manipulację danymi, natomiast wszystkie systemy baz danych pracujące na serwerach dostępne są wyłącznie w języku SQL.

Pakiet JDBC możemy uważać za rodzaj interfejsu programowego umożliwiającego przesyłanie poleceń języka SQL do systemu bazy danych. W podrozdziale tym przedstawimy krótkie wprowadzenie do języka SQL. Jeśli nie spotkałeś się nigdy wcześniej z językiem SQL, to może ono okazać się niewystarczające. W takim przypadku powinieneś uzupełnić swoją wiedzę, korzystając z jednej z wielu dostępnych książek. Polecamy zwłaszcza *Learning SQL* autorstwa Alana Beaulieu (O'Reilly 2005) lub *A Guide to the SQL Standard, Fourth Edition* napisaną przez Chrisa J. Date i Hugh Darwena (Addison-Wesley, 1997).

Bazę danych możemy wyobrazić sobie jako zbiór tabel składających się z kolumn i wierszy. Każda tabela posiada swoją nazwę, podobnie jak kolumna. Wiersze tabeli zawierają dane i nazywane są *rekordami*.

Jako przykładową bazę danych wykorzystamy w tej książce zbiór tabel opisujących kolekcje klasycznych książek z dziedziny informatyki (patrz tabele 4.1 – 4.4).

**Tabela 4.1.** Tabela Authors

Author_ID	Name	FName
ALEX	Alexander	Christopher
BROO	Brooks	Frederick P.
...	...	...

**Tabela 4.2.** Tabela Books

Title	ISBN	Publisher_ID	Price
A Guide to the SQL Standard	0-201-96425-0	0201	47.95
A Pattern Language: Towns, Buildings, Construction.	0-19-501919-9	019	65.00
...	...	...	...

**Tabela 4.3.** Tabela BooksAuthors

ISBN	Author_ID	Seq_no
0-201-96425-0	DATE	1
0-201-96425-0	DARW	2
0-19-501919-9	ALEX	1
...	...	...

**Tabela 4.4.** Tabela Publishers

Publisher_ID	Name	URL
0201	Addison-Wesley	www.aw-bc.com
0407	John Wiley & Sons	www.wiley.com
...	...	...

Rysunek 4.3 pokazuje zawartość tabeli Books. Na rysunku 4.4 widzimy natomiast wynik połączenia tej tabeli z tabelą Publishers. Tabele Books oraz Publishers zawierają identyfikator wydawcy. Jeśli za jego pomocą dokonamy połączenia tych tabel, to otrzymamy wynik zapytania złożony z wartości obu tablic. Każdy jego wiersz zawierać będzie informacje o książce razem z nazwą wydawcy i jego adresem w sieci Internet. Zwróćmy przy tym uwagę, że nazwy wydawców i adresy ich stron powtarzać się będą w wielu wierszach, ponieważ w wielu wierszach pojawią się ten sam wydawca.

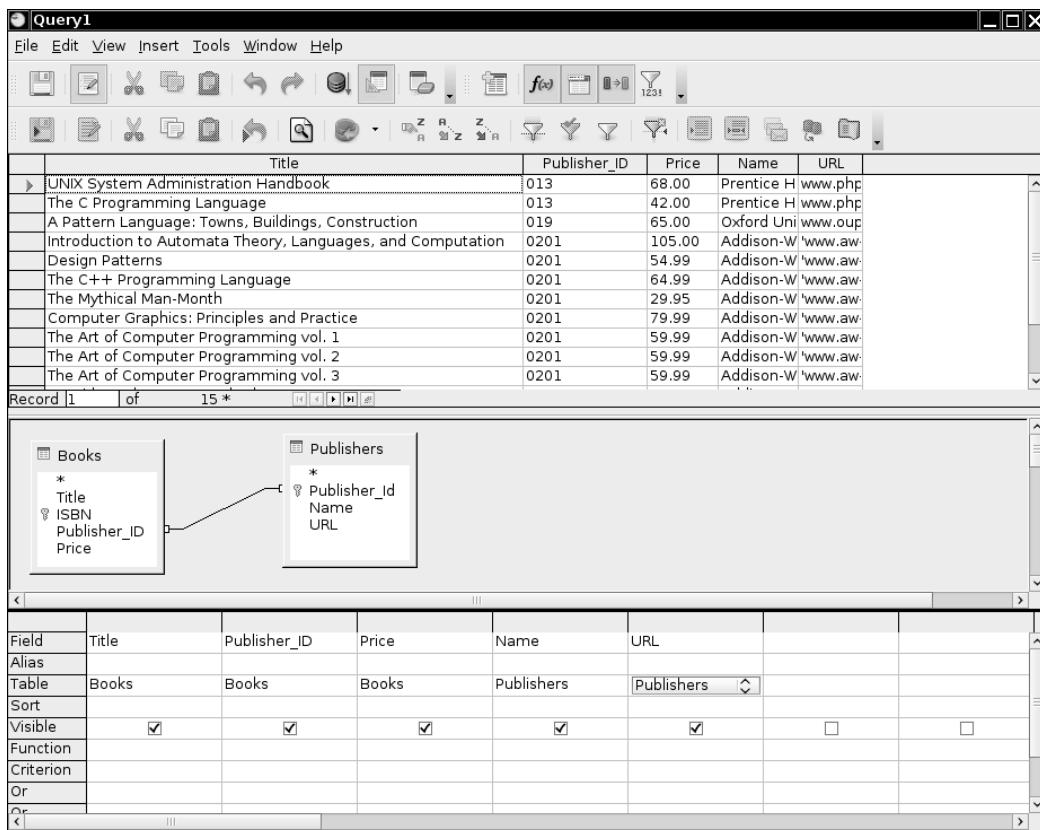
**Rysunek 4.3.**

Przykład tabeli zawierającej informacje o książkach

	Title	ISBN	Publisher_ID	Price
▶	UNIX System Administration Handbook	0-13-020601-6	013	68.00
	The C Programming Language	0-13-110362-8	013	42.00
	A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	019	65.00
	Introduction to Automata Theory, Languages, and Computation	0-201-44124-1	0201	105.00
	Design Patterns	0-201-63361-2	0201	54.99
	The C++ Programming Language	0-201-70073-5	0201	64.99
	The Mythical Man-Month	0-201-83595-9	0201	29.95
	Computer Graphics: Principles and Practice	0-201-84840-6	0201	79.99
	The Art of Computer Programming vol. 1	0-201-89683-4	0201	59.99
	The Art of Computer Programming vol. 2	0-201-89684-2	0201	59.99
	The Art of Computer Programming vol. 3	0-201-89685-0	0201	59.99
	A Guide to the SQL Standard	0-201-96426-0	0201	47.95
	Introduction to Algorithms	0-262-03293-7	0262	80.00
	Applied Cryptography	0-471-11709-9	0471	60.00
	JavaScript: The Definitive Guide	0-596-00048-0	0596	44.95
	The Cathedral and the Bazaar	0-596-00108-8	0596	16.95
	The Soul of a New Machine	0-679-60261-5	0679	18.95
	The Codebreakers	0-684-83130-9	07434	70.00
	Cuckoo's Egg	0-7434-1146-3	07434	13.95
	The UNIX Hater's Handbook	1-56884-203-1	0471	16.95

Operacja połączeń tablic zapobiega zbędnej duplikacji danych w tabelach. Nieco naiwny projekt bazy danych mógłby na przykład przewidywać umieszczenie informacji o nazwie wydawcy i jego stronie internetowej w tabeli Books. Wtedy już sama tabela, a nie dopiero wynik jej połączenia z inną tabelą, zawierałaby wiele duplikatów tej informacji. Jeśli zmieniłby się na przykład adres internetowy jednego wydawcy, to trzeba by zaktualizować wszystkie wiersze, które go zawierają. Operacja ta byłaby oczywiście potencjalnym źródłem błędów. Relacyjny model danych pozwala umieścić dane w wielu tabelach w taki sposób, że informacja nie jest duplikowana. Na przykład informacja o stronie internetowej danego wydawcy występuje tylko raz w tablicy Publishers. Jeśli chcemy uzyskać pełną informację o książce, to wykonujemy operację połączenia tabel.

Na rysunkach widzimy narzędzie umożliwiające inspekcję i łączenie tabel. Wielu producentów dostarcza narzędzi, które pozwalają wyrazić zapytania kierowane do bazy za pomocą łączenia kolumn tabeli bądź wypełniania formularzy. Narzędzia te określają się najczęściej



**Rysunek 4.4.** Połączenie dwóch tabel

wspólnym terminem *zapytania przez przykład* (ang. *query by example*, QBE). W przeciwieństwie do nich zapytania w języku SQL zapisywane są za pomocą tekstu zgodnie ze składnią języka. Na przykład:

```
SELECT Books.Title, Books.Publisher_id, Books.Price, Publishers.Name, Publishers.URL
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

W pozostałej części bieżącego podrozdziału omówimy zasady tworzenia takich zapytań. Jeśli poznałeś już wcześniej język SQL, to możesz pominąć lekturę tego podrozdziału.

Tradycyjnie słowa kluczowe języka SQL zapisuje się wielkimi literami, choć nie jest to obowiązkowe.

Operacja SELECT jest niezwykle uniwersalna. Za jej pomocą możemy pobrać wszystkie elementy tabeli Books:

```
SELECT * FROM Books
```

Słowo kluczowe FROM jest wymagane we wszystkich poleceniach SELECT. Określa ono tabele, na których operuje zapytanie.

Możemy także wybrać określone kolumny tabeli.

```
SELECT ISBN, Price, Title  
FROM Books
```

Zakres zapytania ograniczamy za pomocą klauzuli WHERE.

```
SELECT ISBN, Price, Title,  
FROM Books  
WHERE Price <= 29.95
```

Uwagę należy zwrócić na operacje porównania. Język SQL używa operatorów = oraz <> inaczej niż == oraz != w języku Java.



Niektórzy producenci dopuszczają stosowanie operatora !=. Nie jest to zgodne ze standardem języka SQL.

Klauzula WHERE wykorzystywać może także warunek dopasowania do wzorca przy użyciu operatora LIKE. Wzorzec dopasowania nie składa się jednak ze zwykle stosowanych znaków \* oraz ?, ale stosuje znak % dla oznaczenia zero lub więcej znaków oraz znak \_ dla pojedynczego znaku. Na przykład zapytanie:

```
SELECT ISBN, Price, Title  
FROM Books  
WHERE Title NOT LIKE '%n_x%'
```

wykluczy wszystkie książki, których tytuły zawierają takie słowa jak UNIX lub Linux.

Zwrócmy uwagę, że łańcuchy znaków ujęte są w pojedyncze, a nie podwójne znaki cudzysłowu. Pojedynczy znak cudzysłowu oznaczany jest wewnątrz łańcucha przez parę takich znaków. Na przykład:

```
SELECT Title  
FROM Books  
WHERE Books.Title NOT LIKE '%''%'
```

zwróci jako wynik zapytania wszystkie tytuły zawierające znak pojedynczego cudzysłowu.

Z pomocą polecenia SELECT możemy pobierać dane z wielu tabel.

```
SELECT * FROM Books, Publishers
```

Jednak bez klauzuli WHERE powyższe zapytanie nie jest specjalnie interesujące. Jako wynik zwraca ono *wszystkie kombinacje* wierszy obu tabel. W naszym przypadku, gdzie tabela Books zawiera 20 wierszy, a tabela Publishers 8 wierszy, zwróci ono tabelę o  $20 \times 8$  wierszach zawierającą dużą ilość zduplikowanej informacji. Możemy jednak ograniczyć zapytanie, podając, że interesują nas informacje o wydawcach poszczególnych książek.

```
SELECT * FROM Books, Publishers  
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

Wynikiem tego zapytania będzie 20 wierszy po jednym dla konkretnej książki, ponieważ każda z nich posiada określonego wydawcę.

Kiedy zapytanie operuje na wielu tabelach, może zdarzyć się, że ta sama nazwa kolumny wystąpi w nim wielokrotnie. W naszym przykładzie występuje kolumna Publisher\_Id zarówno

tabeli Publishers, jak i Books. Niejednoznaczności zapobiegamy w tym wypadku, poprzezając nazwę kolumny nazwą tablicy, do której należy, na przykład Books.Publisher\_Id.

Język SQL możemy wykorzystać także do modyfikacji danych w bazie. Założymy na przykład, że powinniśmy zmniejszyć o 5 USD cenę wszystkich książek, których tytuł zawiera słowo „C++”.

```
UPDATE Books
SET Price = Price - 5.00
WHERE Title LIKE '%C++%'
```

Podobnie aby usunąć wszystkie książki poświęcone językowi C++, zastosujemy polecenie DELETE:

```
DELETE FROM Books
WHERE Title LIKE '%C++%'
```

Język SQL posiada także wbudowane funkcje wyznaczania wartości średniej, znajdywania największej i najmniejszej wartości w danej kolumnie i wiele innych. Więcej informacji na ten temat znajdziemy, na przykład, na stronach witryny <http://sqlzoo.net> (zawiera ona również interakcyjny samouczek języka SQL).

Do wstawiania rekordów do tabeli bazy danych używamy polecenia INSERT.

```
INSERT INTO Books
VALUES('A Guide to the SQL Standard', '0-201-96426-0', '0201', 47.95)
```

Dla każdego wiersza musimy zastosować osobne polecenie INSERT.

Zanim zacznijemy pobierać, modyfikować i wstawiać dane do tabeli, musimy ją najpierw utworzyć. W tym celu używamy polecenia CREATE TABLE, określając przy tym nazwę tabeli oraz nazwy i typy jej kolumn.

```
CREATE TABLE Books
{
    Title CHAR(60),
    ISBN CHAR(13),
    Publisher_Id CHAR(6),
    Price DECIMAL(10, 2)
}
```

Tabela 4.5 przedstawia najważniejsze typy dostępne w języku SQL.

W książce tej nie będziemy omawiać dodatkowych klauzul polecenia CREATE TABLE związanych z tworzeniem kluczy i ograniczeń wartości.

## 4.3. Instalacja JDBC

Zanim zacznijemy korzystać z JDBC w tworzonych programach, musimy dysponować odpowiednim oprogramowaniem bazy danych zgodnym z JDBC. Możemy wybierać spośród wielu doskonałych produktów takich jak IBM DB2, Microsoft SQL Server, MySQL, Oracle czy PostgreSQL.

**Tabela 4.5.** Typy danych w języku SQL

Typ danych	Opis
INTEGER lub INT	Liczba całkowita o reprezentacji 32-bitowej (typowo).
SMALLINT	Liczba całkowita o reprezentacji 16-bitowej (typowo).
NUMERIC( <i>m, n</i> ), DECIMAL( <i>m, n</i> ) lub DEC( <i>m, n</i> )	Liczba stałoprzecinkowa o <i>m</i> cyfr, w tym <i>n</i> po przecinku.
FLOAT( <i>n</i> )	Liczba zmiennoprzecinkowa o precyzyji <i>n</i> cyfr binarnych.
REAL	Liczba zmiennoprzecinkowa o reprezentacji 32-bitowej (typowo).
DOUBLE	Liczba zmiennoprzecinkowa o reprezentacji 64-bitowej (typowo).
CHARACTER( <i>n</i> ) lub CHAR( <i>n</i> )	Łańcuch znaków o stałej długości <i>n</i> .
VARCHAR( <i>n</i> )	Łańcuch znaków o zmiennej długości nie większej niż <i>n</i> .
BOOLEAN	Wartość logiczna.
DATE	Data zależna od implementacji.
TIME	Czas zależny od implementacji.
TIMESTAMP	Data i czas zależne od implementacji.
BLOB	Duży obiekt binarny.
CLOB	Duży obiekt znakowy.

Następnie musimy utworzyć testową bazę danych. Przyjmiemy, że nazwą tej ostatniej będzie COREJAVA. Bază tą musimy utworzyć sami bądź poprosić administratora systemu bazy danych o jej wykreowanie wraz z odpowiednimi przywilejami, które pozwolą nam tworzyć, zmieniać i usuwać w niej tabele.

Jeśli nigdy wcześniej nie instalowałeś bazy danych o architekturze klient-serwer, to proces ten może wydać się dość skomplikowany, a ewentualne problemy z jego uruchomieniem trudne do zdiagnozowania. W takim wypadku lepiej poprosić o pomoc eksperta.

Innym rozwiązaniem, jeśli nie masz doświadczenia z bazami danych, będzie wykorzystanie bazy danych Apache Derby, która wchodzi w skład większości wersji pakietu JDK 7. (Jeśli używasz wersji JDK, która nie zawiera Apache Derby, to bazę tą możesz pobrać z witryny <http://db.apache.org/derby>).



Java używa nazwy JavaDB w odniesieniu do wersji bazy Apache Derby dołączonej do pakietu JDK. Aby uniknąć nieporozumień, w tym rozdziale będziemy konsekwentnie stosować nazwę Derby.

Zanim napiszemy pierwszy program korzystający z bazy danych, musimy skompletować szereg elementów, które omawiamy w poniższych podrozdziałach.

### 4.3.1. Adresy URL baz danych

Łącząc się z bazą danych, musimy określić jej adres, a także różne dodatkowe parametry, takie jak na przykład numer portu, gdy korzystamy z sterownika sieciowego czy atrybuty sterownika ODBC.

Do opisu źródła danych JDBC wykorzystuje składnię podobną do adresów URL. Poniżej dwa przykłady:

```
jdbc:derby://localhost:1527/COREJAVA;create=true
jdbc:postgresql:COREJAVA
```

Powyższe URL specyfikują bazę danych COREJAVA w systemie Derby oraz w systemie PostgreSQL.

Ogólna składnia wygląda następująco:

```
jdbc:subprotocol name:other stuff
```

Parametr *subprotocol* wykorzystywany jest do wyboru określonego sterownika.

Format parametru *other stuff* zależy od wartości parametru *subprotocol* i należy go odszukać w dokumentacji producenta.

### 4.3.2. Pliki JAR zawierające sterownik

Na początku musimy ustalić plik JAR, w którym znajduje się sterownik odpowiedni dla naszej bazy danych. W przypadku bazy danych Derby potrzebny będzie plik *derbyclient.jar*. Natomiast sterownik odpowiedni dla bazy danych PostgreSQL odnajdziemy na stronach witryny <http://jdbc.postgresql.org>.

Uruchamiając program korzystający z bazy danych, musimy podać nazwę pliku JAR zawierającego sterownik jako wartość parametru *classpath*. (Informacja o pliku JAR nie jest potrzebna podczas komilacji programu).

Uruchamiając program z wiersza poleceń, zastosujemy zatem następującą komendę:

```
java -classpath ścieżkaSterownika.: NazwaProgramu
```

W systemie Windows bieżący katalog (oznaczony za pomocą kropki) oddzielimy od nazwy pliku JAR za pomocą średnika, a nie dwukropka jak w powyższym przykładzie.

### 4.3.3. Uruchamianie bazy danych

Zanim będziemy mogli połączyć się z bazą danych, musimy najpierw ją uruchomić. Szczegóły tej operacji zależą w znacznym stopniu od konkretnej bazy danych.

W przypadku bazy danych Derby należy podjąć następujące działania:

1. Uruchomić powłokę systemu (wiersz poleceń) i przejść do katalogu zawierającego pliki bazy danych.
2. Odnaleźć plik *derbyrun.jar*. W niektórych wersjach JDK znajduje się on w katalogu *jdk/db/lib*, a w innych w osobnym katalogu instalacji bazy JavaDB. Dalej będziemy używać skrótu *derby* dla określenia katalogu zawierającego plik *lib/derbyrun.jar*.
3. Wykonać polecenie

```
java -jar derby/lib/derbyrun.jar server start
```

4. Sprawdzić, że baza została uruchomiona i działa poprawnie. W tym celu utworzymy plik *ij.properties* zawierający następujące wiersze:

```
ij.driver=org.apache.derby.jdbc.ClientDriver
ij.protocol=jdbc:derby://localhost:1527/
ij.database=COREJAVA:create=true
```

W innym oknie wiersza poleceń uruchamiamy interakcyjne narzędzie skryptów (ij) dostarczane z bazą Derby:

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties
```

Teraz możemy wydać polecenie SQL, Na przykład takie jak poniżej:

```
CREATE TABLE Greetings (Message CHAR(20));
INSERT INTO Greetings VALUES ('Hello, World!');
SELECT * FROM Greetings;
DROP TABLE Greetings;
```

Zwróćmy uwagę, że każde polecenie musi zostać zakończone znakiem średnika. Aby zakończyć pracę z programem ij, wydajemy polecenie:

```
EXIT;
```

5. Gdy nie używamy już bazy danych, możemy zakończyć jej pracę za pomocą poniższego polecenia:

```
java -jar derby/lib/derbyrun.jar server shutdown
```

Jeśli wybrałeś inną bazę danych, to sprawdź w dokumentacji, w jaki sposób należy ją uruchamiać i zatrzymywać oraz w jaki sposób można się z nią połączyć i wydawać polecenia SQL.

#### 4.3.4. Rejestracja klasy sterownika

Niektóre pliki JAR implementujące JDBC (na przykład sterownik bazy Derby dołączonej do Java SE 7) automatycznie rejestrują klasę sterownika. W takim przypadku możesz pominać lekturę tej części, w której omówimy ręczny sposób rejestracji klasy sterownika. Plik JAR może automatycznie zarejestrować klasę sterownika, jeśli zawiera plik *META-INF/services/java.sql.Driver*. Możesz sprawdzić jego obecność, rozpakowując plik JAR.



Mechanizm rejestracji wykorzystuje słabo znaną część specyfikacji plików JAR. Więcej informacji na ten temat znajdziesz na stronie <http://java.sun.com/javase/7/docs/technotes/guides/jar/jar.html#Service%20Provider>. Automatyczna rejestracja jest wymagana dla sterowników zgodnych z JDBC 4.

Jeśli używany przez Ciebie sterownik JDBC nie obsługuje automatycznej rejestracji, to musisz poznać nazwy klas tego sterownika JDBC. Poniżej przedstawiamy dwa przykłady nazw takich klas dla popularnych baz danych:

```
org.apache.derby.jdbc.ClientDriver  
org.postgresql.Driver
```

Istnieją dwa sposoby zarejestrowania sterownika. Pierwszy polega na załadowaniu klasy sterownika przez program w języku Java. Na przykład:

```
Class.forName("org.postgresql.Driver"); // wymusza załadowanie klasy sterownika
```

Powyzsze wywołanie powoduje załadowanie klasy sterownika i tym samym wykonanie statycznej metody inicjalizującej, która rejestruje sterownik.

Alternatywna metoda polega na odpowiednim skonfigurowaniu właściwości `jdbc.property`. Możemy to osiągnąć, stosując odpowiedni argument wywołania programu:

```
java -Djdbc.drivers=org.postgresql.Driver NazwaProgramu
```

Sama aplikacja również może skonfigurować właściwość za pomocą odpowiedniego wywołania, na przykład

```
System.setProperty("jdbc.drivers", "org.postgresql.Driver");
```

Możemy także zarejestrować wiele sterowników, oddzielając ich nazwy znakami dwukropka, jak poniżej.

```
org.postgresql.Driver:org.apache.derby.jdbc.ClientDriver
```

### 4.3.5. Nawiązywanie połączenia z bazą danych

Po zarejestrowaniu sterownika program w języku Java może nawiązać już połączenie z bazą danych przy użyciu kodu zblizonego do pokazanego poniżej:

```
String url = "jdbc:postgresql:COREJAVA";  
String username = "dbuser";  
String password = "secret";  
Connection conn = DriverManager.getConnection(url, username, password);
```

Menedżer sterowników spróbuje odnaleźć sterownik, który wykorzystuje podprotokół wyspecyfikowany w łańcuchu URL, przeglądając listę aktualnie zarejestrowanych sterowników.

Metoda `getConnection` zwraca obiekt klasy `Connection`. W następnych podrozdziałach wykorzystamy go do wykonywania zapytań.

Aby połączyć się z bazą danych, musimy znać nazwę użytkownika bazy i jego hasło.



Domyślnie baza danych Derby pozwala na połączenia z użytkownikiem o dowolnej nazwie i nie sprawdza hasła. Dla każdego użytkownika generowany jest osobny zbiór tabel. Domyślną nazwą użytkownika jest app.

Na listingu 4.1 przedstawiamy program, który wykonuje w praktyce omówione dotąd operacje. Ładuje on parametry połączenia z pliku *database.properties* i nawiązuje połączenie z bazą danych. Plik *database.properties* dołączony do kodu przykładów zawiera informacje niezbędne do nawiązania połączenia z bazą Derby. Jeśli używasz innej bazy danych, to powinieneś zmodyfikować jego zawartość. Poniżej przedstawiamy przykład jego zawartości dla bazy danych PostgreSQL:

```
jdbc.drivers=org.postgresql.Driver
jdbc.url=jdbc:postgresql:COREJAVA
jdbc.username=dbuser
jdbc.password=secret
```

**Listing 4.1.** test/TestDB.java

```
package test;

import java.nio.file.*;
import java.sql.*;
import java.io.*;
import java.util.*;

/**
 * Program sprawdzający poprawność konfiguracji
 * bazy danych i sterownika JDBC.
 * @version 1.02 2012-06-05
 * @author Cay Horstmann
 */
public class TestDB
{
    public static void main(String args[]) throws IOException
    {
        try
        {
            runTest();
        }
        catch (SQLException ex)
        {
            for (Throwable t : ex)
                t.printStackTrace();
        }
    }

    /**
     * Wykonuje test polegający na utworzeniu tabeli, wstawieniu do niej wartości,
     * prezentacji zawartości, usunięciu tabeli.
     */
    public static void runTest() throws SQLException, IOException
    {
        try (Connection conn = getConnection())
        {
            Statement stat = conn.createStatement();
```

```

stat.executeUpdate("CREATE TABLE Greetings (Message CHAR(20))");
stat.executeUpdate("INSERT INTO Greetings VALUES ('Hello, World!')");

try (ResultSet result = stat.executeQuery("SELECT * FROM Greetings"))
{
    if (result.next())
        System.out.println(result.getString(1));
}
stat.executeUpdate("DROP TABLE Greetings");
}

/**
 * Nawiązuje połączenie, korzystając
 * z właściwości w pliku database.properties
 * @return połączenie z bazą danych
 */
public static Connection getConnection() throws SQLException, IOException
{
    Properties props = new Properties();
    try (InputStream in = Files.newInputStream(Paths.get("database.properties")))
    {
        props.load(in);
    }
    String drivers = props.getProperty("jdbc.drivers");
    if (drivers != null) System.setProperty("jdbc.drivers", drivers);
    String url = props.getProperty("jdbc.url");
    String username = props.getProperty("jdbc.username");
    String password = props.getProperty("jdbc.password");

    return DriverManager.getConnection(url, username, password);
}
}

```

Po połączeniu się z bazą danych program testowy wykonuje następujące polecenia SQL:

```

CREATE TABLE Greetings (Message CHAR(20))
INSERT INTO Greetings VALUES ('Hello, World!')
SELECT * FROM Greetings

```

Program wyświetla wynik wykonania polecenia SELECT:

Hello, World!

a następnie usuwa tabelę z bazy danych, używając w tym celu poniższego polecenia:

```
DROP TABLE Greetings
```

Aby wykonać program testowy, uruchamiamy bazę danych i wywołujemy go w następujący sposób:

```
java -classpath .:JARsterownika test.TestDB
```

(Jak zwykle użytkownicy systemu Windows muszą użyć ; zamiast : do rozdzielenia elementów ścieżki dostępu).



Warto włączyć śledzenie na poziomie JDBC, aby łatwo diagnozować pojawiające się problemy. Wywołanie metody `DriverManager.setLogWriter` powoduje, że informacje zawierające szczegóły związane z działaniem JDBC wysyłane są do obiektu `PrintWriter`. Uzyskane tą drogą dane dają szczegółowy obraz aktywności JDBC. Większość implementacji sterowników JDBC dostarcza również dodatkowe mechanizmy śledzenia. W przypadku bazy Derby możemy włączyć taki mechanizm, umieszczając w adresie URL JDBC opcję `traceFile`, na przykład `jdbc:derby://localhost:1527/COREJAVA;create=true;traceFile=trace.out`.

#### **java.sql.DriverManager 1.1**

- `static Connection getConnection(String url, String user, String password)` nawiązuje połączenie z bazą danych i zwraca obiekt klasy `Connection`.

## 4.4. Wykonywanie poleceń języka SQL

Aby wykonać polecenie języka SQL, musimy utworzyć obiekt `Statement`. Możemy w tym celu wykorzystać obiekt `Connection` zwrócony przez wywołanie `DriverManager.getConnection`.

```
Statement stat = conn.createStatement();
```

Następnie tworzymy łańcuch znaków zawierający polecenie języka SQL.

```
String command = "UPDATE Books"
+ " SET Price = Price - 5.00"
+ " WHERE Title NOT LIKE '%Introduction%'";
```

Polecenie wykonujemy za pomocą metody `executeUpdate` interfejsu `Statement`:

```
stat.executeUpdate(command);
```

Metoda ta zwraca liczbę wierszy, które zostały zmodyfikowane na skutek wykonania polecenia. W naszym przykładzie będzie to liczba książek, których cenę obniżono o 5 USD.

Metoda `executeUpdate` może wykonywać operacje języka SQL, takie jak `INSERT`, `UPDATE` i `DELETE` oraz polecenia związane z definiowaniem danych, na przykład `CREATE TABLE` czy `DROP TABLE`. Jednak aby wykonać polecenie `SELECT`, musimy skorzystać z metody `executeQuery`. Istnieje także metoda `execute` umożliwiająca wykonanie dowolnego polecenia języka SQL. Wykorzystuje się ją najczęściej w celu wykonania poleceń języka SQL wprowadzonych interaktywnie przez użytkownika programu.

Wykonując zapytanie, jesteśmy oczywiście zainteresowani jego wynikiem. Metoda `executeQuery` zwraca obiekt typu `ResultSet`, który wykorzystujemy następnie do pobierania kolejnych wierszy wyniku.

```
ResultSet rs = stat.executeQuery("SELECT * FROM Books");
```

Pętla umożliwiająca przeglądanie wyniku zapytań posiada postać, którą przedstawiamy poniżej:

```
while (rs.next())
{
    operacja na pojedynczym wierszu
}
```



Protokół przeglądania klasy ResultSet różni się nieco od protokołu interfejsu java.util.Iterator. Iterator klasy ResultSet znajduje się początkowo na pozycji *przedającej* pierwszy rekord. Aby przesunąć go do pierwszego wiersza, musimy wywołać metodę next. Iterator ten nie dysponuje metodą hasNext i wobec tego wywołujemy metodę next, dopóki nie zwróci ona wartości false.

Uporządkowanie wierszy zbioru ResultSet jest zupełnie dowolne. Dopóki nie użyjemy klauzuli ORDER BY, nie powinniśmy przywiązywać żadnego znaczenia do ich kolejności.

Przeglądając zawartość wiersza tabeli, możemy skorzystać z wielu metod udostępniających wartość poszczególnych pól.

```
String isbn = rs.getString(1)
float price = rs.getDouble("Price");
```

Istnieją metody dostępu odpowiadające każdemu z typów języka Java, na przykład getString czy getDouble. Każda z nich posiada dwie wersje, jedną o parametrze numerycznym i drugą o parametrze w postaci łańcucha znaków. Dostarczając metodzie dostępowej parametr numeryczny, odwołujemy się do kolumny o danym numerze. Na przykład rs.getString(1) zwraca wartość pierwszej kolumny dla danego wiersza.



W przeciwieństwie do indeksów tablic numeracja kolumn rozpoczyna się od wartości 1.

Dostarczając metodzie dostępowej parametr w postaci łańcucha znaków, odwołujemy się do kolumny za pomocą jej nazwy. Na przykład rs.getDouble("Price") zwraca wartość kolumny o nazwie Price dla danego rekordu. Zastosowanie parametru numerycznego jest nieco efektywniejsze, natomiast użycie łańcucha znaków prowadzi do uzyskania bardziej czytelnego i łatwiejszego w utrzymaniu kodu.

Każda z metod dostępowych dokonuje konwersji typu, w przypadku gdy zwracany przez nią typ nie jest zgodny z typem kolumny. Na przykład wywołanie rs.getDouble("Price") dokonuje konwersji wartości zmiennoprzecinkowej na łańcuch znaków.

#### *java.sql.Connection 1.1*

- Statement createStatement()

tworzy obiekt wykorzystywany do wykonywanie zapytań i innych poleceń języka SQL.

- void close()

zamyka natychmiast połączenie do bazy danych.

**API** `java.sql.Statement 1.1`

- `ResultSet executeQuery(String sqlQuery)`  
wykonuje zapytanie języka SQL podane w postaci łańcucha znaków i zwraca obiekt `ResultSet` umożliwiający przeglądanie wyników zapytania.
- `int executeUpdate(String sqlStatement)`  
wykonuje polecenie `INSERT`, `UPDATE` lub `DELETE` języka SQL podane w postaci łańcucha znaków. Może także wykonywać polecenia podzbioru języka SQL — języka definicji danych DDL (*Data Definition Language*), takie jak na przykład `CREATE TABLE`. Zwraca liczbę zmodyfikowanych rekordów lub wartość `-1` w przypadku poleceń DDL.
- `boolean execute(String sqlStatement)`  
wykonuje polecenie języka SQL podane w postaci łańcucha znaków. Rezultatem polecenia może być wiele zbiorów wyników oraz liczniki aktualizacji. Zwraca wartość `true`, jeśli jego wynikiem jest zbiór rekordów, `false` — w przeciwnym razie. Aby uzyskać wynik wykonania polecenia, można zastosować metodę `getResultSet` lub `getUpdateCount`. Więcej informacji na temat przetwarzania wielu zbiorów wyników znajdziesz w punkcie 4.5.4., „Zapytania o wielu zbiorach wyników”.
- `int getUpdateCount()`  
zwraca liczbę rekordów zmodyfikowanych przez poprzednie polecenie lub wartość `-1`, jeśli poprzednie polecenie było innego typu. Metodę tę należy wywoływać tylko raz dla danego polecenia.
- `ResultSet getResultSet()`  
zwraca obiekt `ResultSet` zawierający wynik poprzedniego polecenia lub wartość `null`, jeśli nie tworzyło ono wyniku. Metodę tę należy wywoływać tylko raz dla danego polecenia.
- `void close()`  
zamyka obiekt polecenia i związany z nim zbiór wyników.
- `boolean isClosed() 6`  
zwraca wartość `true`, jeśli obiekt polecenia został zamknięty.
- `void closeOnCompletion() 7`  
powoduje, że obiekt polecenia zostanie zamknięty, gdy zamknięte zostaną wszystkie jego zbiory wyników.

**API** `java.sql.ResultSet 1.1`

- `boolean next()`  
przesuwa się o jeden element w zbiorze wyników zapytania. Zwraca wartość `false` po ostatnim rekordzie. Zwróćmy uwagę, że metodę tę trzeba wywołać, aby przejść do pierwszego rekordu w zbiorze.

- `Xxx getXxx(int columnNumber)`
- `Xxx getXxx(String columnName)`  
(gdzie Xxx jest typem int, double, String, Date itp.)
- `<T> T getObject(int columnNumber, Class<T> type) 7`
- `<T> T getObject(String columnLabel, Class<T> type) 7`  
Zwracają wartość kolumny o podanym numerze lub etykiecie przekształconą do określonego typu. Etykieta kolumny jest etykietą określona w klauzuli AS polecenia SQL lub nazwą kolumny, w przypadku gdy taka klauzula nie została użyta.
- `int findColumn(String columnName)`  
zwraca indeks kolumny o podanej nazwie.
- `void close()`  
zamyka natychmiast zbiór wyników zapytania.
- `boolean isClosed() 6`  
zwraca wartość true, jeśli obiekt polecenia został zamknięty.

#### 4.4.1. Zarządzanie połączeniami, poleceniami i zbiorami wyników

Dla każdego obiektu `Connection` możemy utworzyć jeden lub więcej obiektów `Statement`. Ten sam obiekt `Statement` możemy wykorzystać wiele razy do wykonania różnych, niezwiązanych z sobą poleceń języka SQL. Musimy jednak wiedzieć o tym, że obiekt ten dysponuje *co najwyżej jednym* obiektem typu `ResultSet`. Jeśli więc w programie wykonujemy szereg zapytań, których wyniki musimy później analizować równocześnie, to musimy utworzyć wiele obiektów `Statement`.

Ostrzegamy, że przynajmniej jeden z najczęściej wykorzystywanych systemów baz danych (Microsoft SQL Server) posiada sterownik JDBC, który dopuszcza tylko jeden aktywny obiekt `Statement`. Aby uzyskać informację o liczbie aktywnych obiektów `Statement`, którą dopuszcza używany sterownik JDBC, możemy wykorzystać metodę `getMaxStatement` interfejsu `DatabaseMetaData`.

Choć wygląda to na poważne ograniczenie, to w praktyce powinniśmy jednak unikać równoczesnego wykorzystywania rezultatów wielu zapytań. Jeśli są one ze sobą powiązane, to z reguły możemy utworzyć złożone zapytanie i analizować tylko jego wynik. Zresztą wykonanie złożonych zapytań przez bazy danych jest dużo efektywniejsze niż przeglądanie wielu wyników zapytań za pomocą programu w języku Java.

Po zakończeniu korzystania z obiektów `ResultSet`, `Statement` czy `Connection` powinniśmy natychmiast wywołać metodę `close`. Obiekty te używają rozbudowanych struktur danych i nie należy czekać, aż mechanizm automatycznego odzyskiwania zasobów sam je usunie.

Metoda `close` obiektu `Statement` zamyka automatycznie związkę z nim zbiór wyników (jeśli jest otwarty). Podobnie metoda `close` należąca do klasy `Connection` zamyka wszystkie obiekty polecień związane z danym połączeniem.

Natomiast w Java SE 7 możemy dla obiektu Statement wywołać metodę `closeOnCompletion`, co spowoduje jego automatyczne zamknięcie, gdy tylko zamknięte zostaną wszystkie jego zbiory wyników.

Jeśli korzystamy z połączeń krótkotrwałych, to nie musimy przejmować się zamykaniem obiektów polecień i zbiorów wyników. Aby zagwarantować, że obiekt połączenia nie będzie pozostawał otwarty, należy użyć instrukcji try zarządzającej zasobami:

```
try (Connection conn = . . .)
{
    Statement stat = conn.createStatement();
    ResultSet result = stat.executeQuery(queryString);
    przetwarzanie wyniku zapytania
}
```



Najlepiej użyć bloku instrukcji try zarządzającej zasobami jedynie do zamknięcia połączenia, a do obsługi wyjątków zastosować osobny blok try/catch. Osobne bloki try ułatwiają analizę i pielęgnację kodu.

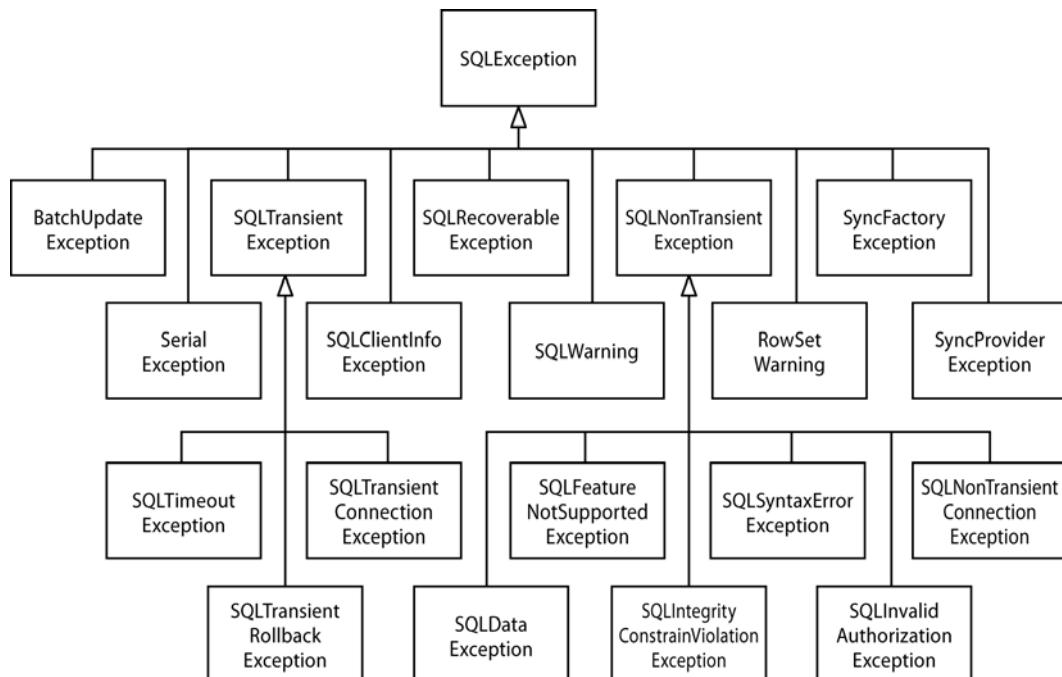
## 4.4.2. Analiza wyjątków SQL

Dla każdego wyjątku `SQLException` istnieje łańcuch obiektów `SQLException`, które pobieramy za pomocą metody `getNextException`. łańcuch ten stanowi dodatek do łańcucha przyczyn wyjątku istniejącego dla każdego obiektu `Throwable`. (Wyjątki języka Java zostały szczegółowo omówione w rozdziale 11. książki *Java. Podstawy*). Do przejrzenia wszystkich wyjątków potrzebowalibyśmy zatem dwóch zagnieżdżonych pętli. Na szczęście w Java SE 6 rozszerzono definicję klasy `SQLException` w taki sposób, że implementuje ona interfejs `Iterable<Throwable>`. Metoda `iterator()` udostępnia iterator `Iterator<Throwable>`, który umożliwia przeglądanie obu wspomnianych łańcuchów, przy czym najpierw przeglądany jest łańcuch przyczyn wyjątku pierwszego obiektu `SQLException`, następnie kolejnego obiektu `SQLException` i tak dalej. Wystarczy zatem użyć odpowiedniej pętli `for`:

```
for (Throwable t : sqlException)
{
    operacje na t
}
```

Aby dokładniej przeanalizować obiekt `SQLException`, możemy wywołać jego metody `getSQLState` i `getErrorCode`. Pierwsza z nich zwraca łańcuch znaków określony przez standard X/Open lub SQL:2003. (Wywołanie metody `getSQLStateType` interfejsu `DatabaseMetaData` pozwoli nam dowiedzieć się, który standard jest używany przez dany sterownik). Natomiast kod błędu zwracany przez drugą z wymienionych metod jest specyficzny dla producenta oprogramowania.

Wyjątki SQL tworzą drzewo dziedziczenia pokazane na rysunku 4.5. Pozwala to na obsługę specyficznych typów błędów w sposób niezależny od konkretnej bazy danych i jej sterownika.



**Rysunek 4.5.** Typy wyjątków SQL

Dodatkowo sterownik bazy danych może raportować pewne mniej poważne problemy jako ostrzeżenia. Ostrzeżenia te możemy pobierać dla połączeń, polecen i zbiorów wyników zapytań. Klasa `SQLWarning` jest klasą pochodną klasy `SQLException` (choćż `SQLWarning` nie jest wyrzucona jako wyjątek). Dokładniejsze informacje na temat ostrzeżenia możemy uzyskać, wywołując metody `getSQLState` i `getErrorCode`. Podobnie jak wyjątki SQL, również ostrzeżenia tworzą łańcuchy. Aby pobrać wszystkie ostrzeżenia, stosujemy pętlę:

```

SQLWarning w = stat.getWarning();
while (w != null)
{
    operacje na w
    w = w.nextWarning();
}
    
```

Klasa `DataTruncation` (będąca klasą pochodną klasy `SQLWarning`) jest używana, gdy dane wczytane z bazy zostały nieoczekiwane obcięte. Jeśli sytuacja ta ma miejsce podczas wykonywania polecenia `UPDATE`, to obiekt klasy `DataTruncation` jest wyrzucany jako wyjątek.

#### API `java.sql.SQLException 1.1`

- `SQLException getNextException()`

zwraca wyjątek połączony z danym wyjątkiem w łańcuchu. Może on zawierać więcej informacji o błędzie.

- `Iterator<Throwable> iterator() 6`

zwraca iterator umożliwiający przeglądanie łańcuchów wyjątków SQL i ich przyczyn.

- `String getSQLState()`  
zwraca SQLState, czyli pięciocyfrowy, standardowy kod związany z błędem.
- `int getErrorCode()`  
zwraca specyficzny kod producenta dla wyjątku.

**API** `java.sql.Warning 1.1`

- `SQLWarning getNextWarning()`  
zwraca kolejne ostrzeżenia w łańcuchu lub `null`, gdy bieżące ostrzeżenie znajduje się na końcu łańcucha.

**API** `java.sql.Connection 1.1`  
`java.sql.Statement 1.1`  
`java.sql.ResultSet 1.1`

- `SQLWarning getWarnings()`
- `SQLWarning getWarnings()`  
zwraca pierwsze oczekujące ostrzeżenie lub `null`, gdy żadne ostrzeżenie nie oczekuje.

**API** `java.sql.DataTruncation 1.1`

- `boolean getParameter()`  
zwraca `true`, gdy obcięcie danych dotyczy parametru: `false` — gdy kolumny.
- `int getIndex()`  
zwraca indeks parametru lub kolumny, której dotyczy obcięcie.
- `int getDataSize()`  
zwraca liczbę bajtów, które powinny zostać przesłane, lub `-1`, gdy wartość ta nie jest znana.
- `int getTransferSize()`  
zwraca liczbę bajtów, które zostały rzeczywiście przesłane, lub `-1`, gdy wartość ta nie jest znana.

#### 4.4.3. Wypełnianie bazy danych

Napiszemy teraz pierwszy przykładowy program operujący na bazie danych i wykorzystujący JDBC. Spełniałby on swoje funkcje dydaktyczne, gdyby wykonywał opisane dotąd operacje i zapytania. Jednak do tego potrzebne są dane w bazie danych, których na razie nie mamy. Musimy zatem wypełnić bazę danymi. W tym celu użyjemy poleceń języka SQL tworzących tabele i wstawiających do nich dane. Większość baz danych potrafi przetwarzać polecenia języka SQL umieszczone w pliku tekstowym. Niestety istnieją jednak drobne różnice co do sposobu kończenia poszczególnych poleceń, a także różnice w ich składni.

Z tego powodu użyjemy JDBC do stworzenia prostego programu, który wczytuje polecenia SQL z pliku, po jednym z każdego wiersza, i wykonuje je.

Program ten będzie czytać polecenia zapisane w pliku tekstowym w następującym formacie:

```
CREATE TABLE Publishers (Publisher_Id CHAR(6), Name CHAR(30), URL CHAR(80))
INSERT INTO Publishers VALUES ('0201', 'Addison-Wesley', 'www.aw-bc.com')
INSERT INTO Publishers VALUES ('0471', 'John Wiley & Sons', 'www.wiley.com')
```

Na listingu 4.2 przedstawiamy kod programu czytającego polecenia języka SQL z pliku i wykonującego je. Nawet jeśli nie jesteś zainteresowany jego implementacją, powinieneś go uruchomić, aby umieścić dane w bazie i wykonać pozostałe programy zamieszczone w tym rozdziale.

Upewnij się, że serwer bazy danych działa, i uruchom czterokrotnie wspomniany program:

```
java -classpath .:ścieżkaSterownika exec.ExecSQL Books.sql
java -classpath .:ścieżkaSterownika exec.ExecSQL Authors.sql
java -classpath .:ścieżkaSterownika exec.ExecSQL Publishers.sql
java -classpath .:ścieżkaSterownika exec.ExecSQL BooksAuthors.sql
```

Przed uruchomieniem programu należy sprawdzić zawartość pliku database.properties — patrz punkt 4.3.5, „Nawiązywanie połączenia z bazą danych”.



Baza danych może udostępniać program narzędziowy umożliwiający bezpośredni odczyt plików poleceń SQL. W przypadku bazy Derby możemy posłużyć się następującym wywołaniem:

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties Books.sql
```

(Plik *ij.properties* został omówiony w punkcie 4.3.3, „Uruchamianie bazy danych”).

W formacie danych dla poleceń wykonywanych przez program ExecSQL przewidzieliśmy opcjonalny znak średnika na końcu każdego wiersza, ponieważ spodziewa się go większość programów narzędziowych dostarczanych z bazami danych.

Poniższe kroki opisują działanie programu ExecSQL.

- 1** Połączenie do bazy danych. Metoda `getConnection`czyta zawartość pliku `database.properties` i dodaje właściwość `jdbc.drivers` do właściwości systemowych. Program zarządzający sterownikami wykorzystuje właściwość `jdbc.drivers`, aby załadować właściwy sterownik. Metoda `getConnection` korzysta z właściwości `jdbc.url`, `jdbc.username` i `jdbc.password` podczas nawiązywania połączenia do bazy danych.
- 2** Otwarcie pliku zawierającego polecenia języka SQL. Jeśli pliku takiego nie określono w linii poleceń, to program prosi użytkownika o interaktywne wprowadzenie poleceń języka SQL.
- 3** Wykonanie poleceń języka SQL za pomocą ogólnej metody `execute`. Jeśli zwraca ona wartość `true`, to utworzony został zbiór wyników. Wszystkie cztery utworzone przez nas pliki tekstowe zawierające polecenia języka SQL i umieszczające dane w bazie kończą się polecienniem `SELECT *`, aby można przekonać się, że dane zostały rzeczywiście pomyślnie umieszczone w bazie.

4. Jeśli polecenie miało wynik, to zostaje on wyświetlony. Ponieważ jest to wynik dowolnego zapytania, to musimy skorzystać z *metadanych*, aby uzyskać informację o liczbie kolumn. Metadane omówimy w podrozdziale 4.8, „Metadane”.
5. Jeśli podczas wykonania polecenia w języku SQL wystąpił wyjątek, to program wyświetla informacje o nim i wszystkich wyjątkach, które mogą być dołączone w postaci łańcucha.
6. Połączenie z bazą danych zostaje zamknięte.

Listing 4.2 zawiera tekst źródłowy omówionego programu.

---

**Listing 4.2. exec/ExecSQL.java**

---

```
package exec;

import java.io.*;
import java.nio.file.*;
import java.util.*;
import java.sql.*;

/**
 * Program wykonuje polecenia języka SQL zapisane w pliku.
 * Uruchomienie programu:
 * java -classpath ścieżkaSterownika:. exec.ExecSQL plikPolecień
 *
 * @version 1.31 2012-06-05
 * @author Cay Horstmann
 */
class ExecSQL
{
    public static void main(String args[]) throws IOException
    {
        try
        {
            Scanner in = args.length == 0 ? new Scanner(System.in) : new
                Scanner(Paths.get(args[0]));

            try (Connection conn = getConnection())
            {
                Statement stat = conn.createStatement();

                while (true)
                {
                    if (args.length == 0) System.out.println("Enter command or EXIT to exit:");

                    if (!in.hasNextLine()) return;

                    String line = in.nextLine();
                    if (line.equalsIgnoreCase("EXIT")) return;
                    if (line.trim().endsWith(";")) // usuwa średnik kończący polecenie
                    {
                        line = line.trim();
                        line = line.substring(0, line.length() - 1);
                    }
                    try
                    {
```

```

        boolean isResult = stat.execute(line);
        if (isResult)
        {
            ResultSet rs = stat.getResultSet();
            showResultSet(rs);
        }
        else
        {
            int updateCount = stat.getUpdateCount();
            System.out.println(updateCount + " rows updated");
        }
    }
    catch (SQLException ex)
    {
        for (Throwable e : ex)
            e.printStackTrace();
    }
}
catch (SQLException e)
{
    for (Throwable t : e)
        t.printStackTrace();
}
}

/**
 * Nawiązuje połączenie z bazą danych,
 * korzystając z właściwości
 * zapisanych w pliku database.properties
 * @return połączenie z bazą danych
 */
public static Connection getConnection() throws SQLException, IOException
{
    Properties props = new Properties();
    try (InputStream in = Files.newInputStream(Paths.get("database.properties")))
    {
        props.load(in);
    }

    String drivers = props.getProperty("jdbc.drivers");
    if (drivers != null) System.setProperty("jdbc.drivers", drivers);

    String url = props.getProperty("jdbc.url");
    String username = props.getProperty("jdbc.username");
    String password = props.getProperty("jdbc.password");

    return DriverManager.getConnection(url, username, password);
}

/**
 * Wyświetla wynik zapytania.
 * @param stat polecenie, którego wynik jest wyświetlany
 */
public static void showResultSet(ResultSet result) throws SQLException
{
    ResultSetMetaData metaData = result.getMetaData();

```

```
int columnCount = metaData.getColumnCount();

for (int i = 1; i <= columnCount; i++)
{
    if (i > 1) System.out.print(", ");
    System.out.print(metaData.getColumnLabel(i));
}
System.out.println();

while (result.next())
{
    for (int i = 1; i <= columnCount; i++)
    {
        if (i > 1) System.out.print(", ");
        System.out.print(result.getString(i));
    }
    System.out.println();
}
```

---

## 4.5. Wykonywanie zapytań

Nasz następny program przykładowy wykonywać będzie zapytania na bazie danych COREJAVA. W tym celu należy najpierw wypełnić bazę COREJAVA danymi, korzystając ze sposobu przedstawionego w poprzednim podrozdziale.

Wykonując zapytanie, możemy wybrać konkretnego autora oraz wydawcę bądź pozostawić dowolną wartość obu pól.

Możemy także modyfikować dane w bazie. Wybierzmy na przykład jednego z wydawców i wpiszmy pewną kwotę. Ceny wszystkich książek tego wydawcy zostaną zmniejszone o podaną kwotę, a program pokaże liczbę książek, których ceny zostały zmienione. Po zmianie cen możemy wykonać zapytanie, aby zweryfikować nowe ceny.

### 4.5.1. Polecenia przygotowane

W programie tym wykorzystujemy *polecenia przygotowane*. Rozważmy zapytanie o wszystkie książki danego wydawcy napisane przez dowolnego autora. Wyglądać będzie ono jak poniżej:

```
SELECT Books.Price, Books.Title
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
AND Publishers.Name = nazwa wydawnictwa wybrana z listy
```

Zamiast tworzyć tekst tego polecenia za każdym razem od nowa, gdy użytkownik zażąda jego wykonania, możemy *przygotować* zapytanie wyposażone w zmienną i korzystać z niego wiele razy, dostarczając różne wartości zmiennej. Takie rozwiązanie jest także lepsze pod

względem efektywności. Za każdym bowiem razem gdy baza danych wykonać ma zapytanie, musi najpierw ustalić sposób jego najefektywniejszego wykonania. Jeśli będziemy wielokrotnie korzystać z polecenia przygotowanego, to strategia jego wykonania określana będzie tylko raz.

Każda zmienna w poleceniu przygotowanym oznaczana jest znakiem ?. Jeśli w zapytaniu takim występuje więcej niż jedna zmienna, to, podając wartość zmiennej, musimy zadbać o wskazanie właściwej. Zapytanie przygotowane w przypadku naszego programu wyglądać będzie, tak jak poniżej.

```
String publisherQuery =
    "SELECT Books.Price, Books.Title " +
    "FROM Books, Publishers " +
    "WHERE Books.Publisher_Id = Publishers.Publisher_Id AND " +
    "Publishers.Name = ?";
PreparedStatement stat
    = conn.prepareStatement(publisherQuery);
```

Zanim wykonamy polecenie przygotowane, musimy nadać wartości jego zmiennym za pomocą metody set. Podobnie jak w przypadku metod get obiektów ResultSet, istnieją różne metody set dla różnych typów. Poniższy przykład pozwala określić nazwę wydawcy.

```
stat.setString(1, publisher);
```

Pierwszy parametr metody setString określa zmienną, której chcemy nadać wartość. Wartość 1 wskazuje na pierwsze wystąpienie znaku ? w tekście polecenia. Drugi parametr metody stanowi wartość, którą nadajemy zmiennej.

Jeśli korzystamy z polecenia przygotowanego, które zostało już przynajmniej raz wykonane i posiada więcej niż jedną zmienną, to zachowują one wartości z ostatniego wykonania. Wykonując zapytanie po raz kolejny, wystarczy więc nadać wartości przy użyciu metody set tylko tym zmiennym, których wartość uległa zmianie.

Po nadaniu wartości wszystkim zmiennym zapytania możemy je już wykonać.

```
ResultSet rs = stat.executeQuery();
```



„Ręczne” tworzenie zapytania, poprzez konkatenację łańcuchów znaków, jest nie tylko pracochłonne, ale potencjalnie niebezpieczne. Szczególną uwagę musimy zwrócić wtedy na znaki specjalne, takie jak cudzysłów. Jeśli zapytanie jest tworzone w oparciu o dane wprowadzone przez użytkownika, to stanowi ono punkt potencjalnego ataku na zasadzie wstrzykiwania kodu. Dlatego też polecenia przygotowane powinniśmy wykorzystywać zawsze, gdy zapytanie posiada zmienne.

Aktualizację cen zaimplementowaliśmy, wykorzystując polecenie UPDATE. Zwróćmy uwagę, że dla wykonania polecenia zamiast metody executeQuery użyliśmy metody executeUpdate, ponieważ polecenie UPDATE nie zwraca zbioru wyników. Metoda executeUpdate zwraca liczbę zmodyfikowanych rekordów, którą wyświetlamy w polu tekstowym okna programu.

```
int r = stat.executeUpdate(updateStatement);
System.out.println(r + " rows updated");
```



Po zamknięciu obiektu Connection związanego z nim obiekt PreparedStatement przejmuje być ważny. Wiele sterowników baz danych automatycznie buforuje polecenie przygotowane. Jeśli to samo polecenie wykonywane jest powtórnie, to baza danych używa strategii realizacji zapytania przygotowanej za pierwszym razem. Dlatego nie należy martwić się kosztem wywołania prepareStatement.

Poniższe kroki przedstawiają sposób działania programu.

- Tworzymy listy nazwisk autorów i nazw wydawców przy użyciu zapytań zwracających wszystkich autorów i wydawców umieszczonych w bazie danych.
- Zapytania związane z nazwiskami autorów są nieco bardziej skomplikowane. Ponieważ książka może mieć wielu autorów, to tabela BooksAuthors określa tę zależność. Na przykład książka o numerze ISBN równym 1-201-96426-0 posiada dwóch autorów o kodach DATE i DARW. Tabela BooksAuthors zawiera więc następujące wiersze:

```
1-201-96426-0, DATE, 1
1-201-96426-0, DARW, 2
```

Trzecia kolumna zawiera informacje o kolejności autorów. (Nie możemy wykorzystać w tym celu kolejności rekordów w tabeli. Tabela relacyjnej bazy danych nie posiada uporządkowanych „na stałe” rekordów). Zapytanie musi więc dokonać połączenia informacji zawartej w tabelach Books, BooksAuthors i Authors, aby porównać nazwisko autora z wybranym przez użytkownika.

```
SELECT Books.Price, Books.Title
FROM Books, Publishers, BooksAuthors, Authors
WHERE Books.Publisher_Id = Publishers.Publisher_id
AND Publishers.Name = ?
AND Books.ISBN = BooksAuthors.ISBN
AND BooksAuthors.Author = Authors.Author
AND Authors.Name = ?
```



Niektórzy programiści wola unikać tak złożonych poleceń języka SQL. W tym celu tworzą rozbudowany i nieefektywny kod w języku Java analizujący wyniki wielu zapytań. Baza danych jest przecież znacznie efektywniejsza w wykonywaniu zapytań niż najlepszy nawet kod w języku Java. Dlatego też, pisząc programy, powinniśmy zastosować się do następującej zasady. Jeśli możemy zaprogramować pewną operację na bazie danych w języku SQL, to nie powinniśmy próbować tego w języku Java.

- Metoda changePrices wykonuje polecenie UPDATE. Zwróćmy uwagę, że klauzula WHERE używa kodu wydawcy, natomiast użytkownik określa jedynie jego nazwę. Problem ten rozwiążujemy za pomocą zagnieżdzonego zapytania.

```
UPDATE Books
SET Price = Price + ?
WHERE Books.Publisher_Id =
      (SELECT Publisher_Id
       FROM Publishers
       WHERE Name = ?)
```

Listing 4.3 zawiera kompletny tekst źródłowy programu.

**Listing 4.3.** query/QueryDB.java

```

package query;

import java.io.*;
import java.nio.file.*;
import java.sql.*;
import java.util.*;

/**
 * Program wykonujący kilka złożonych operacji na bazie danych.
 * @version 1.30 2012-06-05
 * @author Cay Horstmann
 */
public class QueryTest
{
    private static final String allQuery = "SELECT Books.Price, Books.Title FROM Books";

    private static final String authorPublisherQuery = "SELECT Books.Price, Books.Title"
        + " FROM Books, BooksAuthors, Authors, Publishers"
        + " WHERE Authors.Author_Id = BooksAuthors.Author_Id AND BooksAuthors.ISBN"
        + " => Books.ISBN"
        + " AND Books.Publisher_Id = Publishers.Publisher_Id AND Authors.Name = ?"
        + " AND Publishers.Name = ?";

    private static final String authorQuery = "SELECT Books.Price, Books.Title FROM"
        + " Books, BooksAuthors, Authors"
        + " WHERE Authors.Author_Id = BooksAuthors.Author_Id AND BooksAuthors.ISBN"
        + " => Books.ISBN"
        + " AND Authors.Name = ?";

    private static final String publisherQuery = "SELECT Books.Price, Books.Title"
        + " FROM Books, Publishers"
        + " WHERE Books.Publisher_Id = Publishers.Publisher_Id AND Publishers.Name = ?";

    private static final String priceUpdate = "UPDATE Books " + "SET Price = Price + ? "
        + " WHERE Books.Publisher_Id = (SELECT Publisher_Id FROM Publishers WHERE"
        + " Name = ?)";

    private static Scanner in;
    private static Connection conn;
    private static ArrayList<String> authors = new ArrayList<>();
    private static ArrayList<String> publishers = new ArrayList<>();

    public static void main(String[] args) throws IOException
    {
        try
        {
            conn = getConnection();
            in = new Scanner(System.in);
            authors.add("Any");
            publishers.add("Any");
            try (Statement stat = conn.createStatement())
            {
                // Wypełnia listę autorów
                String query = "SELECT Name FROM Authors";
                try (ResultSet rs = stat.executeQuery(query))
                {

```

```
        while (rs.next())
            authors.add(rs.getString(1));
    }

    // Wypełnia listę wydawców
    query = "SELECT Name FROM Publishers";
    try (ResultSet rs = stat.executeQuery(query))
    {
        while (rs.next())
            publishers.add(rs.getString(1));
    }
}

boolean done = false;
while (!done)
{
    System.out.print("Q)uery C)hange prices E)xit: ");
    String input = in.nextLine().toUpperCase();
    if (input.equals("Q"))
        executeQuery();
    else if (input.equals("C"))
        changePrices();
    else
        done = true;
}
}

catch (SQLException e)
{
    for (Throwable t : e)
        System.out.println(t.getMessage());
}
}

/**
 * Wykonuje wybrane zapytanie.
 */
private static void executeQuery() throws SQLException
{
    String author = select("Authors:", authors);
    String publisher = select("Publishers:", publishers);
    PreparedStatement stat;
    if (!author.equals("Any") && !publisher.equals("Any"))
    {
        stat = conn.prepareStatement(authorPublisherQuery);
        stat.setString(1, author);
        stat.setString(2, publisher);
    }
    else if (!author.equals("Any") && publisher.equals("Any"))
    {
        stat = conn.prepareStatement(authorQuery);
        stat.setString(1, author);
    }
    else if (author.equals("Any") && !publisher.equals("Any"))
    {
        stat = conn.prepareStatement(publisherQuery);
        stat.setString(1, publisher);
    }
}
```

```

        else
            stat = conn.prepareStatement(allQuery);

        try (ResultSet rs = stat.executeQuery())
        {
            while (rs.next())
                System.out.println(rs.getString(1) + ", " + rs.getString(2));
        }
    }

    /**
     * Wykonuje polecenie aktualizacji cen.
     */
    public static void changePrices() throws SQLException
    {
        String publisher = select("Publishers:", publishers.subList(1,
            ↳publishers.size()));
        System.out.print("Change prices by: ");
        double priceChange = in.nextDouble();
        PreparedStatement stat = conn.prepareStatement(priceUpdate);
        stat.setDouble(1, priceChange);
        stat.setString(2, publisher);
        int r = stat.executeUpdate();
        System.out.println(r + " records updated.");
    }

    /**
     * Prosi użytkownika o wprowadzenie łańcucha
     * @param prompt łańcuch zachęty
     * @param options opcje, z których może wybierać użytkownik
     * @return wybrana opcja
     */
    public static String select(String prompt, List<String> options)
    {
        while (true)
        {
            System.out.println(prompt);
            for (int i = 0; i < options.size(); i++)
                System.out.printf("%2d %s%n", i + 1, options.get(i));
            int sel = in.nextInt();
            if (sel > 0 && sel <= options.size())
                return options.get(sel - 1);
        }
    }

    /**
     * Nawiązuje połączenie z bazą danych,
     * wykorzystując właściwości
     * zapisane w pliku database.properties
     * @return połączenie z bazą danych
     */
    public static Connection getConnection() throws SQLException, IOException
    {
        Properties props = new Properties();
        try (InputStream in = Files.newInputStream(Paths.get("database.properties")))
        {
            props.load(in);
        }
    }
}

```

```

String drivers = props.getProperty("jdbc.drivers");
if (drivers != null) System.setProperty("jdbc.drivers", drivers);
String url = props.getProperty("jdbc.url");
String username = props.getProperty("jdbc.username");
String password = props.getProperty("jdbc.password");

return DriverManager.getConnection(url, username, password);
}
}

```

**API *java.sql.Connection* 1.1**

- `PreparedStatement prepareStatement(String sql)`

zwraca obiekt `PreparedStatement` zawierający wstępnie skompilowane polecenie języka SQL. Parametr `sql` zawiera polecenie SQL, które może posiadać kilka parametrów oznaczonych znakami `?`.

**API *java.sql.PreparedStatement* 1.1**

- `void setXxx(int n, Xxx x)`  
(gdzie `Xxx` reprezentuje jeden z typów `int`, `double`, `String`, `Date` itp.) nadaje wartość `x` n-temu parametrowi polecenia.
- `void clearParameters()`  
usuwa wartości parametrów polecenia przygotowanego.
- `ResultSet executeQuery()`  
wykonuje zapytanie przygotowane i zwraca obiekt `ResultSet`.
- `int executeUpdate()`  
wykonuje polecenie przygotowane języka SQL, takie jak `INSERT`, `UPDATE` lub `DELETE` reprezentowane przez obiekt `PreparedStatement`. Zwraca liczbę zmodyfikowanych rekordów lub 0 w przypadku polecen języka DDL takich jak na przykład `CREATE TABLE`.

## 4.5.2. Odczyt i zapis dużych obiektów

Wiele baz danych pozwala na przechowywanie nie tylko wartości numerycznych, łańcuchów znaków i dat, ale też *obiektów o znaczących rozmiarach*, takich jak na przykład grafika czy sekwencje wideo. W terminologii języka SQL takie obiekty binarne otrzymały nazwę **BLOB**, a obiekty znakowe — **CLOB**.

Aby odczytać duży obiekt z bazy, najpierw wykonujemy polecenie `SELECT`, a następnie wywołujemy metodę `getBlob` lub `getClob` wynikowego obiektu `ResultSet`. W ten sposób otrzymujemy obiekt typu `Blob` lub `Clob`. Aby pobrać dane binarne z obiektu `Blob`, wywołujemy metodę `getBytes` lub `getBinaryStream`. Na przykład: jeśli tabela bazy danych zawiera okładki książek, to pobieramy je w poniższy sposób:

```

PreparedStatement stat = conn.prepareStatement("SELECT Cover FROM BookCovers WHERE
→ISBN=?");
stat.setInt(1, isbn);
ResultSet result = stat.executeQuery();
if (result.next())
{
    Blob coverBlob = result.getBlob(1);
    Image coverImage = ImageIO.read(coverBlob.getBinaryStream());
}

```

Podobnie w przypadku obiektu Clob jego dane znakowe uzyskamy, wywołując metodę `getSubString` lub `getCharacterStream`.

Aby umieścić duży obiekt w bazie, wywołujemy metodę `createBlob` lub `createClob` obiektu `Connection`, uzyskujemy odpowiedni strumień wyjściowy lub obiekt zapisu, zapisujemy dane i umieszczaemy obiekt w bazie. Oto przykład sposobu zapisu obrazu okładki w bazie:

```

Blob coverBlob = connection.createBlob();
int offset = 0;
OutputStream out = coverBlob.setBinaryStream(offset);
ImageIO.write(coverImage, "PNG", out);
PreparedStatement stat = conn.prepareStatement("INSERT INTO Cover VALUES (?, ?)");
stat.setInt(1, isbn);
stat.setBlob(2, coverBlob);
stat.executeUpdate();

```

#### `java.sql.ResultSet` 1.1

- `Blob getBlob(int columnIndex)` **1.2**
- `Blob getBlob(String columnLabel)` **1.2**
- `Clob getClob(int columnIndex)` **1.2**
- `Clob getClob(String columnLabel)` **1.2**

pobiera BLOB lub CLOB z podanej kolumny.

#### `java.sql.Blob` 1.2

- `long length()`  
zwraca rozmiar obiektu BLOB.
- `byte[] getBytes(long startPosition, long length)`  
pobiera podany zakres danych z obiektu BLOB.
- `InputStream getBinaryStream()`
- `InputStream getBinaryStream(long startPosition, long length)`  
zwraca strumień pozwalający odczytać dane obiektu BLOB lub ich określony zakres.
- `OutputStream setBinaryStream(long startPosition)` **1.4**  
zwraca strumień wyjściowy umożliwiający zapis danych obiektu BLOB, począwszy od podanej pozycji.

**API java.sql.Clob 1.4**

- `long length()`  
zwraca liczbę znaków, które zawiera obiekt CLOB.
- `String getSubString(long startPosition, long length)`  
zwraca dane znakowe obiektu CLOB z podanego zakresu.
- `Reader getCharacterStream()`
- `Reader getCharacterStream(long startPosition, long length)`  
zwraca obiekt odczytu (a nie strumień) umożliwiający odczyt danych znakowych obiektu CLOB lub z podanego zakresu.
- `Writer setCharacterStream(long startPosition) 1.4`  
zwraca obiekt zapisu (a nie strumień) umożliwiający zapis danych znakowych w obiekcie CLOB, począwszy od podanej pozycji.

**API java.sql.Connection 1.1**

- `Blob createBlob() 6`
- `Clob createClob() 6`  
tworzy pusty obiekt BLOB lub CLOB.

### 4.5.3. Sekwencje sterujące

Zastosowanie sekwencji sterujących w poleceniach języka SQL pozwala na obsługę tych właściwości baz danych, które są powszechnie dostępne, ale ich składnia różni się istotnie pomiędzy poszczególnymi produktami.

Sekwencje sterujące spotykamy w następujących zastosowaniach:

- literały związane z datą i czasem,
- wywoływanie funkcji skalarnych,
- wywoływanie procedur składowanych w bazie danych,
- zewnętrzne połączenia tabel,
- w klauzulach LIKE.

Zapis literalów daty i czasu różni się znacznie w poszczególnych bazach danych. Aby umieścić taki literal w bazie, zapisujemy go w formacie ISO 8601 (<http://www.cl.cam.ac.uk/~mgk25/iso-time.html>). Sterownik JDBC zajmie się przetłumaczeniem tego formatu na format właściwy dla danej bazy. Dla wartości typu DATE, TIME i TIMESTAMP stosujemy odpowiednio sekwencje sterujące d, t i ts:

```
{d '2008-01-24'}
{t '23:59:59'}
{ts '2008-01-24 23:59:59.999'}
```

Przez *funkcję skalarną* rozumiemy funkcję, która zwraca pojedynczą wartość. Systemy baz danych dostarczają zwykle bogaty zestaw takich funkcji, ale ich nazwy różnią się. Specyfikacja JDBC dostarcza standardowych nazw tych funkcji, i zajmuje się ich tłumaczeniem na nazwy specyficzne dla konkretnej bazy. Aby wywołać funkcję skalarną, używamy sekwencji sterującej fn, po której podajemy standardową nazwę funkcji i jej argumenty:

```
{fn left(?, 20)}
{fn user()}
```

Pełną listę obsługiwanych nazw funkcji znajdziesz w specyfikacji JDBC.

*Procedura składowana* jest procedurą wykonywaną przez bazę danych, zapisaną w języku specyficznym dla danej bazy. Aby wywołać taką procedurę, stosujemy sekwencję sterującą. Jeśli procedura nie posiada parametrów, to nie musimy stosować nawiasów. Aby uzyskać wartość zwróconą przez procedurę składowaną, używamy znaku =:

```
{call PROC1(?, ?)}
{call PROC2}
{call ? = PROC3(?)}
```

*Połączenie zewnętrzne* dwóch tabel nie wymaga, aby rekordy każdej tabeli spełniały warunek połączenia. Na przykład wynik zapytania

```
SELECT * FROM {oj Books LEFT OUTER JOIN Publishers ON Books.Publisher_Id =
    →Publisher.Publisher_Id}
```

zawierać będzie również książki, dla których Publisher\_Id nie posiada odpowiedniej wartości w tabeli Publishers (pole rekordu będzie wtedy zawierać wartość NULL). Aby wynik zapytania zawierał również wydawców, dla których nie istnieje żadna książka, musimy zastosować połączenie RIGHT OUTER JOIN. Połączenie FULL OUTER JOIN zwróci nam natomiast rekordy uzyskane w obu poprzednich sytuacjach. Zastosowanie sekwencji sterujących dla połączeń zewnętrznych jest konieczne, ponieważ nie wszystkie bazy danych stosują dla nich standardową formę zapisu.

Znaki \_ i % posiadają specjalne znaczenie w klauzulach LIKE, co umożliwia dopasowanie pojedynczego znaku lub ich sekwencji. Natomiast nie istnieje standardowy sposób dosłownego użycia obu tych znaków. Jeśli na przykład chcemy dopasować wszystkie łańcuchy zawierające znak \_, to stosujemy konstrukcję:

```
... WHERE ? LIKE %!_% {escape '!'}
```

W tym przypadku ! stanowi znak sterujący, a kombinacja !\_ reprezentuje znak podkreślenia.

## 4.5.4. Zapytania o wielu zbiorach wyników

Możliwa jest sytuacja, w której zapytanie zwróci wiele zbiorów wyników. Może ona pojawić się na skutek wykonania procedury składowanej lub w przypadku baz danych, które umożliwiają umieszczenie wielu poleceń SELECT w pojedynczym zapytaniu. Wszystkie zbiory wyników zapytania pobieramy w następujący sposób:

- 1 Stosujemy metodę execute w celu wykonanie polecenia SQL.
- 2 Pobieramy pierwszy zbiór wyników lub licznik aktualizacji.

3. Wielokrotnie wywołujemy metodę `getMoreResults`, aby przejść do kolejnych zbiorów wyników.
4. Kończymy działanie, gdy nie ma więcej zbiorów wyników lub liczników aktualizacji.

Metody `execute` i `getMoreResults` zwracają wartość `true`, jeśli kolejnym elementem łańcucha jest zbiór wyników. Metoda `getUpdateCount` zwraca wartość `-1`, jeśli kolejnym elementem łańcucha nie jest licznik aktualizacji.

Poniższa pętla umożliwia przeglądanie wszystkich wyników zapytania:

```
boolean isResult = stat.execute(command);
boolean done = false;
while (!done)
{
    if (isResult)
    {
        ResultSet result = stat.getResultSet();
        operacje na result
    }
    else
    {
        int updateCount = stat.getUpdateCount();
        if (updateCount >= 0)
            operacje na updateCount
        else
            done = true;
    }
    if (!done) isResult = stat.getMoreResults();
}
```

#### `java.sql.Statement` 1.1

- `boolean getMoreResults()`
- `boolean getMoreResults(int current)` 6

pobiera kolejny zbiór wyników dla danego polecenia. Parametr `current` może przyjmować jedną z wartości: `CLOSE_CURRENT_RESULT` (domyślnie), `KEEP_CURRENT_RESULT` lub `CLOSE_ALL_RESULTS`. Metoda zwraca wartość `true`, jeśli kolejny element łańcucha istnieje i jest zbiorem wyników.

## 4.5.5. Pobieranie wartości kluczy wygenerowanych automatycznie

Większość baz danych udostępnia jakiś mechanizm automatycznego numerowania wstawianych rekordów. Niestety, rozwiązania oferowane przez poszczególnych producentów różnią się istotnie. Chociaż więc JDBC nie oferuje uniwersalnego mechanizmu generowania takich kluczy, to jednak dostarcza efektywny sposób ich pobierania. Jeśli po wstawieniu nowego wiersza do tabeli zostaje automatycznie wygenerowany jego klucz, to możemy pobrać jego wartość za pomocą poniższego kodu:

```
stmt.executeUpdate(insertStatement, Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
if (rs.next())
```

```

{
    int key = rs.getInt(1);
}

```

**API *java.sql.Statement* 1.1**

- `boolean execute(String statement, int autogenerated)` **1.4**
- `int executeUpdate(String statement, int autogenerated)` **1.4**

wykonuje polecenie SQL w omówiony wcześniej sposób. Jeśli parametr `autogenerated` ma wartość `Statement.RETURN_GENERATED_KEYS`, a poleceniem SQL jest polecenie `INSERT`, to pierwsza kolumna tabeli zawiera klucz wygenerowany automatycznie.

## 4.6. Przewijalne i aktualizowalne zbiory wyników zapytań

Jak już pokazaliśmy, metoda `next` interfejsu `ResultSet` umożliwia przeglądanie zbioru wierszy będącego wynikiem zapytania. Przeglądanie wyników za pomocą metody `next` jest wystarczające dla programów analizujących dane. Jeśli jednak program zapewniać ma także przeglądanie zbioru wyników przez użytkownika (patrz rysunek 4.4), to metoda `next` okazuje się niewystarczająca, ponieważ z reguły chcemy zapewnić możliwość przeglądania w obu kierunkach (na przód i wstecz). *Przewijalne* zbiory wyników zapytań umożliwiają swobodne poruszanie się po zbiorze wyników.

Co więcej, często prezentując wyniki zapytania, chcemy także pozwolić je użytkownikowi modyfikować. *Aktualizowalny* zbiór wyników zapytania zapewnia automatyczną aktualizację zmodyfikowanych rekordów w bazie danych. Zagadnienia te omówimy szczegółowo w kolejnych podrozdziałach.

### 4.6.1. Przewijalne zbiory wyników

Domyślnie zbiory wyników zapytań nie są ani przewijalne, ani aktualizowalne. Aby uzyskać przewijalny zbiór wyników zapytania, musimy pobrać obiekt `Statement` innego rodzaju, korzystając z metody:

```
Statement stat = conn.createStatement(type, concurrency);
```

Dla polecenia przygotowanego wywołamy natomiast

```
PreparedStatement stat = conn.prepareStatement(command, type, concurrency);
```

Wartości parametrów `type` i `concurrency` podają tabele 4.6 i 4.7. Istnieją następujące możliwości wyboru.

- Jeśli zbiór wyników nie ma być przewijalny, należy użyć wartości `ResultSet.TYPE_FORWARD_ONLY`.
- Jeśli natomiast zbiór wyników ma być przewijalny, to musimy zdecydować, czy ma być on aktualizowany w oparciu o zmiany, jakie zaszły w bazie danych po wykonaniu zapytania, którego jest on rezultatem? (Dla potrzeb naszego omówienia korzystać będziemy z wartości `ResultSet.TYPE_SCROLL_INSENSITIVE`, która oznacza, że zbiór wyników nie będzie powiadamiany o zmianach zachodzących w bazie).
- Czy modyfikacje wprowadzane przez użytkownika w zbiorze wyników mają być automatycznie umieszczane w bazie danych? (Patrz kolejny podrozdział).

**Tabela 4.6.** Wartości parametru type

Wartość	Opis
<code>TYPE_FORWARD_ONLY</code>	Zbiór wyników nie jest przewijalny (domyślnie).
<code>TYPE_SCROLL_INSENSITIVE</code>	Zbiór wyników jest przewijalny, ale nie odzwierciedla zmian zachodzących w bazie danych.
<code>TYPE_SCROLL_SENSITIVE</code>	Zbiór wyników jest przewijalny i odzwierciedla zmiany zachodzące w bazie danych.

**Tabela 4.7.** Wartości parametru concurrency

Wartość	Opis
<code>CONCUR_READ_ONLY</code>	Zbiór wyników nie jest używany do aktualizacji bazy danych.
<code>CONCUR_UPDATABLE</code>	Zbiór wyników może być używany do aktualizacji bazy danych.

Jeśli zdecydujemy na przykład, że zbiór wyników ma być przewijalny, ale nie chcemy umożliwiać modyfikacji danych, to wywołanie wyglądać tak:

```
Statement stat
= conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
resultSet.CONCUR_READ_ONLY);
```

Dzięki temu wszystkie zbiory wyników zwarcane przez wywołania typu

```
ResultSet rs = stat.executeQuery(query)
```

będą przewijalne. Przewijalny zbiór wyników posiada *kursor* wskazujący bieżącą pozycję w zbiorze.

Przewijanie zbioru wyników jest bardzo proste. Do poruszania się wstecz w zbiorze wyników wykorzystujemy metodę `previous`, tak jak w przykładzie poniżej.

```
if (rs.previous()) . . .
```

Metoda ta zwraca wartość `true`, jeśli kursor wskazuje kolejny element lub `false`, jeśli znajduje się przed pierwszym z elementów zbioru wyników.

Kursor możemy także przesuwać o dowolną liczbę elementów wprzód lub wstecz, korzystając z metody

```
rs.relative(n);
```



W rzeczywistości sterownik bazy danych może nie być w stanie zrealizować naszego żądania i utworzyć zbiór wyników, który jest przewijalny bądź aktualizowalny. (Metody supportsResultSetType i supportsResultSetConcurrency klasy DatabaseMetaData pozwalały uzyskać informacje o rzeczywistych możliwościach sterownika i bazy danych). Nawet jednak jeśli baza danych umożliwia tworzenie wszystkich rodzajów zbiorów wyników, to w szczególnych przypadkach utworzenie zbiór wyników o określonych właściwościach może okazać się niemożliwe. (Na przykład wynik złożonego zapytania może nie być aktualizowalny). W takim przypadku baza danych tworzy zbiór wyników o mniejszych możliwościach niż wymagane i dodaje ostrzeżenie SQLWarning do obiektu reprezentującego połączenie. (W punkcie 4.4.2, „Analiza wyjątków SQL”, pokazaliśmy, w jaki sposób można pobrać takie ostrzeżenie). Alternatywnie możemy też sprawdzić, jakimi właściwościami dysponuje w rzeczywistości utworzony zbiór wyników, korzystając z metod getType oraz getConcurrency klasy ResultSet. Jeśli tego nie zrobimy i spróbujemy wywołać metodę previous dla zbiuru, który nie jest właściwa dla danego zbiuru wyników, na przykład metodę previous dla zbiuru, który nie jest przewijalny, to wyrzuci ona wyjątek SQLException.

Jeśli  $n$  jest wartością dodatnią, kursor porusza się do przodu, jeśli ujemną, to do tyłu. Jeśli  $n$  jest równe 0, to wywołanie metody relative nie ma żadnego skutku. Jeśli spróbujemy przemieścić kursor poza zbiór wyników, to, w zależności od znaku  $n$ , umieszczony on zostanie przed pierwszym lub za ostatnim elementem. Metoda relative zwróci wtedy wartość false. Metoda relative zwraca wartość true tylko wtedy, gdy kursor wskazuje element.

Kursor możemy ustawić także na elemencie o określonej pozycji:

```
rs.absolute(n);
```

Bieżący numer elementu uzyskać możemy, wywołując

```
int n = rs.getRow();
```

Pierwszy element posiada numer 1. Jeśli wywołanie zwróci wartość 0, to kursor nie wskazuje elementu, ale znajduje się przed pierwszym lub za ostatnim z nich.

Metody first, last, beforeFirst i afterLast umożliwiają ustawienie kursora odpowiednio na pierwszym lub ostatnim elemencie oraz przed pierwszym i za ostatnim elementem.

Metody isFirst, isLast, isBeforeFirst i isAfterLast umożliwiają sprawdzenie, czy kursor znajduje się na jednej z wyróżnionych pozycji.

Korzystanie z przewijalnych zbiorów wyników jest bardzo proste. Skomplikowane buforowanie rezultatów zapytania wykonywane jest bowiem przez sterownik bazy danych.

## 4.6.2. Aktualizowalne zbiory rekordów

Jeśli chcemy mieć możliwość edycji zbiuru wyników zapytania i automatycznie wprowadzać modyfikacje do bazy danych, to powinniśmy wykorzystać aktualizowalny zbiór wyników. Zbiór taki nie musi być przewijalny, ale z reguły, umożliwiając użytkownikowi edycję danych, chcemy także pozwolić mu na swobodne przeglądanie zbiuru wyników.

Aby otrzymywać aktualizowalne zbiory wyników, musimy utworzyć obiekt Statement w następujący sposób.

```
Statement stat
= conn.createStatement(resultSet.TYPE_SCROLL_INSENSITIVE,
    resultSet.CONCUR_UPDATABLE);
```

Zbiory wyników, które uzyskamy przez wywołanie metody executeQuery, będą już aktualizowalne.



Nie wszystkie zapytania mogą zwracać aktualizowalne zbiory wyników. Jeśli zapytanie wymaga operacji połączenia wielu tabel, to uzyskany zbiór wyników może nie być aktualizowalny. Jeśli zapytanie dotyczy pojedynczej tabeli lub wykonuje połączenie wielu tabel w oparciu o klucz pierwotny, to otrzymane zbiory wyników są z reguły aktualizowalne. Aby się o tym upewnić, wystarczy wywołać metodę getConcurrency interfejsu ResultSet.

Załóżmy na przykład, że użytkownik programu chce podnieść ceny niektórych książek, ale nie dysponuje prostym kryterium, które pozwoliłoby wykonać pojedynczą komendę UPDATE. Przeglądając listę książek, może więc ręcznie zmieniać ceny niektórych z nich.

```
String query = "SELECT * FROM Books";
ResultSet rs = stat.executeQuery(query);
while (rs.next())
{
    if ( . . . )
    {
        double increase = . . .
        double price = rs.getDouble("Price");
        rs.updateDouble("Price", price + increase);
        rs.updateRow(); // po aktualizacji pól należy wywołać updateRow
    }
}
```

Metody updateXxx istnieją dla wszystkich typów odpowiadających typom języka SQL, na przykład updateDouble, updateString itd. Podobnie jak w przypadku metod getXxx ich pierwszym parametrem jest numer lub nazwa kolumny. Kolejnym parametrem metod jest nowa wartość pola rekordu.



Jeśli używamy numeru kolumny jako первого параметра методы updateXxx, то określa on numer kolumny w zbiorze rekordów. Numer ten może być różny od numeru kolumny w bazie danych.

Metoda updateXxx powoduje jedynie zmianę wartości pola w wierszu, a nie w bazie danych. Po zmodyfikowaniu pól danego wiersza musimy wywołać jeszcze metodę updateRow, która aktualizuje dany wiersz w bazie danych. Jeśli wcześniej przemieścimy kursor do następnego wiersza, to wszystkie zmiany w zbiorze wyników zostaną stracone i wobec tego nigdy nie będą zapisane w bazie danych. Możemy także skorzystać z metody cancelRowUpdates, aby odwołać modyfikacje bieżącego rekordu.

W poprzednim przykładzie pokazaliśmy, w jaki sposób zmodyfikować istniejący wiersz. Jeśli chcemy umieścić w bazie nowy wiersz, to najpierw musimy za pomocą metody moveToInsertRow ustawić kursor na specjalnej pozycji zwanej *wierszem wstawiania*. Nowy

wiersz tworzymy, korzystając z wiersza wstawiania i metod updateXXX. Po jego wykreowaniu korzystamy z metody insertRow, która wstawia wiersz do bazy danych. Po wstawieniu wiersza możemy użyć metody moveToCurrentRow, która spowoduje, że kurSOR powróci na pozycję poprzedzającą wywołanie metody moveToInsertRow. Poniżej przedstawiamy przykład wstawienia nowego wiersza.

```
rs.moveToInsertRow();
rs.updateString("Title", title);
rs.updateString("ISBN", isbn);
rs.updateString("Publisher_Id", pubid);
rs.updateDouble("Price", price);
rs.insertRow();
rs.moveToCurrentRow();
```

Zwróćmy uwagę, że nie posiadamy żadnego wpływu na to, na jakiej pozycji nowy wiersz zostanie umieszczony w zbiorze wyników bądź bazie danych.

Jeśli we wstawianym wierszu nie określmy wartości pewnej kolumny, to otrzyma ona wartość NULL. Jeśli jednak kolumna ta posiada ograniczenie NOT NULL, to wyrzucony zostanie wyjątek, a wiersz nie zostanie wstawiony do bazy.

Wiersz wskazywany przez kurSOR możemy usunąć, korzystając z poniższej metody.

```
rs.deleteRow();
```

Usuwa ona natychmiast bieżący wiersz zarówno ze zbioru wyników, jak i z bazy danych.

Metody updateRow, insertRow i deleteRow klasy ResultSet umożliwiają w efekcie wykonanie tych samych operacji co polecenia UPDATE, INSERT i DELETE języka SQL. Z reguły jednak programiści korzystający z języka Java wolą wykonywać operacje na bazie danych za pomocą metod działających na zbiorze wyników, niż stosować polecenia języka SQL.



Korzystając z aktualizowalnych zbiorów wyników zapytań, możemy napisać wyjątkowo nieefektywny program. Zawsze *bardziej* efektywne jest wykonanie polecenia UPDATE, niż wykonanie zapytania i modyfikowanie zbioru wyników. Aktualizowalne zbiory wyników powinny być wykorzystywane jedynie w programach, które przewidują możliwość dowolnej modyfikacji danych przez użytkownika. W każdym innym przypadku zalecamy wykorzystanie polecenia UPDATE.



JDBC 2 udostępnia także możliwość aktualizacji zbioru wyników zapytania, jeśli w międzyczasie nowe dane w bazie zostaną zmodyfikowane przez inne połączenie. JDBC 3 dodaje jeszcze możliwość określenia zachowania zbioru wyników w momencie zatwierdzenia transakcji. Możemy wybrać, czy zbiór wyników powinien zostać zamknięty w momencie zatwierdzenia transakcji, czy też pozostać otwarty. Te zaawansowane możliwości JDBC nie są jednak tematem naszego wprowadzającego w tematykę rozdziału. Wszystkich, którzy chcieliby je poznać, odsyłamy do książki *JDBC API Tutorial and Reference* autorstwa Maydene Fisher, Jona Ellisa i Jonathana Bruce'a (Addison-Wesley, 2003) oraz dokumentów specyfikacji JDBC na stronie [www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html](http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html).

**API** `java.sql.Connection 1.1`

- `Statement createStatement(int type, int concurrency)` **1.2**
- `PreparedStatement prepareStatement(String command, int type, int concurrency)` **1.2**

tworzą polecenie lub polecenie przygotowane, którego rezultatem jest zbiór wyników o określonych właściwościach.

*Parametry:*

<code>command</code>	polecenie przygotowane,
<code>type</code>	jedna ze stałych <code>TYPE_FORWARD_ONLY</code> , <code>TYPE_SCROLL_INSENSITIVE</code> , <code>TYPE_SCROLL_SENSITIVE</code> zdefiniowanych przez interfejs <code>ResultSet</code> ,
<code>concurrency</code>	jedna ze stałych <code>CONCUR_READ_ONLY</code> , <code>CONCUR_UPDATABLE</code> zdefiniowanych przez interfejs <code>ResultSet</code> .

**API** `java.sql.ResultSet 1.1`

- `int getType()` **1.2**

Zwraca dla danego zbioru wyników jedną z wartości `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE` lub `TYPE_SCROLL_SENSITIVE`.

- `int getConcurrency()` **1.2**

Zwraca dla danego zbioru wyników wartość `CONCUR_READ_ONLY` lub `CONCUR_UPDATABLE`.

- `boolean previous()` **1.2**

Przesuwa kursor do poprzedniego rekordu. Zwraca wartość `true`, jeśli kursor ustawiony został na rekordzie.

- `int getRow()` **1.2**

Zwraca numer bieżącego rekordu. Rekordy ponumerowane są, zaczynając od wartości 1.

- `boolean absolute(int r)` **1.2**

Przesuwa kursor do rekordu o numerze `r`. Zwraca wartość `true`, jeśli kursor ustawiony został na rekordzie.

- `boolean relative(int d)` **1.2**

Przesuwa kursor o `d` rekordów. Jeśli `d` jest ujemne, to kursor przesuwany jest wstecz. Zwraca wartość `true`, jeśli kursor ustawiony został na rekordzie.

- `boolean first()` **1.2**

- `boolean last()` **1.2**

Ustawiają kursor na pierwszym lub ostatnim rekordzie. Zwracają wartość `true`, jeśli kursor ustawiony został na rekordzie.

- `void beforeFirst()` **1.2**

■ **void afterLast() 1.2**

Ustawiają kursor przed pierwszym lub za ostatnim rekordem.

■ **boolean isFirst() 1.2**

■ **boolean isLast() 1.2**

Sprawdzają, czy kursor znajduje się na pierwszym bądź ostatnim rekordzie.

■ **boolean isBeforeFirst() 1.2**

■ **boolean isAfterLast() 1.2**

Sprawdzają, czy kursor znajduje się przed pierwszym lub za ostatnim rekordem.

■ **void moveToInsertRow() 1.2**

Ustawia kursor na pozycji *rekordu wstawiania*. Rekord wstawiania umożliwia wstawianie nowych rekordów do bazy za pomocą metod updateXXX i insertRow.

■ **void moveToCurrentRow() 1.2**

Przesuwa kursor z powrotem na pozycję, którą zajmował przed wywołaniem metody moveToInsertRow.

■ **void insertRow() 1.2**

Umieszcza zawartość rekordu wstawiania w zbiorze rekordów i bazie danych.

■ **void deleteRow() 1.2**

Usuwa bieżący rekord z bazy danych i zbioru rekordów.

■ **void updateXXX(int column, Xxx data) 1.2**

■ **void updateXXX(String columnName, Xxx data) 1.2**

(gdzie Xxx jest typem int, double, String, Date itp.)

Aktualizują pole bieżącego rekordu.

■ **void updateRow() 1.2**

Umieszcza modyfikacje bieżącego rekordu w bazie danych.

■ **void cancelRowUpdates() 1.2**

Odwołuje modyfikacje bieżącego rekordu.

**API** `java.sql.DatabaseMetaData 1.1`

■ **boolean supportsResultSetType(int type) 1.2**

Zwraca wartość true, jeśli baza danych umożliwia tworzenie zbiorów wyników danego typu.

*Parametry:* type jedna ze stałych TYPE\_FORWARD\_ONLY,  
TYPE\_SCROLL\_INSENSITIVE lub TYPE\_SCROLL\_SENSITIVE  
zdefiniowanych przez interfejs ResultSet.

- boolean supportsResultSetConcurrency(int type, int concurrency) **1.2**

Zwraca wartość true, jeśli baza danych umożliwia tworzenie zbiorów wyników danego typu.

Parametry:	type	jedna ze stałych TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE lub TYPE_SCROLL_SENSITIVE zdefiniowanych przez interfejs ResultSet,
	concurrency	jedna ze stałych CONCUR_READ_ONLY lub CONCUR_UPDATABLE zdefiniowanych przez interfejs ResultSet.

## 4.7. Zbiory rekordów

Przewijalne zbiory wyników oferują duże możliwości, ale mają poważną wadę. Połączenie z bazą danych musi być otwarte podczas całej sesji przeglądania wyników przez użytkownika. Użytkownik może opuścić stanowisko na dłuższy czas, pozostawiając tym samym zajęte połączenie z bazą danych. Połączenia takie stanowią cenny zasób i nie powinny być blokowane w opisany sposób. W takiej sytuacji możemy użyć zbioru rekordów. Interfejs RowSet stanowi rozszerzenie interfejsu ResultSet, ale nie wymaga utrzymywania połączenia z bazą danych.

Zbiory rekordów przydają się również, gdy musimy przenieść wynik zapytania do innej warstwy skomplikowanej aplikacji lub innego urządzenia, na przykład telefonu komórkowego. W takiej sytuacji nie możemy posłużyć się zbiorem wyników, ponieważ jego struktury danych są rozbudowane i jest on związany z połączeniem z bazą danych.

### 4.7.1. Tworzenie zbiorów rekordów

Pakiet javax.sql.rowset dostarcza następujących interfejsów rozszerzających interfejs RowSet:

- CachedRowSet umożliwia działanie bez połączenia. Buforowane zbiory rekordów omówione zostaną w następnym podrozdziale.
- WebRowSet jest buforowanym zbiorem rekordów, który może zostać zapisany w pliku XML. Plik XML może zostać przeniesiony do innej warstwy aplikacji, gdzie może zostać otwarty przez inny obiekt WebRowSet.
- FilteredRowSet i JoinRowSet umożliwiają przeprowadzanie prostych operacji na zbiorach rekordów stanowiących odpowiednik poleceń SELECT i JOIN języka SQL. Operacje te przeprowadzane są na danych przechowywanych w zbiorze rekordów i nie wymagają połączenia z bazą danych.
- JdbcRowSet jest prostym opakowaniem interfejsu ResultSet. Dodając do zbioru wyników metody get i set pochodzące z interfejsu RowSet, przekształca go w komponent JavaBean (więcej informacji o ziarnkach znajduje się w rozdziale 8.).

W Java SE 7 wprowadzono standardowy sposób uzyskiwania zbioru rekordów:

```
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet crs = factory.createCachedRowSet();
```

Dla pozostałych typów zbiorów rekordów istnieją podobne metody ich uzyskiwania.

W wersjach poprzedzających Java SE 7 metody tworzenia zbiorów były specyficzne dla poszczególnych dostawców. Dodatkowo JDK oferuje również referencyjne implementacje w pakiecie com.sun.rowset. Nazwy ich klas opatrzone zostały przyrostkiem `Impl`, na przykład `CachedRowSetImpl`. Jeśli nie możemy użyć `RowSetProvider`, powinniśmy skorzystać wtedy z tych klas:

```
CachedRowSet crs = new com.sun.rowset.CachedRowSetImpl();
```

## 4.7.2. Buforowane zbiory rekordów

Buforowany zbiór rekordów zawiera wszystkie dane zbioru wyników zapytania. Ponieważ interfejs `CachedRowSet` stanowi rozszerzenie interfejsu `ResultSet`, to buforowanym zbiorem rekordów możemy posługiwać się w taki sam sposób jak zbiorem wyników zapytania. Jednak zaletą buforowanego zbioru rekordów jest to, że możemy go używać po zamknięciu połączenia z bazą danych. W programie na listingu 4.4 pokażemy, że możliwość ta istotnie upraszcza implementację interaktywnych aplikacji. Każde polecenie użytkownika takiej aplikacji powoduje nawiązanie połączenia z bazą danych, wykonanie zapytania, umieszczenie wyniku w buforowanym zbiорze rekordów i zamknięcie połączenia.

Istnieje nawet możliwość modyfikacji danych znajdujących się w buforowanym zbiorniku rekordów. Oczywiście modyfikacje nie znajdują natychmiast odzwierciedlenia w bazie danych. W tym celu należy jawnie zażądać akceptacji dokonanych modyfikacji. Wtedy `CachedRowSet` łączy się ponownie z bazą danych i wykonuje odpowiednie polecenia SQL.

Buforowany zbiór rekordów tworzymy na podstawie zbiuru wyników zapytania w następujący sposób:

```
ResultSet result = . . .;
CachedRowSet crs = new com.sun.rowset.CachedRowSetImpl();
// lub implementacja pochodząca od producenta bazy danych
crs.populate(result);
conn.close(); // można zamknąć połączenie
```

Alternatywnie możemy pozwolić samemu obiekutowi `CachedRowSet` na automatyczne nawiązanie połączenia. Najpierw należy skonfigurować parametry bazy danych:

```
crs.setURL("jdbc:derby://localhost:1527/COREJAVA");
crs.setUsername("dbuser");
crs.setPassword("secret");
```

Następnie konfigurujemy zapytanie i jego ewentualne parametry:

```
crs.setCommand("SELECT * FROM Books WHERE PUBLISHER = ?");
crs.setString(1, publisherName);
```

Na końcu wypełniamy zbiór rekordów:

```
crs.execute();
```

Powyzsze wywołanie nawiązuje połączenie z bazą danych, wykonuje zapytanie, wypełnia zbiór rekordów i kończy połączenie.

Jeśli wynik zapytania jest wyjątkowo obszerny, to nie ma sensu umieszczać go w całości w zbiorze rekordów, zwłaszcza że użytkownik będzie zwykle zainteresowany tylko kilkoma rekordami. W takim przypadku możemy określić rozmiar strony:

```
CachedRowSet crs = . . .;
crs.setCommand(command);
crs.setPageSize(20);
. . .
crs.execute();
```

Dzięki temu otrzymamy teraz jedynie 20 rekordów. Kolejną ich porcję otrzymamy wywołując

```
crs.nextPage();
```

Zbiory rekordów możemy sprawdzać i modyfikować za pomocą tych samych poleceń co zbiory wyników zapytań. Jeśli zmodyfikujemy zawartość zbioru rekordów, to musimy zapisać ją w bazie danych, wywołując:

```
crs.acceptChanges(conn);
```

lub:

```
crs.acceptChanges();
```

Drugie z wywołań działa jedynie wtedy, gdy dla zbioru rekordów skonfigurowaliśmy informacje (URL, nazwę użytkownika i hasło) potrzebne do nawiązania połączenia z bazą danych.

W punkcie 4.6.2, „Aktualizowalne zbiory rekordów”, pokazaliśmy, że nie wszystkie zbiory wyników zapytań mogą być aktualizowane. Podobnie zbiór rekordów zawierający wynik skomplikowanego zapytania może nie być w stanie zapisać modyfikacji z powrotem do bazy danych. Nie powinno się jednak zdarzyć, jeśli zbiór rekordów zawiera dane pojedynczej tabeli.



Jeśli zbiór rekordów został zapełniony zbiorem wyników zapytania, to nie zna on nazwy tabeli, którą należy zaktualizować. Nazwę tę należy skonfigurować za pomocą wywołania metody `setTable`.

Inny problem powstaje, gdy dane w bazie zostały zmodyfikowane już po wypełnieniu zbioru rekordów, co może prowadzić do niespójności danych. Implementacja referencyjna sprawdza, czy oryginalne wartości zbioru rekordów (czyli przed modyfikacją zbioru rekordów) są identyczne z bieżącymi wartościami w bazie danych. Jeśli tak, to wartości w bazie danych zostają zastąpione zmodyfikowanymi wartościami ze zbioru rekordów. W przeciwnym razie zostaje wygenerowany wyjątek `SyncProviderException`. Inne implementacje mogą stosować inne strategie synchronizacji.

#### `javax.sql.RowSet 1.4`

- `String getURL()`
- `void setURL(String url)`

Konfiguruje lub zwraca URL bazy danych.

- `String getUsername()`

- void setUsername(String username)

Zwraca lub konfiguruje nazwę użytkownika dla połączenia z bazą danych.

- String getPassword()

- void setPassword(String password)

Zwraca lub konfiguruje hasło dla połączenia z bazą danych.

- String getCommand()

- void setCommand(String command)

Zwraca lub konfiguruje polecenie wykonywane dla wypełnienia zbioru rekordów.

- void execute()

Wypełnia zbiór rekordów, wykonując polecenie skonfigurowane za pomocą metody setCommand. Aby menedżer sterowników mógł nawiązać połączenie z bazą danych, musi być skonfigurowany URL bazy danych, nazwa użytkownika i hasło.

#### **java.sql.RowSet.CachedRowSet 5.0**

- void execute(Connection conn)

Wypełnia zbiór rekordów, wykonując polecenie skonfigurowane za pomocą metody setCommand. Metoda ta używa podanego połączenia *i zamyka je*.

- void populate(ResultSet result)

Wypełnia buforowany zbiór rekordów danymi podanego zbioru wyników.

- String getTableName()

- void setTableName(String tableName)

Zwraca lub konfiguruje nazwę tabeli, której danymi wypełniony został buforowany zbiór rekordów.

- int getPageSize()

- void setPageSize(int size)

Zwraca lub konfiguruje rozmiar strony.

- boolean nextPage()

- boolean previousPage()

Ładuje następną lub poprzednią stronę rekordów. Jeśli strona taka istnieje, zwraca wartość true.

- void acceptChanges()

- void acceptChanges(Connection conn)

Łączy się ponownie z bazą danych, aby zapisać w niej modyfikacje dokonane w zbiorze rekordów. Może zgłosić wyjątek SyncProviderException, jeśli dane nie mogą zostać zapisane, ponieważ baza została wcześniej zmodyfikowana.

**API** javax.sql.rowset.RowSetProvider 7

- static RowSetFactory newFactory()
   
tworzy fabrykę zbiorów rekordów.

**API** javax.sql.rowset.RowSetFactory 7

- CachedRowSet createCachedRowSet()
- FilteredRowSet createFilteredRowSet()
- JdbcRowSet createJdbcRowSet()
- JoinRowSet createJoinRowSet()
- WebRowSet createWebRowSet()

tworzą zbiór rekordów określonego typu.

## 4.8. Metadane

W poprzednich podrozdziałach pokazaliśmy, w jaki sposób wstawać dane do bazy, tworzyć i wykonywać zapytania oraz modyfikować dane w bazie. JDBC umożliwia także uzyskanie informacji o *strukturze* bazy danych i jej tabelach. Możemy uzyskać w ten sposób listę tabel bazy danych lub nazwy i typy kolumn tabeli. Informacje te są mało przydatne, jeśli tworzymy typową aplikację bazy danych, ponieważ znamy oczywiście strukturę bazy danych, którą sami zaprojektowaliśmy. Natomiast mają zasadnicze znaczenie dla programistów, którzy tworzą narzędzia pracujące z dowolną bazą danych.

W języku SQL dane, które opisują bazę danych lub jej elementy, nazywamy *metadanymi* (dla odróżnienia od rzeczywistych danych znajdujących się w bazie). Rozróżniamy trzy rodzaje metadanych: dotyczące bazy danych, opisujące zbiór wyników oraz związane z parametrami polecień przygotowanych.

Aby uzyskać informacje o bazie danych, musimy utworzyć obiekt DatabaseMetaData dla danego połączenia.

```
DatabaseMetaData meta = conn.getMetaData();
```

Za jego pomocą możemy już uzyskać metadane. Na przykład wywołanie

```
ResultSet mrs = meta.getTables(null, null, null, new String[] { "TABLE" });
```

zwraca zbiór wyników zawierających informacje o wszystkich tabelach bazy danych. (Pozostałe parametry wywołania przedstawiamy, opisując metodę na końcu podrozdziału).

Każdy z rekordów zawiera informacje o tabeli. Nas interesować będzie jedynie trzecie jego pole zawierające nazwę tabeli (pozostałe pola przedstawiamy, opisując metodę na końcu podrozdziału). Wywołanie mrs.getString(3) zwróci więc nazwę tabeli. Poniżej kod, który wypełnia listę rozwijalną nazwami tabel.

```
while (mrs.next())
    tableNames.addItem(mrs.getString(3));
```

Metadane opisujące bazę danych znajdują również inne, ważne zastosowanie. Bazy danych są skomplikowanymi systemami, a standard SQL pozostawia producentom spore pole do popisu. Dlatego też klasa DatabaseMetaData udostępnia ponad 100 metod pozwalających uzyskać szczegółowe informacje o bazie danych. Wśród metod tych niektóre mają zupełnie egzotyczne nazwy:

```
meta.supportsCatalogsInPrivilegeDefinitions()
```

czy

```
meta.nullPlusNonNullIsNull()
```

Oczywiście metody te stworzono z myślą o zaawansowanych użytkownikach i ich specyficznych potrzebach. W szczególności o programistach, którzy tworzą przenośny kod działający z wieloma różnymi bazami danych.

Klasa DatabaseMetaData umożliwia uzyskanie informacji o bazie danych. Druga z klas metadanych, ResultSetMetaData, umożliwia uzyskanie informacji o zbiorze wyników. Za każdym razem, gdy w rezultacie wykonania zapytania otrzymamy zbiór wyników, możemy uzyskać informację o liczbie kolumn oraz nazwie, typie i rozmiarze każdej z nich. Poniżej przedstawiamy typową pętlę przeglądającą te metadane:

```
ResultSet mrs = stat.executeQuery("SELECT * FROM " + tableName);
ResultSetMetaData meta = mrs.getMetaData();
for (int i = 1; i <= meta.getColumnCount(); i++)
{
    String columnName = meta.getColumnName(i);
    int columnWidth = meta.getColumnDisplaySize(i);
    ...
}
```

W tym podrozdziale przedstawiamy sposób implementacji prostego programu narzędziowego wykorzystującego metadane. Program pokazany na listingu 4.4 wykorzystuje metadane i pozwala przeglądać informacje o wszystkich tabelach istniejących w bazie danych. Program ten stanowi również ilustrację zastosowania buforowanych zbiorów rekordów.

#### **Listing 4.4.** view/ViewDB.java

```
package view;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.nio.file.*;
import java.sql.*;
import java.util.*;
import javax.sql.*;
import javax.sql.rowset.*;
import javax.swing.*;

/**
 * Program wykorzystujący metadane
 * do prezentacji dowolnych tabel bazy danych.

```

```
* @version 1.32 2012-01-26
* @author Cay Horstmann
*/
public class ViewDB
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new ViewDBFrame();
                frame.setTitle("ViewDB");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

/**
 * Ramka zawierająca panel danych i przyciski nawigacji.
 */
class ViewDBFrame extends JFrame
{
    private JButton previousButton;
    private JButton nextButton;
    private JButton deleteButton;
    private JButton saveButton;
    private DataPanel dataPanel;
    private Component scrollPane;
    private JComboBox<String> tableNames;
    private Properties props;
    private CachedRowSet crs;

    public ViewDBFrame()
    {
        tableNames = new JComboBox<String>();
        tableNames.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                showTable((String) tableNames.getSelectedItem());
            }
        });
        add(tableNames, BorderLayout.NORTH);

        try
        {
            readDatabaseProperties();
            try (Connection conn = getConnection())
            {
                DatabaseMetaData meta = conn.getMetaData();
                ResultSet mrs = meta.getTables(null, null, null, new String[] { "TABLE" });
                while (mrs.next())
                    tableNames.addItem(mrs.getString(3));
            }
        }
    }
}
```

```

        catch (SQLException e)
        {
            JOptionPane.showMessageDialog(this, e);
        }
        catch (IOException e)
        {
            JOptionPane.showMessageDialog(this, e);
        }

JPanel buttonPanel = new JPanel();
add(buttonPanel, BorderLayout.SOUTH);

previousButton = new JButton("Previous");
previousButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        showPreviousRow();
    }
});
buttonPanel.add(previousButton);

nextButton = new JButton("Next");
nextButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        showNextRow();
    }
});
buttonPanel.add(nextButton);

deleteButton = new JButton("Delete");
deleteButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        deleteRow();
    }
});
buttonPanel.add(deleteButton);

saveButton = new JButton("Save");
saveButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        saveChanges();
    }
});
buttonPanel.add(saveButton);
pack();
}

/**
 * Przygotowuje pola tekstowe do prezentacji nowej tabeli
 * i wyświetla zawartość jej pierwszego rekordu.
 * @param tableName nazwa prezentowanej tabeli
 */

```

```
public void showTable(String tableName)
{
    try
    {
        try (Connection conn = getConnection())
        {
            //pobiera zbiór wyników
            Statement stat = conn.createStatement();
            ResultSet result = stat.executeQuery("SELECT * FROM " + tableName);
            //kopiuje je do buforowanego zbioru rekordów
            RowSetFactory factory = RowSetProvider.newFactory();
            crs = factory.createCachedRowSet();
            crs.setTableName(tableName);
            crs.populate(result);
        }

        if (scrollPane != null) remove(scrollPane);
        dataPanel = new DataPanel(crs);
        scrollPane = new JScrollPane(dataPanel);
        add(scrollPane, BorderLayout.CENTER);
        validate();
        showNextRow();
    }
    catch (SQLException e)
    {
        JOptionPane.showMessageDialog(this, e);
    }
}

/**
 * Pokazuje poprzedni rekord tabeli.
 */
public void showPreviousRow()
{
    try
    {
        if (crs == null || crs.isFirst()) return;
        crs.previous();
        dataPanel.showRow(crs);
    }
    catch (SQLException e)
    {
        for (Throwable t : e)
            t.printStackTrace();
    }
}

/**
 * Pokazuje następny rekord tabeli.
 */
public void showNextRow()
{
    try
    {
        if (crs == null || crs.isLast()) return;
        crs.next();
        dataPanel.showRow(crs);
    }
}
```

```

        catch (SQLException e)
        {
            JOptionPane.showMessageDialog(this, e);
        }
    }

    /**
     * Usuwa bieżący rekord tabeli.
     */
    public void deleteRow()
    {
        try
        {
            try (Connection conn = getConnection())
            {
                crs.deleteRow();
                crs.acceptChanges(conn);
                if (crs.isAfterLast())
                    if (!crs.last()) crs = null;
                dataPanel.showRow(crs);
            }
        }
        catch (SQLException e)
        {
            JOptionPane.showMessageDialog(this, e);
        }
    }

    /**
     * Zapisuje wszystkie modyfikacje.
     */
    public void saveChanges()
    {
        try
        {
            try (Connection conn = getConnection())
            {
                dataPanel.setRow(crs);
                crs.acceptChanges(conn);
            }
        }
        catch (SQLException e)
        {
            JOptionPane.showMessageDialog(this, e);
        }
    }

    private void readDatabaseProperties() throws IOException
    {
        props = new Properties();
        try (InputStream in = Files.newInputStream(Paths.get("database.properties")))
        {
            props.load(in);
        }
        String drivers = props.getProperty("jdbc.drivers");
        if (drivers != null) System.setProperty("jdbc.drivers", drivers);
    }
}

```

```

    /**
     * Nawiązuje połączenie z bazą danych,
     * korzystając z właściwości zapisanych
     * w pliku database.properties.
     * @return połączenie z bazą danych
    */
    private Connection getConnection() throws SQLException
    {
        String url = props.getProperty("jdbc.url");
        String username = props.getProperty("jdbc.username");
        String password = props.getProperty("jdbc.password");

        return DriverManager.getConnection(url, username, password);
    }

    /**
     * Panel wyświetlający zawartość zbioru rekordów.
    */
    class DataPanel extends JPanel
    {
        private java.util.List<JTextField> fields;

        /**
         * Tworzy panel danych.
         * @param rs zbiór rekordów prezentowany przez panel
        */
        public DataPanel(ResultSet rs) throws SQLException
        {
            fields = new ArrayList<>();
            setLayout(new GridBagLayout());
            GridBagConstraints gbc = new GridBagConstraints();
            gbc.gridwidth = 1;
            gbc.gridheight = 1;

            ResultSetMetaData rsmd = rs.getMetaData();
            for (int i = 1; i <= rsmd.getColumnCount(); i++)
            {
                gbc.gridy = i - 1;

                String columnName = rsmd.getColumnName(i);
                gbc.gridx = 0;
                gbc.anchor = GridBagConstraints.EAST;
                add(new JLabel(columnName), gbc);

                int columnWidth = rsmd.getColumnDisplaySize(i);
                JTextField tb = new JTextField(columnWidth);
                if (!rsmd.getColumnClassName(i).equals("java.lang.String"))
                    tb.setEditable(false);

                fields.add(tb);

                gbc.gridx = 1;
                gbc.anchor = GridBagConstraints.WEST;
                add(tb, gbc);
            }
        }
    }
}

```

```

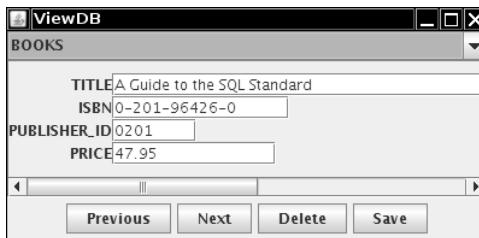
    /**
     * Pokazuje rekord bazy danych,
     * wypełniając pola tekstowe danymi kolejnych kolumn.
     */
    public void showRow(ResultSet rs) throws SQLException
    {
        for (int i = 1; i <= fields.size(); i++)
        {
            String field = rs == null ? "" : rs.getString(i);
            JTextField tb = fields.get(i - 1);
            tb.setText(field);
        }
    }

    /**
     * Aktualizuje dane zmodyfikowane w bieżącym rekordzie.
     */
    public void setRow(ResultSet rs) throws SQLException
    {
        for (int i = 1; i <= fields.size(); i++)
        {
            String field = rs.getString(i);
            JTextField tb = fields.get(i - 1);
            if (!field.equals(tb.getText()))
                rs.updateString(i, tb.getText());
        }
        rs.updateRow();
    }
}

```

Lista umieszczona w górnej części okna pozwala na wybór dowolnej z tabel istniejących w bazie danych. Po wybraniu tabeli program prezentuje nazwy pól wraz z wartościami pochodząymi z pierwszego rekordu (patrz rysunek 4.6). Przyciski *Next* i *Previous* umożliwiają przeglądanie rekordów tabeli. Program pozwala również na edycję pól rekordów oraz usunięcie całego rekordu. Modyfikacje te możemy zapisać w bazie danych, jeśli wybierzemy przycisk *Save*.

**Rysunek 4.6.**  
Program ViewDB  
w działaniu



Wiele baz danych dostarcza znacznie bardziej zaawansowane narzędzia umożliwiające przeglądanie i modyfikację tabel. Jeśli używasz bazy, która nie ma takiego narzędzia, to zapoznaj się z możliwościami programów iSQLViewer (<http://isql.sourceforge.net>) lub SQuirrel (<http://squirrel-sql.sourceforge.net>). Programy te potrafią przeglądać tabele dowolnej bazy zgodnej z JDBC. Nasz przykładowy program nie próbuje z nimi konkurować, a jedynie stanowi ilustrację sposobu implementacji narzędzi potrafiących działać na dowolnych tabelach.

**API** `java.sql.Connection 1.1`

- `DatabaseMetaData getMetaData()`

zwraca obiekt umożliwiający pobieranie metadanych dla danego połączenia.

**API** `java.sql.DatabaseMetaData 1.1`

- `ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String types[])`

pobiera z katalogu opis wszystkich tabel zgodnych ze wzorcami nazwy i schematu oraz kryterium typu. (*Schemat* opisuje grupę powiązanych ze sobą tabel i praw dostępu. *Katalog* opisuje grupę powiązanych ze sobą schematów. Obie koncepcje opracowano w celu strukturyzacji dużych baz danych).

Parametry *catalog* i *schemaPattern* mogą zawierać pusty łańcuch w celu pobrania opisu tablic nieposiadających katalogu ani schematu lub wartość `null` w celu pobrania tablic z pominięciem katalogu i schematu.

Tablica *types* zawiera nazwy typów tablic, które powinny zostać zwrócone. Mogą to być TABLE, VIEW, SYSTEM TABLE, GLOBAL TEMPORARY, LOCAL TEMPORARY, ALIAS i SYNONYM. Jeśli *types* jest `null`, wtedy zwracane są tablice wszystkich typów.

Zbiór rekordów będący wynikiem działania metody składa się z pięciu kolumn, wszystkich typu `String`, co pokazuje tabela 4.8.

**Tabela 4.8.** Kolumny wynikowego zbioru rekordów

Kolumna	Nazwa	Opis
1	TABLE_CAT	Katalog tabel (może być null).
2	TABLE_SCHEM	Schemat tabel (może być null).
3	TABLE_NAME	Nazwa tabeli.
4	TABLE_TYPE	Typ tabeli.
5	REMARKS	Komentarz.

- `int getJDBCMajorVersion() 1.4`

- `int getJDBCMinorVersion() 1.4`

Zwraca bardziej i mniej znaczącą część numeru wersji sterownika JDBC dla danego połączenia. Na przykład dla sterownika JDBC 3.0 zwróci odpowiednio wartości 3 oraz 0.

- `int getMaxConnections()`

zwraca maksymalną liczbę równoczesnych połączeń z bazą danych.

- `int getMaxStatements()`

zwraca maksymalną liczbę obiektów poleceń, które można otworzyć dla danego połączenia z bazą danych, lub wartość 0, jeśli liczba ta jest nieograniczona lub nieznana.

**API** `java.sql.ResultSet 1.1`

- `ResultSetMetaData getMetaData()`  
zwraca obiekt metadanych dla danego zbioru rekordów.

**API** `java.sql.ResultSetMetaData 1.1`

- `int getColumnCount()`  
zwraca liczbę kolumn zbioru rekordów.
- `int getColumnDisplaySize(int column)`  
zwraca maksymalną szerokość określonej kolumny.  
*Parametry:* column numer kolumny
- `String getColumnLabel(int column)`  
zwraca sugerowany tytuł kolumny.  
*Parametry:* column numer kolumny
- `String getColumnName(int column)`  
zwraca nazwę kolumny o danym numerze.  
*Parametry:* column numer kolumny

## 4.9. Transakcje

Z wielu operacji przeprowadzanych na bazie danych możemy stworzyć *transakcję*. Transakcja zostaje *zatwierdzona* tylko wtedy, gdy wszystkie operacje zostały wykonane pomyślnie. Jeśli podczas wykonywania jednej z operacji wystąpił błąd, to cała transakcja może zostać *odwołana*. Odwołanie transakcji równoważne jest sytuacji, w której żadna z operacji wchodzących w jej skład nie została wykonana.

Głównym powodem tworzenia transakcji jest konieczność zachowania *spójności bazy danych*. Założymy, na przykład, że pewna kwota przelewana jest z jednego konta w banku na inne. Dla zachowania spójnego stanu kont musi być wykonana operacja zmniejszenia jednego konta i operacja zwiększenia drugiego konta albo żadna z nich. Jeśli podczas wykonywania drugiej operacji wystąpi błąd, to należy odwołać pierwszą operację.

Jeśli operacje wykonywane na bazie danych tworzy będą transakcję, to albo wykonanie wszystkich operacji powiedzie się i transakcja zostanie *zatwierdzona*, albo jedno z nich zakończy się błędem i transakcja zostanie *odwołana*. W takim przypadku odwołane zostaną wszystkie modyfikacje bazy danych wprowadzone od momentu zatwierdzenia poprzedniej transakcji.

Domyślnie baza danych pracuje w trybie *automatycznego zatwierdzania* poleceń, czyli wynik każdego polecenia w języku SQL jest natychmiast zatwierdzany w bazie. Po zatwierdzeniu polecenia nie możemy już go odwołać. Tryb automatycznego zatwierdzania poleceń możemy wyłączyć za pomocą wywołania

```
conn.setAutoCommit(false);
```

Utwórzmy teraz obiekt polecenia w zwykły sposób:

```
Statement stat = conn.createStatement();
```

I wywołajmy metodę executeUpdate dowolną liczbę razy:

```
stat.executeUpdate(command1);
stat.executeUpdate(command2);
stat.executeUpdate(command3);
```

Po pomyślnym wykonaniu wszystkich poleceń zatwierdzimy je za pomocą metody commit:

```
conn.commit();
```

Natomiast jeśli wystąpił błąd, wywołamy

```
conn.rollback();
```

co spowoduje odwołanie rezultatów wszystkich poleceń, które zostały wykonane od poprzedniego wywołania metody commit. Metodę rollback wywołujemy zwykle, obsługując wyjątek SQLException, który przerwał wykonywanie transakcji.

## 4.9.1. Punkty kontrolne

Używając niektórych sterowników, możemy uzyskać precyzyjniejszą kontrolę nad odwoływaniem transakcji, jeśli zastosujemy *punkty kontrolne*. Punkt kontrolny oznacza punkt transakcji, do którego możemy powrócić bez konieczności ponownego uruchamiania transakcji, na przykład:

```
Statement stat = conn.createStatement(); // rozpoczęcie transakcji; wywołanie rollback() spowoduje
                                         // powrót do tego punktu
stat.executeUpdate(command1);
Savepoint svpt = conn.setSavepoint(); // punkt kontrolny; wywołanie rollback(svpt) spowoduje
                                         // powrót do tego punktu
stat.executeUpdate(command2);
if ( . . . ) conn.rollback(svpt); // odwołuje command2
.
.
.
conn.commit();
```

Gdy punkt kontrolny nie jest już potrzebny, należy go zwolnić:

```
stat.releaseSavepoint(svpt);
```

## 4.9.2. Aktualizacje wsadowe

Efektywność działania programów, które wykonują serie poleceń INSERT w celu wypełnienia bazy danymi, możemy poprawić, korzystając z aktualizacji wsadowych. Aktualizacje wsadowe tworzą sekwencję poleceń i wykonują je wsadowo.



Aby sprawdzić, czy baza danych umożliwia wykonywanie aktualizacji wsadowych, korzystamy z metody supportsBatchUpdates należącej do interfejsu DatabaseMetaData.

Przetwarzane wsadowo mogą być polecenia języka SQL, takie jak INSERT, UPDATE i DELETE oraz polecenia definicji danych, jak CREATE TABLE czy DROP TABLE. Wsadowo nie mogą być natomiast przetwarzane zapytania SELECT, ponieważ tworzą one zbiory rekordów i nie modyfikują bazy danych.

Aby wykonać serię poleceń jako wsad, tworzymy najpierw obiekt poleceń w zwykły sposób:

```
Statement stat = conn.createStatement();
```

Następnie zamiast wywoływać metodę executeUpdate, wywołujemy metodę addBatch:

```
String command = "CREATE TABLE . . . "
stat.addBatch(command);

while ( . . . )
{
    command = "INSERT INTO . . . VALUES (" + . . . + ")";
    stat.addBatch(command);
}
```

W końcu wykonujemy cały wsad poleceń.

```
int [] counts = stat.executeBatch();
```

Wywołanie metody executeBatch zwraca tablicę liczników rekordów dla wszystkich wykonywanych poleceń.

Z punktu widzenia właściwej obsługi błędów aktualizacje wsadowe traktować musimy jak pojedynczą transakcję. Jeśli podczas wykonania aktualizacji wsadowej wystąpi błąd, to powinniśmy odwołać ją w całości.

W tym celu należy kolejno: wyłączyć tryb automatycznego zatwierdzania poleceń, utworzyć wsad poleceń, wykonać go i zatwierdzić, przywrócić tryb automatycznego zatwierdzania poleceń:

```
boolean autoCommit = conn.getAutoCommit();
conn.setAutoCommit(false);
Statement stat = conn.createStatement();

//wywołania metody stat.addBatch(. . .)

stat.executeBatch();
conn.commit();
conn.setAutoCommit(autoCommit);
```

#### **API** java.sql.Connection 1.1

- void setAutoCommit(boolean b)  
włącza lub wyłącza tryb automatycznego zatwierdzania poleceń.
- boolean getAutoCommit()  
sprawdza, czy tryb automatycznego zatwierdzania poleceń jest włączony.

- `void commit()`  
zatwierdza wykonanie wszystkich poleceń wydanych od momentu poprzedniego wywołania metody `commit`.
- `void rollback()`  
odwołuje wykonanie wszystkich poleceń wydanych od momentu poprzedniego wywołania metody `commit`.
- `Savepoint setSavepoint() 1.4`  
wstawia punkt kontrolny bez nazwy.
- `Savepoint setSavepoint(String name) 1.4`  
wstawia punkt kontrolny o podanej nazwie.
- `void rollback(Savepoint svpt) 1.4`  
odwołuje transakcję do podanego punktu kontrolnego.
- `void releaseSavepoint(Savepoint svpt) 1.4`  
zwalnia podany punkt kontrolny.

**API `java.sql.Savepoint 1.4`**

- `int getSavepointId()`  
zwraca identyfikator punktu kontrolnego nieposiadającego nazwy lub zgłasza wyjątek `SQLException`, gdy punkt ten posiada nazwę.
- `String getSavepointName()`  
zwraca nazwę punktu kontrolnego lub zgłasza wyjątek `SQLException`, gdy punkt ten nie posiada nazwy.

**API `java.sql.Statement 1.1`**

- `void addBatch(String command) 1.2`  
dodaje komendę do bieżącego wsadu danego obiektu polecenia.
- `int [] executeBatch() 1.2`  
wykonuje wszystkie polecenia wsadu. Zwraca tablicę liczników rekordów zmodyfikowanych przez poszczególne polecenia wsadu. Jeśli wartość licznika nie jest ujemna, to oznacza dokładnie liczbę zmodyfikowanych rekordów. Jeśli wartość ta jest równa `SUCCESS_NO_INFO`, to informacja o liczbie zmodyfikowanych rekordów nie jest dostępna. Wartość `EXECUTE_FAILED` oznacza, że wykonanie polecenia nie powiodło się.

**API `java.sql.DatabaseMetaData 1.1`**

- `boolean supportsBatchUpdates() 1.2`  
zwraca wartość `true`, jeśli sterownik JDBC umożliwia wykonywanie aktualizacji wsadowych.

### 4.9.3. Zaawansowane typy języka SQL

W tabeli 4.9 przedstawione zostały typy języka SQL obsługiwane w standardzie JDBC oraz ich odpowiedniki w języku Java.

**Tabela 4.9.** Typy danych języka SQL i odpowiadające im typy języka Java

Typ danych języka SQL	Typ danych języka Java
INTEGER lub INT	int
SMALLINT	short
NUMERIC( <i>m, n</i> ), DECIMAL( <i>m, n</i> ) lub DEC( <i>m, n</i> )	java.math.BigDecimal
FLOAT( <i>n</i> )	double
REAL	float
DOUBLE	double
CHARACTER( <i>n</i> ) lub CHAR( <i>n</i> )	String
VARCHAR( <i>n</i> )	String
BOOLEAN	boolean
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
BLOB	java.sql.Blob
CLOB	java.sql.Clob
ARRAY	java.sql.Array
ROWID	java.sql.RowId
NCHAR( <i>n</i> ), NVARCHAR( <i>n</i> ), LONG NVARCHAR	String
NCLOB	java.sql.NClob
SQLXML	java.sql.SQLXML

Typ ARRAY w języku SQL reprezentuje sekwencję wartości. Na przykład tabela Student zawierać może kolumnę Scores typu ARRAY OF INTEGER. Metoda getArray zwróci obiekt typu java.sql.Array. Interfejs java.sql.Array posiada metody umożliwiające pobranie wartości przechowywanych przez obiekt.

Jeśli potrzebujemy uzyskać z bazy danych rekordy zawierające pola kolumn typu BLOB lub ARRAY, to zawartość tych pól pobierana jest dopiero wtedy, gdy zażądamy wartości pojedynczego pola. Jest to uzasadnione efektywnością, ponieważ wartości tych typów mogą rzeczywiście posiadać reprezentację pokaźnych rozmiarów.

Niektóre bazy danych udostępniają wartości typu ROWID, które opisują położenie rekordu w bazie i pozwalają na jego natychmiastowe pobranie. W JDBC 4 wprowadzono odpowiadający im typ java.sql.RowId i udostępniono metody umożliwiające posługiwanie się identyfikatorami ROWID w zapytaniach oraz ich pobieranie ze zbiorów wyników.

*Lańcuchy znaków narodowych* (typ NCHAR i jego warianty) umożliwiają przechowywanie łańcuchów wykorzystujących lokalne kodowanie znaków i sortowanie danych w bazie zgodnie z lokalnym uporządkowaniem znaków. JDBC 4 dostarcza metod umożliwiających dokonywanie konwersji pomiędzy łańcuchami NCHAR i obiektami języka Java, zarówno w zapytaniach, jak i zbiorach wyników.

Niektoře bazy danych umożliwiają przechowywanie złożonych typów strukturalnych definiowanych przez użytkownika. JDBC 3 dostarcza automatyczny mechanizm tworzenia odwzorowań strukturalnych typów języka SQL na obiekty języka Java.

Jeszcze inne systemy baz danych obsługują bezpośrednio dane XML. W JDBC 4 wprowadzono interfejs SQLXML spełniający rolę mediatora pomiędzy wewnętrzną reprezentacją danych XML w bazie a interfejsami DOM Source/Result, a także strumieniami binarnymi. Szczegóły związane z tą mediacją można odnaleźć w dokumentacji interfejsu SQLXML.

W podrozdziale tym, ze względu na jego charakter wprowadzenia, nie będziemy szczegółowo omawiać zaawansowanych typów języka SQL. Informacje na ten temat można znaleźć w książce *JDBC API Tutorial and Reference* oraz specyfikacji JDBC 4.

## 4.10. Zaawansowane zarządzanie połączeniami

Rozwiązanie, które zastosowaliśmy w naszych przykładach w celu łączenia się do bazy danych, korzystające z pliku właściwości database.properties nie skaluje się dobrze w przypadku dużych aplikacji.

W przypadku aplikacji JDBC działających w skali przedsiębiorstwa lub w sieci Web, zarządzanie połączeniemi z bazami danych integrujemy z interfejsem JNDI (*Java Naming and Directory Interface*). Wykorzystywany przez niego katalog zawiera informacje o lokalizacji źródeł danych w firmie. Wykorzystanie usługi katalogowej umożliwia centralizację zarządzania nazwami i hasłami użytkowników, nazwami baz danych oraz adresami URL wykorzystywanymi przez JDBC.

Aby w środowisku tym nawiązać połaczenie do bazy danych, korzystamy z następującego fragmentu kodu:

```
Context jndiContext = new InitialContext();
DataSource source = (DataSource) jndiContext.lookup("java:comp/env/jdbc/corejava");
Connection conn = source.getConnection();
```

Zwróćmy uwagę, że nie musimy w tym przypadku używać już klasy DriverManager. Usługa JNDI sama znajduje źródło danych. Źródło danych udostępnia przy tym interfejs, który umożliwia zarówno proste połączenia JDBC, jak i bardziej zaawansowane rozwiązania, na przykład wykonywanie rozproszonych transakcji działających na wielu bazach danych. Interfejs źródła danych DataSource zdefiniowany jest w pakiecie rozszerzeń javax.sql.

Oczywiście źródło danych musi zostać gdzieś skonfigurowane. Jeśli tworzymy aplikacje baz danych działające w kontenerze serwletów, takim jak na przykład Apache Tomcat, lub na serwerze aplikacji, takim jak GlassFish, to konfigurację bazy danych (w tym nazwę JNDI,

URL JDBC, nazwę użytkownika i hasło) umieszczamy w pliku konfiguracyjnym lub konfigurujemy za pośrednictwem graficznego interfejsu administratora.



W przypadku kontenera Java EE nie musimy nawet używać wyszukiwania źródła danych za pomocą JNDI. Wystarczy zastosować zapis Resource dla pola `DataSource`, a referencja źródła danych zostanie automatycznie skonfigurowana podczas ładowania aplikacji:

```
@Resource("jdbc/corejava")
private DataSource source;
```

Zarządzanie nazwami użytkowników jest zagadniением wymagającym szczególnej uwagi, podobnie jak koszt związany z nawiązywaniem połączeń z bazami danych. W naszych prostych przykładach stosowaliśmy dotąd dwie strategie uzyskiwania połączenia z bazą danych. W programie QueryDB przedstawionym na listingu 4.3 nawiązywaliśmy pojedyncze połączenie z bazą danych na początku programu i zamkaliśmy je, kończąc wykonanie programu. Natomiast program ViewDB przedstawiony na listingu 4.4 otwierał nowe połączenie z bazą danych za każdym razem, gdy było ono potrzebne.

Jednak istnieje wiele sytuacji, w których żadne z tych rozwiązań nie sprawdza się. Połączenia z bazą danych stanowią ograniczony zasób. Jeśli użytkownik pozostawi uruchomioną aplikację, to połączenie z bazą danych nie powinno być cały czas otwarte. Rozwiązanie polegające na tworzeniu nowego połączenia dla każdego żądania i zamknięciu go po obsłudzeniu żądania będzie natomiast wyjątkowo kosztowne.

Rozwiązanie tego problemu polega na utworzeniu *puli* połączeń. Oznacza to, że połączenia z bazą danych nie są zamknięte, lecz utrzymywane w kolejce w celu ponownego wykorzystania. Udostępnianie puli połączeń jest bardzo ważną usługą i specyfikacja JDBC dostarcza mechanizm umożliwiający jej implementację. Jednak sam pakiet JDK Java nie zawiera żadnej implementacji puli połączeń, a producenci baz danych zwykle nie dostarczają jej także ze sterownikiem JDBC. Implementacje puli połączeń dostarczają natomiast producenci kontenerów web i serwerów aplikacji.

Wykorzystanie puli połączeń jest zupełnie niewidoczne dla programisty. Połączenie pobierane jest automatycznie z puli w momencie uzyskania źródła danych i wywołania metody `getConnection`. Zakończenie wykorzystania, czyli połączenie metodą `close`, nie oznacza w tym przypadku rzeczywistego zamknięcia połączenia, a jedynie zwraca je do puli.

W rozdziale tym omówiliśmy podstawy wykorzystania pakietu JDBC, dostarczając wiedzę pozwalającą tworzyć proste aplikacje baz danych. Jak zaznaczyliśmy we wstępie, problematyka baz danych jest niezwykle rozbudowana i istnieje szereg bardziej zaawansowanych zagadnień, których omówienie wykracza poza materiał tego rozdziału. Przegląd zaawansowanych możliwości pakietu JDBC można znaleźć w *JDBC API Tutorial and Reference* oraz specyfikacji JDBC.

W tym rozdziale przedstawiliśmy sposoby tworzenia aplikacji Java wykorzystujących relacyjne bazy danych, a także zamieściliśmy wprowadzenie do hierarchicznych baz danych. Kolejny rozdział poświęciliśmy ważnemu zagadnieniu internacjonalizacji aplikacji. Pokażemy w nim, w jaki sposób można przystosować programy dla użytkowników w różnych częściach świata.



# 5

## Internacjonalizacja

W tym rozdziale:

- Lokalizatory.
- Formaty liczb.
- Data i czas.
- Porządek alfabetyczny.
- Formatowanie komunikatów.
- Pliki tekstowe i zbiory znaków.
- Komplety zasobów.
- Kompletny przykład.

Potencjalnymi odbiorcami naszych aplikacji i appletów są użytkownicy komputerów na całym świecie. I choć Internet nie zna granic, to tworząc programy wyłącznie przy użyciu zbioru kodów ASCII, sami ograniczamy grono odbiorców programów.

Język Java był pierwszym językiem programowania, który zaprojektowano z myślą o umożliwieniu internacjonalizacji tworzonych w nim programów. Od samego początku dysponował podstawową z punktu widzenia internacjonalizacji właściwością: do reprezentacji łańcuchów znakowych wykorzystywał kod Unicode. Użycie kodu Unicode umożliwia implementację programów, które operują na łańcuchach znaków zapisanych za pomocą znaków różnych alfabetów.

Wielu programistów uważa, że zadanie internacjonalizacji programu sprowadza się do wykorzystania kodu Unicode i przetłumaczenia komunikatów pojawiających się w interfejsie użytkownika. Jak pokażemy jednak w tym rozdziale, internacjonalizacja programów jest o wiele bardziej złożonym zadaniem. Daty, czas, waluty a nawet liczby zapisywane są w różnych krajach w odmiennych formatach. Musimy też dysponować sposobem łatwej konfiguracji menu, nazw przycisków, tekstów komunikatów i skrótów dla różnych języków.

W rozdziale tym omówimy sposoby internacjonalizacji aplikacji i appletów tworzonych w języku Java. Pokażemy, jak poddawać lokalizacji prezentację daty i czasu, liczb i tekstu, graficznego interfejsu użytkownika oraz przedstawimy narzędzia pakietu JDK wspomagające proces internacjonalizacji. Rozdział zakończymy kompletnym przykładem implementacji kalkulatora emerytalnego wyposażonego w interfejs użytkownika w językach: angielskim, niemieckim i chińskim.

## 5.1. Lokalizatory

Kiedy uruchomimy aplikację przeznaczoną dla użytkowników różnej narodowości, pierwszą różnicą, którą zauważymy, będzie lokalizacja języka aplikacji. Z punktu widzenia pełnej internacjonalizacji programu obserwacja ta jest jednak niewystarczająca: użytkownicy w różnych krajach mogą posługiwać się tym samym językiem, ale stosować na przykład różne formaty zapisu daty.

W każdym przypadku internacjonalizacji programu muszą zostać przetłumaczone nazwy pozycji menu, etykiety przycisków i komunikaty. Istnieje jednak wiele innych, dość subtelnego różnic, którymi także należy się zajęć. Na przykład w językach angielskim i niemieckim różny jest sposób zapisu liczb. Liczba

123.456.78

powinna być zapisana w języku niemieckim jako

123.456,78

W przykładzie tym następuje więc zamiana funkcji kropki i przecinka dziesiętnego. Podobne różnice istnieją w przypadku prezentacji daty. W USA daty zapisywane są w nieco irracjonalnym formacie w postaci miesiąc/dzień/rok. W Niemczech stosuje się bardziej sensowy format w postaci dzień/miesiąc/rok, a na przykład w Chinach format — rok/miesiąc/dzień. Data zapisana w USA jako

3/22/61

powinna więc zostać zaprezentowana niemieckiemu użytkownikowi jako

22.03.1961

W przypadku gdy nazwy miesięcy zapisywane są słownie, różnice pomiędzy formatami zapisu daty w różnych językach stają się bardziej oczywiste. Data zapisana w języku angielskim jako

March 22, 1961

powinna zostać zapisana w języku niemieckim jako

22. März 1961

lub jako

1961年3月22日

w języku chińskim.

Istnieje szereg klas formatujących, które uwzględniają te różnice. Sposób działania formatowania kontrolujemy za pomocą klasy `Locale`. *Lokalizator* klasy `Locale` opisuje:

- język,
- lokalizację,
- opcjonalny skrypt (dostępny od wersji Java SE 7),
- opcjonalny wariant pozwalający określić na przykład dialekt czy zasady pisowni.

Na przykład dla USA będzie on zawierał informacje:

język = angielski, lokalizacja = USA,

a dla Niemiec:

język = niemiecki, lokalizacja = Niemcy.

Szwajcaria posiada cztery języki oficjalne (niemiecki, francuski, włoski i retoromański). W Szwajcarii dla użytkownika posługującego się językiem niemieckim użyjemy więc lokalizatora zawierającego poniższą informację:

język = niemiecki, lokalizacja = Szwajcaria.

Zastosowanie takiego lokalizatora spowoduje formatowanie tekstów, liczb i dat jak dla lokalizatora niemieckiego, jednak walutą będzie frank szwajcarski zamiast euro.

Jeśli określmy tylko język, na przykład:

język = niemiecki

to taki lokalizator nie może zostać użyty do formatowania walut i innych zagadnień specyficznych dla poszczególnych krajów.

Aby opisać język i lokalizację w standardowej formie, Java wykorzystuje kody opracowane przez organizację ISO (*International Standardization Organization*). Język zapisany jest za pomocą kodu złożonego z dwóch małych liter zgodnie ze standardem ISO 639-1, a lokalizacja — przy użyciu kodu kraju złożonego z dwóch wielkich liter zgodnie ze standardem ISO 3166-1. Tabele 5.1 i 5.2 prezentują najczęściej używane wartości obu kodów.

**Tabela 5.1. Najczęściej spotykane kody języków ISO-639-1**

Język	Kod	Język	Kod	Język	Kod
angielski	en	hiszpański	sp	polski	pl
chiński	zh	holenderski	nl	portugalski	pt
duński	da	japoński	ja	szwedzki	sv
fiński	fi	koreański	ko	turecki	tr
francuski	fr	niemiecki	de	włoski	it
grecki	el	norweski	no		

**Tabela 5.2.** Najczęściej spotykane kody krajów ISO-3166-1

Kraj	Kod	Kraj	Kod	Kraj	Kod
Austria	AT	Irlandia	IE	Szwajcaria	CH
Belgia	BE	Japonia	JP	Szwecja	SE
Chiny	CN	Kanada	CA	Tajwan	TW
Dania	DK	Korea	KR	Turcja	TR
Finlandia	FI	Niemcy	DE	USA	US
Grecja	GR	Norwegia	NO	Wielka Brytania	GB
Hiszpania	ES	Polska	PL	Włochy	IT
Holandia	NL	Portugalia	PT		

Niektóre kody mogą wydawać się dość przypadkowe, ponieważ zostały utworzone w oparciu o nazwy kraju w lokalnym języku (na przykład Niemcy = Deutschland = de, Chiny = zhong-wen = zh), ale przynajmniej są standaryzowane.

Tworząc lokalizator klasy Locale, przekazujemy mu sam kod języka lub dodatkowo kod kraju.

```
Locale german = new Locale("de");
Locale germanGermany = new Locale("de", "DE");
Locale germanSwitzerland = new Locale("de", "CH");
```

Dla wygody programistów Java SE zawiera szereg predefiniowanych lokalizatorów:

```
Locale.CANADA
Locale.CANADA_FRENCH
Locale.CHINA
Locale.FRANCE
Locale.GERMANY
Locale.ITALY
Locale.JAPAN
Locale.KOREA
Locale.PRC
Locale.TAIWAN
Locale.UK
Locale.US
```

Java SE definiuje też szereg lokalizatorów języków, które specyfikują tylko sam język i nie zawierają informacji o kraju.

```
Locale.CHINESE
Locale.ENGLISH
Locale.FRENCH
Locale.GERMAN
Locale.ITALIAN
Locale.JAPANESE
Locale.KOREAN
Locale.SIMPLIFIED_CHINESE
Locale.TRADITIONAL_CHINESE
```



Zastosowania lokalizatorów wykraczające poza określenie języka i kraju wymagają zrozumienia standardu IETF BCP 47 (dobre wprowadzenie można znaleźć na stronie [www.w3.org/International/articles/language-tags](http://www.w3.org/International/articles/language-tags)). Począwszy od wersji Java SE 7, Java obsługuje znaczniki języków zgodne z IETF BCP 47. Dzięki temu można na przykład utworzyć lokalizator na podstawie znacznika języka w poniższy sposób:

```
Locale chineseTraditionalCharactersHongKong = Locale.forLanguageTag("zh-Hant-HK");
```

Oprócz samodzielnego utworzenia obiektu klasy `Locale` lub skorzystania z predefiniowanego istnieją jeszcze dwa sposoby uzyskania lokalizatora.

Metoda statyczna `getDefault` klasy `Locale` zwraca domyślny lokalizator dla lokalnego systemu operacyjnego. Domyślny lokalizator Javy możemy zmienić, wywołując metodę `setLocale`, ale zmiana ta dotyczy tylko naszego programu, a nie systemu operacyjnego.

Również różnego rodzaju klasy użytkowe, których działanie zależne jest od lokalizatora, zwracają tablicę obsługiwanych lokalizatorów. Na przykład wywołanie

```
Locale[] supportedLocales = DateFormat.getAvailableLocales();
```

zwraca wszystkie lokalizatory, które potrafi obsłużyć klasa `DateFormat`.



Testując program możemy zmienić domyślny lokalizator. W tym celu podczas jego uruchamiania należy dostarczyć odpowiednich wartości właściwości określających język i region. Poniżej przykład dla języka niemieckiego i Szwajcarii:

```
java -Duser.language=de -Duser.region=CH Program
```

Gdy mamy już odpowiedni lokalizator, to sam w sobie posiada on ograniczoną funkcjonalność. Jedyne jego użyteczne metody pozwalają zidentyfikować język i lokalizację. Najważniejszą z metod jest `getDisplayName`, która zwraca łańcuch opisujący lokalizator. łańcuch ten nie zawiera kodów, lecz ma postać, którą można zaprezentować użytkownikowi, na przykład

```
German (Switzerland)
```

Z wykorzystaniem tej metody związany jest pewien problem. Zwraca ona łańcuch, korzystając z domyślnego lokalizatora, co nie zawsze jest pożądane. Na przykład jeśli użytkownik wybrał już niemiecki jako język aplikacji, to należy pokazać mu opis lokalizatora w języku niemieckim. Możemy to osiągnąć, przekazując metodzie lokalizatora języka niemieckiego jako parametr. Wykonanie poniższego fragmentu kodu

```
Locale loc = new Locale("de", "CH");
System.out.println(loc.getDisplayName(Locale.GERMAN));
```

spowoduje wyświetlenie następującego opisu lokalizatora

```
Deutsch (Schweiz)
```

Prawdziwym powodem, dla którego tworzymy lokalizator, jest więc konieczność dostarczenia go różnym metodom. Metody te tworzą tekst prezentowany użytkownikom w różnych miejscach świata. W kolejnych podrozdziałach pokażemy wiele przykładów takich zastosowań.

**API java.util.Locale 1.1**

- `Locale(String language)`
- `Locale(String language, String country)`
- `Locale(String language, String country, String variant)`  
tworzą lokalizator dla podanego języka, kraju i wariantu.
- `static Locale forLanguageTag(String languageTag) 7`  
tworzy lokalizator dla podanej etykiety języka.
- `static Locale getDefault`  
zwraca domyślny lokalizator.
- `static void setDefault(Locale loc)`  
określa domyślny lokalizator.
- `String getDisplayName()`  
zwraca nazwę lokalizatora wyrażoną przy użyciu bieżącego lokalizatora.
- `String getDisplayName(Locale loc)`  
zwraca nazwę lokalizatora wyrażoną przy użyciu podanego lokalizatora.
- `String getLanguage()`  
zwraca kod ISO-639 języka w postaci pary małych liter.
- `String getDisplayLanguage()`  
zwraca nazwę języka wyrażoną przy użyciu bieżącego lokalizatora.
- `String getDisplayLanguage(Locale loc)`  
zwraca nazwę języka wyrażoną przy użyciu podanego lokalizatora.
- `String getCountry()`  
zwraca kod ISO-3166 kraju w postaci pary wielkich liter.
- `String getDisplayCountry()`  
zwraca nazwę kraju wyrażoną przy użyciu bieżącego lokalizatora.
- `String getDisplayCountry(Locale loc)`  
zwraca nazwę kraju wyrażoną przy użyciu podanego lokalizatora.
- `String toLanguageTag() 7`  
zwraca etykietę języka zgodną z IETF BCP 47, na przykład "de-CH".
- `String toString()`  
zwraca opis lokalizatora w postaci konkatenacji łańcuchów reprezentujących język, kraj i wariant połączonych znakiem podkreślenia (na przykład "de\_CH").  
Metody tej należy używać jedynie w celach uruchomieniowych.

## 5.2. Formaty liczb

Wspomnieliśmy już, że sposób formatowania liczb oraz waluta zależą od lokalizacji użytkownika. Język Java dostarcza zbioru obiektów formatujących, które potrafią formatować i parsować wartości numeryczne w pakiecie `java.text`. Aby sformatować liczbę za pomocą lokalizatora, musimy wykonać następujące operacje.

1. Pobieramy lokalizator w sposób opisany w poprzednim podrozdziale.
2. Korzystamy z metody fabryki w celu uzyskania obiektu formatującego.
3. Korzystamy z obiektu formatującego do formatowania i parsowania.

Metody fabryki są statycznymi metodami klasy `NumberFormat` i posiadają parametr klasy `Locale`. Istnieją trzy takie metody: `getNumberInstance`, `getCurrencyInstance` i `getPercentInstance`. Metody te zwracają odpowiednio obiekty formatujące i parsujące liczby, kwoty i wartości procentowe. Poniżej przedstawiamy sposób, w jaki możemy sformatować kwotę zgodnie z zasadami obowiązującymi w Niemczech.

```
Locale loc = new Locale("de", "DE");
NumberFormat currFmt = NumberFormat.getCurrencyInstance(loc);
double amt = 123456.78;
System.out.println(currFmt.format(amt));
```

Wykonanie powyższego fragmentu kodu spowoduje wyświetlenie napisu

123.456,78€

Zwróćmy uwagę, że symbol waluty € umieszczony został na końcu łańcucha, a także na role znaków kropki i przecinka.

Z metody `parse` korzystamy, gdy chcemy przeczytać liczbę, która została wprowadzona lub zapisana przy użyciu lokalizatora. Przedstawiony poniżej kod parsuje wartość, którą użytkownik wpisał w polu tekstowym. Metoda `parse` radzi sobie z różnymi sposobami wykorzystania znaków kropki i przecinka oraz cyframi w różnych językach.

```
TextField inputField;
...
NumberFormat fmt = NumberFormat.getNumberInstance();
// pobiera obiekt formatujący i parsujący dla domyślnego lokalizatora
Number input = fmt.parse(inputField.getText().trim());
double x = input.doubleValue();
```

Metoda `parse` zwraca obiekt abstrakcyjnego typu `Number`. Obiekt ten w rzeczywistości jest typu `Double` lub `Long` w zależności od tego, czy parsowana liczba była zmienoprzecinkowa. Jeśli nie jest to dla nas istotne, to możemy po prostu skorzystać z metody `doubleValue` klasy `Number` i pobrać uzyskaną wartość.



Obiekty typu `Number` nie są rozpakowywane automatycznie. Nie możemy bezpośrednio przypisać takiego obiektu zmiennej typu podstawowego. Konieczne jest zastosowanie metody `doubleValue` lub `intValue`.

Jeśli parsowana liczba nie ma poprawnego formatu, to wyrzucony zostaje wyjątek `ParseException`. *Nie* jest na przykład dozwolone poprzedzanie liczb znakami spacji. (Aby je usunąć, wystarczy wywołać metodę `trim`). Natomiast wszystkie znaki, które następują po liczbie, są ignorowane i nie powodują wyrzucenia wyjątku.

Zwróćmy uwagę, że obiekty zwarcane przez metody fabryki `getXXXInstance` nie są w rzeczywistości typu `NumberFormat`. Klasa `NumberFormat` jest klasą abstrakcyjną, a obiekty formatujące należą do jej klas pochodnych. Metody fabryki wiedzą jedynie, w jaki sposób zlokalizować obiekt należący do odpowiedniego lokalizatora.

Listę lokalizatorów, dla których dostępne są obiekty formatujące, możemy uzyskać za pomocą metody statycznej `getAvailableLocales`. Metoda ta zwraca tablicę lokalizatorów, dla których można uzyskać obiekty formatujące.

Przykładowy program, który zamieszczamy w bieżącym podrozdziale, pozwala eksperymentować z różnymi obiektami formatującymi (patrz rysunek 5.1). Lista rozwijalna umieszczona w górnej części okna programu zawiera wszystkie lokalizatory, dla których dostępne są obiekty formatujące. Możemy wybrać obiekty formatujące liczby, kwoty oraz wartości procentowe. Za każdym razem gdy zmienimy jedną z opcji, formatowana jest zawartość pola tekstowego. Sprawdzenie efektów działania kilku lokalizatorów pozwoli zobaczyć, w jak różny sposób może zostać sformatowana liczba lub kwota. Możemy wprowadzić w nim własną wartość i spróbować ją sparsować. Wybranie przycisku `Parse` powoduje wywołanie metody `parse`. Jeśli parsowanie przebiegnie pomyślnie, to zostanie wywołana metoda `format` i wyświetlany jest wynik jej działania. W przeciwnym razie w polu tekstowym umieszczany jest komunikat o błędzie parsowania.

**Rysunek 5.1.**

Program  
`NumberFormatTest`  
w działaniu



Działanie programu, którego kod przedstawiliśmy na listingu 5.1, jest dość oczywiste. Konstruktor wywołuje metodę `NumberFormat.getAvailableLocales`. Dla każdego z uzyskanych lokalizatorów wywołujemy metodę `getDisplayName`, a uzyskany łańcuch umieszczamy na liście rozwijalnej. (Łańcuchy te nie są posortowane, problemem tym zajmiemy się w podrozdziale 5.4, „Porządek alfabetyczny”.) Za każdym razem gdy użytkownik wybiera inny lokalizator z listy lub inny przycisk wyboru obiektu formatującego, tworzymy nowy obiekt formatujący i aktualizujemy zawartość pola tekstowego. Jeśli użytkownik wybierze przycisk `Parse`, to wywołujemy metodę `parse` dla aktualnie wybranego lokalizatora.

**Listing 5.1.** `numberFormat/NumberFormatTest.java`

```
package numberFormat;

import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;

import javax.swing.*;
```

```

/*
 * Program demonstrujący formatowanie liczb
 * dla różnych lokalizatorów.
 * @version 1.13 2007-07-25
 * @author Cay Horstmann
 */
public class NumberFormatTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new NumberFormatFrame();
                frame.setTitle("NumberFormatTest");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }

    /**
     * Ramka zawierająca przyciski wyboru obiektów formatujących,
     * listę rozwijalną lokalizatorów, pole tekstowe prezentujące wynik formatowania
     * oraz przycisk umożliwiający parsowanie zawartości pola tekstowego.
     */
    class NumberFormatFrame extends JFrame
    {
        private Locale[] locales;
        private double currentNumber;
        private JComboBox<String> localeCombo = new JComboBox<>();
        private JButton parseButton = new JButton("Parse");
        private JTextField numberText = new JTextField(30);
        private JRadioButton numberRadioButton = new JRadioButton("Number");
        private JRadioButton currencyRadioButton = new JRadioButton("Currency");
        private JRadioButton percentRadioButton = new JRadioButton("Percent");
        private ButtonGroup rbGroup = new ButtonGroup();
        private NumberFormat currentNumberFormat;

        public NumberFormatFrame()
        {
            setLayout(new GridBagLayout());

            ActionListener listener = new ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    updateDisplay();
                }
            };

            JPanel p = new JPanel();
            addRadioButton(p, numberRadioButton, rbGroup, listener);
            addRadioButton(p, currencyRadioButton, rbGroup, listener);
            addRadioButton(p, percentRadioButton, rbGroup, listener);
        }
    }
}

```

```

        add(new JLabel("Locale:"), new GBC(0, 0).setAnchor(GBC.EAST));
        add(p, new GBC(1, 1));
        add(parseButton, new GBC(0, 2).setInsets(2));
        add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
        add(numberText, new GBC(1, 2).setFill(GBC.HORIZONTAL));
        locales = (Locale[]) NumberFormat.getAvailableLocales().clone();
        Arrays.sort(locales, new Comparator<Locale>()
        {
            public int compare(Locale l1, Locale l2)
            {
                return l1.getDisplayName().compareTo(l2.getDisplayName());
            }
        });
        for (Locale loc : locales)
            localeCombo.addItem(loc.getDisplayName());
        localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());
        currentNumber = 123456.78;
        updateDisplay();

        localeCombo.addActionListener(listener);

        parseButton.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                String s = numberText.getText().trim();
                try
                {
                    Number n = currentNumberFormat.parse(s);
                    if (n != null)
                    {
                        currentNumber = n.doubleValue();
                        updateDisplay();
                    }
                    else
                    {
                        numberText.setText("Parse error: " + s);
                    }
                }
                catch (ParseException e)
                {
                    numberText.setText("Parse error: " + s);
                }
            }
        });
        pack();
    }

    /**
     * Umieszcza przycisk wyboru w kontenerze.
     * @param p the kontener, w którym umieszczamy przycisk
     * @param b przycisk
     * @param g grupa przycisków
     * @param listener obiekt nasłuchujący przycisku
     */
    public void addRadioButton(Container p, JRadioButton b, ButtonGroup g, ActionListener
        ↴listener)
    {

```

```

        b.setSelected(g.getButtonCount() == 0);
        b.addActionListener(listener);
        g.add(b);
        p.add(b);
    }

    /**
     * Aktualizuje prezentowaną informację i formatuje liczbę
     * zgodnie z wyborem użytkownika.
     */
    public void updateDisplay()
    {
        Locale currentLocale = locales[localeCombo.getSelectedIndex()];
        currentNumberFormat = null;
        if (numberRadioButton.isSelected()) currentNumberFormat = NumberFormat
            .getNumberInstance(currentLocale);
        else if (currencyRadioButton.isSelected()) currentNumberFormat = NumberFormat
            .getCurrencyInstance(currentLocale);
        else if (percentRadioButton.isSelected()) currentNumberFormat = NumberFormat
            .getPercentInstance(currentLocale);
        String n = currentNumberFormat.format(currentNumber);
        numberText.setText(n);
    }
}

```

**API** `java.text.NumberFormat 1.1`■ `static Locale[] getAvailableLocales()`

zwraca tablicę obiektów klasy `Locale`, dla których istnieją obiekty formatujące typu `NumberFormat`.

■ `static NumberFormat getInstance()`■ `static NumberFormat getInstance(Locale l)`■ `static NumberFormat getCurrencyInstance()`■ `static NumberFormat getCurrencyInstance(Locale l)`■ `static NumberFormat getPercentInstance()`■ `static NumberFormat getPercentInstance(Locale l)`

Zwracają obiekt formatujący liczby, kwoty lub wartości procentowe dla bieżącego lub podanego lokalizatora.

■ `String format(double x)`■ `String format(long x)`

Zwracają łańcuch będący wynikiem formatowania liczby zmiennoprzecinkowej lub całkowitej.

■ `Number parse(String s)`

parsuje podany łańcuch i zwraca liczbę jako obiekt klasy `Double`, jeśli łańcuch reprezentował liczbę zmiennoprzecinkową bądź jako obiekt klasy `Long` w pozostałych przypadkach. Początek łańcucha musi być początkiem reprezentacji liczby, nie są

dozwolone poprzedzające ją odstępy. Po liczbie w łańcuchu mogą występować inne znaki, które są ignorowane. Wyrzuca wyjątek ParseException, jeśli proces parsowania się nie powiodł.

- void setParseIntegerOnly(boolean b)
- boolean isParseIntegerOnly()

Ustawiają lub pobierają znacznik, który określa, czy obiekt formatujący powinien parsować jedynie wartości całkowite.

- setGroupingUsed(boolean b)
- boolean isGroupingUsed()

Ustawiają lub pobierają znacznik, który określa, czy obiekt formatujący rozpoznaje i stosuje grupowanie cyfr liczby (na przykład 100.000.000)

- void setMinimumIntegerDigits(int n)
- int getMinimumIntegerDigits()
- void setMaximumIntegerDigits(int n)
- int getMaximumIntegerDigits()
- void setMinimumFractionDigits(int n)
- int getMinimumFractionDigits()
- void setMaximumFractionDigits(int n)
- int getMaximumFractionDigits()

Ustawiają lub pobierają największą lub najmniejszą liczbę cyfr dozwoloną w całkowitej lub dziesiętnej części liczby.

## 5.2.1. Waluty

Do formatowania kwoty wyrażonej w pewnej walucie możemy zastosować metodę NumberFormat.getCurrencyInstance. Metoda ta nie jest jednak zbyt elastyczna, gdyż zwraca obiekt formatujący dla jednej waluty. Założymy na przykład, że przygotowujemy rachunek dla klienta w USA, na którym niektóre kwoty podane będą w dolarach, a inne w euro. Nie możemy wtedy użyć dwóch obiektów formatujących:

```
NumberFormat dollarFormater =  
    NumberFormat.getCurrencyInstance(Locale.US);  
NumberFormat euroFormater =  
    NumberFormat.getCurrencyInstance(Locale.GERMANY);
```

Uzyskany w ten sposób rachunek będzie wyglądać dziwnie, ponieważ niektóre wartości będą miały postać \$100,000, a inne 100.000€. (Zwróćmy uwagę, że kwota wyrażona w euro oddziela tysiące znakiem kropki zamiast przecinka).

Dlatego w przypadku walut do kontroli obiektów formatujących stosujemy klasę Currency. Obiekt Currency uzyskujemy, przekazując identyfikator waluty metodzie statycznej Cur

`Currency.getInstance`. Następnie dla każdego obiektu formatującego wywołujemy metodę `setCurrency`. Oto sposób, w jaki należy skonfigurować obiekt formatujący kwoty w euro dla klienta w USA:

```
NumberFormat euroFormater =
    NumberFormat.getCurrencyInstance(Locale.US);
euroFormatter.setCurrency(Currency.getInstance("EUR"));
```

Identyfikatory walut określone są przez standard ISO 4217. Ich listę można odnaleźć, na przykład, na stronie <http://www.currency-iso.org/en/home/tables/table-a1.html>. W tabeli 5.3 przedstawione zostały wybrane identyfikatory.

**Tabela 5.3.** Przykładowe identyfikatory walut

Waluta	Identyfikator	Waluta	Identyfikator
Dolar amerykański	USD	Juan	CNY
Euro	EUR	Rubel	RUB
Funt brytyjski	GBP	Rupia	INR
Jen	JPY	Złoty polski	PLN

#### API java.util.Currency 1.4

- `static Currency getInstance(String currencyCode)`
- `static Currency getInstance(Locale locale)`

zwraca instancję klasy `Currency` dla podanego kodu waluty zgodnego ze standardem ISO 4217 lub kraju określonego za pomocą lokalizatora.

- `String toString()`
- `String getCurrencyCode()`

zwracają kod ISO 4217 dla danej waluty.

- `String getSymbol()`
- `String getSymbol(Locale locale)`

zwracają symbol waluty dla domyślnego lub podanego lokalizatora. Na przykład dla dolara amerykańskiego symbolem tym będzie "\$" lub "US\$", w zależności od lokalizatora.

- `int getDefaultFractionsDigits()`
- zwraca domyślną liczbę cyfr po przecinku dla danej waluty.
- `static Set<Currency> getAvailableCurrencies()`
- zwraca wszystkie dostępne waluty.

## 5.3. Data i czas

Z formatowaniem daty i czasu związane są cztery następujące zagadnienia.

- Nazwy miesięcy i dni tygodnia muszą być prezentowane w odpowiednim języku.
- Uporządkowanie roku, miesiąca i dnia w zapisie daty może być różne.
- Nie zawsze musi obowiązywać kalendarz gregoriański.
- Strefa czasowa lokalizacji musi być wzięta pod uwagę.

Klasa `DateFormat` uwzględnia te zagadnienia. Posługiwanie się nią jest proste i przypomina sposób wykorzystania klasy `NumberFormat`. Najpierw musimy pobrać lokalizator. Możemy wykorzystać lokalizator domyślny lub wywołać metodę statyczną `getAvailableLocales`, aby uzyskać tablicę lokalizatorów, które umożliwiają formatowanie daty. Następnie wywołujemy jedną z poniższych metod fabryki:

```
fmt = DateFormat.getDateInstance(dateStyle, loc);
fmt = DateFormat.getTimeInstance(timeStyle, loc);
fmt = DateFormat.getDateInstance(dateStyle, timeStyle, loc);
```

Parametry tych metod pozwalają określić sposób prezentacji daty za pomocą jednej z następujących stałych:

- 1.** `DateFormat.DEFAULT`,
- 2.** `DateFormat.FULL` (na przykład `Wednesday, September 15, 2004 8:42:46 PM PDT` dla lokalizatora USA),
- 3.** `DateFormat.LONG` (na przykład `September 15, 2004 8:42:46 PM PDT` dla lokalizatora USA),
- 4.** `DateFormat.MEDIUM` (na przykład `Sep 15, 2004 8:42:46 PM` dla lokalizatora USA),
- 5.** `DateFormat.SHORT` (na przykład `9/15/04 8:42 PM` dla lokalizatora USA).

Metody fabryki zwracają obiekt formatujący, który możemy wykorzystać do formatowania dat.

```
Date now = new Date();
String s = fmt.format(now);
```

Podobnie jak w przypadku klasy `NumberFormat`, możemy skorzystać też z metody `parse` i parsować datę wprowadzoną przez użytkownika. Poniższy fragment kodu parsuje wartość wprowadzoną przez użytkownika w polu tekstowym, używając domyślnego lokalizatora.

```
TextField inputField;
...
DateFormat fmt = DateFormat.getDateInstance(DateFormat.MEDIUM);
Date input = fmt.parse(inputField.getText().trim());
```

Niedogodnością zastosowania metody `parse` jest wymóg, by użytkownik wprowadził datę dokładnie w oczekiwany formacie. Na przykład dla lokalizatora USA i stylu określonego jako `MEDIUM` oczekuje się, że użytkownik wprowadzi datę w następującej postaci

Sep 12, 2007

Jeśli użytkownik pominie znak przecinka i wpisze

Sep 12 2007

lub skorzysta ze skróconego stylu

9/12/07

to wyrzucony zostanie wyjątek parsowania ParseException.

Znacznik lenient umożliwia mniej restrykcyjną interpretację dat. Na przykład data February 30, 2007 zostanie wtedy automatycznie poddana konwersji do daty March 2, 2007. Ten mało bezpieczny sposób działania jest niestety domyślny i dlatego należy go wyłączyć. Obiekt kalendarza wykorzystywany do interpretacji parsowanych dat wyrzuca wyjątek IllegalArgumentException, jeśli użytkownik wprowadzi niedozwoloną kombinację roku, miesiąca i dnia.

Program przedstawiony na listingu 5.2 wykorzystuje klasę DateFormat. W oknie programu możemy wybrać lokalizator i obserwować, w jaki sposób wpływa on na format daty. Jeśli podczas prezentacji daty pojawią się znaki zapytania, oznacza to, że system nie posiada zainstalowanej czcionki umożliwiającej wyświetlanie znaków alfabetu danego języka. Na przykład jeśli wybierzemy lokalizator Chin (rysunek 5.2), to data zostanie zaprezentowana jako

2007年9月12日

**Rysunek 5.2.**

Program  
DateFormatTest  
w działaniu



**Listing 5.2.** dateFormat/DateFormatTest.java

```
package dateFormat;

import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;
import javax.swing.*;

/**
 * Program demonstrujący formatowanie dat
 * dla różnych lokalizatorów.
 * @version 1.13 2007-07-25
 * @author Cay Horstmann
 */
public class DateFormatTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new DateFormatFrame();
                frame.setTitle("DateFormatTest");
            }
        });
    }
}
```

```

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
}

/**
 * Ramka zawierająca listę rozwijaną wyboru lokalizatora,
 * listy rozwijane wyboru stylu prezentacji daty i czasu,
 * pola tekstowe prezentacji daty i czasu,
 * przyciski parsowania zawartości pól tekstowych oraz pole
 * wyboru znacznika mniej restrykcyjnego interpretowania dat.
 */
class DateFormatFrame extends JFrame
{
    private Locale[] locales;
    private Date currentDate;
    private Date currentTime;
    private DateFormat currentDateFormat;
    private DateFormat currentDateFormat;
    private JComboBox<String> localeCombo = new JComboBox<>();
    private JButton dateParseButton = new JButton("Parse date");
    private JButton timeParseButton = new JButton("Parse time");
    private JTextField dateText = new JTextField(30);
    private JTextField timeText = new JTextField(30);
    private JCheckBox lenientCheckbox = new JCheckBox("Parse lenient", true);
    private EnumCombo dateStyleCombo = new EnumCombo(DateFormat.class, "Default",
        "Full", "Long", "Medium", "Short");
    private EnumCombo timeStyleCombo = new EnumCombo(DateFormat.class, "Default",
        "Full", "Long", "Medium", "Short");

    public DateFormatFrame()
    {
        setLayout(new GridBagLayout());
        add(new JLabel("Locale"), new GBC(0, 0).setAnchor(GBC.EAST));
        add(new JLabel("Date style"), new GBC(0, 1).setAnchor(GBC.EAST));
        add(new JLabel("Time style"), new GBC(2, 1).setAnchor(GBC.EAST));
        add(new JLabel("Date"), new GBC(0, 2).setAnchor(GBC.EAST));
        add(new JLabel("Time"), new GBC(0, 3).setAnchor(GBC.EAST));
        add(localeCombo, new GBC(1, 0, 2, 1).setAnchor(GBC.WEST));
        add(dateStyleCombo, new GBC(1, 1).setAnchor(GBC.WEST));
        add(timeStyleCombo, new GBC(3, 1).setAnchor(GBC.WEST));
        add(dateParseButton, new GBC(3, 2).setAnchor(GBC.WEST));
        add(timeParseButton, new GBC(3, 3).setAnchor(GBC.WEST));
        add(lenientCheckbox, new GBC(0, 4, 2, 1).setAnchor(GBC.WEST));
        add(dateText, new GBC(1, 2, 2, 1).setFill(GBC.HORIZONTAL));
        add(timeText, new GBC(1, 3, 2, 1).setFill(GBC.HORIZONTAL));

        locales = (Locale[]) DateFormat.getAvailableLocales().clone();
        Arrays.sort(locales, new Comparator<Locale>()
        {
            public int compare(Locale l1, Locale l2)
            {
                return l1.getDisplayName().compareTo(l2.getDisplayName());
            }
        });
        for (Locale loc : locales)
            localeCombo.addItem(loc.getDisplayName());
    }
}

```

```
localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());
currentDate = new Date();
currentTime = new Date();
updateDisplay();

ActionListener listener = new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        updateDisplay();
    }
};

localeCombo.addActionListener(listener);
dateStyleCombo.addActionListener(listener);
timeStyleCombo.addActionListener(listener);

dateParseButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        String d = dateText.getText().trim();
        try
        {
            currentDateFormat.setLenient(lenientCheckbox.isSelected());
            Date date = currentDateFormat.parse(d);
            currentDate = date;
            updateDisplay();
        }
        catch (ParseException e)
        {
            dateText.setText("Parse error: " + d);
        }
        catch (IllegalArgumentException e)
        {
            dateText.setText("Argument error: " + d);
        }
    }
});

timeParseButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        String t = timeText.getText().trim();
        try
        {
            currentDateFormat.setLenient(lenientCheckbox.isSelected());
            Date date = currentTimeFormat.parse(t);
            currentTime = date;
            updateDisplay();
        }
        catch (ParseException e)
        {
            timeText.setText("Parse error: " + t);
        }
        catch (IllegalArgumentException e)
        {

```

```

        timeText.setText("Argument error: " + t);
    }
}
);
pack();
}

/**
 * Aktualizuje prezentowaną informację i formatuje pola
 * tekstowe zgodnie z wyborem użytkownika.
 */
public void updateDisplay()
{
    Locale currentLocale = locales[localeCombo.getSelectedIndex()];
    int dateStyle = dateStyleCombo.getValue();
    currentDateFormat = DateFormat.getDateInstance(dateStyle, currentLocale);
    String d = currentDateFormat.format(currentDate);
    dateText.setText(d);
    int timeStyle = timeStyleCombo.getValue();
    currentTimeFormat = DateFormat.getTimeInstance(timeStyle, currentLocale);
    String t = currentTimeFormat.format(currentTime);
    timeText.setText(t);
}
}

```

Korzystając z programu, możemy też eksperymentować z parsowaniem dat. Po wpisaniu daty lub czasu możemy zaznaczyć pole wyboru *Check lenient*, a następnie wybrać przycisk *Parse date* lub *Parse time*.

Klasy pomocniczej `EnumCombo` użyliśmy do rozwiązania problemu natury technicznej (patrz listing 5.3). Listę rozwijalną stylów prezentacji dat chcieliśmy wypełnić wartościami `Short`, `Medium` i `Long`, a następnie automatycznie przetworzyć wybraną przez użytkownika wartość na odpowiadającą jej wartość `DateFormat.SHORT`, `DateFormat.MEDIUM` lub `DateFormat.LONG`. W tym celu wybraną przez użytkownika pozycję listy przetwarzamy tak, że wszystkie jej litery zostają zamienione na wielkie litery, a odstępy — na znaki podkreślenia. Następnie korzystamy z refleksji, aby odnaleźć wartość składowej statycznej klasy o takiej nazwie. (Więcej informacji na temat refleksji odnajdziemy w 5. rozdziale książki *Java. Podstawy*.)

**Listing 5.3.** `dateFormat/EnumCombo.java`

```

package dateFormat;

import java.util.*;
import javax.swing.*;

/**
 * Lista rozwijana pozwalająca użytkownikowi wybrać
 * jedną spośród wartości pól statycznych,
 * których nazwy zostały przekazane konstruktorowi.
 * @version 1.14 2012-01-26
 * @author Cay Horstmann
 */
public class EnumCombo extends JComboBox<String>
{
    private Map<String, Integer> table = new TreeMap<>();

```

```

    /**
     * Tworzy EnumCombo.
     * @param cl klasa
     * @param labels tablica nazw pól statycznych klasy cl
     */
    public EnumCombo(Class<?> cl, String... labels)
    {
        for (String label : labels)
        {
            String name = label.toUpperCase().replace(' ', '_');
            int value = 0;
            try
            {
                java.lang.reflect.Field f = cl.getField(name);
                value = f.getInt(cl);
            }
            catch (Exception e)
            {
                label = "(" + label + ")";
            }
            table.put(label, value);
            addItem(label);
        }
        setSelectedItem(labels[0]);
    }

    /**
     * Zwraca wartość pola wybranego przez użytkownika.
     * @return wartość pola statycznego
     */
    public int getValue()
    {
        return table.get(getSelectedItem());
    }
}

```

**API** `java.text.DateFormat 1.1`

- `static Locale[] getAvailableLocales()`  
zwraca tablicę obiektów klasy `Locale`, dla których dostępne są obiekty klasy `DateFormat`.
  - `static DateFormat getDateInstance(int dateStyle)`
  - `static DateFormat getDateInstance(int dateStyle, Locale l)`
  - `static DateFormat getTimeInstance(int timeStyle)`
  - `static DateFormat getTimeInstance(int timeStyle, Locale l)`
  - `static DateFormat getDateInstance(int dateStyle, int timeStyle)`
  - `static DateFormat getDateInstance(int dateStyle, int timeStyle, Locale l)`
- Zwracają obiekt formatujący daty, czas lub i daty, i czas dla domyślnego lub podanego lokalizatora.

*Parametry:* dateStyle, timeStyle jedna z wartości DEFAULT, FULL, LONG, MEDIUM lub SHORT.

- `String format(Date d)`

zwraca łańcuch będący wynikiem formatowania podanej daty.

- `Date parse(String s)`

parsuje podany łańcuch i zwraca opisaną przez niego datę i (lub) czas. Początek łańcucha musi zawierać reprezentację daty lub czasu. Nie są dozwolone poprzedzające ją odstępy. Znaki następujące po reprezentacji daty są ignorowane. Metoda wyrzuca wyjątek ParseException, jeśli parsowanie nie powiedzie się.

- `void setLenient(boolean b)`

- `boolean isLenient()`

Ustawiają lub pobierają znacznik, który określa dokładność parsowania. Jeśli znacznik zostanie ustawiony, to na przykład data February 30, 1999 zostanie automatycznie poddana konwersji do daty March 2, 1999. Domyślnie znacznik jest ustawiony.

- `void setCalendar(Calendar cal)`

- `Calendar getCalendar()`

Określają lub pobierają obiekt kalendarza wykorzystywany do pobrania wartości roku, miesiąca, dnia, godziny, minut i sekundy z obiektu Date. Metody te należy zastosować, jeśli nie chcemy korzystać z domyślnego kalendarza (zwykle gregoriańskiego) dla danego lokalizatora.

- `void setTimezone(TimeZone tz)`

- `TimeZone getTimeZone()`

Ustawiają lub pobierają obiekt reprezentujący strefę czasową i wykorzystywany do formatowania czasu. Metody te należy zastosować, jeśli nie chcemy korzystać ze strefy czasowej domyślnej dla danego lokalizatora. Domyślana strefa czasowa jest strefą czasową domyślnego lokalizatora uzyskaną od systemu operacyjnego. Dla innych lokalizatorów strefa czasowa odpowiada lokalizacji geograficznej określonej przez lokalizator.

- `void setNumberFormat(NumberFormat f)`

- `NumberFormat getNumberFormat()`

Ustawiają lub pobierają obiekt formatujący wykorzystywany do formatowania liczb reprezentujących rok, miesiąc, dzień, godzinę, minutę i sekundę.

#### API java.util.TimeZone 1.1

- `static String[] getAvailableIDs()`

zwraca identyfikatory wszystkich obsługiwanych stref czasowych.

- `static TimeZone getDefault()`

zwraca domyślną strefę czasową.

- static TimeZone getTimeZone(String timeZoneId)  
zwraca obiekt TimeZone dla podanego identyfikatora strefy czasowej.
- String getID()  
zwraca identyfikator strefy czasowej.
- String getDisplayName()
- String getDisplayName(Locale locale)
- String getDisplayName(boolean daylight, int style)
- String getDisplayName(boolean daylight, int style, Locale locale)  
zwracają nazwę strefy czasowej przy zastosowaniu domyślnego lub podanego lokalizatora. Jeśli parametr daylight posiada wartość true, to zwracana jest nazwa uwzględniająca zmianę czasu. Parametr style może przyjmować wartości SHORT lub LONG.
- boolean useDaylightTime()  
zwraca wartość true, jeśli strefa czasowa stosuje zmianę czasu dla oszczędności energii elektrycznej.
- boolean inDaylightTime(Date date)  
zwraca wartość true, jeśli podana data przypada w okresie zmiany czasu dla oszczędności energii elektrycznej.

## 5.4. Porządek alfabetyczny

Dwa łańcuchy możemy porównać za pomocą metody compareTo klasy String. Wartość zwrócona przez wywołanie a.compareTo(b) jest ujemna, jeśli łańcuch a poprzedza łańcuch b w porządku leksykograficznym, jest równa 0, jeśli łańcuchy są identyczne lub dodatnia — w pozostałych przypadkach.

Metoda ta jest przydatna jedynie wtedy, gdy łańcuchy zawierają wyłącznie wielkie litery kodu ASCII języka angielskiego. Wykorzystuje ona kod Unicode, w przypadku którego wartości kodu dla małych liter są większe od wartości kodu wielkich liter, a litery akcentowane posiadają jeszcze większe wartości kodu. Prowadzi to do absurdalnych wyników. Przykładem może być uporządkowanie poniższych łańcuchów uzyskane za pomocą metody compareTo:

America  
Zulu  
able  
zebra  
Ångström

Uporządkowanie słownikowe nie powinno rozróżniać małych i wielkich liter. Dla osoby używającej na co dzień języka angielskiego uporządkowanie tych łańcuchów powinno wyglądać następująco:

```
able
America
Ångström
zebra
Zulu
```

Jednak porządek taki będzie nie do zaakceptowania w języku szwedzkim. Szwedzi odróżniają literę A od Å, która umieszczona jest w alfabetie po literze Z! Dlatego też Szwed zaakceptuje jedynie poniższe uporządkowanie.

```
able
America
zebra
Zulu
Ångström
```

Gdy zdamy sobie sprawę z istnienia tego problemu, to rozwiązywanie okaże się łatwe. Jak zwykle rozpoczęmy od uzyskania odpowiedniego lokalizatora. Następnie skorzystamy z metody fabryki getInstance, by uzyskać obiekt typu Collator. Porównując łańcuchy, wykorzystamy jego metodę compare zamiast metody compareTo klasy String.

```
Locale loc = . . . ;
Collator coll = Collator.getInstance(loc);
if (coll.compare(a, b) < 0) //a poprzedza b . . .
```

Klasa Collator implementuje interfejs Comparator. Jej obiekty możemy więc przekazać metodzie Collections.sort sortującej listę łańcuchów:

```
Collections.sort(strings, coll);
```

## 5.4.1. Moc uporządkowania

Dla obiektu klasy Collator możemy opisać *moc*, określającą stopień jego selektywności. Różnice między znakami klasyfikowane są jako *pierwszo-, drugo- i trzeciorzędne*. Na przykład w języku angielskim różnica pomiędzy „A” i „Z” uważana jest za pierwszorzędną, między „A” i „Å” — za drugorzędną, a między „A” i „a” — za trzeciorzędną.

Określając moc obiektu klasy Collator za pomocą stałej Collator.PRIMARY, powodujemy, że będzie on brał pod uwagę jedynie pierwszorzędne różnice. Określenie mocy za pomocą stałej Collator.SECONDARY spowoduje także uwzględnienie różnic drugorzędnych. W tym przypadku wzrośnie więc prawdopodobieństwo, że dwa łańcuchy uznane zostaną za różne. Tabela 5.4 przedstawia sposób uporządkowania zbioru łańcuchów dla różnych mocy obiektu klasy Collator.

**Tabela 5.4.** Uporządkowanie łańcuchów dla różnej mocy obiektu Collator

PRIMARY	SECONDARY	TERCIARY
Angstrom = Ångström	Angstrom ≠ Ångström	Angstrom ≠ Ångström
Able = able	Able = able	Able ≠ able

Określając moc jako `Collator.IDENTICAL`, zabraniamy występowania jakichkolwiek różnic. W praktyce możliwość ta jest użyteczna jedynie w połączeniu z *trybem rozkładu*, który omawiamy w następnym podrozdziale.

## 5.4.2. Rozkład

Unicode umożliwia w niektórych przypadkach opis znaku lub sekwencji znaków na kilka sposobów. Na przykład znak "À" możemy zapisać w Unicode jako pojedynczy znak o kodzie U+00C5 lub jako kombinację znaku A (U+0065) ze znakiem ° (U+030A). Jeszcze bardziej zaskakująca jest możliwość zapisania sekwencji trzech znaków "ffí" jako pojedynczego znaku o kodzie U+FB03.

Standard Unicode definiuje cztery *formy normalizacji* (D, KD, C i KC) dlałańcuchów znakowych. Szczegóły na ten temat można znaleźć na stronie <http://www.unicode.org/unicode/reports/tr15/tr15-23.html>. Dwie z nich są stosowane przy określaniu uporządkowania. W przypadku formy normalizacji D znaki akcentowane podlegają rozkładowi na znak podstawowy i akcent. Na przykład znak À zostanie rozłożony na sekwencję znaku A i znaku °. Forma normalizacji KD stosuje jeszcze dalej idący rozkład, któremu zostają poddane np. wspomniana wcześniej ligatura ffí czy symbol ™.

Możemy sami określić stopień normalizacji podczas uporządkowania. Wartość `Collator.NO_DECOMPOSITION` powoduje, żełańcuchy nie są wcale poddawane normalizacji. Przypięsza to ich uporządkowanie, ale może nie być poprawne w przypadku tekstów zapisujących znaki w różnych postaciach. Domyślna wartość `Collator.CANONICAL_DECOMPOSITION` oznacza zastosowanie formy normalizacji D. Jest ona najważniejsza dla tekstów, które stosują znaki akcentowane, ale nie zawierają ligatur. Pełna dekompozycja jest możliwa przez zastosowanie formy normalizacji KD. Zagadnienie to ilustrują przykłady pokazane w tabeli 5.5.

**Tabela 5.5.** Różnice pomiędzy trybami rozkładu

Brak rozkładu	Rozkład kanoniczny	Pien rozkład
À ≠ A°	À = A°	À = A°
TM ≠ TM	TM ≠ TM	TM = TM

Wielokrotna dekompozycja tego samegołańcucha znaków stanowi niepotrzebne marnotrawstwo czasu procesora. Jeśli więc danyłańcuch jest wielokrotnie porównywany z innymiłańcuchami, to jego dekompozycję możemy przechować za pomocą obiektu `CollationKey`. Metoda `getCollationKey` zwraca taki obiekt, który możemy wykorzystać do dalszych porównań. Poniżej przykład jego wykorzystania.

```
String a = ...;
CollationKey aKey = coll.getCollationKey(a);
if (aKey.compareTo(coll.getCollationKey(b)) == 0)//efektywne porównanie
```

Normalizacjałańcuchów znakowych może być również przydatna w sytuacjach innych niż ich alfabetyczne uporządkowanie; na przykład do zapisułańcuchów w bazie danych lub podczas ich wymiany z inną aplikacją. Klasa `java.text.Normalizer` umożliwia nam wtedy przeprowadzenie procesu normalizacji. Na przykład:

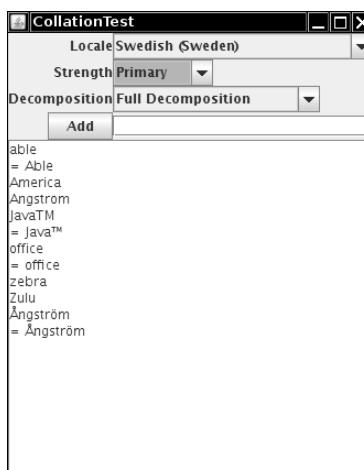
```
String name = "Ångström";
String normalized = Normalizer.normalize(name, Normalizer.Form.NFD);
// stosuje formę normalizacji D
```

Znormalizowany łańcuch będzie zawierać dziesięć znaków. Znaki "Å" i "ö" zostaną zastąpione sekwencjami "A°" i "o·".

Taka forma normalizacji nie jest jednak najlepsza do przechowywania i transmisji łańcuchów znakowych. Forma normalizacji C umożliwia najpierw zastosowanie rozkładu, a następnie przywrócenie akcentów w standardowy sposób. Zgodnie z zaleceniami W3C taki właśnie sposób powinien być stosowany podczas transmisji danych w internecie.

Program, którego kod źródłowy zawiera listing 5.4 pozwala eksperymentować z uporządkowaniem łańcuchów znakowych. Nowy łańcuch wpisujemy w polu tekstowym i wybieramy przycisk *Add*, dodając go w ten sposób do listy. Za każdym razem gdy dodamy nowy łańcuch, zmienimy lokalizator, moc lub tryb dekompozycji, lista słów zostanie uporządkowana od nowa. Znak = pojawiający się na liście oznacza, że słowa są uważane za identyczne z punktu widzenia uporządkowania (patrz rysunek 5.3).

**Rysunek 5.3.**  
Program  
*CollationTest*  
w działaniu



**Listing 5.4.** *collation/CollationTest.java*

```
package collation;

import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;
import java.util.List;

import javax.swing.*;

/**
 * Program demonstrujący uporządkowanie łańcuchów znakowych
 * dla różnych lokalizatorów.
 * @version 1.14 2012-01-26
 * @author Cay Horstmann
 */
```

```

public class CollationTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new CollationFrame();
                frame.setTitle("CollationTest");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

/**
 * Ramka zawierająca listę rozwijaną wyboru lokalizatora,
 * listy rozwijane wyboru mocy i trybu dekompozycji,
 * pole tekstowe i przycisk umożliwiające dodawanie nowych łańcuchów
 * oraz obszar tekstowy prezentujący uporządkowaną listę słów.
 */
class CollationFrame extends JFrame
{
    private Collator collator = Collator.getInstance(Locale.getDefault());
    private List<String> strings = new ArrayList<>();
    private Collator currentCollator;
    private Locale[] locales;
    private JComboBox<String> localeCombo = new JComboBox<>();
    private JTextField newWord = new JTextField(20);
    private JTextArea sortedWords = new JTextArea(20, 20);
    private JButton addButton = new JButton("Add");
    private EnumCombo strengthCombo = new EnumCombo(Collator.class, "Primary",
        "Secondary", "Tertiary", "Identical");
    private EnumCombo decompositionCombo = new EnumCombo(Collator.class,
        "Canonical Decomposition", "Full Decomposition", "No Decomposition");

    public CollationFrame()
    {
        setLayout(new GridBagLayout());
        add(new JLabel("Locale"), new GBC(0, 0).setAnchor(GBC.EAST));
        add(new JLabel("Strength"), new GBC(0, 1).setAnchor(GBC.EAST));
        add(new JLabel("Decomposition"), new GBC(0, 2).setAnchor(GBC.EAST));
        add(addButton, new GBC(0, 3).setAnchor(GBC.EAST));
        add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
        add(strengthCombo, new GBC(1, 1).setAnchor(GBC.WEST));
        add(decompositionCombo, new GBC(1, 2).setAnchor(GBC.WEST));
        add(newWord, new GBC(1, 3).setFill(GBC.HORIZONTAL));
        add(new JScrollPane(sortedWords), new GBC(0, 4, 2, 1).setFill(GBC.BOTH));

        locales = (Locale[]) Collator.getAvailableLocales().clone();
        Arrays.sort(locales, new Comparator<Locale>()
        {
            public int compare(Locale l1, Locale l2)
            {
                return collator.compare(l1.getDisplayName(), l2.getDisplayName());
            }
        });
    }
}

```

```

    });
    for (Locale loc : locales)
        localeCombo.addItem(loc.getDisplayName());
    localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());

    strings.add("America");
    strings.add("able");
    strings.add("Zulu");
    strings.add("zebra");
    strings.add("\u00C5ngstr\u00F6m");
    strings.add("A\u030angstro\u0308m");
    strings.add("Angstrom");
    strings.add("Able");
    strings.add("office");
    strings.add("o\uFB03ce");
    strings.add("Java\u2122");
    strings.add("JavaTM");
    updateDisplay();

    addButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            strings.add(newWord.getText());
            updateDisplay();
        }
    });
}

ActionListener listener = new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        updateDisplay();
    }
};

localeCombo.addActionListener(listener);
strengthCombo.addActionListener(listener);
decompositionCombo.addActionListener(listener);
pack();
}

/**
 * Aktualizuje prezentowaną informację i porządkuje łańcuchy
 * zgodnie z wyborem użytkownika.
 */
public void updateDisplay()
{
    Locale currentLocale = locales[localeCombo.getSelectedIndex()];
    localeCombo.setLocale(currentLocale);

    currentCollator = Collator.getInstance(currentLocale);
    currentCollator.setStrength(strengthCombo.getValue());
    currentCollator.setDecomposition(decompositionCombo.getValue());

    Collections.sort(strings, currentCollator);

    sortedWords.setText("");
}

```

```

        for (int i = 0; i < strings.size(); i++)
    {
        String s = strings.get(i);
        if (i > 0 && currentCollator.compare(s, strings.get(i - 1)) == 0) sortedWords
            .append("= ");
        sortedWords.append(s + "\n");
    }
    pack();
}
}

```

Łańcuchy nazw lokalizatorów zostają posortowane zgodnie z porządkiem leksykograficznym domyślnego lokalizatora. Jeśli lokalizatorem tym będzie US English, to na liście lokalizatorów "Norwegian (Norway,Nynorsk)" poprzedzi "Norwegian (Norway)", mimo że wartość kodu Unicode dla znaku przecinka jest większa niż dla zamykającego nawiasu.

#### java.text.Collator 1.1

- static Locale[] getAvailableLocales()
 

zwraca tablicę obiektów klasy Locale, dla których dostępne są obiekty klasy Collator.
- static Collator getInstance()
- static Collator getInstance(Locale l)
 

Zwracają obiekt klasy Collator dla domyślnego lub podanego lokalizatora.
- int compare(String a, String b)
 

zwraca wartość ujemną, gdy łańcuch a poprzedza łańcuch b, 0, jeśli łańcuchy są identyczne z punktu widzenia uporządkowania oraz wartość dodatnią — w pozostałych przypadkach.
- boolean equals(String a, String b)
 

zwraca wartość true, jeśli łańcuchy są identyczne, wartość false — w przeciwnym przypadku.
- void setStrength(int strength)
- int getStrength()
 

Ustawiają lub pobierają moc obiektu klasy Collator. Efektem większej mocy jest rozróżnianie większej ilości słów. Moc może przyjmować jedną z wartości Collator.PRIMARY, Collator.SECONDARY oraz Collator.TERTIARY.

- void setDecomposition(int decomp)
- int getDecomposition()

Ustawiają lub pobierają tryb rozkładu wykonywanego przez obiekt klasy Collator. Im pełniejszy stopień rozkładu, tym większa dokładność decyzji o identyczności łańcuchów. Tryb rozkładu może być określony za pomocą jednej z wartości Collator.NO\_DECOMPOSITION, Collator.CANONICAL\_DECOMPOSITION lub Collator.FULL\_DECOMPOSITION.

- `CollationKey getCollationKey(String a)`

zwraca obiekt klasy `CollationKey`, który zawiera rozkład łańcucha wykorzystywany do wykonania szybkiego porównania z obiektem `CollationKey` innego łańcucha.

#### `java.text.CollationKey 1.1`

- `int compareTo(CollationKey b)`

zwraca wartość ujemną, gdy dany klucz poprzedza klucz b, 0, jeśli są identyczne, wartość dodatnią — w pozostałych przypadkach.

#### `java.text.Normalizer 6`

- `static String normalize(CharSequence str, Normalizer.Form form)`

zwraca znormalizowaną postać łańcucha str. Parametr `form` może przyjmować wartość ND, NKD, NC lub NKC.

## 5.5. Formatowanie komunikatów

Język Java udostępnia klasę `MessageFormat` umożliwiającą formatowanie tekstu zawierającego znaczniki, na przykład:

```
"On {2}, a {0} destroyed {1} houses and caused {3} of damage."
```

Numery w nawiasach klamrowych zastępowane są później właściwymi nazwami i wartościami. Metoda statyczna `MessageFormat.format` umożliwia podstawienie wartości za znaczniki. W JDK 5.0 metoda ta posiada zmienną liczbę parametrów, wobec czego wartości znaczników możemy dostarczyć w następujący sposób:

```
String msg = MessageFormat.format("On {2}, a {0} destroyed {1} houses and caused {3} of damage.", "hurricane", 99, new GregorianCalendar(1999, 1, 1).getTime(), 10.0E8);
```

W powyższym przykładzie znacznik `{0}` zostanie zastąpiony łańcuchem "hurricane", `{1}` liczbą 99 i tak dalej.

Wykonanie omówionego fragmentu kodu spowoduje wyświetlenie następującego napisu:

```
On 1/1/99 12:00 AM, a hurricane destroyed 99 houses  
and caused 100,000,000 of damage.
```

Całkiem nieźle jak na początek, ale wolelibyśmy, aby komunikat nie podawał niepotrzebnie czasu oraz wyrastał wielkość szkód jako kwotę w określonej walucie. Możemy to osiągnąć, dostarczając opcjonalnej informacji o formacie niektórych znaczników.

```
"On {2,date,long}, a {0} touched down at {2} and destroyed {1} houses  
and caused {3,number,currency} of damage."
```

W tym przypadku wyświetlony zostanie napis:

```
On January 1, 1999, a hurricane destroyed 99 houses  
and caused $100,000,000 of damage.
```

Indeks znacznika możemy opatrzyć informacjami o *typie* i *stylu*, rozdzielając je znakiem przecinka. Typ może być następujący:

number  
time  
date  
choice

Jeśli typem jest number, to stylem może być:

integer  
currency  
percent

lub wzorzec formatu liczby — na przykład `$.\#\#0.` (Więcej informacji o formatach liczb można znaleźć w dokumentacji klasy `DecimalFormat`).

Jeśli typem jest time lub date, to stylem może być

short  
medium  
long  
full

lub wzorzec formatu daty taki jak na przykład `yyyy-MM-dd`. (Więcej informacji o formatach daty odnajdziemy w dokumentacji klasy `SimpleDateFormat`).

Formatowanie z wariantami jest znacznie bardziej skomplikowane. Omówimy je w następnym podrozdziale.

 Metoda statyczna `MessageFormat.format` używa lokalizatora bieżącego. Zastosowanie dowolnego lokalizatora wymaga nieco więcej wysiłku, ponieważ nie istnieje metoda o zmiennej liczbie parametrów, której moglibyśmy użyć w tym celu. Aby zastosować inny lokalizator, należy umieścić formatowane wartości w tablicy `Object[]` i posłużyć się następującym fragmentem kodu:

```
MessageFormat mf = new MessageFormat(pattern, loc);
String msg = mf.format(new Object[] { values });
```

#### API `java.text.MessageFormat 1.1`

- `MessageFormat(String pattern)`
- `MessageFormat(String pattern, Locale loc)`
- tworzy obiekt klasy `MessageFormat` z podanego wzorca.
- `void applyPattern(String pattern)`
- określa wzorzec obiektu klasy `MessageFormat`.
- `void setLocale(Locale loc)`
- `Locale getLocale()`

Okręślają lub pobierają lokalizator używany do formatowania znaczników komunikatu. Podany lokalizator będzie wykorzystany *tylko* dla kolejnych wzorców określonych za pomocą metody `applyPattern`.

- static String format(String pattern, Object... args)  
Formatuje komunikat, korzystając z obiektu args[i] dla znacznika {i}.
- StringBuffer format(Object args, StringBuffer result, FieldPosition pos)  
Formatuje komunikat za pomocą obiektu MessageFormat. Parametr args musi być tablicą obiektów. Sformatowany łańcuch zostaje dołączony do bufora result. Metoda zwraca bufor result. Jeśli pos równy jest new FieldPosition(MessageFormat.Field. ARGUMENT), to jego właściwości beginIndex i endIndex określają położenie tekstu, który zastępuje znacznik {1}. Jeśli nie jesteśmy zainteresowani tą informacją, to przekazujemy null jako wartość parametru pos.

**API java.text.Format 1.1**

- String format(Object obj)  
formatuje podany obiekt za pomocą obiektu formatującego. Metoda ta wywołuje format(obj, new StringBuffer(), new FieldPosition(1)).toString().

## 5.5.1. Formatowanie z wariantami

Przyjrzyjmy się bliżej wzorcowi z poprzedniego rozdziału:

"On {2}, a {0} destroyed {1} houses and caused {3} of damage."

Jeśli znacznik {0} zastąpimy łańcuchem "earthquake", to otrzymamy zdanie w języku angielskim, które nie jest poprawne gramatycznie.

On January 1, 1999, a earthquake destroyed . . .

Musimy więc zintegrować rodzajnik nieokreślony ze znacznikiem:

"On {2}, {0} destroyed {1} houses and caused {3} of damage."

Odtąd możemy zastąpić znacznik {0} łańcuchem "a hurricane" lub "an earthquake". Ma to szczególne znaczenie, jeśli przetłumaczymy komunikat na język, w którym rodzaj rzeczownika ma wpływ na postać rodzajnika. Na przykład w języku niemieckim nasz wzorzec będzie wyglądać następująco.

"{0} zerstörte am {2} {1} Häuser und richtete einen Schaden von {3} an."

Następnie znacznik {0} zostanie zastąpiony poprawną gramatycznie kombinacją rodzajnika i rzeczownika, na przykład "Ein Wirbelsturm" czy "Eine Naturkatastrophe".

Zajmijmy się teraz znacznikiem {1}. Jeśli komunikat będzie opisywał wypadek, a nie klęskę żywiołową, to znacznik ten może zostać zastąpiony wartością 1:

On January 1, 1999, a mudslide destroyed 1 houses and . . .

Musimy więc zmienić postać wzorca, tak by liczba rzeczownika była zgodna z wartością znacznika, jak poniżej:

no houses  
one house  
2 houses

.

W tym celu zaprojektowana została opcja formatowania choice.

Jest ona sekwencją par składających się z:

- dolnego ograniczenia,
- łańcucha formatującego.

Elementy te oddzielone są znakiem #, a kolejne pary sekwencji oddziela się znakiem |.

Na przykład:

```
{1.choice.0#no houses|1#one house|2#{1} houses}
```

W tabeli 5.6 przedstawione zostały wyniki zastosowania powyższego łańcucha formatującego dla różnych wartości znacznika {1}.

**Tabela 5.6.** Łańcuch sformatowany przy użyciu opcji choice

①	Wynik
0	"no houses"
1	"one house"
3	"3 houses"
-1	"no houses"

Dlaczego łańcuch formatujący zawiera znowu znacznik {1}? Gdy obiekt klasy MessageFormat wykorzysta obiekt klasy ChoiceFormat, to uzyska w rezultacie łańcuch w postaci "{1} houses". Łańcuch ten zostanie jeszcze raz sformatowany przez obiekt klasy MessageFormat i dopiero ten wynik będzie umieszczony w tekście komunikatu.



Przykład ten nie świadczy zbyt dobrze o projektantach klasy ChoiceFormat. Dla oddzielenia trzech łańcuchów formatujących wystarczą bowiem dwie wartości graniczne. W ogólnym przypadku potrzebujemy zawsze o jedną wartość graniczną mniej niż wynosi liczba łańcuchów formatujących. W tabeli 5.6 pokazaliśmy, że klasa MessageFormat ignoruje pierwsze ograniczenie.

Składnia wariantów formatowania uprościłaby się, gdyby projektanci zauważyli, że wartości graniczne znajdują się pomiędzy wariantami, na przykład

```
no houses|1|one house|2|{1} houses //format ten nie jest dozwolony
```

Możemy użyć także znaku < dla zaznaczenia, że wariant powinien zostać wybrany, jeśli wartość znacznika jest większa od wartości granicznej.

Znak ≤ (wyrażony za pomocą kodu Unicode \u2264) możemy zastosować jako synonim znaku #. Możemy nawet określić dolne ograniczenie jako -∞ (czyli -\u221E).

Na przykład

-∞<no houses|0<one house|2≤{1} houses

lub za pomocą kodów Unicode

-\u221E< no houses|0<one house|2\u2264{1} houses

Powróćmy do naszego przykładowego komunikatu. Jeśli umieścimy w nim łańcuch wyboru, to otrzymamy w rezultacie następującą instrukcję formatowania:

```
String pattern =
"On {2,date,long}. {0} destroyed {1.choice.0#no house|1#one
house|2#{1} houses} +
and caused {3,number,currency} of damage."
```

Podobnie dzieje się w języku niemieckim:

```
String pattern =
"{0} zerstörte am {2,date,long} {1.choice.0#kein Haus|1#ein
Haus|2#{1} Häuser} +
und richtete einen Schaden von {3,number,currency} an."
```

Zwrócmy uwagę, że choć porządek słów w języku niemieckim jest inny, to przekazujemy metodzie format *tę samą* tablicę obiektów. Uporządkowanie znaczników we wzorcu odpowiada bowiem uporządkowaniu słów.

## 5.6. Pliki tekstowe i zbiory znaków

Chociaż platforma Java została w pełni oparta o zestaw znaków Unicode, to jednak poszczególne systemy operacyjne wykorzystują zwykle własne, często niezgodne ze sobą, zestawy znaków, jak na przykład ISO 8859-1 (8-bitowy kod zwany czasami kodem ANSI) w USA czy BIG5 na Tajwanie.

Zapisując dane w pliku tekstowym, powinniśmy uwzględnić lokalny kod znaków, aby użytkownicy programu mogli otworzyć plik tekstowy za pomocą innych aplikacji. Kod ten określamy, wywołując konstruktor klasy `FileWriter`, na przykład:

```
out = new FileWriter(filename, "ISO-8859-1");
```

Kompletną listę obsługiwanych kodów zamieściliśmy w tabeli 1.1.

Niestety obecnie nie istnieje żadna zależność pomiędzy lokalizatorami a kodowaniem znaków. Na przykład jeśli użytkownik wybrał lokalizator chiński zh\_TW, to nie jest dostępna żadna metoda, która poinformowałaby nas, że najodpowiedniejszym sposobem kodowania znaków będzie BIG5.

### 5.6.1. Internacjonalizacja a pliki źródłowe programów

Kiedy tworzymy program w języku Java, to komunikujemy się z kompilatorem, korzystając z narzędzi lokalnego systemu operacyjnego. Tekst źródłowy programu możemy na przykład pisać za pomocą chińskiej wersji programu Notepad. Powstałe w ten sposób pliki źródłowe

nie są przenośne, ponieważ używają lokalnego kodowania znaków (GB lub BIG5 w zależności od wersji chińskiego systemu operacyjnego). Jedynie pliki klas są przenośne, ponieważ automatycznie stosują kod „zmodyfikowany” UTF-8 dlałańcuchów znakowych. W przypadku każdego skompilowanego i działającego programu w języku Java wykorzystywane są więc trzy różne kodowania znaków:

- lokalny kod dla plików źródłowych,
- kod „zmodyfikowany” UTF-8 dla plików klas,
- UTF-16 dla maszyny wirtualnej Java.

(Definicje kodów „zmodyfikowany” UTF-8 i UTF-16 zamieściliśmy w rozdziale 1.)



Sposób kodowania plików źródłowych możemy określić za pomocą znacznika -encoding, na przykład:

```
java -encoding Big5 Myfile.java
```

Aby można było swobodnie przenosić pliki źródłowe, powinniśmy ograniczyć się do wykorzystania w nich wyłącznie podstawowych znaków ASCII. Wszystkie pozostałe znaki należy zastąpić odpowiadającymi im kodami Unicode. Na przykład zamiast łańcucha "Häuser" należy użyć łańcucha "H\u00e4user". Pakiet JDK zawiera program narzędziowy native2ascii, który możemy wykorzystać do zamiany kodów takich znaków na zwykły kod ASCII. Program ten zamienia każdy znak spoza podstawowego zestawu ASCII na łańcuch \u, po którym następują 4 cyfry szesnastkowe kodu Unicode. Uruchamiając program native2ascii, podajemy nazwę pliku wejściowego i wyjściowego.

```
native2ascii Myfile.java Myfile.temp
```

Konwersję tę możemy przeprowadzić także w kierunku odwrotnym, korzystając z opcji -reverse.

```
native2ascii -reverse Myfile.java Myfile.temp
```

Natomiast za pomocą opcji -encoding możemy określić inny rodzaj kodowania. Nazwa kodu musi być jedną z nazw, które podaliśmy w tabeli 1.1.

```
native2ascii -encoding Big5 Myfile.java Myfile.temp
```



Zdecydowanie zalecamy także ograniczenie się do nazw klas zapisywanych tylko za pomocą podstawowych znaków ASCII. Ponieważ nazwa klasy staje się na skutek komplikacji nazwą pliku klasy, to w przypadku znaków spoza zestawu podstawowego ASCII jesteśmy zdani wyłącznie na łaskę lokalnego systemu plików, który z reguły nie obsługuje takiej nazwy poprawnie. Na przykład system Windows 95 stosuje dla nazw plików własny sposób kodowania o nazwie *Code Page 437*. Jeśli na przykład nadamy klasie nazwę Bär, to program ładujący poinformuje nas, że nie może odnaleźć klasy o nazwie B  r.

## 5.7. Komplety zasobów

Poddając aplikację procesowi lokalizacji, szybko uzyskujemy przerzążająco dużą liczbę łańcuchów znaków reprezentujących teksty komunikatów, etykiety przycisków i inne elementy interfejsu użytkownika. Wszystkie one muszą zostać przetłumaczone. Najlepiej będzie, jeśli zdefiniujemy je w osobnym pliku zwanym *zasobem*. Osoba wykonująca tłumaczenie może wtedy edytować plik zasobu bez konieczności ingerowania w kod programu.

W języku Java zasoby w postaci łańcuchów znakowych implementujemy za pomocą plików właściwości, a zasoby innych typów za pomocą klas.



Zasoby w języku Java różnią się od zasobów wykorzystywanych przez programy systemu Windows czy Macintosh. Program wykonywalny systemu Windows przechowuje zasoby (menu, okna dialogowe, ikony, komunikaty) w sekcji oddzielonej od kodu programu. Dzięki temu możemy poddawać je edycji za pomocą edytora zasobów bez konieczności ingerowania w kod programu.



W 10. rozdziale książki *Java. Podstawy* przedstawiliśmy umieszczanie plików zasobów (zawierających na przykład dźwięki, obrazy) w pojedynczym pliku JAR. Metoda `getResource` klasy `Class` odnajduje odpowiedni plik, otwiera go i zwraca adres URL zasobu. W ten sposób program ładujący klasy wykonuje dla nas zadanie odnalezienia właściwego pliku zasobów. Mechanizm ten jednak nie obsługuje lokalizatorów.

### 5.7.1. Wyszukiwanie kompletów zasobów

Lokalizując aplikację, musimy utworzyć zbiór *kompletów zasobów*. Każdy komplet zasobów jest plikiem właściwości lub klasą opisującą elementy specyficzne (takie jak komunikaty, etykiety itd.) dla lokalizatorów, które zamierzamy obsługiwać. Dla każdego kompletu musimy dostarczyć wersji dla wszystkich obsługiwanych lokalizatorów.

Dla nazw kompletów musimy zastosować także odpowiednią konwencję. Na przykład zasoby wykorzystywane przez użytkowników z Niemiec umieścimy w pliku *nazwaKompletu\_de\_DE*, a dla użytkowników ze wszystkich krajów niemieckojęzycznych w klasie *nazwaKompletu\_de*. W ogólnym przypadku używamy nazw kompletów postaci:

*nazwaKompletu\_język\_kraj*

dla zasobów specyficznych dla poszczególnych krajów oraz nazw:

*nazwaKompletu\_język*

dla zasobów specyficznych dla języków. Wartości domyślne możemy umieścić w pliku, którego nazwa nie zawiera żadnych przyrostków.

Komplet zasobów ładujemy w następujący sposób:

```
ResourceBundle currentResources =
    ResourceBundle.getBundle(nazwaKompletu, bieżącyLokalizator);
```

Metoda `getBundle` próbuje załadować komplet, którego nazwa odpowiada bieżącemu lokalizatorowi ze względu na język, kraj i wariant. Jeśli się to nie powiedzie, to przy kolejnych próbach nie jest brana pod uwagę informacja o kraju, a potem o języku. Następnie ten sam sposób zostaje zastosowany dla domyślnego lokalizatora. Jeśli i te działania nie zakończą się sukcesem, to pod uwagę brany jest jeszcze komplet domyślny. Jeśli i jego załadowanie zakończy się niepowodzeniem, to metoda `getBundle` zgłosi wyjątek `MissingResourceException`.

Metoda `getBundle` próbuje więc kolejno załadować poniższe klasy:

```
nazwaKompletu_bieżącyLokalizatorJęzyk_bieżącyLokalizatorKraj
nazwaKompletu_bieżącyLokalizatorJęzyk
nazwaKompletu_domyślnyLokalizatorJęzyk_domyślnyLokalizatorKraj_
nazwaKompletu_domyślnyLokalizatorJęzyk
nazwaKompletu
```

Jeśli metoda `getBundle` zlokalizuje pomyślnie komplet zasobów, na przykład `nazwaKompletu_de_DE`, to będzie próbowała zlokalizować także zasoby `nazwaKompletu_de` oraz `nazwaKompletu`. Jeśli operacja ta powiedzie się, komplety te zostaną uznane za komplety nadrzędne dla kompletu `nazwaKompletu_de_DE` w hierarchii zasobów. Zostaną one później wykorzystane do odnalezienia zasobu, jeśli jego odnalezienie w bieżącym komplecie nie powiedzie się. Jeśli zasób nie zostanie odnaleziony w komplecie `nazwaKompletu_de_DE`, to będzie poszukiwany w kompletach `nazwaKompletu_de` oraz `nazwaKompletu`.

Usługa taka jest bardzo przydatna, a jej samodzielna implementacja wymagałaby sporo pracy. Mechanizm kompletów zasobów w języku Java pozwala wyszukać najodpowiedniejsze zasoby dla danego lokalizatora. Kolejne wersje językowe programu możemy więc tworzyć, dodając jedynie nowe komplety zasobów.



Omwienie wyszukiwania kompletów zasobów zostało nieco uproszczone. Jeśli lokalizator ma skrypt lub wariant, to proces wyszukiwania jest bardziej skomplikowany. Szczegóły można znaleźć w dokumentacji metody  `ResourceBundle.Control.getCandidateLocales`.



Wszystkich zasobów aplikacji nie musimy umieszczać w pojedynczym komplecie. Możemy utworzyć osobny komplet dla etykiet przycisków, osobny dla komunikatów o błędach i tak dalej.

## 5.7.2. Pliki właściwości

Internacjonalizacja łańcuchów znaków jest zadaniem nieskomplikowanym. Polega zwykle na umieszczeniu wszystkich łańcuchów znaków w pliku właściwości, na przykład o nazwie `MyProgramStrings.properties`. Plik taki jest po prostu plikiem tekstowym, którego każdy wiersz stanowi para klucz-wartość. Przykładowa zawartość pliku będzie wyglądać jak poniżej:

```
computeButton=Rechnen
backgroundColor=black
defaultPageSize=210x297
```

Plikom właściwości nadajemy nazwy zgodnie z wzorcem omówionym w poprzednim podrozdziale:

```
MyProgramStrings.properties
MyProgramStrings_en.properties
MyProgramStrings_de_DE.properties
```

Następnie komplet zasobów ładujemy w poniższy sposób:

```
ResourceBundle bundle =
 ResourceBundle.getBundle("MyProgramStrings", locale);
```

Konkretny łańcuch wyszukujemy za pomocą następującego wywołania:

```
String computeButtonLabel = bundle.getString("computeButton");
```

 Pliki właściwości umożliwiają zapis znaków w kodzie ASCII. Jeśli plik taki powinien zawierać znaki kodu Unicode, to należy je zapisać w postaci \uxxxxx, na przykład zamiast łańcucha "colorName=Grün" należy umieścić w pliku łańcuch:

```
colorName=Gr\u00FCn
```

Możemy wykorzystać w tym celu program native2ascii.

### 5.7.3. Klasa kompletów zasobów

W przypadku zasobów, które nie są łańcuchami znaków, komplety zasobów tworzymy, definiując klasy pochodne klasy ResourceBundle. Nazwy tych klas tworzymy, stosując znaną już nam konwencję, na przykład:

```
MyProgramResources.java
MyProgramResources_en.java
MyProgramResources_de_DE.java
```

Klasę taką ładujemy za pomocą tej samej metody getBundle co w przypadku plików właściwości:

```
ResourceBundle bundle = ResourceBundle.getBundle("MyProgramResources", locale);
```



Podczas wyszukiwania kompletów zasobów klasy są preferowane w stosunku do plików właściwości, gdy dwa komplety zasobów posiadają tę samą nazwę bazową.

Każda klasa kompletu zasobów implementuje tabelę wyszukiwania. Tworząc komplety zasobów, dostarczamy łańcuch klucza dla każdego zasobu. Klucz ten wykorzystujemy później do pobrania zasobu, na przykład:

```
Color backgroundColor
    = (Color)resources.getObject("backgroundColor");
double[] paperSize
    = (double[])resources.getObject("defaultPaperSize");
```

Najprostszy sposób implementacji klas kompletów zasobów polega na rozszerzeniu klasy ListResourceBundle. Klasa ListResourceBundle pozwala nam umieścić wszystkie zasoby w tablicy obiektów i sama je wyszukuje. Musimy jedynie wypełnić poniższy szkielet:

```

public class nazwaKompletu_język_kraj
    extends ListResourceBundle
{
    private static final Object[][] contents =
    {
        { klucz1, wartość1 },
        { klucz2, wartość2 },
        ...
    }
    public Object[][] getContents() { return contents; }
}

```

Na przykład:

```

public class ProgramResources_de
    extends ListResourceBundle
{
    private static final Object[][] contents =
    {
        { "backgroundColor", Color.black },
        { "defaultPageSize", new double[] { 210, 297 } }
    }
    public Object[][] getContents() { return contents; }
}

public class ProgramResources_en_US
    extends ListResourceBundle
{
    private static final Object[][] contents =
    {
        { "backgroundColor", Color.blue },
        { "defaultPageSize", new double[] { 216, 279 } }
    }
    public Object[][] getContents() { return contents; }
}

```



Rozmiary arkuszy papieru podaje się w milimetrach. Wszystkie kraje z wyjątkiem USA i Kanady wykorzystują rozmiary papieru określone standardem ISO 216. Więcej informacji na temat standardu znaleźć można na stronie [www.cl.cam.ac.uk/~mgk25/iso-paper.html](http://www.cl.cam.ac.uk/~mgk25/iso-paper.html). Do chwili obecnej jedynie trzy kraje nie przyjęły standardu metrycznego: Liberia, Myanmar (Burma) i USA (źródło: <http://lamar.colostate.edu/~hillger>).

Inny sposób polega na rozszerzeniu klasy ResourceBundle. Jednak wtedy konieczna staje się implementacja dwóch metod, z których jedna tworzy wyliczenie kluczy, a druga wyszukuje wartość dla podanego klucza:

```

Enumeration<String> getKeys()
Object handleGetObject(String key)

```

Metoda getObject klasy ResourceBundle wywołuje zaimplementowaną przez nas metodę handleGetObject.

#### java.util.ResourceBundle 1.1

- static ResourceBundle getBundle(String basename, Locale loc)

- static ResourceBundle getBundle(String baseName)
 

Ładują klasę kompletu zasobów o podanej nazwie dla określonego lub domyślnego lokalizatora oraz jej klasy nadzędne. Jeśli klasy kompletów zasobów zostały umieszczone w pakuiecie, to należy podać ich pełną nazwę, na przykład "intl.ProgramResources". Klasy kompletów zasobów muszą posiadać dostęp publiczny, aby zapewnić możliwość dostępu do nich metodzie getBundle.
- Object getObject(String name)
 

wyszukuje obiekt w komplecie zasobów lub jego kompletach nadzędnych.
- String getString(String name)
 

wyszukuje obiekt w komplecie zasobów lub jego kompletach nadzędnych, a następnie rzutuje go na typ String.
- String[] getStringArray(String name)
 

wyszukuje obiekt w komplecie zasobów lub jego kompletach nadzędnych, a następnie rzutuje go na typ String[].
- Enumeration<String> getKeys()
 

zwraca obiekt wyliczenia umożliwiający przeglądanie kluczy kompletu zasobów oraz jego kompletów nadzędnych.
- Object handleGetObject(String key)
 

metodę te należy zastąpić własną wersją, jeśli definiujemy swój mechanizm wyszukiwania zasobów.

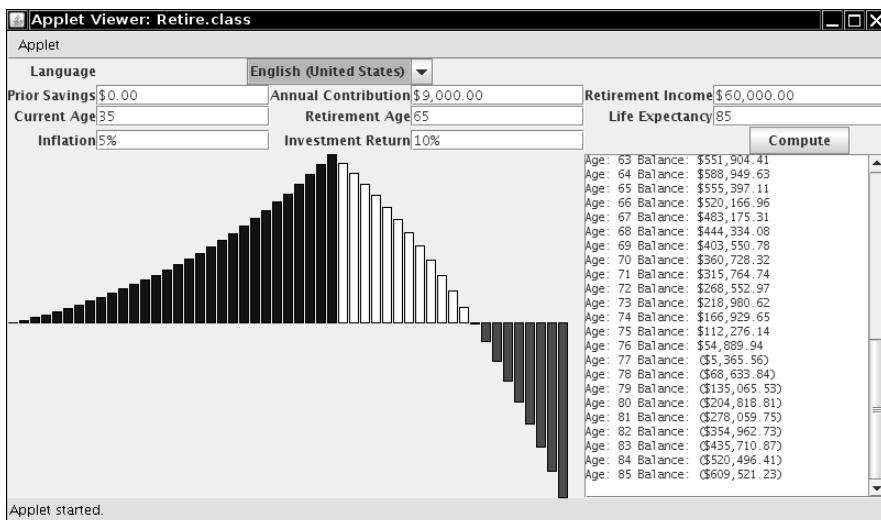
## 5.8. Kompletny przykład

Zdobyta w trakcie lektury tego rozdziału wiedzę wykorzystamy teraz do lokalizacji kalkulatora emerytalnego. Umożliwia on sprawdzenie, czy odkładamy co miesiąc wystarczającą ilość pieniędzy na naszą przyszłą emeryturę. W tym celu musimy podać nasz wiek, ile oszczędzamy miesięcznie itd. (patrz rysunek 5.4).

Wykres i obszar tekstowy prezentują stan naszego konta emerytalnego w każdym roku. Jeśli zacznie on przyjmować wartości ujemne musimy coś z tym zrobić: oszczędzać więcej pieniędzy, opóźnić przejście na emeryturę, umrzeć w młodszym wieku lub odmłodzić się.

Kalkulator emerytalny działać będzie dla trzech lokalizatorów (angielskiego, niemieckiego i chińskiego). Poniżej omówimy kilka zagadnień związanych ze sposobem jego lokalizacji.

- Etykiety, przyciski i komunikaty zostały przetłumaczone na język niemiecki i chiński umieszczone odpowiednio w klasach RetireResources\_de oraz RetireResources\_zh. Oryginalne łańcuchy w języku angielskim zawiera natomiast klasa RetireResources. Teksty w języku chińskim wpisaliśmy najpierw w programie Notepad działającym w chińskiej wersji systemu Windows, po czym poddaliśmy je konwersji do kodu Unicode za pomocą programu native2ascii.



Rysunek 5.4. Kalkulator emerytalny w języku angielskim

- Za każdym razem gdy użytkownik zmienia lokalizator, resetujemy etykiety i reformatujemy zawartość pól tekstowych.
- Pola tekstowe prezentują liczby, kwoty i wielkości procentowe w lokalnych formatach.
- Pole prezentujące wyniki obliczeń korzysta z klasy MessageFormat. Łańcuch formatujący jest przechowywany w komplecie zasobów dla każdego z języków.
- Wraz ze zmianą lokalizatora zmieniamy też kolory słupków wykresu, jedynie po to, aby pokazać, że jest to możliwe.

Listingi 5.5 do 5.8 zawierają kod źródłowy. Listingi 5.9 do 5.11 prezentują pliki właściwości zawierające zlokalizowane łańcuchy znakowe. Rysunki 5.5 i 5.6 pokazują działanie kalkulatora po wybraniu języka niemieckiego i chińskiego. Aby widoczne były znaki alfabetu chińskiego, musimy uruchomić aplet w chińskiej wersji systemu lub zainstalować dodatkowo chińskie czcionki. W przeciwnym razie zamiast chińskich znaków pojawią się ikony informujące o braku znaków.

Listing 5.5. retire/Retire.java

```
package retire;

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.util.*;
import java.text.*;
import javax.swing.*;

/**
 * Kalkulator emerytalny posiadający
 * interfejs użytkownika w języku angielskim,
 * niemieckim i chińskim.
 */

```

```
* @version 1.23 2012-06-07
* @author Cay Horstmann
*/
public class Retire
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new RetireFrame();
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

class RetireFrame extends JFrame
{
    private JTextField savingsField = new JTextField(10);
    private JTextField contribField = new JTextField(10);
    private JTextField incomeField = new JTextField(10);
    private JTextField currentAgeField = new JTextField(4);
    private JTextField retireAgeField = new JTextField(4);
    private JTextField deathAgeField = new JTextField(4);
    private JTextField inflationPercentField = new JTextField(6);
    private JTextField investPercentField = new JTextField(6);
    private JTextArea retireText = new JTextArea(10, 25);
    private RetireComponent retireCanvas = new RetireComponent();
    private JButton computeButton = new JButton();
    private JLabel languageLabel = new JLabel();
    private JLabel savingsLabel = new JLabel();
    private JLabel contribLabel = new JLabel();
    private JLabel incomeLabel = new JLabel();
    private JLabel currentAgeLabel = new JLabel();
    private JLabel retireAgeLabel = new JLabel();
    private JLabel deathAgeLabel = new JLabel();
    private JLabel inflationPercentLabel = new JLabel();
    private JLabel investPercentLabel = new JLabel();
    private RetireInfo info = new RetireInfo();
    private Locale[] locales = { Locale.US, Locale.CHINA, Locale.GERMANY };
    private Locale currentLocale;
    private JComboBox<Locale> localeCombo = new LocaleCombo(locales);
    private ResourceBundle res;
    private ResourceBundle resStrings;
    private NumberFormat currencyFmt;
    private NumberFormat numberFmt;
    private NumberFormat percentFmt;

    public RetireFrame()
    {
        setLayout(new GridBagLayout());
        add(languageLabel, new GBC(0, 0).setAnchor(GBC.EAST));
        add(savingsLabel, new GBC(0, 1).setAnchor(GBC.EAST));
        add(contribLabel, new GBC(2, 1).setAnchor(GBC.EAST));
        add(incomeLabel, new GBC(4, 1).setAnchor(GBC.EAST));
    }
}
```

```

add(currentAgeLabel, new GBC(0, 2).setAnchor(GBC.EAST));
add(retireAgeLabel, new GBC(2, 2).setAnchor(GBC.EAST));
add(deathAgeLabel, new GBC(4, 2).setAnchor(GBC.EAST));
add(inflationPercentLabel, new GBC(0, 3).setAnchor(GBC.EAST));
add(investPercentLabel, new GBC(2, 3).setAnchor(GBC.EAST));
add(localeCombo, new GBC(1, 0, 3, 1));
add(savingsField, new GBC(1, 1).setWeight(100, 0).setFill(GBC.HORIZONTAL));
add(contribField, new GBC(3, 1).setWeight(100, 0).setFill(GBC.HORIZONTAL));
add(incomeField, new GBC(5, 1).setWeight(100, 0).setFill(GBC.HORIZONTAL));
add(currentAgeField, new GBC(1, 2).setWeight(100, 0).setFill(GBC.HORIZONTAL));
add(retireAgeField, new GBC(3, 2).setWeight(100, 0).setFill(GBC.HORIZONTAL));
add(deathAgeField, new GBC(5, 2).setWeight(100, 0).setFill(GBC.HORIZONTAL));
add(inflationPercentField, new GBC(1, 3).setWeight(100, 0).setFill(GBC.HORIZONTAL));
add(investPercentField, new GBC(3, 3).setWeight(100, 0).setFill(GBC.HORIZONTAL));
add(retireCanvas, new GBC(0, 4, 4, 1).setWeight(100, 100).setFill(GBC.BOTH));
add(new JScrollPane(retireText), new GBC(4, 4, 2, 1).setWeight(0,
➥100).setFill(GBC.BOTH));

computeButton.setName("computeButton");
computeButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        getInfo();
        updateData();
        updateGraph();
    }
});
add(computeButton, new GBC(5, 3));

retireText.setEditable(false);
retireText.setFont(new Font("Monospaced", Font.PLAIN, 10));

info.setSavings(0);
info.setContrib(9000);
info.setIncome(60000);
info.setCurrentAge(35);
info.setRetireAge(65);
info.setDeathAge(85);
info.setInvestPercent(0.1);
info.setInflationPercent(0.05);

int localeIndex = 0; // lokalizator USA jest domyślny
for (int i = 0; i < locales.length; i++)
    // jeśli bieżący lokalizator jest jednym z możliwych do wyboru, to zostaje wybrany
    if (getLocale().equals(locales[i])) localeIndex = i;
setCurrentLocale(locales[localeIndex]);

localeCombo.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        setCurrentLocale((Locale) localeCombo.getSelectedItem());
        validate();
    }
});
pack();
}

```

```
/***
 * Wybiera bieżący lokalizator.
 * @param locale lokalizator
 */
public void setCurrentLocale(Locale locale)
{
    currentLocale = locale;
    localeCombo.setSelectedItem(currentLocale);
    localeCombo.setLocale(currentLocale);

    res = ResourceBundle.getBundle("retire.RetireResources", currentLocale);
    resStrings = ResourceBundle.getBundle("retire.RetireStrings", currentLocale);
    currencyFmt = NumberFormat.getCurrencyInstance(currentLocale);
    numberFmt = NumberFormat.getNumberInstance(currentLocale);
    percentFmt = NumberFormat.getPercentInstance(currentLocale);

    updateDisplay();
    updateInfo();
    updateData();
    updateGraph();
}

/***
 * Aktualizuje wszystkie wyświetlane etykiety.
 */
public void updateDisplay()
{
    languageLabel.setText(resStrings.getString("language"));
    savingsLabel.setText(resStrings.getString("savings"));
    contribLabel.setText(resStrings.getString("contrib"));
    incomeLabel.setText(resStrings.getString("income"));
    currentAgeLabel.setText(resStrings.getString("currentAge"));
    retireAgeLabel.setText(resStrings.getString("retireAge"));
    deathAgeLabel.setText(resStrings.getString("deathAge"));
    inflationPercentLabel.setText(resStrings.getString("inflationPercent"));
    investPercentLabel.setText(resStrings.getString("investPercent"));
    computeButton.setText(resStrings.getString("computeButton"));
}

/***
 * Aktualizuje zawartość pól tekstowych.
 */
public void updateInfo()
{
    savingsField.setText(currencyFmt.format(info.getSavings()));
    contribField.setText(currencyFmt.format(info.getContrib()));
    incomeField.setText(currencyFmt.format(info.getIncome()));
    currentAgeField.setText(numberFmt.format(info.getCurrentAge()));
    retireAgeField.setText(numberFmt.format(info.getRetireAge()));
    deathAgeField.setText(numberFmt.format(info.getDeathAge()));
    investPercentField.setText(percentFmt.format(info.getInvestPercent()));
    inflationPercentField.setText(percentFmt.format(info.getInflationPercent()));
}

/***
 * Aktualizuje zawartość obszaru tekstowego.
 */
public void updateData()
```

```

{
    retireText.setText("");
    MessageFormat retireMsg = new MessageFormat("");
    retireMsg.setLocale(currentLocale);
    retireMsg.applyPattern(resStrings.getString("retire"));

    for (int i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
    {
        Object[] args = { i, info.getBalance(i) };
        retireText.append(retireMsg.format(args) + "\n");
    }
}

<**
 * Aktualizuje wykres.
 */
public void updateGraph()
{
    retireCanvas.setColorPre((Color) res.getObject("colorPre"));
    retireCanvas.setColorGain((Color) res.getObject("colorGain"));
    retireCanvas.setColorLoss((Color) res.getObject("colorLoss"));
    retireCanvas.setInfo(info);
    repaint();
}

<**
 * Odczytuje dane wprowadzone przez użytkownika.
 */
public void getInfo()
{
    try
    {
        info.setSavings(currencyFmt.parse(savingsField.getText()).doubleValue());
        info.setContrib(currencyFmt.parse(contribField.getText()).doubleValue());
        info.setIncome(currencyFmt.parse(incomeField.getText()).doubleValue());
        info.setCurrentAge(numberFmt.parse(currentAgeField.getText()).intValue());
        info.setRetireAge(numberFmt.parse(retireAgeField.getText()).intValue());
        info.setDeathAge(numberFmt.parse(deathAgeField.getText()).intValue());

        info.setInvestPercent(percentFmt.parse(investPercentField.getText()).doubleValue());
        info.setInflationPercent(percentFmt.parse(inflationPercentField.getText()).doubleValue());
    }
    catch (ParseException ex)
    {
        ex.printStackTrace();
    }
}

<**
 * Klasa zawierająca informacje potrzebne do obliczenia emerytury.
 */
class RetireInfo
{
    private double savings;
    private double contrib;
    private double income;
}

```

```
private int currentAge;
private int retireAge;
private int deathAge;
private double inflationPercent;
private double investPercent;
private int age;
private double balance;

/**
 * Zwraca stan konta dla podanego roku.
 * @param year rok, dla którego obliczany jest stan konta
 * @return dostepna lub wymagana kwota w danym roku
 */
public double getBalance(int year)
{
    if (year < currentAge) return 0;
    else if (year == currentAge)
    {
        age = year;
        balance = savings;
        return balance;
    }
    else if (year == age) return balance;
    if (year != age + 1) getBalance(year - 1);
    age = year;
    if (age < retireAge) balance += contrib;
    else balance -= income;
    balance = balance * (1 + (investPercent - inflationPercent));
    return balance;
}

/**
 * Zwraca wysokość oszczędności.
 * @return kwota oszczędności
 */
public double getSavings()
{
    return savings;
}

/**
 * Określa wysokość oszczędności.
 * @param newValue kwota oszczędności
 */
public void setSavings(double newValue)
{
    savings = newValue;
}

/**
 * Zwraca roczny przychód na koncie emerytalnym.
 * @return kwota przychodu
 */
public double getContrib()
{
    return contrib;
}
```

```
/***
 * Określa roczny przychód na koncie emerytalnym.
 * @param newValue kwota przychodu
 */
public void setContrib(double newValue)
{
    contrib = newValue;
}

/***
 * Zwraca roczny przychód.
 * @return wysokość przychodu
 */
public double getIncome()
{
    return income;
}

/***
 * Określa roczny przychód.
 * @param newValue wysokość przychodu
 */
public void setIncome(double newValue)
{
    income = newValue;
}

/***
 * Zwraca aktualny wiek.
 * @return wiek
 */
public int getCurrentAge()
{
    return currentAge;
}

/***
 * Określa aktualny wiek.
 * @param newValue wiek
 */
public void setCurrentAge(int newValue)
{
    currentAge = newValue;
}

/***
 * Zwraca oczekiwany wiek przejścia na emeryturę.
 * @return wiek
 */
public int getRetireAge()
{
    return retireAge;
}

/***
 * Określa oczekiwany wiek przejścia na emeryturę.
 * @param newValue wiek
 */
public void setRetireAge(int newValue)
```

```
{  
    retireAge = newValue;  
}  
  
/**  
 * Zwraca spodziewany wiek w momencie śmierci.  
 * @return wiek  
 */  
public int getDeathAge()  
{  
    return deathAge;  
}  
  
/**  
 * Określa spodziewany wiek w momencie śmierci.  
 * @param newValue wiek  
 */  
public void setDeathAge(int newValue)  
{  
    deathAge = newValue;  
}  
  
/**  
 * Zwraca spodziewaną wysokość inflacji.  
 * @return wysokość inflacji w procentach  
 */  
public double getInflationPercent()  
{  
    return inflationPercent;  
}  
  
/**  
 * Określa spodziewaną wysokość inflacji.  
 * @param newValue wysokość inflacji w procentach  
 */  
public void setInflationPercent(double newValue)  
{  
    inflationPercent = newValue;  
}  
  
/**  
 * Zwraca oczekiwany zwrot inwestycji.  
 * @return zwrot inwestycji w procentach  
 */  
public double getInvestPercent()  
{  
    return investPercent;  
}  
  
/**  
 * Określa oczekiwany zwrot inwestycji.  
 * @param newValue zwrot inwestycji w procentach  
 */  
public void setInvestPercent(double newValue)  
{  
    investPercent = newValue;  
}
```

```

/**
 * Komponent rysujący wykres inwestycji.
 */
class RetireComponent extends JComponent
{
    private static final int DEFAULT_WIDTH = 800;
    private static final int DEFAULT_HEIGHT = 600;
    private static final int PANEL_WIDTH = 400;
    private static final int PANEL_HEIGHT = 200;

    private RetireInfo info = null;
    private Color colorPre;
    private Color colorGain;
    private Color colorLoss;

    public RetireComponent()
    {
        setSize(PANEL_WIDTH, PANEL_HEIGHT);
    }

    /**
     * Określa informację prezentowaną na wykresie.
     * @param newInfo informacja o emeryturze
     */
    public void setInfo(RetireInfo newInfo)
    {
        info = newInfo;
        repaint();
    }

    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;
        if (info == null) return;

        double minValue = 0;
        double maxValue = 0;
        int i;
        for (i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
        {
            double v = info.getBalance(i);
            if (minValue > v) minValue = v;
            if (maxValue < v) maxValue = v;
        }
        if (maxValue == minValue) return;

        int barWidth = getWidth() / (info.getDeathAge() - info.getCurrentAge() + 1);
        double scale = getHeight() / (maxValue - minValue);

        for (i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
        {
            int xl = (i - info.getCurrentAge()) * barWidth + 1;
            int yl;
            double v = info.getBalance(i);
            int height;
            int yOrigin = (int) (maxValue * scale);

            if (v >= 0)

```

```
{  
    y1 = (int) ((maxValue - v) * scale);  
    height = yOrigin - y1;  
}  
else  
{  
    y1 = yOrigin;  
    height = (int) (-v * scale);  
}  
  
if (i < info.getRetireAge()) g2.setPaint(colorPre);  
else if (v >= 0) g2.setPaint(colorGain);  
else g2.setPaint(colorLoss);  
Rectangle2D bar = new Rectangle2D.Double(x1, y1, barWidth - 2, height);  
g2.fill(bar);  
g2.setPaint(Color.black);  
g2.draw(bar);  
}  
}  
  
/**  
 * Określa kolor słupków wykresu przed przejściem na emeryturę.  
 * @param color wymagany kolor  
 */  
public void setColorPre(Color color)  
{  
    colorPre = color;  
    repaint();  
}  
  
/**  
 * Określa kolor słupków wykresu po przejściu na emeryturę.  
 * gdy stan konta jest dodatni.  
 * @param color wymagany kolor  
 */  
public void setColorGain(Color color)  
{  
    colorGain = color;  
    repaint();  
}  
  
/**  
 * Określa kolor słupków wykresu po przejściu na emeryturę.  
 * gdy stan konta jest ujemny.  
 * @param color wymagany kolor  
 */  
public void setColorLoss(Color color)  
{  
    colorLoss = color;  
    repaint();  
}  
  
public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH,  
DEFAULT_HEIGHT); }  
}
```

---

**Listing 5.6.** *retire/RetireResources.java*


---

```
package retire;

import java.awt.*;

/**
 * Zasoby dla języka niemieckiego,
 * które nie sąłańcuchami znaków.
 * @author Cay Horstmann
 */
public class RetireResources_de extends java.util.ListResourceBundle
{
    private static final Object[][] contents = {
        //BEGIN LOCALIZE
        { "colorPre", Color.yellow }, { "colorGain", Color.black }, { "colorLoss",
        Color.red }
        //END LOCALIZE
    };

    public Object[][] getContents()
    {
        return contents;
    }
}
```

---

**Listing 5.7.** *retire/RetireResources\_de.java*


---

```
package retire;

import java.awt.*;

/**
 * Zasoby dla języka niemieckiego,
 * które nie sąłańcuchami znaków.
 * @version 1.21 2001-08-27
 * @author Cay Horstmann
 */
public class RetireResources_de extends java.util.ListResourceBundle
{
    private static final Object[][] contents = {
        //BEGIN LOCALIZE
        { "colorPre", Color.yellow }, { "colorGain", Color.black }, { "colorLoss",
        ↳Color.red }
        //END LOCALIZE
    };

    public Object[][] getContents()
    {
        return contents;
    }
}
```

---

**Listing 5.8.** *retire/RetireResources\_zh.java*


---

```
package retire;

import java.awt;
```

```
/*
 * Zasoby dla języka chińskiego,
 * które nie sąłańcuchami znaków.
 * @version 1.21 2001-08-27
 * @author Cay Horstmann
 */
public class RetireResources_zh extends java.util.ListResourceBundle
{
    private static final Object[][] contents = {
        //BEGIN LOCALIZE
        { "colorPre", Color.red }, { "colorGain", Color.blue }, { "colorLoss",
            ↴Color.yellow }
        //END LOCALIZE
    };

    public Object[][] getContents()
    {
        return contents;
    }
}
```

---

**Listing 5.9.** *retire/RetireStrings.properties*

```
language=Language
computeButton=Compute
savings=Prior Savings
contrib=Annual Contribution
income=Retirement Income
currentAge=Current Age
retireAge=Retirement Age
deathAge=Life Expectancy
inflationPercent=Inflation
investPercent=Investment Return
retire=Age: {0,number} Balance: {1,number,currency}
```

---

**Listing 5.10.** *retire/RetireStrings\_de.properties*

```
language=Sprache
computeButton=Rechnen
savings=Vorherige Ersparnisse
contrib=J\u00fcl\u00e4hrliche Einzahlung
income=Einkommen nach Ruhestand
currentAge=Jetziges Alter
retireAge=Ruhestandsalter
deathAge=Lebenserwartung
inflationPercent=Inflation
investPercent=Investitionsgewinn
retire=Alter: {0,number} Guthaben: {1,number,currency}
```

---

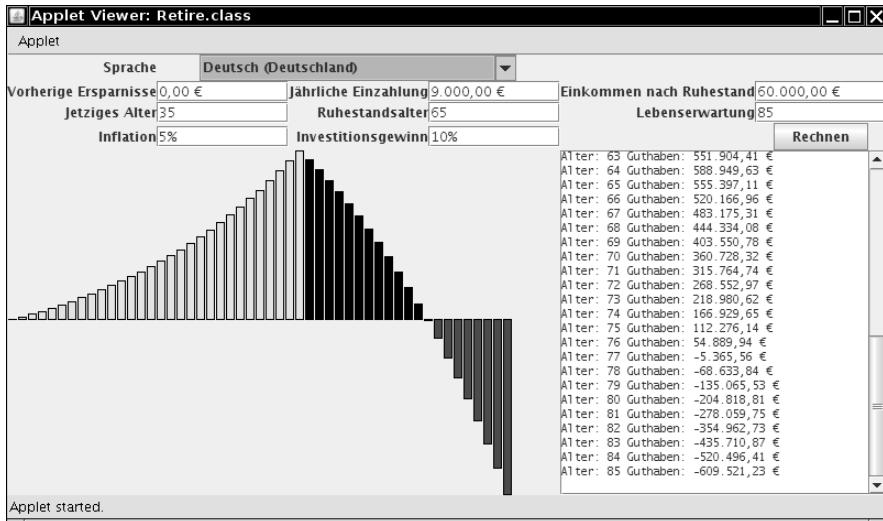
**Listing 5.11.** *retire/RetireStrings\_zh.properties*

```
language=\u8bed\u8a00
computeButton=\u8ba1\u7b97
savings=\u65e2\u5b58
contrib=\u6bcf\u5e74\u5b58\u91d1
income=\u9000\u4f11\u6536\u5165
currentAge=\u73b0\u5cad
```

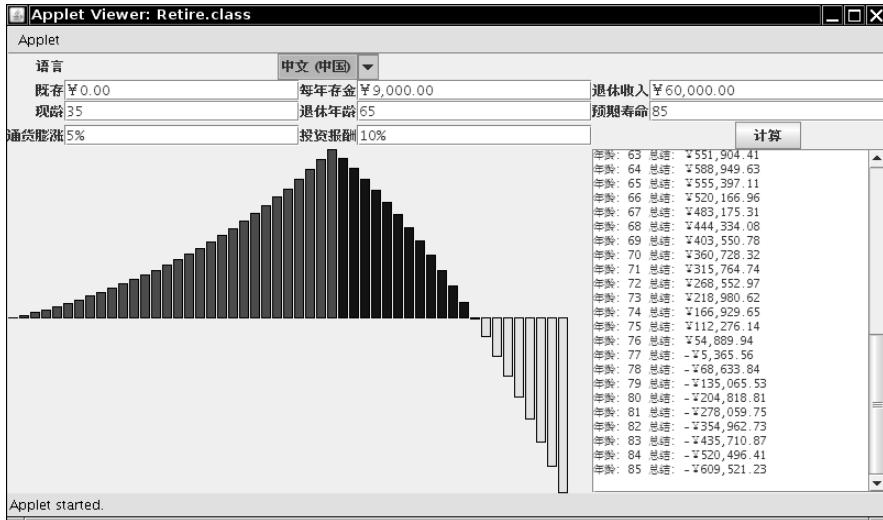
```

retireAge=\u9000\u4f11\u5e74\u9f84
deathAge=\u9884\u671f\u5bff\u547d
inflationPercent=\u901a\u8d27\u81a8\u6da8
investPercent=\u6295\u8d44\u62a5\u916c
retire=\u5e74\u9f84: {0,number} \u603b\ued3: {1,number,currency}

```



Rysunek 5.5. Kalkulator emerytalny w języku niemieckim



Rysunek 5.6. Kalkulator emerytalny w języku chińskim

W tym rozdziale przedstawiliśmy możliwości internacjonalizacji aplikacji na platformie Java. Zastosowanie kompletów zasobów pozwala nam tworzyć różne wersje językowe aplikacji, a obiektem formatującym przetwarzać teksty w różnych językach.

W kolejnym rozdziale zajmiemy się zaawansowanym programowaniem aplikacji Swing.



# 6

## Zaawansowane możliwości pakietu Swing

W tym rozdziale:

- Listy.
- Tabele.
- Drzewa.
- Komponenty formatujące tekst.
- Wskaźniki postępu.
- Organizatory komponentów i dekoratory.

W rozdziale tym kontynuować będziemy rozpoczęte w książce *Java. Podstawy* omówienie pakietu Swing wykorzystywanego do tworzenia interfejsu użytkownika. Pakiet Swing posiada bardzo rozbudowane możliwości, a w książce *Java. Podstawy* zdołaliśmy przedstawić jedynie komponenty najczęściej używane. Większość niniejszego rozdziału poświęcimy złożonym komponentom, takim jak listy, drzewa i tabele. Następnie zajmiemy się komponentami tekstowymi i pokażemy sposoby kontroli ich zawartości oraz przedstawimy komponent JSpinner. Komponenty umożliwiające formatowanie tekstu, na przykład HTML, posiadają jeszcze bardziej złożoną implementację. Omówimy sposób ich praktycznego wykorzystania. Następnie poświęcimy uwagę sposobom przedstawiania postępu wykonania czasochłonnych operacji przez osobny wątek. Rozdział zakończymy przedstawieniem organizatorów komponentów, takich jak panele z zakładkami i panele z wewnętrznymi ramkami.

### 6.1. Listy

Prezentując użytkownikowi zbiór elementów do wyboru, możemy skorzystać z różnych komponentów. Jeśli zbiór ten zawierać będzie wiele elementów, to ich przedstawienie za pomocą pól wyboru zajmie zdecydowanie za dużo miejsca w oknie programu. Skorzystamy

wtedy zwykle z listy bądź listy rozwijalnej. Listy rozwijalne są stosunkowo prostymi komponentami i dlatego omówiliśmy je już w książce *Java. Podstawy*. Natomiast listy reprezentowane przez komponent `JList` posiadają dużo większe możliwości, a sposób ich użycia przypomina korzystanie z innych złożonych komponentów, takich jak drzewa czy tabele. Dlatego właśnie od list rozpoczęliśmy omówienie złożonych komponentów pakietu Swing.

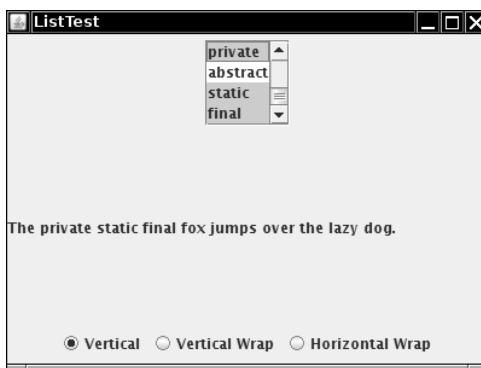
Listy często składają się z łańcuchów znaków, ale w praktyce zawierać mogą dowolne obiekty i kontrolować przy tym sposób ich prezentacji. Wewnętrzna architektura listy, która umożliwia taki stopień ogólności, prezentuje się dość elegancko. Niestety projektanci z firmy Sun postanowili pochwalić się elegancją tworzonych rozwiązań, zamiast ukryć ją przed programistami korzystającymi z komponentu. Skutkiem tego posługiwanie się listami w najprostszych przypadkach jest trochę skomplikowane, ponieważ programista musi manipulować mechanizmami, które umożliwiają wykorzystanie list w bardziej złożonych przypadkach. Omówienie rozpoczęliśmy od przedstawienia najprostszego i najczęściej spotykanego zastosowania tego komponentu — listy, której elementami są łańcuchy znaków. Później przejdziemy do bardziej złożonych przykładów ilustrujących uniwersalność komponentu.

## 6.1.1. Komponent `JList`

Komponent `JList` prezentuje użytkownikowi zbiór elementów do wyboru. Rysunek 6.1 pokazuje najprostszy przykład takiego komponentu. Użytkownik może z niej wybrać atrybuty opisujące lisa, takie jak „quick”, „brown”, „hungry” i „wild” oraz, z braku innych pomysłów, „static”, „private” i „final”.

**Rysunek 6.1.**

Komponent klasy `JList`



Począwszy od Java SE 7, `JList` jest typem generycznym, którego parametrem jest typ wartości wybieranych przez użytkownika z listy. W tym przykładzie używamy zatem typu `JList<String>`.

Tworzenie listy rozpoczynamy od skonstruowania tablicy łańcuchów znaków, którą następnie przekazujemy konstruktorowi klasy `JList`:

```
String[] words = { "quick", "brown", "hungry", "wild", . . . };
JList<String> wordList = new JList<>(words);
```

Komponenty `JList` nie przewijają automatycznie swojej zawartości. W tym celu musimy umieścić listę w panelu przewijalnym:

```
JScrollPane scrollPane = new JScrollPane(wordList);
```

Panel ten, a nie listę, umieszczamy następnie na docelowym panelu okna.

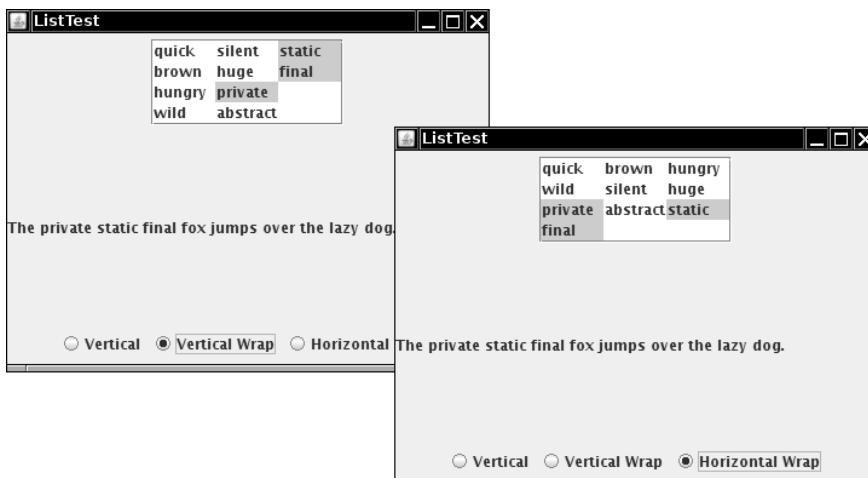
Rozdzielenie prezentacji listy od mechanizmu przewijania jest z pewnością rozwiązaniem eleganckim, ale mało praktycznym. Właściwie prawie wszystkie listy wymagają przewijania. Zmuszanie programistów, by za każdym razem, gdy tworzą najprostszą listę, podziwiali działanie tego mechanizmu, jest okrutne.

Domyślnie lista mieści osiem elementów widocznych jednocześnie. Możemy to zmienić, korzystając z metody setVisibleRowCount:

```
wordList.setVisibleRowCount(4); // pokazuje jednocześnie 4 elementy
```

Możemy również wybrać jeden z trzech sposobów prezentacji elementów listy:

- `JList.VERTICAL` (domyślny) — elementy zostają ułożone w pionie czyli jeden nad drugim,
- `JList.VERTICAL_WRAP` — jeśli lista zawiera więcej elementów niż liczba jednocześnie widocznych elementów, to tworzona jest nowa kolumna listy (patrz rysunek 6.2),



**Rysunek 6.2.** Listy o pionowym i poziomym rozmieszczeniu elementów

- `JList.HORIZONTAL_WRAP` — również tworzy nowe kolumny listy, ale elementy rozmieszczane są wierszami. Różnicę pomiędzy tym a poprzednim sposobem prezentacji elementów listy możemy zaobserwować na rysunku 6.2, przyglądając się położeniu słów „quick”, „brown” i „hungry”.

Domyślnie użytkownik może wybierać wiele elementów listy. Wymaga to od niego zaawansowanego posługiwania się myszą: wybierając kolejne elementy, musi jednocześnie wcisnąć klawisz `Ctrl`. Aby wytypować pewien ciągły zakres elementów, użytkownik powinien zaznaczyć pierwszy z nich a następnie, wciskając klawisz `Shift`, wybrać ostatni z elementów.

Możliwość wyboru elementów przez użytkownika możemy ograniczyć, korzystając z metody `setSelectionMode`:

```

wordList.setSelectionMode
    (ListSelectionModel.SINGLE_SELECTION);
// możliwość wyboru pojedynczego elementu
wordList.setSelectionMode
    (ListSelectionModel.SINGLE_INTERVAL_SELECTION);
// możliwość wyboru pojedynczego elementu lub pojedynczego zakresu elementów

```

Z lektury książki *Java. Podstawy* przypominamy sobie z pewnością, że podstawowe elementy interfejsu użytkownika generują zdarzenia akcji w momencie ich aktywacji przez użytkownika. Listy wykorzystują jednak inny mechanizm powiadomień. Zamiast nasłuchiwać zdarzeń akcji, w przypadku list nasłuchiwać będziemy zdarzeń wyboru na liście. W tym celu musimy dodać do komponentu listy obiekt nasłuchujący wyboru i zaimplementować następującą metodę obiektu nasłuchującego:

```
public void valueChanged(ListSelectionEvent evt)
```

Podczas dokonywania wyboru na liście generuje się sporo zdarzeń. Założymy na przykład, że użytkownik przechodzi do kolejnego elementu listy. Gdy naciska klawisz myszy, generowane jest zdarzenie zmiany wyboru na liście. Zdarzenie to jest przejściowe i wywołanie metody

```
event.isAdjusting()
```

zwraca wartość true, gdy wybór nie jest ostateczny. Gdy użytkownik puszczas klawisz myszy, generowane jest kolejne zdarzenie, dla którego wywołanie metody `isAdjusting` zwróci tym razem wartość false. Jeśli nie jesteśmy zainteresowani przejściowymi zdarzeniami na liście, to wystarczy poczekać jedynie na zdarzenie, dla którego metoda `isAdjusting` zwróci właśnie wartość false. W przeciwnym razie musimy obsługiwać wszystkie zdarzenia.

Zwykle po zawiadomieniu o zdarzeniu będziemy chcieli się dowiedzieć, które elementy zostały wybrane. Jeśli lista działa w trybie wyboru pojedynczego elementu, metoda `getSelectedValues` zwraca wartość w postaci elementu listy. W przeciwnym razie wywołujemy metodę `getSelectedValuesList`, która zwraca listę zawierającą wszystkie wybrane elementy. Listę tę możemy przetwarzać w typowy sposób:

```
for(String value: getSelectedValuesList)
    // przetwarzanie elementu value;
```



Listy nie obsługują dwukrotnych kliknięć myszą. Projektanci pakietu Swing założyli, że na liście dokonuje się jedynie wyboru, a następnie używa się komponentu przycisku, aby wykonać jakąś akcję. Niektóre interfejsy użytkownika posiadają możliwość dwukrotnego kliknięcia elementu listy w celu dokonania jego wyboru i wykonania na nim pewnej akcji. Jeśli chcemy zaimplementować takie zachowanie listy, to musimy dodać do niej obiekt nasłuchujący zdarzeń myszy i obsługiwać je w następujący sposób:

```

public void mouseClicked(MouseEvent evt)
{
    if (evt.getClickCount()== 2)
    {
        JList source = (JList)evt.getSource();
        Object[] selection = source.getSelectedValues();
        doAction(selection);
    }
}

```

Listing 6.1 zawiera tekst źródłowy programu demonstrującego wykorzystanie listy wypełnionej łańcuchami znaków. Zwróćmy uwagę na sposób, w jaki metoda valueChanged tworzy łańcuch komunikatu z wybranych elementów listy.

#### **Listing 6.1. list/ListFrame.java**

```
package list;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Ramka zawierająca listę słów i etykietę pokazującą zdanie złożone
 * z wybranych słów. Przytrzymując klawisz Ctrl, wybrać można wiele słów,
 * a klawisz Shift pozwala na wybór całego zakresu słów.
 */
class ListFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 300;

    private JPanel listPanel;
    private JList<String> wordList;
    private JLabel label;
    private JPanel buttonPanel;
    private ButtonGroup group;
    private String prefix = "The ";
    private String suffix = "fox jumps over the lazy dog.';

    public ListFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        String[] words = { "quick", "brown", "hungry", "wild", "silent", "huge", "private",
                           "abstract", "static", "final" };

        wordList = new JList<>(words);
        wordList.setVisibleRowCount(4);
        JScrollPane scrollPane = new JScrollPane(wordList);

        listPanel = new JPanel();
        listPanel.add(scrollPane);
        wordList.addListSelectionListener(new ListSelectionListener()
        {
            public void valueChanged(ListSelectionEvent event)
            {
                StringBuilder text = new StringBuilder(prefix);
                for (String value : wordList.getSelectedValuesList())
                {
                    text.append(value);
                    text.append(" ");
                }
                text.append(suffix);

                label.setText(text.toString());
            }
        });
    }
}
```

```

}};

buttonPanel = new JPanel();
group = new ButtonGroup();
makeButton("Vertical", JList.VERTICAL);
makeButton("Vertical Wrap", JList.VERTICAL_WRAP);
makeButton("Horizontal Wrap", JList.HORIZONTAL_WRAP);

add(listPanel, BorderLayout.NORTH);
label = new JLabel(prefix + suffix);
add(label, BorderLayout.CENTER);
add(buttonPanel, BorderLayout.SOUTH);
}

/**
 * Tworzy przycisk wyboru układu listy.
 * @param label etykieta przycisku
 * @param orientation układ listy
 */
private void makeButton(String label, final int orientation)
{
    JRadioButton button = new JRadioButton(label);
    buttonPanel.add(button);
    if (group.getButtonCount() == 0) button.setSelected(true);
    group.add(button);
    button.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            wordList.setLayoutOrientation(orientation);
            listPanel.revalidate();
        }
    });
}
}
}

```

### API javax.swing.JList<E> 1.2

- **JList(E[] items)**  
tworzy listę wyświetlającą podane elementy.
- **int getVisibleRowCount()**
- **void setVisibleRowCount(int c)**  
zwraca lub konfiguruje liczbę elementów widocznych jednocześnie na liście (bez przewijania).
- **int getLayoutOrientation() 1.4**
- **void setLayoutOrientation(int orientation) 1.4**  
zwraca lub konfiguruje sposób rozmieszczenia elementów.  
*Parametry:* mode jedna z wartości VERTICAL, VERTICAL\_WRAP,  
HORIZONTAL\_WRAP.

- int getSelectionMode()
- void setSelectionMode(int mode)

zwraca lub konfiguruje możliwość wyboru pojedynczego lub wielu elementów.

*Parametry:* mode jedna z wartości SINGLE\_SELECTION, SINGLE\_INTERVAL\_SELECTION, MULTIPLE\_INTERVAL\_SELECTION.

- void addListSelectionListener(ListSelectionListener listener)
- dodaje do listy obiekt nasłuchujący zdarzeń wyboru na liście.
- List<E> getSelectedValues() **7**
- zwraca elementy wybrane lub pustą listę, jeśli żaden element nie został wybrany.
- E getSelectedValue()
- zwraca pierwszy element wybrany na liście lub wartość null, jeśli żaden element nie został wybrany.

#### API javax.swing.event.ListSelectionListener 1.2

- void valueChanged(ListSelectionEvent e)
- metoda wywoływana za każdym razem, gdy wybór na liście uległ zmianie.

## 6.1.2. Modele list

W poprzednim podrozdziale pokazaliśmy najczęściej spotykany sposób wykorzystania list polegający na:

1. utworzeniu niezmennego zbioru elementów listy (łańcuchów znaków),
2. umożliwieniu przewijania listy,
3. obsłudze zdarzeń wyboru elementów listy.

W dalszej części przedstawimy bardziej skomplikowane sposoby wykorzystania list, czyli:

- listy o bardzo dużej liczbie elementów,
- listy o zmiennej zawartości,
- listy zawierające elementy inne niż łańcuchy znaków.

W naszym pierwszym przykładzie utworzyliśmy listę zawierającą określony zbiór łańcuchów znaków. Często jednak zachodzi potrzeba dodawania nowych elementów listy bądź usuwania elementów już umieszczonych na liście. Zaskakujący może wydać się fakt, że klasa `JList` nie zawiera metod umożliwiających takie operacje na liście. Aby zrozumieć tego przyczynę, musimy zapoznać się bliżej z wewnętrzna architekturą komponentu listy. Podobnie jak w przypadku komponentów tekstowych także i lista jest przykładem zastosowania wzorca model-widok-nadzorca w celu oddzielenia wizualizacji listy (czyli kolumny elementów wyświetlonych w pewien sposób) od danych (kolekcji obiektów).

Klasa `JList` odpowiedzialna jest jedynie za wizualizację danych i niewiele wie na temat ich reprezentacji. Potrafi jedynie pobrać dane, korzystając z obiektu implementującego interfejs `ListModel`:

```
public interface ListModel<E>
{
    int getSize();
    E getElementAt(int i);
    void addListDataListener(ListDataListener l);
    void removeListDataListener(ListDataListener l);
}
```

Wykorzystując ten interfejs, komponent klasy `JList` może określić liczbę elementów i pobrać każdy z nich. Może także dodać się jako `ListDataListener`. Dzięki temu będzie powiadamiany o każdej zmianie w kolekcji elementów i będzie mógł aktualizować reprezentację list na ekranie.

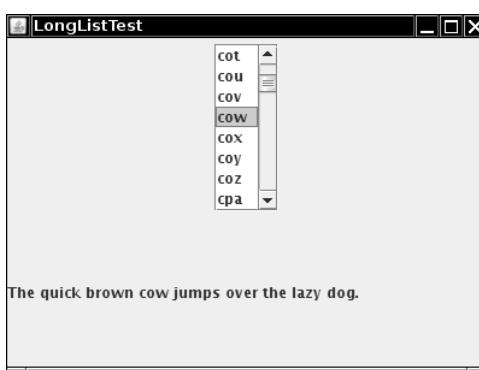
Jaki jest cel takiego rozwiązania? Dlaczego komponent `JList` nie przechowuje po prostu swoich elementów jako tablicy obiektów?

Zwrócić uwagę, że interfejs nie określa sposobu przechowywania obiektów. W szczególności nie wymaga on nawet, by obiekty były przechowywane! Metoda `getElementAt` może wyznaczać wartość elementu od nowa, za każdym razem gdy jest wywoływana. Może okazać się to przydatne, jeśli lista prezentować ma bardzo dużą kolekcję danych, których nie chcemy przechowywać.

A oto najprostszy przykład: lista umożliwiać będzie użytkownikowi wybór spośród wszystkich możliwych kombinacji trzech liter (patrz rysunek 6.3).

**Rysunek 6.3.**

Wybór z listy zawierającej dużą liczbę elementów



Istnieje  $26 \times 26 \times 26 = 17\,576$  takich kombinacji. Zamiast przechowywać je wszystkie, program będzie tworzył je podczas przewijania listy przez użytkownika.

Implementacja programu okazuje się bardzo prosta. Zadanie dodania i usuwania odpowiednich obiektów nasłuchujących wykoną za nas klasa `AbstractListModel`, którą rozszerzymy. Naszym zadaniem będzie jedynie dostarczenie implementacji metod `getSize` i `getElementAt`:

```
class WordListModel extends AbstractListModel<String>
{
    public WordListModel(int n) { length = n; }
    public int getSize() { return (int) Math.pow(26, length); }
```

```

public Object getElementAt(int n)
{
    // wyznacza n-ty łańcuch
    . . .
}
. . .
}

```

Wyznaczenie *n*-tego łańcucha jest trochę skomplikowane — szczegółowo znajdziemy w tekście programu umieszczonym w listingu 6.3.

#### **Listing 6.2.** LongList/LongListFrame.java

```

package longList;

import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Ramka zawierająca długą listę słów i etykiety pokazującą zdanie
 * złożone z wybranych słów.
 */
public class LongListFrame extends JFrame
{
    private JList<String> wordList;
    private JLabel label;
    private String prefix = "The quick brown ";
    private String suffix = " jumps over the lazy dog. ";

    public LongListFrame()
    {
        wordList = new JList<String>(new WordListModel(3));
        wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        wordList.setPrototypeCellValue("www");
        JScrollPane scrollPane = new JScrollPane(wordList);

        JPanel p = new JPanel();
        p.add(scrollPane);
        wordList.addListSelectionListener(new ListSelectionListener()
        {
            public void valueChanged(ListSelectionEvent evt)
            {
                setSubject(wordList.getSelectedValue());
            }
        });
        Container contentPane = getContentPane();
        contentPane.add(p, BorderLayout.NORTH);
        label = new JLabel(prefix + suffix);
        contentPane.add(label, BorderLayout.CENTER);
        setSubject("fox");
        pack();
    }

    /**
     * Określa podmiot zdania pokazywanego za pomocą etykiety.
     * @param word nowy podmiot zdania
     */

```

```
public void setSubject(String word)
{
    String text = new StringBuilder(prefix);
    text.append(word);
    text.append(suffix);
    label.setText(text.toString());
}
```

---

**Listing 6.3.** longList/WordListModel.java

```
package longList;

import javax.swing.*;

/**
 * Model listy dynamicznie generujący kombinacje n liter.
 */
public class WordListModel extends AbstractListModel<String>
{
    private int length;
    public static final char FIRST = 'a';
    public static final char LAST = 'z';

    /**
     * Tworzy model.
     * @param n długość kombinacji (słowa)
     */
    public WordListModel(int n)
    {
        length = n;
    }

    public int getSize()
    {
        return (int) Math.pow(LAST - FIRST + 1, length);
    }

    public String getElementAt(int n)
    {
        StringBuilder r = new StringBuilder();
        for (int i = 0; i < length; i++)
        {
            char c = (char) (FIRST + n % (LAST - FIRST + 1));
            r.insert(0, c);
            n = n / (LAST - FIRST + 1);
        }
        return r.toString();
    }
}
```

---

Po utworzeniu modelu listy łatwo wykreować taką listę, która pozwoli użytkownikowi na przeglądanie wartości dostarczanych przez model:

```
JList<String> wordList = new JList<>(new WordListModel(3));
wordList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
JScrollPane scrollPane = new JScrollPane(wordList);
```

Zaletą programu jest to, że prezentowane na liście łańcuchy nie są nigdzie *przechowywane*. Generowane są jedynie elementy listy widoczne w danym momencie dla użytkownika.

Musimy jeszcze dostarczyć do listy informację, że każdy z jej elementów posiada stałą szerokość i wysokość. Najprostszy sposób określenia rozmiarów komórki listy polega na wykorzystaniu *prototypowej wartości komórki*:

```
wordList.setPrototypeCellValue("www");
```

Prototypowa wartość komórki zostaje użyta do określenia rozmiarów wszystkich komórek. (Użyliśmy w tym celu łańcucha "www", ponieważ "w" jest najszerszą małą literą dla większości czcionek).

Inny sposób polega na bezpośrednim określeniu rozmiarów komórki:

```
wordList.setFixedCellWidth(50);
wordList.setFixedCellHeight(15);
```

W przeciwnym razie lista będzie wyznaczać te wartości dla każdego elementu, co będzie zbyt czasochłonne.

Na listingu 6.2 przedstawiliśmy główną klasę programu.

W praktyce listy zawierające tak dużą liczbę elementów są rzadko przydatne, ponieważ przeglądanie ich jest kłopotliwe dla użytkownika. Dlatego też uważamy, że projektanci list pakietu Swing przesadzili nieco z ich uniwersalnością. Liczba elementów, które użytkownik może wygodnie przeglądać na ekranie, jest tak mała, że mogłyby one być przechowywane po prostu wewnątrz komponentu listy. Oszczędziłoby to programistom tworzenia modeli list. Z drugiej jednak strony zastosowane rozwiązanie sprawia, że sposób wykorzystania komponentu `JList` jest spójny ze sposobami używania komponentów `JTree` i `JTable`, w przypadku których taka uniwersalność okazuje się przydatna.

#### API javax.swing.JList<E> 1.2

- `JList(ListModel<E> dataModel)`  
tworzy listę, która wyświetla elementy dostarczane przez określony model.
- `void setPrototypeCell(E newValue)`
- `E getPrototypeCell()`  
konfiguruje lub zwraca prototypową wartość komórki używaną do wyznaczenia szerokości i wysokości każdej komórki listy. Domyslnie prototypowa wartość komórki równa jest `null`, co wymusza wyznaczenie rozmiarów każdej komórki z osobna.
- `void setFixedCellWidth(int width)`
- `void setFixedCellHeight(int height)`  
jeśli parametr `width` lub `height` ma wartość większą od zera, to określa on szerokość lub wysokość każdej komórki listy. Wartością domyślną jest `-1`, co wymusza wyznaczenie rozmiarów każdej komórki z osobna.

**API javax.swing.ListModel<E> 1.2**

- `int getSize()`  
zwraca liczbę elementów w danym modelu.
- `E getElementAt(int position)`  
zwraca element modelu.

### 6.1.3. Wstawianie i usuwanie

Kolekcji elementów listy nie możemy modyfikować bezpośrednio. Operacje te możemy wykonywać jedynie za pośrednictwem *modelu*. Jak się znowu okaże, także w tym przypadku łatwiej powiedzieć, niż zrobić. Założymy, że chcemy umieścić na liście kolejne elementy. Najpierw pobierzemy referencję odpowiedniego modelu:

```
ListModel<String> model = list.getModel();
```

Jednak interfejs `ListModel` nie zawiera metod umożliwiających wstawianie i usuwanie elementów, ponieważ nie wymaga on przechowywania elementów listy.

Spróbujmy więc inaczej. Jeden z konstruktorów klasy `JList` korzysta z wektora obiektów:

```
Vector<String> values = new Vector<String>();
values.addElement("quick");
values.addElement("brown");
...
JList<String> list = new JList<>(values);
```

Elementy wektora możemy następnie usuwać lub dodawać, ale lista nie zarejestruje tych operacji i wobec tego nie będzie odzwierciedlać zmian w zbiorze elementów. Zatem również ten konstruktor nie stanowi rozwiązania problemu.

Rozwiązanie polega na utworzeniu modelu klasy `DefaultListModel`, wypełnieniu go początkowymi elementami i związaniu z listą. Klasa `DefaultListModel` implementuje interfejs `ListModel`, umożliwiając zarządzanie kolekcją obiektów.

```
DefaultListModel<String> model = new DefaultListModel();
model.addElement("quick");
model.addElement("brown");
...
JList<String> list = new JList<>(model);
```

Możemy teraz dodawać i usuwać elementy modelu, który będzie powiadamiać listę o tych zmianach, a ona zaktualizuje prezentację na ekranie.

```
model.removeElement("brown");
model.addElement("slow");
```

Jak łatwo zauważyc, klasa `DefaultModelList` stosuje nazwy metod odmienne niż klasy kolekcji.

Domyślny model listy wykorzystuje wektor do przechowania danych.



Dostępne są konstruktory klasy `JList` tworzące listę w oparciu o tablicę lub wektor obiektów lub łańcuchów znaków. Wydawać się może, że wykorzystują one model `DefaultListModel` w celu przechowywania elementów listy. Nie jest to prawda. Konstruktory te korzystają z uproszczonego modelu, który umożliwia jedynie dostęp do elementów, ale nie posiada mechanizmu powiadamiania o zmianach. Poniżej prezentujemy kod konstruktora klasy `JList` tworzącego listę na podstawie wektora:

```
public JList(final Vector<? extends E> listData)
{
    this (new AbstractListModel<E>()
    {
        public int getSize()
        { return listData.size(); }
        public E getElementAt(int i)
        { return listData.elementAt(i); }
    });
}
```

Jeśli zmienimy zawartość wektora po utworzeniu listy, to pokazywać ona będzie mylącą mieszankę starych i nowych elementów, aż do momentu gdy cała lista zostanie odrysowana. (Słowo kluczowe `final` w deklaracji konstruktora nie zabrania wprowadzania zmian zawartości wektora w innych fragmentach kodu. Oznacza ono jedynie, że konstruktor nie modyfikuje referencji `listData`. Słowo kluczowe `final` jest wymagane w tej deklaracji, ponieważ obiekt `listData` wykorzystywany jest przez klasę wewnętrzną).

#### `javax.swing.JList<E>` 1.2

- `ListModel<E> getModel()`  
pobiera model listy.

#### `javax.swing.DefaultListModel<E>` 1.2

- `void addElement(E obj)`  
umieszcza obiekt na końcu danych modelu.
- `boolean removeElement(Object obj)`  
usuwa pierwsze wystąpienie obiektu z modelu. Zwraca wartość `true`, jeśli obiekt został odnaleziony w modelu, wartość `false` — w przeciwnym razie.

### 6.1.4. Odrysowywanie zawartości listy

Jak dotąd wszystkie przykłady wykorzystywały listy zawierające łańcuchy znaków. W praktyce równie łatwo możemy utworzyć listę ikon, dostarczając konstruktorowi tablicę lub wektor wypełniony obiektami klasy `Icon`. W ogóle możemy reprezentować elementy listy za pomocą dowolnych rysunków.

Klasa `JList` automatycznie wyświetla łańcuchy znaków oraz ikony, ale w pozostałych przypadkach należy dostarczyć jej *obiekt odrysowujący zawartość komórek listy*. Obiekt taki musi należeć do klasy, która implementuje poniższy interfejs:

```
interface ListCellRenderer<E>
{
    Component getListCellRendererComponent(JList<? extends E> list,
                                           E value, int index,
                                           boolean isSelected, boolean cellHasFocus);
}
```

Metoda tego interfejsu wywoływana jest dla każdej komórki listy i zwraca komponent, który rysuje zawartość komórki. Komponent ten zostaje umieszczony we właściwym miejscu, gdy zachodzi konieczność odrysowania zawartości komórki.

Najprostszy sposób implementacji obiektu odrysowującego zawartość komórki polega na utworzeniu klasy pochodnej komponentu JComponent:

```
class MyCellRenderer extends JComponent implements ListCellRenderer<Type>
{
    public Component getListCellRendererComponent(JList<? extends Type> list,
                                                   Type value, int index,
                                                   boolean isSelected, boolean cellHasFocus)
    {
        przechowuje informację potrzebną do wyznaczenia rozmiarów i rysowania
        return this;
    }
    public void paintComponent(Graphics g)
    {
        kod rysujący element
    }
    public Dimension getPreferredSize()
    {
        kod wyznaczający rozmiary
    }
    zmienne instancji
}
```

Program, którego kod źródłowy zawiera listing 6.4, umożliwia wybór czcionki, korzystając z jej rzeczywistego przedstawienia na liście (patrz rysunek 6.4). Metoda paintComponent wyświetla nazwę danej czcionki, wykorzystując ją samą. Musi także dopasować kolorystykę do aktualnego wyglądu klasy JList. Uzyskujemy ją, posługując się metodami getForeground/→getBackground oraz getSelectionForeground/getSelectionBackground klasy JList. Metoda getPreferredSize mierzy łańcuch znaków w sposób opisany w książce *Java. Podstawy* w rozdziale 7.

**Rysunek 6.4.**  
Lista o komórkach  
rysowanych  
przez program



**Listing 6.4.** listRendering/FontCellRenderer.java

```

package listRendering;

import java.awt.*;
import javax.swing.*;

/**
 * Obiekt rysujący komórki listy przy użyciu czcionki, której nazwę zawiera komórka.
 */
public class FontCellRenderer extends JComponent implements ListCellRenderer<Font>
{
    private Font font;
    private Color background;
    private Color foreground;

    public Component getListCellRendererComponent(JList<? extends Font> list,
                                                Font value, int index, boolean isSelected, boolean cellHasFocus)
    {
        font = value;
        background = isSelected ? list.getSelectionBackground() : list.getBackground();
        foreground = isSelected ? list.getSelectionForeground() : list.getForeground();
        return this;
    }

    public void paintComponent(Graphics g)
    {
        String text = font.getFamily();
        FontMetrics fm = g.getFontMetrics(font);
        g.setColor(background);
        g.fillRect(0, 0, getWidth(), getHeight());
        g.setColor(foreground);
        g.setFont(font);
        g.drawString(text, 0, fm.getAscent());
    }

    public Dimension getPreferredSize()
    {
        String text = font.getFamily();
        Graphics g = getGraphics();
        FontMetrics fm = g.getFontMetrics(font);
        return new Dimension(fm.stringWidth(text), fm.getHeight());
    }
}

```

Obiekt rysujący komórki instalujemy za pomocą metody setCellRenderer:

```
fontList.setCellRenderer(new FontCellRenderer());
```

Od tego momentu wszystkie komórki listy będą rysowane przez ten obiekt.

W wielu przypadkach sprawdza się prostsza metoda tworzenia obiektów rysujących komórki list. Jeśli komórka składa się z tekstu, ikony i zmienia swój kolor, to wszystkie te możliwości uzyskać możemy, korzystając z obiektu klasy JLabel. Aby na przykład pokazać nazwę czcionki daną czcionką, możemy skorzystać z następującego obiektu:

```

class FontCellRenderer extends JLabel implements ListCellRenderer<Font>
{
    public Component getListCellRendererComponent(JList<? extends Font> list,
                                                Font value, int index, boolean isSelected,
                                                boolean cellHasFocus)
    {
        Font font = (Font)value;
        setText(font.getFamily());
        setFont(font);
        setOpaque(true);
        setBackground(isSelected
                      ? list.getSelectionBackground()
                      : list.getBackground());
        setForeground(isSelected
                      ? list.getSelectionForeground()
                      : list.getForeground());
        return this;
    }
}

```

Zwróćmy uwagę, że w tym przypadku nie implementujemy wcale metod `paintComponent` oraz `getPreferredSize`. Implementacje tych metod posiada bowiem klasa `JLabel`. Nasze zadanie polega jedynie na skonfigurowaniu tekstu, czcionki i koloru etykiety klasy `JLabel` zgodnie z naszymi wymaganiami.

Kod taki stanowi wygodny skrót w sytuacjach, w których istnieje komponent (`JLabel` w naszym przypadku) o funkcjonalności wystarczającej do narysowania zawartości komórki listy.

Komponent `JLabel` mogliśmy również zastosować w naszym programie przykładowym, ale zdecydowaliśmy się zaprezentować bardziej ogólny kod, który pozwala na rysowanie dowolnej zawartości komórek listy.



Nie należy tworzyć nowego komponentu za każdym razem, gdy wywoływana jest metoda `getListCellRendererComponent`. Jeśli użytkownik przewija listę, to ciągle będą tworzone nowe komponenty. Efektywniejsze rozwiązanie polega na rekonfiguracji istniejącego komponentu.

### javax.swing.JList<E> 1.2

- `Color getBackground()`  
zwraca kolor tła komórki listy, która nie jest wybrana.
- `Color getSelectionBackground()`  
zwraca kolor tła komórki listy, która została wybrana.
- `Color getForeground()`  
zwraca kolor komórki listy, która nie jest wybrana.
- `Color getSelectionForeground()`  
zwraca kolor komórki listy, która została wybrana.

- void setCellRenderer(ListCellRenderer<? super E> cellRenderer)  
instaluje obiekt wykorzystywany do rysowania zawartości komórek listy.

**API javax.swing.ListCellRenderer<E> 1.2**

- Component getListCellRendererComponent(JList<? extends E> list, E item, int index, boolean isSelected, boolean hasFocus)

zwraca komponent, którego metoda paint rysuje zawartość komórek. Jeśli komórki listy nie posiadają stałych rozmiarów, to komponent ten musi także implementować metodę getPreferredSize.

<i>Parametry:</i>	list	lista, której komórki mają zostać narysowane,
	item	rysowany element,
	index	indeks elementu w modelu,
	isSelected	wartość true, jeśli komórka jest wybrana,
	hasFocus	wartość true, jeśli komórka jest bieżąca.

## 6.2. Tabele

Komponent JTable wyświetla dwuwymiarową siatkę obiektów. Tabele stanowią często wykorzystywany komponent interfejsu użytkownika. Projektanci biblioteki Swing włożyli wiele wysiłku w uniwersalne zaprojektowanie komponentu tabel. Tabele są skomplikowanym komponentem, ale w ich przypadku projektantom klas JTable udało się ukryć tę złożoność. W pełni funkcjonalne tabele o dużych możliwościach tworzymy już za pomocą kilku wierszy kodu. Oczywiście kod ten możemy rozbudowywać, dostosowując wygląd i zachowanie tabeli do naszych specyficznych potrzeb.

W podrozdziale tym przedstawimy sposób tworzenia najprostszych tabel, ich interakcje z użytkownikiem i najczęstsze modyfikacje komponentu. Podobnie jak w przypadku innych złożonych komponentów biblioteki Swing, omówienie wszystkich aspektów korzystania z tabel przekracza możliwości tego rozdziału. Więcej informacji na ten temat znaleźć można w książce *Graphic Java 2*, wyd. trzecie, napisanej przez Davida M. Geary'ego (Prentice Hall 1999) lub *Core Swing* autorstwa Kim Topley (Prentice Hall 1999).

### 6.2.1. Najprostsze tabele

Podobnie jak komponent drzewa, także i klasa JTable nie przechowuje danych tabeli, ale uzyskuje je, korzystając z *modelu tabeli*. Klasa JTable dysponuje konstruktorem, który umożliwia obudowanie dwuwymiarowej tablicy obiektów za pomocą domyślnego modelu. Z takiego rozwiązania skorzystamy w naszym pierwszym przykładzie. W dalszej części rozdziału zajmiemy się innymi modelami tabel.

Rysunek 6.5 przedstawia typową tabelę opisującą cechy planet układu słonecznego. (Planeta posiada cechę *gaseous*, jeśli składa się w większości z wodoru i helu. Kolumnę *Color* wykorzystamy dopiero w kolejnych przykładach programów).

**Rysunek 6.5.**

Przykład  
najprostszej tabeli

Planet	Radius	Moons	Gaseous	Color
Mercury	2440.0	0	false	java.awt.C...
Venus	6052.0	0	false	java.awt.C...
Earth	6378.0	1	false	java.awt.C...
Mars	3397.0	2	false	java.awt.C...
Jupiter	71492.0	16	true	java.awt.C...
Saturn	60268.0	18	true	java.awt.C...
Uranus	25559.0	17	true	java.awt.C...
Neptune	24766.0	14	true	java.awt.C...

Jak można zauważyć, analizując kod programu zawarty na listingu 6.5, dane tabeli przechowywane są za pomocą dwuwymiarowej tablicy obiektów:

```
private Object[][] cells =
{
    { "Mercury", 2440.0, 0, false, Color.YELLOW },
    { "Venus", 6052.0, 0, false, Color.YELLOW },
    .
    .
}
```

**Listing 6.5.** table/TableTest.java

```
package table;

import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import javax.swing.*;

/**
 * Program demonstrujący przykład prostej tabeli.
 * @version 1.12 2012-06-09
 * @author Cay Horstmann
 */
public class TableTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new PlanetTableFrame();
                frame.setTitle("TableTest");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }

    /**
     * Ramka zawierająca tabelę danych o planetach.
     */
}
```

```

class PlanetTableFrame extends JFrame
{
    private String[] columnNames = { "Planet", "Radius", "Moons", "Gaseous", "Color" };
    private Object[][] cells = { { "Mercury", 2440.0, 0, false, Color.YELLOW },
        { "Venus", 6052.0, 0, false, Color.YELLOW }, { "Earth", 6378.0, 1, false,
        ↳Color.BLUE },
        { "Mars", 3397.0, 2, false, Color.RED }, { "Jupiter", 71492.0, 16, true,
        ↳Color.ORANGE },
        { "Saturn", 60268.0, 18, true, Color.ORANGE },
        { "Uranus", 25559.0, 17, true, Color.BLUE }, { "Neptune", 24766.0, 8, true,
        ↳Color.BLUE },
        { "Pluto", 1137.0, 1, false, Color.BLACK } };

    public PlanetTableFrame()
    {
        final JTable table = new JTable(cells, columnNames);
        table.setAutoCreateRowSorter(true);
        add(table, BorderLayout.CENTER);
        JButton printButton = new JButton("Print");
        printButton.addActionListener(EventHandler.create(ActionListener.class, table,
        ↳"print"));
        JPanel buttonPanel = new JPanel();
        buttonPanel.add(printButton);
        add(buttonPanel, BorderLayout.SOUTH);
        pack();
    }
}

```



Wykorzystaliśmy tutaj właściwość automatycznego opakowywania, dzięki której elementy w drugiej, trzeciej i czwartej kolumnie są automatycznie zamieniane na obiekty typu Double, Integer i Boolean.

Tabela wywołuje metodę `toString` każdego z tych obiektów i wyświetla uzyskany rezultat. Wyjaśnia to między innymi sposób prezentacji danych w kolumnie `Color` w postaci `java.awt.↳Color[r=...,g=...,b=...]`.

Nazwy kolumn tabeli przechowywane są w osobnej tablicy łańcuchów znaków:

```

String[] columnNames =
{
    "Planet", "Radius", "Moons", "Gaseous", "Color"
};

```

Korzystając z obu przedstawionych wyżej tablic, tworzymy tabelę.

```
JTable table = new JTable(cells, columnNames);
```



W przeciwieństwie do `JList` `JTable` nie jest typem generycznym. Powód jest dość oczywisty. W przypadku listy spodziewamy się, że wszystkie jej elementy będą jednego typu, ale w przypadku tabeli zwykle trudno określić pojedynczy typ dla wszystkich jej elementów. W naszym przykładzie nazwa planety jest łańcuchem znaków, kolor jest typu `java.awt.Color` i tak dalej.

Przewijanie tabeli umożliwiamy, obudowując ją panelem klasy JScrollPane.

```
JScrollPane pane = new JScrollPane(table);
```

Otrzymana tabela posiada zaskakująco bogate możliwości. Zmniejszymy jej rozmiar tak, by pokazały się paski przewijania. Zwrócić uwagę, że podczas przewijania zawartości tabeli nazwy kolumn pozostają na właściwym miejscu!

Następnie wybierzmy jeden z nagłówków kolumn i przeciągnijmy go w lewo lub w prawo. Spowoduje to przesunięcie całej kolumny (patrz rysunek 6.6), którą umieścić możemy w innym miejscu tabeli. Zmiana ta dotyczy *jedynie widoku* tabeli i pozostaje bez wpływu na dane modelu.

**Rysunek 6.6.**

Przesunięcie kolumny

Planet	Radius	Moons	Gaseous	Color
Mercury	2440.0	0	false	java.awt.C...
Venus	6052.0	0	false	java.awt.C...
Earth	6378.0	1	false	java.awt.C...
Mars	3397.0	2	false	java.awt.C...
Jupiter	71492.0	16	true	java.awt.C...
Saturn	60268.0	18	true	java.awt.C...
Uranus	25559.0	17	true	java.awt.C...
Neptune	24766.0	14	true	java.awt.C...

Aby zmienić szerokość kolumny, wystarczy umieścić kurSOR myszy na linii oddzielającej kolumny. KurSOR przybierze wtedy kształt strzałki, umożliwiając przesunięcie linii oddzielającej kolumny (patrz rysunek 6.7).

**Rysunek 6.7.**

Zmiana szerokości kolumny

Planet	Radius	Moons	Gas	Color
Mercury	2440.0	0	false	java.awt.Color[r=...
Venus	6052.0	0	false	java.awt.Color[r=...
Earth	6378.0	1	false	java.awt.Color[r=...
Mars	3397.0	2	false	java.awt.Color[r=...
Jupiter	71492.0	16	true	java.awt.Color[r=...
Saturn	60268.0	18	true	java.awt.Color[r=...
Uranus	25559.0	17	true	java.awt.Color[r=...
Neptune	24766.0	14	true	java.awt.Color[r=...

Użytkownik może wybierać wiersze tabeli za pomocą myszy. Wybrane wiersze zostają podświetlone. Sposób obsługi takiego zdarzenia pokażemy w dalszej części rozdziału. Użytkownik może także wybrać komórkę tabeli i zmodyfikować ją. W bieżącym przykładzie modyfikacja ta nie spowoduje jeszcze zmiany danych modelu. W praktyce w programach należy wyłączyć możliwość edycji komórek tabeli lub obsługiwać zdarzenia edycji i odpowiednio modyfikować dane modelu. Zagadnienia te omówimy nieco później.

Na koniec kliknijmy nagłówek kolumny. Wiersze zostaną posortowane. Jeśli klikniemy nagłówek jeszcze raz, porządek wierszy zostanie odwrócony. Taki sposób działania tabeli możemy aktywować za pomocą wywołania

```
table.setAutoCreateRowSorter(true);
```

Wydruk tabeli umożliwia poniższe wywołanie:

```
table.print();
```

Powoduje ona pojawienie się okna dialogowego wydruku, a następnie wysłanie tabeli do drukarki. Więcej na temat opcji drukowania w rozdziale 7.



Jeśli zmienimy rozmiar ramki TableTest w taki sposób, że jej wysokość będzie większa od wysokości tabeli, to poniżej tabeli pojawi się szary obszar. Tabela bowiem, w przeciwieństwie do komponentów JList i JTree, nie wypełnia obszaru panelu przewijania. Może to okazać się problemem podczas obsługi operacji typu przeciagnij-i-upuść. (Więcej informacji na temat trybu przeciagnij-i-upuść podamy w rozdziale 7.). W takim przypadku należy skorzystać z następującego wywołania:

```
table.setFillsViewportHeight(true);
```

#### javax.swing.JTable 1.2

- `JTable(Object[][][] entries, Object[] columnNames)`  
tworzy tabelę, wykorzystując domyślny model.
- `void print() 5.0`  
wyświetla okno dialogowe drukowania i drukuje tabelę.
- `boolean getAutoCreateRowSorter() 6`
- `void setAutoCreateRowSorter(boolean newValue) 6`  
pobiera lub nadaje wartość właściwości autoCreateRowSorter. Domyślnie właściwość ta ma wartość false. Gdy otrzyma wartość true, każda zmiana modelu spowoduje zastosowanie domyślnego obiektu sortującego.
- `boolean getFillsViewportHeight() 6`
- `void setFillsViewportHeight(boolean newValue) 6`  
pobiera lub nadaje wartość właściwości fillsViewportHeight. Domyślnie właściwość ta ma wartość false. Gdy otrzyma wartość true, tabela będzie zawsze wypełniać obszar obejmującego ją komponentu.

### 6.2.2. Modele tabel

W poprzednim przykładzie obiekty tabeli przechowywane były za pomocą dwuwymiarowej tablicy. Rozwiążanie takie nie jest jednak zalecane. Jeśli kod nasz umieszcza dane w tablicy, aby zaprezentować je następnie w postaci tabeli, oznacza to, że powinniśmy zastanowić się nad implementacją własnego modelu tabeli.

Implementacja własnego modelu tabeli nie jest trudna, ponieważ wykorzystać możemy klasę AbstractTableModel, która dostarcza implementacji większości potrzebnych metod. Sami zaimplementować musimy jedynie trzy poniższe metody:

```
public int getRowCount();
public int getColumnCount();
public Object getValueAt(int row, int column);
```

Istnieje wiele sposobów implementacji metody `getValueAt`. Na przykład: jeśli chcemy użyć tabeli do prezentacji obiektu `RowSet` zawierającego wynik zapytania skierowanego do bazy danych, metodę `getValueAt` możemy zaimplementować w następujący sposób:

```
public Object getValueAt(int r, int c)
{
    try
    {
        rowSet.absolute(r + 1);
        return rowSet.getObject(c + 1);
    }
    catch (SQLException e)
    {
        e.printStackTrace();
        return null;
    }
}
```

Nasz przykładowy program używa jeszcze prostszej implementacji. Konstruuje on tabelę przedstawiającą wzrost inwestycji w różnych scenariuszach (patrz rysunek 6.8).

**Rysunek 6.8.**

Tabela  
reprezentująca  
wzrost inwestycji

InvestmentTable					
5%	6%	7%	8%	9%	10%
100000.00	100000.00	100000.00	100000.00	100000.00	100000.00
105000.00	106000.00	107000.00	108000.00	109000.00	110000.00
110250.00	112360.00	114490.00	116640.00	118810.00	121000.00
115762.50	119101.60	122504.30	125971.20	129502.90	133100.00
121550.63	126247.70	131079.60	136048.90	141158.16	146410.00
127628.16	133822.56	140255.17	146932.81	153862.40	161051.00
134009.56	141851.91	150073.04	158687.43	167710.01	177156.10
140710.04	150363.03	160578.15	171382.43	182803.91	194871.71
147745.54	159384.81	171818.62	185093.02	199256.26	214358.88
155132.82	168947.90	183845.92	199900.46	217189.33	235794.77
162889.46	179084.77	196715.14	215892.50	236736.37	259374.25
171033.94	189829.86	210485.20	233163.90	258042.64	285311.67
179585.63	201219.65	225219.16	251817.01	281266.48	313842.84
188564.91	213292.83	240984.50	271962.37	306580.46	345227.12
197993.16	226090.40	257853.42	293719.36	334172.70	379749.83
207892.82	239655.82	275903.15	317216.91	364248.25	417724.82

Metoda `getValueAt` wyznacza odpowiednią wartość komórki i formatuje ją:

```
public Object getValueAt(int r, int c)
{
    double rate = (c + minRate) / 100.0;
    int nperiods = r;
    double futureBalance = INITIAL_BALANCE
        * Math.pow(1 + rate, nperiods);
    return
        String.format("%.2f", futureBalance);
}
```

Metody `getRowCount` i `getColumnCount` zwracają odpowiednio liczbę wierszy i kolumn tabeli.

```
public int getRowCount()
{
    return years;
}

public int getColumnCount()
```

```

    {
        return maxRate - minRate + 1;
    }
}

```

Jeśli nie podamy nazw kolumn, to metoda `getColumnName` klasy `AbstractTableModel` nazwie kolejne kolumny A, B, C itd. Aby zmienić nazwy kolumn, przesłonimy metodę `getColumnName` i wykorzystamy procentowy przyrost inwestycji jako nazwę kolumn.

```

public String getColumnName(int c)
{
    return (c + minRate) + "%";
}

```

Kompletny kod źródłowy programu zawiera listing 6.6.

#### **Listing 6.6.** *tableModel/InvestmentTable.java*

```

package tableModel;

import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;

/**
 * Program tworzący tabelę w oparciu o własny model.
 * @version 1.02 2007-08-01
 * @author Cay Horstmann
 */
public class InvestmentTable
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new InvestmentTableFrame();
                frame.setTitle("InvestmentTable");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

/**
 * Ramka zawierająca tabelę inwestycji.
 */
class InvestmentTableFrame extends JFrame
{
    public InvestmentTableFrame()
    {
        TableModel model = new InvestmentTableModel(30, 5, 10);
        JTable table = new JTable(model);
        add(table);
        pack();
    }
}

```

```
/*
 * Model tabeli wyliczający wartości komórek na żądanie.
 * Tabela pokazuje przyrost inwestycji w kolejnych latach
 * w różnych scenariuszach.
 */
class InvestmentTableModel extends AbstractTableModel
{
    private static double INITIAL_BALANCE = 100000.0;

    private int years;
    private int minRate;
    private int maxRate;

    /**
     * Tworzy model tabeli inwestycji.
     * @param y liczba lat
     * @param r1 najniższa stopa oprocentowania
     * @param r2 najwyższa stopa oprocentowania
     */
    public InvestmentTableModel(int y, int r1, int r2)
    {
        years = y;
        minRate = r1;
        maxRate = r2;
    }

    public int getRowCount()
    {
        return years;
    }

    public int getColumnCount()
    {
        return maxRate - minRate + 1;
    }

    public Object getValueAt(int r, int c)
    {
        double rate = (c + minRate) / 100.0;
        int nperiods = r;
        double futureBalance = INITIAL_BALANCE * Math.pow(1 + rate, nperiods);
        return String.format("%.2f", futureBalance);
    }

    public String getColumnName(int c)
    {
        return (c + minRate) + "%";
    }
}
```

---

**API javax.swing.table.TableModel 1.2**

- int getRowCount()
  - int getColumnCount()
- zwracają liczbę wierszy i kolumn w modelu tabeli.

- `Object getValueAt(int row, int column)`  
zwraca wartość w danym wierszu i kolumnie.
- `void setValueAt(Object newValue, int row, int column)`  
nadaje wartość obiektowi w danym wierszu i kolumnie.
- `boolean isCellEditable(int row, int column)`  
zwraca wartość true, jeśli komórka w danym wierszu i kolumnie może być edytowana.
- `String getColumnName(int column)`  
zwraca nazwę (tytuł) kolumny.

## 6.2.3. Wiersze i kolumny

W kolejnym podrozdziale zajmiemy się manipulacjami wierszami i kolumnami tabeli. Podczas lektury tego materiału należy pamiętać, że tabele Swing są asymetryczne i wobec tego operacje, które można wykonywać na wierszach różnią się od operacji na kolumnach. Komponent tabeli zaprojektowano z myślą o prezentacji wierszy o takiej samej strukturze, na przykład rekordów bazy danych, a nie o prezentacji dwuwymiarowej siatki dowolnych obiektów. Asymetria ta będzie się pojawiać we wszystkich zagadnieniach omawianych w tym podrozdziale.

### 6.2.3.1. Klasa kolumn

Kolejny przykład znowu będzie prezentował dane o planetach, ale tym razem wyposażymy tabelę w informację o *typie kolumn*. Jeśli zdefiniujemy metodę

```
Class<?> getColumnClass(int columnIndex)
```

modelu tabeli tak, by zwracała klasę opisującą typ kolumny, to klasa `JTable` będzie mogła wybrać właściwy *obiekt rysujący* dla danej klasy. Tabela 6.1 przedstawia sposób prezentacji kolumn różnych typów przez domyślne obiekty rysujące wykorzystywane przez klasę `JTable`.

**Tabela 6.1.** Domyślne obiekty rysujące

Typ kolumny	Reprezentacja w postaci
Icon	obrazka
Boolean	pola wyboru
Object	łańcucha znaków

Pola wyboru i obrazki w komórkach tabeli zobaczyć możemy na rysunku 6.9 (dziękujemy w tym miejscu Jimowi Evinsowi, <http://snaught.com/JimsCoolIcons/Planets>, za udostępnienie obrazków planet).

W przypadku innych typów dostarczyć możemy własne obiekty rysujące komórki tabeli — patrz punkt 6.2.4, „Rysowanie i edycja komórek”.

**Rysunek 6.9.**

Tabela zawierająca dane o planetach

	us	Moons	Gaseous	Color	Image
Mercury	0	0	<input type="checkbox"/>	java.awt.Color[r=255,g=255,b=0]	
Earth	6,378	1	<input type="checkbox"/>	java.awt.Color[r=0,g=0,b=255]	
Mars	3,397	2	<input type="checkbox"/>	java.awt.Color[r=255,g=0,b=0]	
Jupiter	71,492	16	<input checked="" type="checkbox"/>	java.awt.Color[r=255,g=200,b=0]	
Saturn	60,268	18	<input checked="" type="checkbox"/>	java.awt.Color[r=255,g=200,b=0]	

**6.2.3.2. Dostęp do kolumn tabeli**

Klasa `JTable` przechowuje informację o kolumnach tabeli, korzystając z obiektów typu `TableColumn`. Klasa `TableColumnModel` zarządza natomiast kolumnami. (Rysunek 6.10 przedstawia zależności między najważniejszymi klasami związanymi z tabelami). Jeśli nie planujemy dynamicznie umieszczać w tabeli nowych kolumn bądź usuwać istniejących, to nie musimy korzystać z usług modelu kolumn tabeli z wyjątkiem sytuacji, w której chcemy uzyskać obiekt `TableColumn` dla pewnej kolumny:

```
int columnIndex = . . .;
TableColumn column = table.getColumnModel().getColumn(columnIndex);
```

**6.2.3.3. Zmiana rozmiaru kolumn**

Klasa `TableColumn` udostępnia metody umożliwiające nadzór nad zmianami szerokości kolumny wykonywanymi przez użytkownika. Programista może określić preferowaną, najmniejszą i największą szerokość kolumny, korzystając z poniższych metod.

```
void setPreferredWidth(int width)
void setMinWidth(int width)
void setMaxWidth(int width)
```

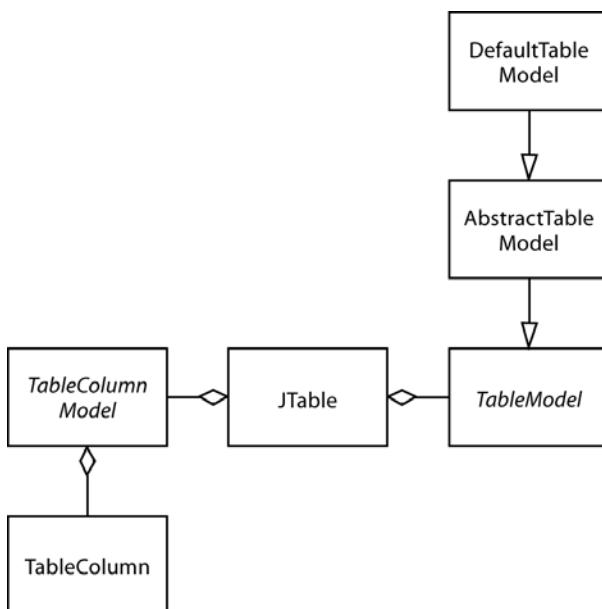
Informacja ta wykorzystywana jest przez komponent tabeli podczas jej wyświetlania.

Metoda

```
void setResizable(boolean resizable)
```

zezwala lub zabrania użytkownikowi zmieniać szerokość kolumny.

**Rysunek 6.10.**  
Zależności między klasami tabel



Szerokość kolumny można także zmieniać programowo, korzystając z poniższej metody.

```
void setWidth(int width)
```

Gdy zmieniana jest szerokość kolumny, to domyślnie całkowita szerokość tabeli nie ulega zmianie. Oznacza to, że zmiana szerokości kolumny spowoduje także zmianę szerokości kolumn tabeli, leżących od niej na prawo. Zachowanie takie jest o tyle rozsądne, że pozwala użytkownikowi dostosować szerokość kolejnych kolumn, poruszając się od lewej strony tabeli ku prawej.

Zachowanie to możemy zmienić na jedno z wymienionych w tabeli 6.2 za pomocą metody

```
void setAutoResizeMode(int mode)
```

udostępnianej przez klasę `JTable`.

**Tabela 6.2.** Tryby zmiany szerokości kolumn

Tryb	Zachowanie
AUTO_RESIZE_OFF	Nie zmienia szerokości innych kolumn, ale szerokość całej tabeli.
AUTO_RESIZE_NEXT_COLUMN	Zmienia jedynie szerokość następnej kolumny.
AUTO_RESIZE_SUBSEQUENT_COLUMNS	Zmienia równo szerokość wszystkich następnych kolumn. Zachowanie domyślne.
AUTO_RESIZE_LAST_COLUMN	Zmienia jedynie szerokość ostatniej kolumny.
AUTO_RESIZE_ALL_COLUMNS	Zmienia szerokość wszystkich kolumn tabeli. Zachowanie to najczęściej nie jest właściwe, ponieważ uniemożliwia użytkownikowi określenie szerokości więcej niż jednej kolumny.

### 6.2.3.4. Zmiana rozmiaru wierszy

Wysokością wierszy zarządza bezpośrednio klasa `JTable`. Jeśli chcemy zmienić wysokość komórek tabeli, skorzystamy z poniższej metody.

```
table.setRowHeight(height);
```

Domyślnie wszystkie wiersze tabeli posiadają tę samą wysokość. Możemy jednak zmienić wysokość poszczególnych wierszy, posługując się wywołaniem:

```
table.setRowHeight(row, height);
```

Rzeczywista wysokość komórki pomniejszona będzie o jej margines, który domyślnie wynosi 1. Wielkość marginesu możemy zmienić, używając poniższej metody.

```
table.setRowMargin(margin);
```

### 6.2.3.5. Wybór wierszy, kolumn i komórek

W zależności od trybu wyboru użytkownik wybierać może wiersze, kolumny bądź komórki tabeli. Domyślnym trybem wyboru jest tryb zezwalający na wybór wierszy tabeli. Wybranie komórki powoduje automatycznie wybranie całego wiersza (patrz rysunek 6.9). Wywołanie

```
table.setRowSelectionAllowed(false)
```

wyłącza możliwość wyboru wierszy.

Jeśli tryb wyboru wierszy jest włączony, to możemy określić, czy użytkownikowi wolno wybrać pojedynczy wiersz, ciągły zakres wierszy bądź dowolny zbiór wierszy. W tym celu musimy pobrać *model wyboru* i skorzystać z jego metody `setSelectionMode`:

```
table.getSelectionModel().setSelectionMode(mode);
```

Parametr `mode` przyjmować może jedną z trzech wartości:

```
ListSelectionModel.SINGLE_SELECTION  
ListSelectionModel.SINGLE_INTERVAL_SELECTION  
ListSelectionModel.MULTIPLE_INTERVAL_SELECTION
```

Wybór kolumn jest domyślnie zabroniony. Możemy go umożliwić, wywołując

```
table.setColumnSelectionAllowed(true)
```

Zezwolenie na równoczesny wybór wierszy i kolumn oznacza możliwość wyboru komórek tabeli. Użytkownik może wtedy wybierać zakresy komórek, jak pokazano na rysunku 6.11. Wybór komórek umożliwić możemy także, korzystając z metody

```
table.setCellSelectionEnabled(true)
```

Program, którego kod źródłowy zawiera listing 6.7, umożliwia obserwację sposobu wyboru elementów tabeli. Jego menu umożliwia włączanie lub wyłączanie możliwości wyboru wierszy, kolumn i komórek tabeli.

**Rysunek 6.11.**

Wybór zakresu komórek tabeli

Planet	Radius	Moons	Gaseous	Color	Image
Mars	3,397	2	<input type="checkbox"/>	java.awt.Color[r=255,g=0,b=0]	
Jupiter	71,492	16	<input checked="" type="checkbox"/>	java.awt.Color[r=255,g=200,b=0]	
Saturn	60,268	18	<input checked="" type="checkbox"/>	java.awt.Color[r=255,g=200,b=0]	
Uranus	25,559	17	<input checked="" type="checkbox"/>	java.awt.Color[r=0,g=0,b=255]	

Informację o wybranych wierszach i kolumnach możemy uzyskać, wywołując metody `getSelectedRows` i `getSelectedColumns`. Metody te zwracają tablice `int[]` zawierające indeksy wybranych wierszy bądź kolumn. Zwróćmy uwagę, że wartości indeksów dotyczą w tym przypadku widoku tabeli, a nie używanego przez nią modelu tabeli. Spróbuj wybrać różne kolumny i wiersze, przeciągnąć kolumnę w inne miejsce czy posortować wiersze, klikając nagłówki kolumn. Menu *Print Selection* pozwoli Ci sprawdzić, które wiersze i kolumny są raportowane jako wybrane.

Jeśli zachodzi potrzeba przełożenia wartości indeksów tablicy na wartości indeksów modelu, to stosujemy wtedy metody `convertRowIndexToModel` i `convertColumnIndexToModel` klasy `JTable`.

### 6.2.3.6. Sortowanie wierszy

Jak pokazaliśmy w naszym pierwszym przykładowym programie działającym na tabelach, możemy łatwo dodać sortowanie do tabeli `JTable`, wywołując po prostu metodę `setAutoCreateRowSorter`. Jeśli jednak chcemy sami kontrolować sposób sortowania wierszy tabeli, musimy zainstalować dla tabeli `JTable` obiekt `TableRowSorter<TableModel>` i dopasować jego działanie do naszych potrzeb. Parametr typu `M` oznacza w tym przypadku model tabeli, który musi być typem pochodnym interfejsu `TableModel`.

```
TableRowSorter<TableModel> sorter = new TableRowSorter<TableModel>(model);
table.setRowSorter(sorter);
```

Niektóre kolumny nie powinny umożliwiać sortowania, na przykład kolumna zawierająca obrazki planet w naszym programie. Sortowanie wyłączamy, wywołując:

```
sorter.setSortable(IMAGE_COLUMN, false);
```

Dla każdej kolumny możemy zainstalować własny komparator. W naszym przykładzie użyjemy go do posortowania kolumny Color w taki sposób, że kolory niebieski i zielony będą faworyzowane względem czerwonego. Jeśli klikniesz kolumnę Color, to zobaczysz, że niebieskie planety znajdują się na końcu tabeli. Osiągamy to za pomocą poniższego wywołania:

```
sorter.setComparator(COLOR_COLUMN, new Comparator<Color>()
{
    public int compare(Color c1, Color c2)
    {
        int d = c1.getBlue() - c2.getBlue();
        if (d != 0) return d;
        d = c1.getGreen() - c2.getGreen();
        if (d != 0) return d;
        return c1.getRed() - c2.getRed();
    }
});
```

Jeśli nie podamy własnego komparatora, to uporządkowanie określone zostaje w następujący sposób:

1. Jeśli klasą kolumny jest String, zostanie zastosowany domyślny obiekt zwrocony przez metodę `Collator.getInstance()`. Sortuje on łańcuchy znaków w sposób właściwy dla bieżącego lokalizatora. (Więcej informacji o lokalizatorach i obiektach sortujących znajdziesz w rozdziale 5.).
2. Jeśli klasa kolumny implementuje interfejs `Comparable`, zostaje zastosowana metoda `compareTo`.
3. Jeśli dla komparatora został skonfigurowany konwerter `TableStringConverter`, to łańcuchy zwrocone przez metodę `toString` zostaną posortowane przy użyciu domyślnego obiektu sortującego. Jeśli chcemy skorzystać z tej możliwości, musimy zdefiniować konwerter w poniższy sposób:

```
sorter.setStringConverter(new TableStringConverter()
{
    public String toString(TableModel model, int row, int column)
    {
        Object value = model.getValueAt(row, column);
        zamień value na łańcuch i zwróć go
    }
});
```

4. W pozostałych przypadkach zostanie wywołana metoda `toString` dla wartości komórek i zostaną one posortowane za pomocą domyślnego obiektu sortującego.

### 6.2.3.7. Filtrowanie wierszy

`TableRowSorter` pozwala na sortowanie wierszy tabeli, a także na ich selektywne ukrywanie, czyli proces zwany *filtrowaniem*. Aby uruchomić filtrowanie, musimy skonfigurować filtr `RowFilter`. Na przykład: jeśli tabela ma prezentować tylko planety posiadające co najmniej jeden księżyc, posłużymy się następującym wywołaniem:

```
sorter.setRowFilter(RowFilter.numberFilter(ComparisonType.NOT_EQUAL, 0,
→MOONS_COLUMN));
```

W tym przypadku użyliśmy wstępnie zdefiniowanego filtra działającego na wartościach liczbowych. Aby skonstruować taki filtr, musimy określić:

- Typ porównania (jedna ze stałych EQUAL, NOT\_EQUAL, AFTER lub BEFORE).
- Obiekt klasy pochodnej od klasy Number (na przykład Integer lub Double). Pod uwagę będą brane tylko te obiekty, które mają taką samą klasę jak podany obiekt Number.
- Zero lub więcej indeksów kolumn. Jeśli nie podamy żadnego indeksu kolumny, to przeszukiwane będą wszystkie kolumny tabeli.

Metoda statyczna RowFilter.dateFilter konstruuje w ten sam sposób filtr działający na danych. Zatem zamiast obiektu klasy Number dostarczamy jej obiekt klasy Date.

Metoda statyczna RowFilter.regexFilter tworzy filtr poszukujący łańcuchów pasujących do wyrażenia regularnego. Na przykład

```
sorter.setRowFilter(RowFilter.regexFilter(".*[^\s]$". PLANET_COLUMN));
```

spowoduje wyświetlenie w tabeli tylko tych planet, których nazwa nie kończy się literą "s". (Więcej informacji na temat wyrażeń regularnych znajdziesz w rozdziale 1.).

Metody andFilter, orFilter i notFilter pozwalają tworzyć kombinacje filtrów. Poniższe wywołanie stworzy kombinację filtrów wyświetlającą w tabeli planety, których nazwa nie kończy się literą "s" i które posiadają co najmniej jeden księżyca:

```
sorter.setRowFilter(RowFilter.andFilter(Arrays.asList(
    RowFilter.regexFilter(".*[^\s]$". PLANET_COLUMN),
    RowFilter.numberFilter(ComparisonType.NOT_EQUAL, 0, MOONS_COLUMN)));
```



Niestety, metody andFilter i orFilter nie posiadają zmiennej liczby argumentów, lecz używają pojedynczego parametru typu Iterable.

Aby zaimplementować własny filtr, musimy dostarczyć klasę pochodną klasy RowFilter i zaimplementować metodę include, która pozwoli określić wyświetlane wiersze tabeli. Chociaż wspaniała ogólność klasy RowFilter może działać na programistę trochę paraliżująco, samo zadanie jest dość łatwe do wykonania.

Klasa RowFilter<M, I> ma dwa parametry typu: typ modelu i typ identyfikatora wiersza. W przypadku tabel typ modelu jest zawsze podtypem typu TableModel, a typem identyfikatora jest Integer. (W przyszłości być może również inne komponenty będą obsługiwać filtrowanie. Na przykład aby filtrować wiersze komponentu JTree, będzie się używać klasy RowFilter ↗<TreeModel, TreePath>).

Filtr wierszy musi implementować metodę

```
public boolean include(RowFilter.Entry<? extends M, ? extends I> entry)
```

Klasa RowFilter.Entry dostarcza metody pozwalające uzyskać model, identyfikator wiersza i wartość o podanym indeksie. Dlatego też możemy filtrować wiersze zarówno na podstawie identyfikatora wiersza, jak i zawartości wiersza.

Poniższy filtr wyświetla co drugi wiersz:

```
RowFilter<TableModel, Integer> filter = new RowFilter<TableModel, Integer>()
{
    public boolean include(Entry<? extends TableModel, ? extends Integer> entry)
    {
        return entry.getIdentifier() % 2 == 0;
    }
};
```

Jeśli chcemy, aby tabela zawierała tylko planety o parzystej liczbie księżyców, powinniśmy sprawdzać następujący warunek:

```
((Integer) entry.getValue(MOONS_COLUMN)) % 2 == 0
```

W naszym przykładowym programie umożliwimy użytkownikowi ukrywanie dowolnych wierszy. Indeksy ukrytych wierszy będziemy umieszczać w zbiorze. Filtr wierszy będzie dopuszczał jedynie te wiersze, których indeksy nie znajdują się w tym zbiorze.

Mechanizm filtrowania nie został zaprojektowany z myślą o kryteriach filtrowania zmieniających się w czasie. Dlatego w naszym programie wywołujemy

```
sorter.setRowFilter(filter);
```

za każdym razem, gdy zmienia się zawartość zbioru ukrytych wierszy. Wywołanie to powoduje natychmiastowe zastosowanie filtra.

### 6.2.3.8. Ukrywanie i wyświetlanie kolumn

W poprzednim punkcie pokazaliśmy, że możliwe jest filtrowanie wierszy tabeli na podstawie ich identyfikatora bądź zawartości. W przypadku ukrywania kolumn zastosowano natomiast zupełnie inny mechanizm.

Metoda `removeColumn` klasy `JTable` usuwa kolumnę z widoku tabeli. Dane kolumny nie są usuwane z modelu tabeli, a jedynie ukrywane przed jej widokiem. Parametrem metody `removeColumn` jest obiekt klasy  `TableColumn`. Jeśli dysponujemy numerem kolumny (na przykład zwróconym przez metodę `getSelectedColumn`), to musimy najpierw pobrać od modelu kolumn tabeli odpowiedni obiekt reprezentujący kolumnę:

```
TableColumnModel columnModel = table.getColumnModel();
TableColumn column = columnModel.getColumn();
table.removeColumn(column);
```

Jeśli zapamiętamy obiekt kolumny, to możemy później wstawić go z powrotem do tabeli:

```
table.addColumn(column);
```

Wywołanie tej metody spowoduje dodanie kolumny jako ostatniej kolumny tabeli. Jeśli chcemy ją umieścić w innym miejscu tabeli, to musimy skorzystać jeszcze z metody `moveColumn`.

Nową kolumnę możemy dodać także, tworząc nowy obiekt klasy  `TableColumn` o indeksie odpowiadającym numerowi kolumny w modelu.

```
table.addColumn(new TableColumn(modelColumnIndex));
```

Możemy utworzyć wiele obiektów tej klasy, które stanowić będą reprezentacje jednej i tej samej kolumny modelu.

Program przedstawiony na listingu 6.7 prezentuje możliwości wyboru i filtrowania wierszy i kolumn tabeli.

**Listing 6.7.** *tableRowColumn/PlanetTableFrame.java*

```
package tableRowColumn;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.table.*;

/**
 * Ramka zawierająca tabelę z danymi planet.
 */
public class PlanetTableFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 600;
    private static final int DEFAULT_HEIGHT = 500;

    public static final int COLOR_COLUMN = 4;
    public static final int IMAGE_COLUMN = 5;

    private JTable table;
    private HashSet<Integer> removedRowIndices;
    private ArrayList<TableColumn> removedColumns;
    private JCheckBoxMenuItem rowsItem;
    private JCheckBoxMenuItem columnsItem;
    private JCheckBoxMenuItem cellsItem;

    private String[] columnNames = { "Planet", "Radius", "Moons", "Gaseous", "Color",
        "Image" };

    private Object[][][] cells = {
        { "Mercury", 2440.0, 0, false, Color.YELLOW,
            new ImageIcon(getClass().getResource("Mercury.gif")) },
        { "Venus", 6052.0, 0, false, Color.YELLOW,
            new ImageIcon(getClass().getResource("Venus.gif")) },
        { "Earth", 6378.0, 1, false, Color.BLUE,
            new ImageIcon(getClass().getResource("Earth.gif")) },
        { "Mars", 3397.0, 2, false, Color.RED,
            new ImageIcon(getClass().getResource("Mars.gif")) },
        { "Jupiter", 71492.0, 16, true, Color.ORANGE,
            new ImageIcon(getClass().getResource("Jupiter.gif")) },
        { "Saturn", 60268.0, 18, true, Color.ORANGE,
            new ImageIcon(getClass().getResource("Saturn.gif")) },
        { "Uranus", 25559.0, 17, true, Color.BLUE,
            new ImageIcon(getClass().getResource("Uranus.gif")) },
        { "Neptune", 24766.0, 8, true, Color.BLUE,
            new ImageIcon(getClass().getResource("Neptune.gif")) },
        { "Pluto", 1137.0, 1, false, Color.BLACK,
            new ImageIcon(getClass().getResource("Pluto.gif")) } };
}
```

```
public PlanetTableFrame()
{
    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

    TableModel model = new DefaultTableModel(cells, columnNames)
    {
        public Class<?> getColumnClass(int c)
        {
            return cells[0][c].getClass();
        }
    };

    table = new JTable(model);

    table.setRowHeight(100);
    table.getColumnModel().getColumn(COLOR_COLUMN).setMinWidth(250);
    table.getColumnModel().getColumn(IMAGE_COLUMN).setMinWidth(100);

    final TableRowSorter<TableModel> sorter = new TableRowSorter<>(model);
    table.setRowSorter(sorter);
    sorter.setComparator(COLOR_COLUMN, new Comparator<Color>()
    {
        public int compare(Color c1, Color c2)
        {
            int d = c1.getBlue() - c2.getBlue();
            if (d != 0) return d;
            d = c1.getGreen() - c2.getGreen();
            if (d != 0) return d;
            return c1.getRed() - c2.getRed();
        }
    });
    sorter.setSortable(IMAGE_COLUMN, false);
    add(new JScrollPane(table), BorderLayout.CENTER);

    removedRowIndices = new HashSet<>();
    removedColumns = new ArrayList<>();

    final RowFilter<TableModel, Integer> filter = new RowFilter<TableModel, Integer>()
    {
        public boolean include(Entry<? extends TableModel, ? extends Integer> entry)
        {
            return !removedRowIndices.contains(entry.getIdentifer());
        }
    };
}

// tworzy menu

JMenuBar menuBar = new JMenuBar();
setJMenuBar(menuBar);

JMenu selectionMenu = new JMenu("Selection");
menuBar.add(selectionMenu);

rowsItem = new JCheckBoxMenuItem("Rows");
columnsItem = new JCheckBoxMenuItem("Columns");
cellsItem = new JCheckBoxMenuItem("Cells");

rowsItem.setSelected(table.getRowSelectionAllowed());
```

```

columnsItem.setSelected(table.getColumnNameSelectionAllowed());
cellsItem.setSelected(table.getCellSelectionEnabled());

rowsItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        table.clearSelection();
        table.setRowSelectionAllowed(rowsItem.isSelected());
        updateCheckboxMenuItems();
    }
});
selectionMenu.add(rowsItem);

columnsItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        table.clearSelection();
        table.setColumnSelectionAllowed(columnsItem.isSelected());
        updateCheckboxMenuItems();
    }
});
selectionMenu.add(columnsItem);

cellsItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        table.clearSelection();
        table.setCellSelectionEnabled(cellsItem.isSelected());
        updateCheckboxMenuItems();
    }
});
selectionMenu.add(cellsItem);

JMenu tableMenu = new JMenu("Edit");
menuBar.add(tableMenu);

JMenuItem hideColumnsItem = new JMenuItem("Hide Columns");
hideColumnsItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        int[] selected = table.getSelectedColumns();
        TableColumnModel columnModel = table.getColumnModel();

        // usuwa kolumny z widoku tabeli, począwszy od
        // najwyższego indeksu, aby nie zmieniać numerów kolumn

        for (int i = selected.length - 1; i >= 0; i--)
        {
            TableColumn column = columnModel.getColumn(selected[i]);
            table.removeColumn(column);

            // przechowuje ukryte kolumny do kolejnej prezentacji
            removedColumns.add(column);
        }
    }
});

```

```
        }
    });
tableMenu.add(hideColumnsItem);

JMenuItem showColumnsItem = new JMenuItem("Show Columns");
showColumnsItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        // przywraca wszystkie usunięte dotąd kolumny
        for (TableColumn tc : removedColumns)
            table.addColumn(tc);
        removedColumns.clear();
    }
});
tableMenu.add(showColumnsItem);

JMenuItem hideRowsItem = new JMenuItem("Hide Rows");

hideRowsItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        int[] selected = table.getSelectedRows();
        for (int i : selected)
            removedRowIndices.add(table.convertRowIndexToModel(i));
        sorter.setRowFilter(filter);
    }
});
tableMenu.add(hideRowsItem);

JMenuItem showRowsItem = new JMenuItem("Show Rows");
showRowsItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        removedRowIndices.clear();
        sorter.setRowFilter(filter);
    }
});
tableMenu.add(showRowsItem);

JMenuItem printSelectionItem = new JMenuItem("Print Selection");
printSelectionItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        int[] selected = table.getSelectedRows();
        System.out.println("Selected rows: " + Arrays.toString(selected));
        selected = table.getSelectedColumns();
        System.out.println("Selected columns: " + Arrays.toString(selected));
    }
});
tableMenu.add(printSelectionItem);
}

private void updateCheckboxMenuItems()
```

```

    {
        rowsItem.setSelected(table.getRowSelectionAllowed());
        columnsItem.setSelected(table.getColumnSelectionAllowed());
        cellsItem.setSelected(table.getCellSelectionEnabled());
    }
}

```

**API javax.swing.table.TableModel 1.2**

- **Class getColumnClass(int columnIndex)**  
zwraca klasę wartości danej kolumny. Informacja ta jest używana podczas sortowania i wyświetlania tabeli.

**API javax.swing.JTable 1.2**

- **TableColumnModel getColumnModel()**  
zwraca „model kolumn” opisujący porządek kolumn w tabeli.
- **void setAutoResizeMode(int mode)**  
określa tryb zmiany szerokości kolumn.  
*Parametry:* mode jedna z wartości AUTO\_RESIZE\_OFF, AUTO\_RESIZE\_NEXT\_COLUMN, AUTO\_RESIZE\_SUBSEQUENT\_COLUMNS, AUTO\_RESIZE\_LAST\_COLUMN, AUTO\_RESIZE\_ALL\_COLUMNS.
- **int getRowMargin()**
- **void setRowMargin(int margin)**  
zwraca lub określa wolną przestrzeń między sąsiednimi wierszami.
- **int getRowHeight()**
- **void setRowHeight(int height)**  
zwraca lub określa wysokość wszystkich wierszy tabeli.
- **int getRowHeight(int row)**
- **void setRowHeight(int row, int height)**  
zwraca lub określa wysokość danego wiersza tabeli.
- **ListSelectionModel getSelectionModel()**  
zwraca model wyboru, który umożliwia następnie określenie trybu wyboru wierszy, kolumn lub komórek.
- **boolean getRowSelectionAllowed()**
- **void setRowSelectionAllowed(boolean b)**  
zwraca lub nadaje wartość właściwości rowSelectionAllowed. Wartość true oznacza, że kliknięcie komórki tabeli powoduje wybranie całego wiersza.
- **boolean getColumnSelectionAllowed()**

- `void setColumnSelectionAllowed(boolean b)`  
zwraca lub nadaje wartość właściwości `columnSelectionAllowed`. Wartość true oznacza, że kliknięcie komórki tabeli powoduje wybranie całego wiersza.
- `boolean getCellSelectionEnabled()`  
zwraca wartość true, jeśli dozwolony jest wybór wierszy i kolumn tabeli (czyli obie właściwości `rowSelectionEnabled` i `columnSelectionEnabled` mają wartość true).
- `void setCellSelectionEnabled(boolean b)`  
nadaje właściwościom `rowSelectionEnabled` i `columnSelectionEnabled` wartość b.
- `void addColumn(TableColumn column)`  
dodaje kolumnę do widoku tabeli.
- `void moveColumn(int from, int to)`  
przesuwa kolumnę o indeksie `from` w taki sposób, że jej indeksem staje się `to`. Operacja ta dotyczy jedynie widoku tabeli.
- `void removeColumn(TableColumn column)`  
usuwa kolumnę z widoku tabeli.
- `int convertRowIndexToModel(int index) 6`
- `int convertColumnIndexToModel(int index)`  
zwraca indeks modelu dla wiersza lub kolumny o podanym indeksie. Wartość ta różni się od indeksu, gdy wiersze są filtrowane lub sortowane albo gdy kolumny są przesuwane lub usuwane.
- `void setRowSorter(RowSorter<? extends TableModel> sorter)`  
określa obiekt `RowSorter`.

**API `javax.swing.table.TableColumnModel 1.2`**

- `TableColumn getColumn(int index)`  
zwraca obiekt kolumny tabeli o danym indeksie.

**API `javax.swing.table.TableColumn 1.2`**

- `TableColumn(int modelColumnIndex)`  
tworzy obiekt reprezentujący kolumnę tabeli o danym indeksie.
- `void setPreferredWidth(int width)`
- `void setMinWidth(int width)`
- `void setMaxWidth(int width)`  
Określają preferowaną, najmniejszą i największą szerokość danej kolumny.

- void setWidth(int width)  
ustawia bieżącą szerokość danej kolumny.
- void setResizable(boolean b)  
jeśli b posiada wartość true, to użytkownik może zmieniać szerokość kolumny.

**API** javax.swing.ListSelectionModel 1.2

- void setSelectionMode(int mode)  
określa tryb wyboru.  
*Parametry:* mode jedna z wartości SINGLE\_SELECTION,  
SINGLE\_INTERVAL\_SELECTION i MULTIPLE\_INTERVAL\_SELECTION.

**API** javax.swing.DefaultRowSorter<M, I> 6

- void setComparator(int column, Comparator<?> comparator)  
określa komparator używany dla danej kolumny.
- void setSortable(int column, boolean enabled)  
włącza lub wyłącza możliwość sortowania kolumny.
- void setRowFilter(RowFilter<? super M, ? super I> filter)  
określa filtr wierszy.

**API** javax.swing.table.TableRowSorter<M extends TableModel> 6

- void setStringConverter(TableStringConverter stringConverter)  
określa konwerter łańcuchów używany podczas sortowania i filtrowania.

**API** javax.swing.table.TableStringConverter<M extends TableModel> 6

- abstract String toString(TableModel model, int row, int column)  
metodę tę należy zastąpić własną wersją, aby dokonać konwersji wartości modelu dla danej komórki na łańcuch znaków.

**API** javax.swing.RowFilter<M, I> 6

- boolean include(RowFilter.Entry<? extends M, ? extends I> entry)  
metodę tę należy zastąpić własną wersją, aby określić, które wiersze mają zostać zachowane w tabeli.
- static <M, I> RowFilter<M, I> numberFilter(RowFilter.ComparisonType type, Number number, int... indices)

- static <M,I> RowFilter<M,I> dateFilter(RowFilter.ComparisonType type, Date date, int... indices)
 

zwraca filtr dopuszczający wiersze zawierające wartość, którą podane porównanie potrafi uzgodnić z podaną wartością numeryczną lub datą. Typem porównania może być jedna ze stałych EQUAL, NOT\_EQUAL, AFTER lub BEFORE. Jeśli podane zostały indeksy kolumn, to przeszukiwane będą jedynie kolumny o tych indeksach. W przypadku filtra numerycznego klasa wartości komórki musi zgadzać się z klasą podanej wartości.
- static <M,I> RowFilter<M,I> regexFilter(String regex, int... indices)
 

zwraca filtr dopuszczający wiersze zawierające łańcuch znaków pasujący do podanego wyrażenia regularnego. Jeśli podane zostały indeksy kolumn, to przeszukiwane będą jedynie kolumny o tych indeksach. W przeciwnym razie przeszukiwane są wszystkie kolumny. Zwróćmy uwagę, że uzgadniany jest łańcuch zwracany przez metodę getStringValue obiektu RowFilter.Entry.
- static <M,I> RowFilter<M,I> andFilter(Iterable<? extends RowFilter<? super M,? super I>> filters)
 

zwraca filtr dopuszczający wiersze dopuszczone przez wszystkie filtry lub co najmniej jeden filtr.
- static <M,I> RowFilter<M,I> orFilter(Iterable<? extends RowFilter<? super M,? super I>> filters)
 

zwraca filtr dopuszczający wiersze, które nie zostały dopuszczone przez podany filtr.

**API | javax.swing.RowFilter.Entry<M, I>** 6

- I getIdentifier()
 

zwraca identyfikator wiersza.
- M getModel()
 

zwraca model wiersza.
- Object getValue(int index)
 

zwraca wartość o podanym indeksie należącą do wiersza.
- int getCount()
 

zwraca liczbę wartości w wierszu.
- String getStringValue()
 

zwraca wartość o podanym indeksie należącą do wiersza, przekształconą na łańcuch znaków. Klasa TableRowSorter tworzy wiersze, których metoda getStringValue wywołuje konwerter łańcuchów należący do obiektu sortującego.

## 6.2.4. Rysowanie i edycja komórek

Na początku części 6.2.3.2., „Dostęp do kolumn tabeli”, pokazaliśmy, że typ kolumny określa sposób prezentacji jej komórek. W przypadku typów Boolean czy Icon istnieją domyślne obiekty rysujące, które prezentują zawartość komórki jako pole wyboru lub ikonę. W przypadku innych typów dostarczyć możemy własnych obiektów rysujących komórki tabeli.

Obiekty rysujące komórki tabeli przypominają w działaniu obiekty rysujące komórki list, które omówiliśmy już wcześniej. Implementują one interfejs TableCellRenderer posiadający pojedynczą metodę:

```
Component getTableCellRendererComponent(JTable table,
    Object value, boolean isSelected, boolean hasFocus,
    int row, int column)
```

Metoda ta wywoływana jest za każdym razem, kiedy komórka tabeli wymaga narysowania. Zwraca komponent, którego metoda paint wykorzystywana jest do narysowania komórki tabeli.

Tabela przedstawiona na rysunku 6.12 zawiera komórki typu Color. Aby wyświetlić komórkę typu Color, wystarczy, że zwrócimy panel, którego kolor tła ustawiony będzie zgodnie z kolorem określonym przez obiekt Color znajdujący się w komórce tabeli. Obiekt ten zostanie przekazany metodzie za pośrednictwem parametru value.

```
class ColorTableCellRenderer extends JPanel implements TableCellRenderer
{
    public Component getTableCellRendererComponent(JTable table,
        Object value, boolean isSelected, boolean hasFocus,
        int row, int column)
    {
        setBackground((Color) value);
        if (hasFocus)
            setBorder(
                UIManager.getBorder(Table.focusCellHighlightBorder));
        else
            setBorder(null);
    }
}
```

Jak łatwo zauważyc, obiekt rysujący nadaje komórce obramowanie, gdy została ona wybrana przez użytkownika. (Odpowiednie obramowanie zwraca nam obiekt klasy UIManager. Aby odnaleźć właściwy klucz obramowania, podejrzeliśmy kod źródłowy klasy TableCell->Renderer).

W ogólnym przypadku również tło komórki powinno wskazywać, że jest ona aktualnie wybrana. W naszym przykładzie pominiemy ten element, ponieważ kolidowałby on z wyświetlonym kolorem. Przykład ListRenderingTest pokazany na listingu 6.4 ilustruje sposób sygnalizacji wyboru przez obiekt rysujący.



Jeśli obiekt rysujący wyświetla łańcuch znaków lub ikonę, to możemy go utworzyć, rozszerzając klasę DefaultTableCellRenderer, która wykona działania związane z obsługą stanu wyboru i przeglądania komórki.

**Rysunek 6.12.**

Tabela wykorzystująca obiekty rysujące

Planet	Radius	Image	Gaseous	Color	Image
Mars	3,397	● ●	<input type="checkbox"/>		
Jupiter	71,492		<input checked="" type="checkbox"/>		
Saturn	60,268		<input checked="" type="checkbox"/>		

Musimy jeszcze przekazać do tabeli informację, aby skorzystała z obiektu rysującego powyższej klasy w przypadku wszystkich komórek zawierających obiekty klasy Color. Użyjemy w tym celu metody setDefaultRenderer klasy JTable, której parametrami będą obiekt klasy Class i obiekt rysujący.

```
table.setDefaultRenderer(Color.class,
    new ColorTableCellRenderer());
```

Odtąd nowy obiekt rysujący będzie wykorzystywany dla obiektów danego typu.

Jeśli chcemy wybrać obiekt rysujący w oparciu o inne kryterium, musimy utworzyć klasę pochodną klasy JTable i zastąpić jej metodę getCellRenderer własną wersją.

#### 6.2.4.1. Wyświetlanie nagłówka

Aby wyświetlić ikonę w nagłówku kolumny, użyjemy następującego wywołania:

```
moonColumn.setHeaderValue(new ImageIcon("Moons.gif"));
```

Jednak tabela sama nie potrafi wybrać właściwego obiektu rysującego dla nagłówka. Obiekt ten należy zainstalować ręcznie. Na przykład: aby w nagłówku kolumny wyświetlana była ikona, wywołamy:

```
moon.Column.setHeaderRenderer(table.getDefaultRenderer(ImageIcon.class));
```

#### 6.2.4.2. Edycja komórek

Aby umożliwić edycję komórek, model tabeli musi definiować metodę isCellEditable wskazującą, czy dana komórka tabeli może być edytowana. Zwykle zezwala się raczej od razu na edycję całej kolumny niż poszczególnych komórek. W programie przykładowym pozwolimy na edycję komórek czterech kolumn tabeli.

```
public boolean isCellEditable(int r, int c)
{
    return c == PLANET_COLUMN
        || c == MOONS_COLUMN
        || c == GASEOUS_COLUMN
        || c == COLOR_COLUMN;
}
```



Klasa `AbstractTableModel` definiuje metodę `isCellEditable`, która zawsze zwraca wartość `false`. Klasa `DefaultTableModel` zastępuje ją z kolejną implementacją, która zawsze zwraca wartość `true`.

Jeśli uruchomimy program, którego tekst źródłowy przedstawiają listingi 6.8 do 6.11, to zauważymy, że w kolumnie *Gaseous* możemy edytować pole wyboru. Natomiast w kolumnie *Moons* możemy wybierać wartość z listy rozwijalnej (patrz rysunek 6.13). Wkrótce pokażemy, w jaki sposób zainstalować listę rozwijalną jako edytor wartości komórki.

**Rysunek 6.13.**  
Edytor komórki

Planet	Radius	Gaseous	Color	Image
Mercury	2,440	0		
Venus	6,052	0 1 2 3 4 5 6 7		
Earth	6,378	1		

**Listing 6.8.** `tableCellRender/TableCellRenderFrame.java`

```
package tableCellRender;

import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;

/**
 * Ramka zawierająca tabelę danych o planetach.
 */
public class TableCellRenderFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 600;
    private static final int DEFAULT_HEIGHT = 400;

    public TableCellRenderFrame()
    {

```

```
setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

TableModel model = new PlanetTableModel();
JTable table = new JTable(model);
table.setRowSelectionAllowed(false);

// instaluje obiekty rysujące i edytory
table.setDefaultRenderer(Color.class, new ColorTableCellRenderer());
table.setDefaultEditor(Color.class, new ColorTableCellEditor());

JComboBox<Integer> moonCombo = new JComboBox<>();
for (int i = 0; i <= 20; i++)
    moonCombo.addItem(i);

TableColumnModel columnModel = table.getColumnModel();
TableColumn moonColumn = columnModel.getColumn(PlanetTableModel.MOONS_COLUMN);
moonColumn.setCellEditor(new DefaultCellEditor(moonCombo));
moonColumn.setHeaderRenderer(table.getDefaultRenderer(ImageIcon.class));
moonColumn.setHeaderValue(new ImageIcon(getClass().getResource("Moons.gif")));

// pokazuje tabelę

table.setRowHeight(100);
add(new JScrollPane(table), BorderLayout.CENTER);
}
}
```

---

**Listing 6.9.** *tableCellRender/PlanetTableModel.java*

```
package tableCellRender;

import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;

/**
 * Model tabeli planet określający wartości danych
 * oraz sposób ich rysowania i edycji.
 */
public class PlanetTableModel extends AbstractTableModel
{
    public static final int PLANET_COLUMN = 0;
    public static final int MOONS_COLUMN = 2;
    public static final int GASEOUS_COLUMN = 3;
    public static final int COLOR_COLUMN = 4;

    private Object[][] cells = {
        { "Mercury", 2440.0, 0, false, Color.YELLOW,
            new ImageIcon(getClass().getResource("Mercury.gif")) },
        { "Venus", 6052.0, 0, false, Color.YELLOW,
            new ImageIcon(getClass().getResource("Venus.gif")) },
        { "Earth", 6378.0, 1, false, Color.BLUE,
            new ImageIcon(getClass().getResource("Earth.gif")) },
        { "Mars", 3397.0, 2, false, Color.RED,
            new ImageIcon(getClass().getResource("Mars.gif")) },
        { "Jupiter", 71492.0, 16, true, Color.ORANGE,
            new ImageIcon(getClass().getResource("Jupiter.gif")) },
    };
}
```

```

        { "Saturn", 60268.0, 18, true, Color.ORANGE,
          new ImageIcon(getClass().getResource("Saturn.gif")) },
        { "Uranus", 25559.0, 17, true, Color.BLUE,
          new ImageIcon(getClass().getResource("Uranus.gif")) },
        { "Neptune", 24766.0, 8, true, Color.BLUE,
          new ImageIcon(getClass().getResource("Neptune.gif")) },
        { "Pluto", 1137.0, 1, false, Color.BLACK,
          new ImageIcon(getClass().getResource("Pluto.gif")) } };

private String[] columnNames = { "Planet", "Radius", "Moons", "Gaseous", "Color",
→"Image" };

public String getColumnName(int c)
{
    return columnNames[c];
}

public Class<?> getColumnClass(int c)
{
    return cells[0][c].getClass();
}

public int getColumnCount()
{
    return cells[0].length;
}

public int getRowCount()
{
    return cells.length;
}

public Object getValueAt(int r, int c)
{
    return cells[r][c];
}

public void setValueAt(Object obj, int r, int c)
{
    cells[r][c] = obj;
}

public boolean isCellEditable(int r, int c)
{
    return c == PLANET_COLUMN || c == MOONS_COLUMN || c == GASEOUS_COLUMN || c ==
→COLOR_COLUMN;
}
}

```

**Listing 6.10.** *tableCellRender/ColorTableCellRenderer.java*

```

package tableCellRender;

import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;

```

```


    /**
     * Klasa obiektu rysującego kolorowy panel wewnątrz komórki.
     */
    public class ColorTableCellRenderer extends JPanel implements TableCellRenderer
    {
        public Component getTableCellRendererComponent(JTable table, Object value,
            boolean isSelected,
            boolean hasFocus, int row, int column)
        {
            setBackground((Color) value);
            if (hasFocus)
                setBorder(UIManager.getBorder("Table.focusCellHighlightBorder"));
            else setBorder(null);
            return this;
        }
    }


```

**Listing 6.11.** *tableCellRender/ColorTableCellEditor.java*

```


package tableCellRender;

import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.util.*;
import javax.swing.*;
import javax.swing.table.*;

/**
 * Edytor otwierający okno dialogowe wyboru koloru.
 */
public class ColorTableCellEditor extends AbstractCellEditor implements
TableCellEditor
{
    private JColorChooser colorChooser;
    private JDialog colorDialog;
    private JPanel panel;

    public ColorTableCellEditor()
    {
        panel = new JPanel();
        // przygotowuje okno dialogowe

        colorChooser = new JColorChooser();
        colorDialog = JColorChooser.createDialog(null, "Planet Color", false, colorChooser,
            EventHandler.create(ActionListener.class, this, "stopCellEditing"),
            EventHandler.create(ActionListener.class, this, "cancelCellEditing"));
    }

    public Component getTableCellEditorComponent(JTable table, Object value, boolean
        isSelected,
        int row, int column)
    {
        // Tutaj uzyskujemy bieżącą wartość Color.
        // Przechowujemy ją w obiekcie okna dialogowego.
        colorChooser.setColor((Color) value);
        return panel;
    }
}


```

```

        }

    public boolean shouldSelectCell(EventObject anEvent)
    {
        // rozpoczęcie edycji
        colorDialog.setVisible(true);

        // informuje metodę wywołującą o rozpoczęciu edycji
        return true;
    }

    public void cancelCellEditing()
    {
        // edycja przerwana - zamyka okno dialogowe
        colorDialog.setVisible(false);
        super.cancelCellEditing();
    }

    public boolean stopCellEditing()
    {
        // edycja zakończona — zamyka okno dialogowe
        colorDialog.setVisible(false);
        super.stopCellEditing();

        // informuje metodę wywołującą, że wartość koloru jest dozwolona
        return true;
    }

    public Object getCellEditorValue()
    {
        return colorChooser.getColor();
    }
}

```

Wybierając komórki pierwszej kolumny tabeli, możemy zmieniać ich zawartość, wpisując dowolny ciąg znaków.

Wszystkie powyższe przykłady stanowią odmiany klasy DefaultCellEditor. Obiekt klasy DefaultCellEditor może zostać utworzony z obiektu klasy JTextField, JCheckBox lub JComboBox. Klasa JTable sama automatycznie instaluje edytor pól wyboru dla kolumn klasy Boolean oraz edytor pól tekstowych dla pozostałych typów kolumn, które nie posiadają własnego obiektu rysującego. Edytory pól tekstowych pozwalają w takim przypadku na modyfikację łańcucha znaków będącego wynikiem zastosowania metody `toString` do obiektu zwróconego przez metodę `getValueAt` modelu tabeli.

Po zakończeniu edycji komórki jej nowa wartość zostaje pobrana przez wywołanie metody `getCellEditorValue` edytora komórki. Metoda ta powinna zwrócić wartość odpowiedniego typu (czyli typu zwracanego przez metodę modelu `getColumnType`).

Aby użyć w komórkach tabeli edytora listy rozwijalnej, musimy go zainstalować samodzielnie, ponieważ klasa JTable nie może sama ustalić zbioru wartości dla danego typu komórki. W przypadku komórek kolumny *Moons* naszej tabeli umożliwimy użytkownikowi wybór wartości z przedziału od 0 do 20. Poniżej fragment kodu inicjujący listę rozwijalną.

```
JComboBox moonCombo = new JComboBox();
for (int i = 0; i <= 20; i++)
    moonCombo.addItem(i);
```

Następnie utworzymy obiekt klasy `DefaultCellEditor`, przekazując listę jako parametr jego konstruktora:

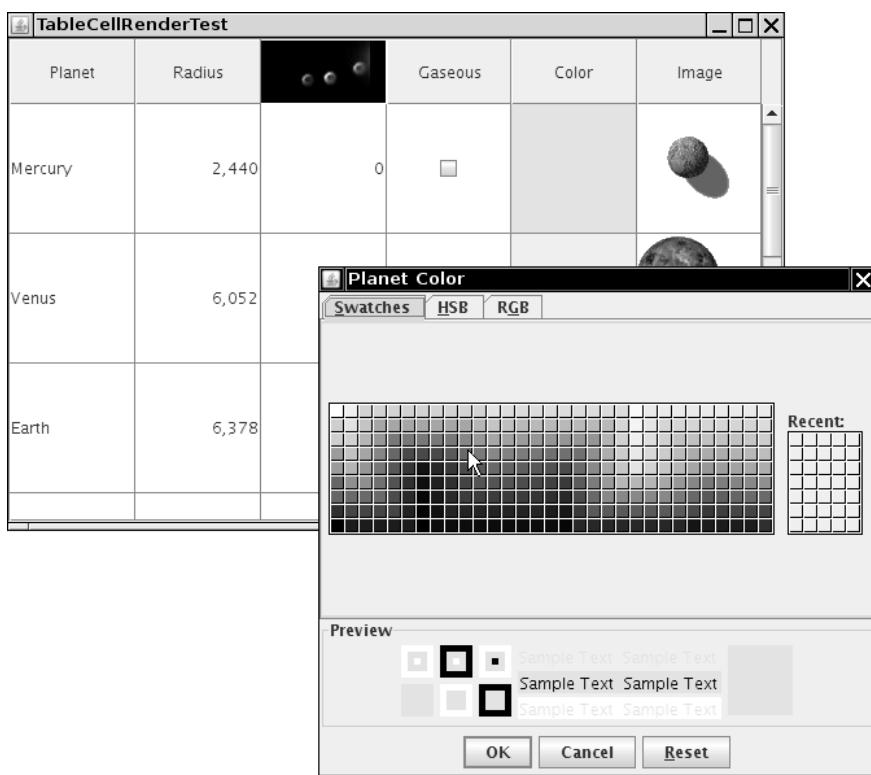
```
TableCellEditor moonEditor = new DefaultCellEditor(moonCombo);
```

Musimy jeszcze zainstalować utworzony edytor. W przeciwnieństwie do edytora kolorów nie zwiążemy go z określonym *typem*, ponieważ nie chcemy, by używany był przez tabelę dla wszystkich komórek typu `Integer`. Zamiast tego zainstalujemy go jedynie dla określonej kolumny tabeli.

```
moonColumn.setCellEditor(moonEditor);
```

### 6.2.4.3. Tworzenie własnych edytorów

Jeśli uruchomimy przykładowy program i wybierzemy za pomocą myszy komórkę zawierającą kolor, to otworzy się okno dialogowe wyboru koloru. Użytkownik może wybrać nowy kolor i zaakceptować go przyciskiem *OK*, co spowoduje zmianę koloru komórki tabeli (patrz rysunek 6.14).



Rysunek 6.14. Wybór koloru komórki

Edytor koloru komórek nie jest standardowym edytorem tabeli. Aby utworzyć własny edytor, należy zaimplementować interfejs TableCellEditor. Jest to dość pracochłonne zadanie i dla tego wersja Java SE 1.3 wprowadziła klasę AbstractCellEditor zawierającą implementację obsługi zdarzeń.

Metoda getTableCellEditorComponent interfejsu TableCellEditor pobiera komponent rysujący komórkę tabeli. Jest zdefiniowana tak samo jak metoda getTableCellRendererComponent interfejsu TableCellRenderer, z tą różnicą, że nie posiada parametru hasFocus. Ponieważ komórka jest edytowana, to automatycznie przyjmuje się, że wartością tego parametru jest true. Komponent edytora *zastępuje* obiekt rysujący podczas edycji komórki. W naszym przykładzie wywołanie metody zwraca pusty, niepokolorowany panel, sygnalizując użytkownikowi w ten sposób, że komórka jest edytowana.

Edytor musi rozpoczęć swoje działanie po kliknięciu komórki przez użytkownika.

Klasa JTable wywołuje edytor dla danego zdarzenia (na przykład kliknięcia myszą), aby sprawdzić, czy spowoduje ono rozpoczęcie procesu edycji. Klasa AbstractCellEditor definiuje metodę akceptującą wszystkie zdarzenia.

```
public boolean isCellEditable(EventObject anEvent)
{
    return true;
}
```

Jeśli zastąpimy tę metodę, tak by zwracała wartość false, to tabela w ogóle nie umieści komponentu edytora.

Po zainstalowaniu edytora wywoływana jest metoda shouldSelectCell, prawdopodobnie dla tego samego zdarzenia. Metoda ta rozpoczęć powinna proces edycji, na przykład otwierając okno dialogowe.

```
public boolean shouldSelectCell(EventObject anEvent)
{
    colorDialog.setVisible(true);
    return true;
}
```

Jeśli użytkownik anuluje proces edycji, wywołana zostanie metoda cancelCellEditing. Jeśli użytkownik kliknie inną komórkę tabeli, wywołana zostanie metoda stopCellEditing. W obu przypadkach powinniśmy zamknąć okno dialogowe. Wywołanie metody stopCellEditing oznacza, że tabela chce zachować wartość zmodyfikowaną w procesie edycji. Metoda powinna zwrócić wartość true, jeśli wartość komórki jest dozwolona. W przypadku wyboru kolorów dozwolona będzie dowolna wartość. Jednak jeśli tworzymy edytor innych rodzajów danych, to powinniśmy zawsze sprawdzać, czy powstała w procesie edycji wartość jest dozwolona.

Po zakończeniu edycji należy także wywołać metodę klasy bazowej, która obsługuje dla nas zdarzenia.

```
public void cancelCellEditing()
{
    colorDialog.setVisible(false);
    super.cancelCellEditing();
}
```

Musimy dostarczyć także metodę, która umożliwi pobranie wartości powstałej w procesie edycji:

```
public Object getCellEditorValue()
{
    return colorChooser.getColor();
}
```

Podsumowując, edytor powinien:

- 1.** Rozszerzać klasę `AbstractCellEditor` i implementować interfejs `TableCellEditor`.
- 2.** Definiować metodę `getTableCellEditorComponent`, która zwraca nieinteraktywny komponent, gdy edytor otworzy własne okno dialogowe lub komponent umożliwiający edycję wewnątrz komórki (na przykład lista rozwijalna bądź pole tekstowe).
- 3.** Definiować metody `shouldSelectCell`, `stopCellEditing` i `cancelCellEditing` obsługujące rozpoczęcie, zakończenie i anulowanie procesu edycji; metody `stopCellEditing` i `cancelCellEditing` wywoływać powinny te same metody klasy bazowej, aby zapewnić powiadomienie obiektów nasłuchujących.
- 4.** Definiować metodę `getCellEditorValue` zwracającą nową wartość komórki powstałą w procesie edycji.

Na koniec musimy jeszcze wywołać metody `stopCellEditing` i `cancelCellEditing`, gdy użytkownik zakończy edycję. Tworząc okno dialogowe wyboru kolorów, zainstalujemy metody wywoływane zwrotnie, które spowodują odpowiednie zdarzenia.

```
colorDialog = JColorChooser.createDialog(null,
    "Planet Color", false, colorChooser,
    EventHandler.create(ActionListener.class, this, "stopCellEditing"),
    EventHandler.create(ActionListener.class, this, "cancelCellEditing"));
```

W ten sposób zakończyliśmy implementację własnego edytora komórek.

Wiemy już, w jaki sposób umożliwić edycję komórek i zainstalować edytor. Pozostaje jeszcze tylko powiadomienie modelu tabeli o zmianie wartości edytowanej komórki. Po zakończeniu edycji komórki klasa `JTable` wywołuje następującą metodę modelu tabeli:

```
void setValueAt(Object value, int r, int c)
```

Musimy zastąpić tę metodę, aby przekazać do modelu nową wartość. Parametr `value` jest obiektem zwróconym przez edytor komórki. W przypadku edytora, który sami zaimplementowaliśmy, znamy typ obiektu zwróconego za pomocą metody `getCellEditorValue`. W przypadku edytora klasy `DefaultCellEditor` istnieją natomiast trzy możliwości. Może to być typ `Boolean`, jeśli edytor był polem wyboru, lub łańcuch znaków, jeśli edytorem było pole tekstowe lub obiekt wybrany przez użytkownika z listy rozwijalnej.

Jeśli obiekt `value` nie posiada odpowiedniego typu, należy go w taki przekształcić. Sytuacja taka najczęściej zdarza się, gdy w polu tekstowym edytowana jest liczba. W naszym przykładzie lista rozwijalna wypełniona została obiektami klasy `Integer`, dlatego też nie ma potrzeby konwersji typu.

**API javax.swing.JTable 1.2**

- TableCellRenderer getDefaultRenderer(Class<?> type)  
zwraca domyślny obiekt rysujący dla komórek podanego typu.
- TableCellEditor getDefaultEditor(Class<?> type)  
zwraca domyślny edytor dla komórek podanego typu.

**API javax.swing.table.TableCellRenderer 1.2**

- Component getTableCellRendererComponent(JTable table, Object value, boolean selected, boolean hasFocus, int row, int column)  
zwraca komponent, którego metoda paint wywoływana jest w celu narysowania komórki tabeli.

*Parametry:*



  
                   value         obiekt rysowanej komórki;  
                   selected      wartość true, jeśli komórka jest wybrana;  
                   hasFocus     wartość true, jeśli komórka jest przeglądana;  
                   row, column   wiersz i kolumna komórki.

**API javax.swing.table.TableColumn 1.2**

- void setCellEditor(TableCellEditor editor)
- void setCellRenderer(TableCellRenderer renderer)  
instaluje edytor i obiekt rysujący dla wszystkich komórek danej kolumny.
- void setHeaderRenderer(TableCellRenderer renderer)  
instaluje obiekt rysujący dla nagłówka danej kolumny.
- void setHeaderValue(Object value)  
określa wartość wyświetlaną w nagłówku danej kolumny.

**API javax.swing.DefaultCellEditor 1.2**

- DefaultCellEditor(JComboBox comboBox)  
tworzy edytor komórek wykorzystujący listę rozwijalną do wyboru wartości.

**API javax.swing.table.TableCellEditor 1.2**

- Component getTableCellEditorComponent(JTable table, Object value, boolean selected, int row, int column)  
zwraca komponent, którego metoda paint wywoywana jest w celu narysowania komórki tabeli.

*Parametry:*



  
                   value         obiekt rysowanej komórki;

selected      wartość true, jeśli komórka jest wybrana;  
 row, column    wiersz i kolumna komórki.

#### *javax.swing.CellEditor 1.2*

- `boolean isCellEditable(EventObject event)`  
zwraca wartość true, jeśli zdarzenie rozpoczęnie proces edycji komórki.
- `boolean shouldSelectCell(EventObject anEvent)`  
rozpoczyna proces edycji. Zwraca wartość true, jeśli edytowana komórka powinna zostać *wybrana*. Wartość false powinna być zwrócona, jeśli nie chcemy, by proces edycji zmieniał wybór komórek.
- `void cancelCellEditing()`  
przerywa proces edycji. Wartość powstała na skutek edycji może być porzucona.
- `boolean stopCellEditing()`  
kończy proces edycji. Wartość powstała na skutek edycji może być wykorzystana. Zwraca wartość true, jeśli wartość powstała na skutek edycji jest dozwolona i może być pobrana.
- `Object getCellEditorValue()`  
zwraca edytowaną wartość.
- `void addCellEditorListener(CellEditorListener l)`
- `void removeCellEditorListener(CellEditorListener l)`  
dodaje i usuwa obowiązkowy obiekt nasłuchujący edytora.

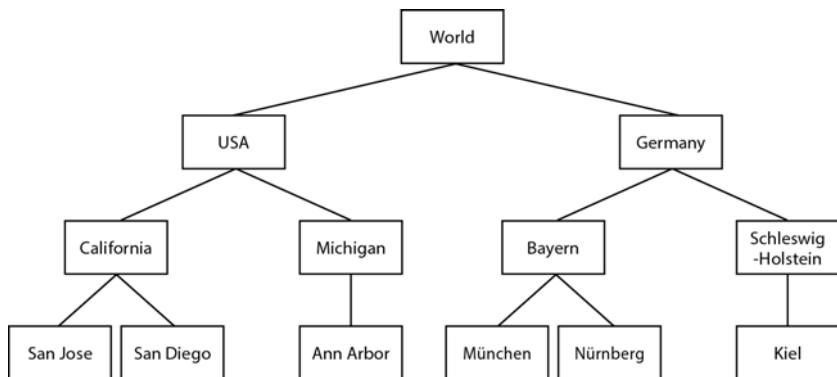
## 6.3. Drzewa

Każdy użytkownik komputera, którego system operacyjny posiada hierarchicznie zbudowany system plików, spotkał się w praktyce z jego reprezentacją za pomocą *drzewa*. Także w życiu codziennym często spotykamy struktury drzewiaste, takie jak hierarchie administracyjne państw, stanów, miast itd. (patrz rysunek 6.15).

W programach często trzeba zaprezentować dane w postaci struktury drzewiastej. Biblioteka Swing dostarcza w tym celu klasę `JTree`. Klasa `JTree` (wraz z klasami pomocniczymi) służy do tworzenia reprezentacji graficznej drzewa i przetwarzania akcji użytkownika polegających na rozwijaniu i zwijaniu węzłów drzewa. W podrozdziale tym nauczymy się korzystać z możliwości klasy `JTree`.

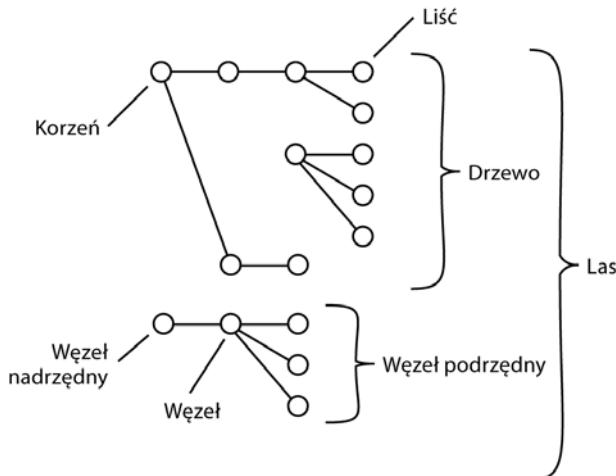
Podobnie jak w przypadku innych złożonych komponentów biblioteki Swing, skoncentrujemy się na omówieniu najczęstszych i najbardziej przydatnych przypadków zastosowań klasy `JTree`. Jeśli spotkasz się z nietypowym zastosowaniem drzewa, to polecamy książki *Graphic Java 2* napisaną przez Davida M. Geary'ego (Prentice-Hall, 1999), *Core Swing* autorstwa Kima Topleya (Prentice-Hall, 1999).

**Rysunek 6.15.**  
Hierarchia państw,  
stanów i miast



Zanim przejdziemy do konkretów, warto usystematyzować terminologię związaną z opisem drzew (patrz rysunek 6.16). Drzewo składa się z *węzłów*. Każdy węzeł jest albo *liściem*, albo posiada *węzły podzielne*. Każdy węzeł drzewa z wyjątkiem jego *korzenia*, posiada dokładnie jeden *węzeł nadzędny*. Każde drzewo posiada jeden korzeń. Czasami występuje kolekcja drzew, z których każda posiada własny korzeń. Nazywamy ją *lasem*.

**Rysunek 6.16.**  
Terminologia drzew



### 6.3.1. Najprostsze drzewa

Pierwszy przykładowy program wyświetlać będzie drzewo o kilku węzłach (patrz rysunek 6.18). Podobnie jak inne komponenty biblioteki Swing, także i klasa JTree jest przykładem zastosowania wzorca model-widok-nadzorca. W praktyce oznacza to, że komponentowi interfejsu użytkownika musimy dostarczyć model danych. W przypadku klasy JTree będzie on parametrem konstruktora:

```
TreeModel model = . . . ;
JTree tree = new JTree(model);
```



Dostępne są także konstruktory tworzące drzewo na podstawie kolekcji elementów:

```
JTree(Object[] nodes)
JTree(Vector<?> nodes)
JTree(Hashtable<?,?> nodes) // wartości tablicy stają się węzłami drzewa
```

Konstruktory te są jednak mało przydatne, gdyż tworzą las drzew, z których każde posiada jeden węzeł. Ostatni z konstruktorów jest wyjątkowo nieprzydatny, ponieważ węzły umieszczane są w drzewie w praktycznie przypadkowy sposób określony przez kody mieszające elementów.

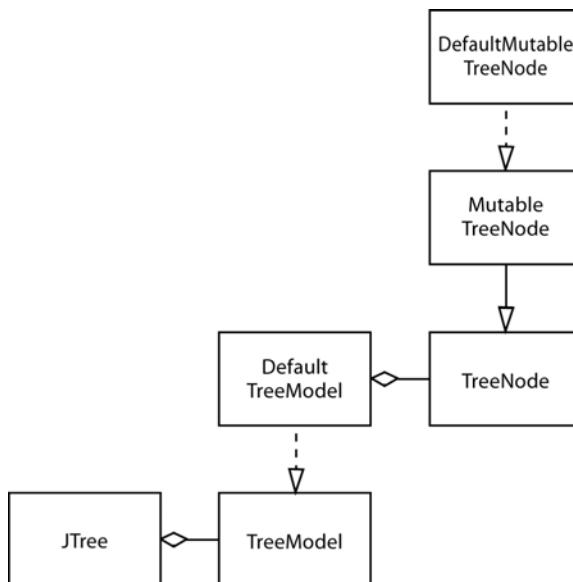
Model drzewa tworzymy, definiując klasę implementującą interfejs TreeModel. Z możliwości tej skorzystamy w dalszej części rozdziału, a na początku użyjemy klasy DefaultTreeModel dostarczanej przez bibliotekę Swing.

Kreując model tej klasy, musimy dostarczyć mu korzeń.

```
TreeNode root = . . . ;
DefaultTreeModel model = new DefaultTreeModel(root);
```

TreeNode jest kolejnym z interfejsów związanych z drzewami. Drzewo powstaje z węzłów dowolnych klas implementujących interfejs TreeNode. Na razie wykorzystamy w tym celu konkretną klasę DefaultMutableTreeNode udostępnianą przez bibliotekę Swing. Klasa ta implementuje interfejs MutableTreeNode będący specjalizacją interfejsu TreeNode (patrz rysunek 6.17).

**Rysunek 6.17.**  
Zależności pomiędzy klasami drzewa



Węzeł klasy DefaultMutableTreeNode przechowuje obiekt zwany *obiektem użytkownika*. Drzewo rysuje reprezentację obiektów użytkownika dla wszystkich węzłów. Dopóki nie zostanie zainstalowany specjalizowany obiekt rysujący węzły, to drzewo wyświetla po prostu łańcuch znaków będący wynikiem wywołania metody `toString`.

Nasz pierwszy przykład wykorzystywać będzie łańcuchy znaków jako obiekty użytkownika. W praktyce drzewo tworzą zwykle bardziej złożone obiekty, na przykład drzewo reprezentujące system plików składając się może z obiektów klasy File.

Obiekt użytkownika możemy przekazać jako parametr konstruktora węzła bądź później za pomocą metody setUserObject.

```
DefaultMutableTreeNode node
    = new DefaultMutableTreeNode(" Texas ")
...
node.setUserObject(" California ");
```

Następnie musimy utworzyć powiązania pomiędzy węzłami nadzewnętrznymi i podrzędnymi. Konstrukcję drzewa rozpoczęliśmy od korzenia, a później dodamy do niego węzły podrzędne:

```
DefaultMutableTreeNode root
    = new DefaultMutableTreeNode("World");
DefaultMutableTreeNode country
    = new DefaultMutableTreeNode("USA");
root.add(country);
DefaultMutableTreeNode state
    = new DefaultMutableTreeNode("California");
country.add(state);
```

Rysunek 6.18 pokazuje drzewo utworzone przez program.

**Rysunek 6.18.**  
Najprostsze drzewo



Po skonstruowaniu i połączeniu wszystkich węzłów możemy utworzyć model drzewa, przekazując mu korzeń, a następnie wykorzystać model do opracowania komponentu JTree.

```
DefaultTreeModel treeModel = new DefaultTreeModel(root);
JTree = new JTree(treeModel);
```

Mogemy nawet przekazać korzeń bezpośrednio konstruktorowi klasy JTree, który w takim przypadku sam utworzy domyślny model drzewa:

```
JTree = new JTree(root);
```

Listing 6.12 zawiera kompletny tekst źródłowy programu.

**Listing 6.12.** tree/SimpleTreeFrame.java

```
package tree;

import javax.swing.*;
import javax.swing.tree.*;
```

```

/**
 * Ramka zawierająca drzewo wyświetlające dane
 * "recznie" utworzonego modelu drzewa.
 */
public class SimpleTreeFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;

    public SimpleTreeFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        // tworzy model drzewa

        DefaultMutableTreeNode root = new DefaultMutableTreeNode("World");
        DefaultMutableTreeNode country = new DefaultMutableTreeNode("USA");
        root.add(country);
        DefaultMutableTreeNode state = new DefaultMutableTreeNode("California");
        country.add(state);
        DefaultMutableTreeNode city = new DefaultMutableTreeNode("San Jose");
        state.add(city);
        city = new DefaultMutableTreeNode("Cupertino");
        state.add(city);
        state = new DefaultMutableTreeNode("Michigan");
        country.add(state);
        city = new DefaultMutableTreeNode("Ann Arbor");
        state.add(city);
        country = new DefaultMutableTreeNode("Germany");
        root.add(country);
        state = new DefaultMutableTreeNode("Schleswig-Holstein");
        country.add(state);
        city = new DefaultMutableTreeNode("Kiel");
        state.add(city);

        // tworzy drzewo i umieszcza je w przewijalnym panelu

        JTree tree = new JTree(root);
        add(new JScrollPane(tree));
    }
}

```

Po uruchomieniu programu powstanie drzewo, które zobaczymy na rysunku 6.19. Widoczny będzie jedynie korzeń drzewa oraz jego węzły podrzędne. Wybranie myszą uchwytu węzła spowoduje rozwinięcie poddrzewa. Odcinki wystające z ikony uchwytu węzła skierowane są w prawo, gdy poddrzewo jest zwinięte i w dół w przeciwnym razie (patrz rysunek 6.20). Nie wiemy, co mieli na myśli projektanci wyglądu interfejsu zwanego Metal, tworząc ikonę uchwytu węzła, ale nam przypomina ona w działaniu klamkę drzwi. Kierujemy ją w dół, aby rozwinąć poddrzewo.



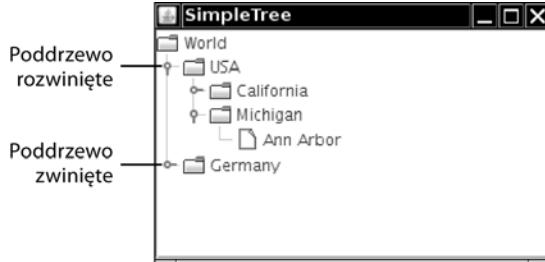
Wygląd drzewa zależy od wybranego wyglądu komponentów interfejsu użytkownika. Opisując dotąd drzewo, korzystaliśmy ze standardowego dla aplikacji Java wyglądu interfejsu Metal. W przypadku interfejsów Motif lub Windows uchwyty węzłów posiadają postać kwadratów zawierających znaki plus lub minus (patrz rysunek 6.21).

**Rysunek 6.19.**

Początkowy  
wygląd drzewa

**Rysunek 6.20.**

Zwinięte i rozwinięte  
poddrzewa

**Rysunek 6.21.**

Drzewo o wyglądzie  
Windows



Rysowanie linii łączących węzły możemy wyłączyć w poniższy sposób (patrz rysunek 6.22):

```
tree.putClientProperty("JTree.lineStyle", "None");
```

**Rysunek 6.22.**

Drzewo bez linii  
łączących węzły



Włączamy je z powrotem za pomocą wywołania

```
tree.putClientProperty("JTree.lineStyle", "Angled");
```

Istnieje także styl linii Horizontal pokazany na rysunku 6.23. Poziome linie oddzielają wtedy węzły podzielone korzenia. Nie jesteśmy przekonani o celowości takiego rozwiązania.

**Rysunek 6.23.**

Drzewo wykorzystujące styl linii Horizontal



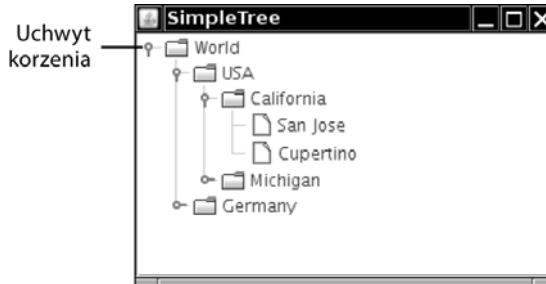
Domyślnie korzeń drzewa nie posiada uchwytu, który umożliwiałby zwinięcie całego drzewa. Możemy go dodać następująco:

```
tree.setShowsRootHandles(true);
```

Rysunek 6.24 pokazuje rezultat. Możemy teraz zwinąć całe drzewo do korzenia.

**Rysunek 6.24.**

Drzewo posiadające uchwyt korzenia



Możemy także w ogóle usunąć reprezentację korzenia drzewa, uzyskując w ten sposób *las* drzew, z których każdy posiada własny korzeń. Tworząc taki las, nadal musimy jednak połączyć wszystkie drzewa wspólnym korzeniem, który następnie ukrywamy, wywołując

```
tree.setRootVisible(false);
```

Efekt przedstawia rysunek 6.25. Występują na nim dwa korzenie oznaczone „USA” i „Germany”. W rzeczywistości są to węzły posiadające wspólny, ukryty korzeń.

**Rysunek 6.25.**

Las

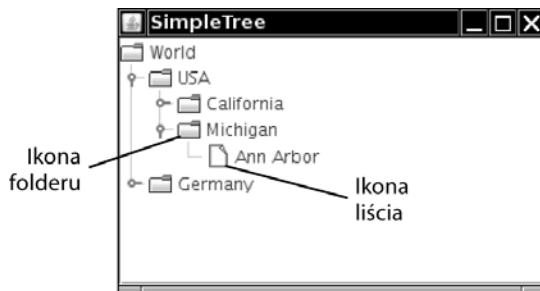


Zajmijmy się teraz liścimi drzewa. Zwróćmy uwagę, że ikony liści różnią się od ikon pozostała węzłów drzewa (patrz rysunek 6.26).

Każdy węzeł drzewa reprezentowany jest za pomocą pewnej ikony. Wyróżnić można trzy rodzaje takich ikon: ikona liścia, ikona węzła, którego poddrzewo jest rozwinięte i ikona węzła, którego poddrzewo jest zwinięte. Dla uproszczenia dwa ostatnie rodzaje ikon będziemy nazywać wspólnie ikonami folderu.

**Rysunek 6.26.**

Ikony liści



Obiekt rysujący węzły drzewa musi otrzymać informację, jakiej ikony powinien użyć dla danego węzła. Domyślny proces decyzyjny przebiega następująco: jeśli metoda `isLeaf` zwraca wartość `true`, to rysowana jest ikona liścia. W przeciwnym razie używana jest ikona folderu.

Metoda `isLeaf` klasy `DefaultMutableTreeNode` zwraca wartość `true` dla każdego węzła, który nie posiada węzłów podrzędnych. W ten sposób węzły posiadające węzły podrzędne otrzymują ikonę folderu, a pozostałe węzły — ikonę liścia.

Czasami rozwiązywanie takie nie jest jednak właściwe. Założymy, że do naszego drzewa dodaliśmy węzeł reprezentujący stan Montana i dopiero zastanawiamy się, jakie miasta powinniśmy umieścić jako jego węzły podrzędne. Nie chcemy przy tym, aby węzeł reprezentujący stan posiadał ikonę liścia, ponieważ koncepcyjnie przysługuje ona tylko miastom.

Klasa `JTree` nie wie, które węzły powinny być liściami. Decyduje o tym model drzewa. Jeśli węzeł, który nie posiada węzłów podrzędnych, nie jest liściem z punktu widzenia koncepcji drzewa, to model drzewa może zastosować inne kryterium rozpoznawania liści. Polega ono na wykorzystaniu właściwości węzła zezwalającej na posiadanie węzłów podrzędnych.

Dla węzłów, które nie będą posiadać węzłów podrzędnych, należy wtedy wywołać:

```
node.setAllowsChildren(false);
```

oraz poinformować model drzewa, by, decydując czy danym węzłem jest liściem, sprawdzał, czy może on posiadać węzły podrzędne. W tym celu wywołujemy metodę `setAsksAllowsChildren` klasy `DefaultTreeModel`:

```
model.setAsksAllowsChildren(true);
```

Od tego momentu węzły, które mogą posiadać węzły podrzędne, otrzymują ikonę folderu, a pozostałe ikonę liścia.

Takie kryterium doboru ikony możemy także uzyskać, tworząc obiekt klasy `JTree` za pomocą odpowiedniego konstruktora:

```
JTree tree = new JTree(root, true);
// węzły, które nie mogą posiadać węzłów podrzędnych otrzymają ikonę liścia
```

### javax.swing.JTree 1.2

■ `JTree(TreeModel model)`

tworzy drzewo na podstawie modelu.

- `JTree(TreeNode root)`
  - `JTree(TreeNode root, boolean asksAllowChildren)`

Tworzą drzewo, korzystając z domyślnego modelu i wyświetlając początkowo korzeń i jego węzły podrzędne.

<i>Parametry:</i>	root	korzeń,
	asksAllowChildren	jeśli posiada wartość true, to węzeł jest liściem, gdy może posiadać węzły podrzędne.

- void setShowsRootHandles(boolean b)  
jeśli b posiada wartość true, to korzeń posiada uchwyt umożliwiający zwinięcie drzewa.
  - void setRootVisible(boolean b)  
jeśli b posiada wartość true, to korzeń jest wyświetlany. W przeciwnym razie jest ukryty.

API *javax.swing.tree.TreeNode* 1-2

- `boolean isLeaf()`  
zwraca wartość true, jeśli dany węzeł reprezentuje liść na poziomie koncepcji.
  - `boolean getAllowsChildren()`  
zwraca wartość true, jeśli dany węzeł może posiadać węzły podziale.

API javax.swing.tree.MutableTreeNode 1-2

- `void setUserObject(Object userObject)`  
określa obiekt użytkownika dla danego wezła.

API javax.swing.tree.TreeModel 1-2

- `boolean isLeaf(Object node)`  
zwraca wartość true, jeśli wezel node zostanie wyświetlony jako liść.

API javax.swing.tree.DefaultTreeModel 1-2

- void setAsksAllowsChildren(boolean b)  
jeśli b posiada wartość true, to węzły wyświetlane są jako liście, w przypadku gdy metoda getAllowsChildren zwraca wartość false. Gdy b posiada wartość false, to węzły wyświetlane są jako liście, jeśli metoda isLeaf zwraca wartość true.

API javax.swing.tree.DefaultMutableTreeNode 1-2

- DefaultMutableTreeNode(Object userObject)  
tworzy wezel drzewa zawierajacy podany obiekt uzytkownika.

- void add(MutableTreeNode child)  
dodaje do węzła węzeł podrzędny.
- void setAllowsChildren(boolean b)  
jeśli b posiada wartość true, to do węzła mogą być dodawane węzły podrzędne.

**API javax.swing.JComponent 1.2**

- void putClientProperty(Object key, Object value)  
dodaje parę key/value do niewielkiej tablicy, którą zarządza każdy komponent.  
Mechanizm ten jest stosowany przez niektóre komponenty Swing w celu przechowywania specyficznych właściwości związanych z wyglądem komponentu.

### 6.3.1.1. Modyfikacje drzew i ścieżek drzew

Następny przykład programu ilustrować będzie sposób modyfikacji drzew. Rysunek 6.27 przedstawia interfejs użytkownika. Jeśli wybierzemy przycisk *Add Sibling* lub *Add Child*, to program doda do drzewa nowy węzeł opisany jako *New*. Jeśli wybierzemy przycisk *Delete*, to usuniemy wybrany węzeł.

**Rysunek 6.27.**  
Modyfikowanie  
drzewa



Aby zaimplementować takie zachowanie, musimy uzyskać informację o tym, który z węzłów drzewa jest aktualnie wybrany. Klasa JTTree zaskoczy nas z pewnością sposobem identyfikacji węzłów w drzewie. Wykorzystuje ona ścieżki do obiektów nazywane ścieżkami drzewa. Ścieżka taka zaczyna się zawsze od korzenia i zawiera sekwencje węzłów podrzędnych (patrz rysunek 6.28).

**Rysunek 6.28.**  
Ścieżka drzewa



Zastanawiać może, w jakim celu klasa JTTree potrzebuje całej ścieżki. Czy nie wystarczyłby jej obiekt klasy TreeNode i możliwość wywołania metody getParent? Okazuje się, że klasa JTTree nie ma pojęcia o istnieniu interfejsu TreeNode. Nie jest on wykorzystywany przez interfejs

`TreeModel`, a dopiero przez implementującą go klasę `DefaultTreeModel`. Możemy tworzyć też inne modele drzewa, które nie będą wykorzystywać interfejsu `TreeNode`. Więc zdarza się, że obiekty w takim modelu nie będą posiadać metod `getParent` i `getChild`, ale wykorzystywać będą inny sposób połączeń pomiędzy węzłami. Łączenie węzłów jest zadaniem modelu drzewa. Klasa `JTree` nie posiada żadnej wiedzy na temat natury tych połączeń. I to jest właśnie przyczyną, dla której klasa `JTree` musi wykorzystywać kompletne ścieżki drzewa.

Klasa `TreePath` zarządza sekwencją referencji do obiektów klasy `Object` (nie `TreeNode`!). Wiele metod klasy `JTree` zwraca obiekty klasy `TreePath`. Gdy dysponujemy już ścieżką, to możemy pobrać węzeł znajdujący się na jej końcu, korzystając z metody `getLastPathComponent`. Aby na przykład odnaleźć aktualnie wybrany węzeł drzewa, korzystamy z metody `getSelectionPath` klasy `JTree`. W rezultacie otrzymujemy obiekt klasy `TreePath`, za pomocą którego możemy z kolei uzyskać aktualnie wybrany węzeł.

```
TreePath selectionPath = tree.getSelectionPath();
DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode)
    selectionPath.getLastPathComponent();
```

Ponieważ potrzeba taka pojawiła się bardzo często, to udostępniono dodatkową metodę, która natychmiast zwraca wybrany węzeł drzewa.

```
DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode)
    tree.getLastSelectedPathComponent();
```

Metody tej nie nazywano `getSelectedNode`, ponieważ drzewo nie operuje na węzłach, a jedynie na ścieżkach.



Ścieżki są jedną z dwu metod wykorzystywanych przez klasę `JTree` do opisu węzłów. Istnieje kilka metod klasy `JTree`, które wykorzystują lub zwracają wartość indeksu określającą pozycję wiersza. Jest to po prostu numer wiersza (numeracja rozpoczyna się od 0), w którym znajduje się dany węzeł. Numerowane są jedynie węzły widoczne w danym momencie i w związku z tym numer wiersza danego węzła ulega zmianie podczas operacji, takich jak wstawianie, zwijanie i rozwijanie, wykonywanych dla węzłów poprzedzających dany węzeł w drzewie. Dlatego też należy unikać korzystania z numerów węzłów. Wszystkie metody klasy `JTree` używające numerów węzłów posiadają odpowiedniki posługujące się ścieżkami.

Po uzyskaniu wybranego węzła możemy dodać do niego węzły podrzędne, ale nie w poniższy sposób:

```
selectedNode.add(newNode); // NIE!
```

Zmieniając strukturę węzła, dokonujemy zmiany jedynie modelu, ale jego widok nie uzyskuje o tym informacji. Możemy sami wysłać odpowiednie zawiadomienie, ale jeśli skorzystamy z metody `insertNodeInto` klasy `DefaultTreeModel`, to zrobi to za nas automatycznie model drzewa. Na przykład poniższe wywołanie doda nowy węzeł jako ostatni węzeł podrzędny wybranego węzła i powiadomi o tym widok drzewa:

```
model.insertNodeInto(newNode, selectedNode,
    selectedNode.getChildCount());
```

Natomiast metoda `removeNodeFromParent` usunie węzeł i powiadomi widok drzewa:

```
model.removeNodeFromParent(selectedNode);
```

Jeśli struktura drzewa pozostaje zachowana, a zmienił się jedynie obiekt użytkownika, wystarczy wywołać:

```
model.nodeChanged(changedNode);
```

Automatyczne powiadamianie widoku drzewa jest główna zaletą korzystania z klasy `DefaultTreeModel`. Jeśli utworzymy własny model drzewa, to sami musimy zawiadniać jego widok o zmianach. (Patrz *Core Swing* autorstwa Kim Topley).



**Klasa DefaultTreeModel posiada metodę `reload`, która powoduje przeładowanie modelu.** Nie należy jednak z niej korzystać w celu aktualizacji widoku drzewa po każdej przeprowadzonej zmianie modelu. Przeładowanie modelu powoduje, że zwijane są wszystkie węzły drzewa z wyjątkiem węzłów podzielnych korzenia. Użytkownik będzie więc zmuszony rozwijać drzewo po każdej wprowadzonej zmianie.

Gdy widok drzewa jest zawiadamiany o zmianie w strukturze węzłów, to aktualizuje odpowiednio reprezentację graficzną drzewa. Nie rozwija jednak przy tym automatycznie węzłów, jeśli nowe węzły zostały dodane jako węzły podzielne do zwiniętego węzła. Szczególnie jeśli użytkownik doda nowy węzeł do węzła, który jest zwinięty, to nie wywoła żadnej zmiany w bieżącej prezentacji drzewa. Użytkownik nie będzie więc wiedzieć, czy nowy węzeł został faktycznie dodany, dopóki sam nie rozwinięcie odpowiednich węzłów, tak by widoczny był nowo dodany węzeł. W tym celu wykorzystać można metodę `makeVisible` klasy `JTree`. Jej parametrem jest ścieżka prowadząca do węzła, który powinien być widoczny.

Musimy więc skonstruować ścieżkę prowadzącą od korzenia do nowego węzła. Wywołamy w tym celu metodę `getPathToRoot` klasy `DefaultTreeModel`, która zwróci tablicę `TreeNode[]` wszystkich węzłów ścieżki od danego węzła do korzenia. Tablicę tę przekażemy jako parametr konstruktora klasy `TreePath`.

Poniżej demonstrujemy przykładowy kod rozwijający ścieżkę do nowego węzła.

```
TreeNode[] nodes = model.getPathToRoot(newNode);
TreePath path = new TreePath(nodes);
tree.makeVisible(path);
```



**Ciekawe, że klasa DefaultTreeModel ignoruje zupełnie istnienie klasy TreePath, mimo że musi komunikować się z klasą JTree. Klasa JTree wykorzystuje ścieżki, ale nigdy nie używa tablic węzłów.**

Założymy teraz, że drzewo nasze umieszczone jest wewnątrz przewijalnego panelu. Po dodaniu nowego węzła może on nadal nie być widoczny, ponieważ znajdzie się poza widocznym fragmentem drzewa. Zamiast wywołać metodę `makeVisible`, wykorzystamy wtedy:

```
tree.scrollPathToVisible(path);
```

Wywołanie to spowoduje nie tylko rozwinięcie węzłów wzduż ścieżki prowadzącej do nowego węzła, ale i takie przewinięcie zawartości panelu, że nowy węzeł będzie widoczny (patrz rysunek 6.29).

**Rysunek 6.29.**

Przewinięcie panelu w celu prezentacji nowego węzła



Domyślnie węzły drzewa nie mogą być modyfikowane. Jeśli jednak wywołamy:

```
tree.setEditable(true);
```

to użytkownik może edytować węzły, klikając je dwukrotnie myszą, zmieniając łańcuch opisujący węzeł i zatwierdzając zmianę klawiszem *Enter*. Dwukrotne kliknięcie węzła myszą powoduje wywołanie *domyślnego edytora komórki* implementowanego przez klasę `DefaultCellEditor` (patrz rysunek 6.30). Można zainstalować własny edytor — w sposób, który omówiliśmy, przedstawiając edytory komórek tabeli.

**Rysunek 6.30.**

Domyślny edytor komórek



Listing 6.13 zawiera kompletny tekst źródłowy programu edycji drzewa. Umożliwia on wstawianie węzłów i edytowanie ich. Zwróćmy przy tym uwagę, w jaki sposób rozwijane są węzły drzewa i przewijany panel, tak aby można było zobaczyć nowe węzły dodane do drzewa.

**Listing 6.13.** *treeEdit/TreeEditFrame.java*

```
package treeEdit;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;

/**
 * Ramka zawierająca przyciski i edytowane drzewo.
 */
public class TreeEditFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 200;

    private DefaultTreeModel model;
    private JTree tree;

    public TreeEditFrame()
    {
        ...
    }
}
```

```

{
    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

    // tworzy drzewo

    TreeNode root = makeSampleTree();
    model = new DefaultTreeModel(root);
    tree = new JTree(model);
    tree.setEditable(true);

    // umieszcza drzewo w przewijalnym panelu

    JScrollPane scrollPane = new JScrollPane(tree);
    add(scrollPane, BorderLayout.CENTER);

    makeButtons();
}

public TreeNode makeSampleTree()
{
    DefaultMutableTreeNode root = new DefaultMutableTreeNode("World");
    DefaultMutableTreeNode country = new DefaultMutableTreeNode("USA");
    root.add(country);
    DefaultMutableTreeNode state = new DefaultMutableTreeNode("California");
    country.add(state);
    DefaultMutableTreeNode city = new DefaultMutableTreeNode("San Jose");
    state.add(city);
    city = new DefaultMutableTreeNode("San Diego");
    state.add(city);
    state = new DefaultMutableTreeNode("Michigan");
    country.add(state);
    city = new DefaultMutableTreeNode("Ann Arbor");
    state.add(city);
    country = new DefaultMutableTreeNode("Germany");
    root.add(country);
    state = new DefaultMutableTreeNode("Schleswig-Holstein");
    country.add(state);
    city = new DefaultMutableTreeNode("Kiel");
    state.add(city);
    return root;
}

/**
 * Tworzy przyciski umożliwiające dodanie węzła siostrzanego,
 * dodanie węzła podzielnego oraz usunięcie wybranego węzła.
 */
public void makeButtons()
{
    JPanel panel = new JPanel();
    JButton addSiblingButton = new JButton("Add Sibling");
    addSiblingButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) tree
                .getLastSelectedPathComponent();

            if (selectedNode == null) return;

```

```
DefaultMutableTreeNode parent = (DefaultMutableTreeNode)
↳selectedNode.getParent();

if (parent == null) return;

DefaultMutableTreeNode newNode = new DefaultMutableTreeNode("New");

int selectedIndex = parent.getIndex(selectedNode);
model.insertNodeInto(newNode, parent, selectedIndex + 1);

// wyświetla nowy węzeł

TreeNode[] nodes = model.getPathToRoot(newNode);
TreePath path = new TreePath(nodes);
tree.scrollPathToVisible(path);
}

});

panel.add(addSiblingButton);

JButton addChildButton = new JButton("Add Child");
addChildButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) tree
            .getLastSelectedPathComponent();
        if (selectedNode == null) return;

        DefaultMutableTreeNode newNode = new DefaultMutableTreeNode("New");
        model.insertNodeInto(newNode, selectedNode, selectedNode.getChildCount());

        // wyświetla nowy węzeł

        TreeNode[] nodes = model.getPathToRoot(newNode);
        TreePath path = new TreePath(nodes);
        tree.scrollPathToVisible(path);
    }
});
panel.add(addChildButton);

JButton deleteButton = new JButton("Delete");
deleteButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) tree
            .getLastSelectedPathComponent();

        if (selectedNode != null && selectedNode.getParent() != null) model
            .removeNodeFromParent(selectedNode);
    }
});
panel.add(deleteButton);
add(panel, BorderLayout.SOUTH);
}
```

**API javax.swing.JTree 1.2**

- **TreePath getSelectionPath()**  
zwraca ścieżkę do aktualnie wybranego węzła (lub pierwszego wybranego, jeśli wybranych zostało wiele węzłów) bądź wartość null, jeśli żaden węzeł nie jest wybrany.
- **Object getLastSelectedPathComponent()**  
zwraca obiekt aktualnie wybranego węzła (lub pierwszego wybranego, jeśli wybranych zostało wiele węzłów) bądź wartość null, jeśli żaden węzeł nie jest wybrany.
- **void makeVisible(TreePath path)**  
rozwija wszystkie węzły wzduż ścieżki.
- **void scrollPathToVisible(TreePath path)**  
rozwija wszystkie węzły wzduż ścieżki oraz, jeśli drzewo jest umieszczone w panelu przewijalnym, przewija panel, tak by widoczny był ostatni węzeł ścieżki.

**API javax.swing.tree.TreePath 1.2**

- **Object getLastPathComponent()**  
zwraca ostatni obiekt ścieżki czyli węzeł, do którego dostęp reprezentuje ścieżka.

**API javax.swing.tree.TreeNode 1.2**

- **TreeNode getParent()**  
zwraca węzeł nadrzędny danego węzła.
- **TreeNode getChildAt(int index)**  
zwraca węzeł podrzędny o danym indeksie. Wartość indeksu musi być z przedziału od 0 do getChildCount() - 1.
- **int getChildCount()**  
zwraca liczbę węzłów podrzędnych danego węzła.
- **Enumeration children()**  
zwraca obiekt wyliczenia umożliwiający przeglądanie wszystkich węzłów podrzędnych danego węzła.

**API javax.swing.DefaultTreeModel 1.2**

- **void insertNodeInto(MutableTreeNode newChild, MutableTreeNode parent, int index)**  
wstawia newChild jako nowy węzeł podrzędny węzła parent o podanym indeksie.

- void removeNodeFromParent(MutableTreeNode node)  
usuwa węzeł node z modelu.
- void nodeChanged(TreeNode node)  
zawiadamia obiekty nasłuchujące modelu o modyfikacji węzła node.
- void nodesChanged(TreeNode parent, int[] changedChildIndexes)  
zawiadamia obiekty nasłuchujące modelu, że uległy modyfikacji węzły podrzędne węzła parent o podanych indeksach.
- void reload()  
ładuje wszystkie węzły do modelu. Operacja ta wykonywana powinna być jedynie, gdy zaszła zasadnicza modyfikacja węzłów spowodowana zewnętrzną przyczyną.

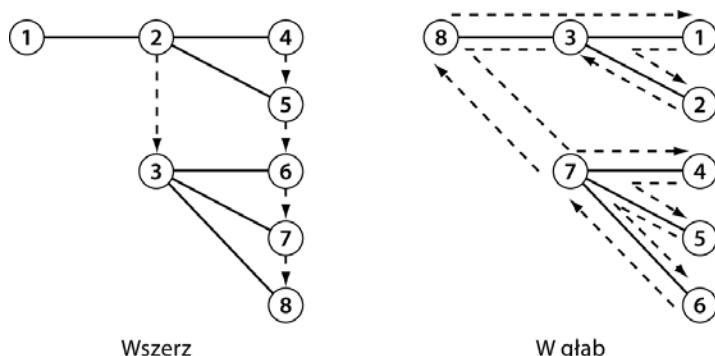
### 6.3.2. Przeglądanie węzłów

Często, aby odnaleźć poszukiwany węzeł, musimy przejrzeć wszystkie węzły. Klasa DefaultMutableTreeNode posiada kilka metod przydatnych w tym celu.

Metody breadthFirstEnumeration i depthFirstEnumeration zwracają obiekty wyliczeń, których metoda nextElement umożliwia przeglądanie wszystkich węzłów podrzędnych bieżącego węzła, korzystając z metody przeglądania wszerz i w głąb. Rysunek 6.31 pokazuje porządek przeglądania węzłów przykładowego drzewa w obu metodach.

**Rysunek 6.31.**

Kolejność  
przeglądania  
węzłów drzewa



Przeglądanie wszerz jest nieco łatwiejsze do wyjaśnienia. Drzewo przeglądane jest warstwami. Najpierw odwiedzany jest korzeń drzewa, potem jego wszystkie węzły podrzędne, a następnie ich węzły podrzędne itd.

W przypadku przeglądania w głąb sposób poruszania się po drzewie przypomina mysz biegającą po labiryncie w kształcie drzewa. Porusza się ona wzduł ścieżek drzewa aż do napotkania liścia, po czym wycofuje się i wybiera następną ścieżkę.

Taki sposób przeglądania drzewa nazywany bywa także *przeglądaniem od końca*, ponieważ węzły podrzędne są przeglądane przed węzłami nadrzędnymi. Zgodnie z tym nazewnictwem

dodano metodę przeszukiwania od końca postOrderTraversal będącą synonimem metody depthFirstTraversal przeszukiwania w głąb. Aby zestaw ten był kompletny, uzupełniono go metodą preOrderTraversal, która przegląda drzewo również w głąb, ale najpierw węzły nadrzędne, a potem podrzędne.

Poniżej typowy przykład użycia metod przeglądania drzewa:

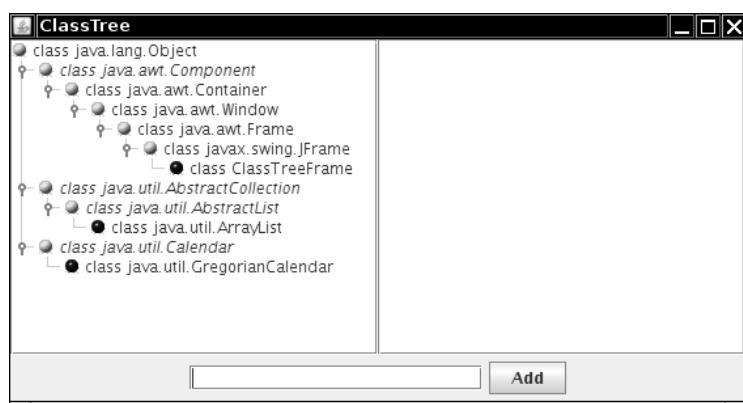
```
Enumeration breadthFirst = node.breadthFirstEnumeration();
while (breadthFirst.hasMoreElements())
    operacje na węzle breadthFirst.nextElement();
```

Z przeglądaniem drzewa związana jest także metoda pathFromAncestorEnumeration, która wyznacza ścieżkę od jednego z przodków węzła do danego węzła i tworzy wyliczenie węzłów znajdujących się na ścieżce. Jej implementacja polega na wywoływaniu metody getParent do momentu znalezienia przodka, a następnie przejściu ścieżki z powrotem.

Nasz następny program ma przeglądać drzewo. Będzie on wyświetlał drzewo dziedziczenia klas. Po wprowadzeniu nazwy klasy w polu tekstowym w dolnej części okna klasa ta zostanie dodana do drzewa wraz z jej wszystkimi klasami bazowymi (patrz rysunek 6.32).

**Rysunek 6.32.**

Drzewo dziedziczenia



W przykładzie tym wykorzystamy fakt, że obiekt użytkownika dla danego węzła może być dowolnego typu. Ponieważ węzły naszego drzewa reprezentować będą klasy, umieścimy w nich obiekty klasy Class.

Ponieważ nie chcemy, by drzewo zawierało wiele wystąpień tej samej klasy, to najpierw będziemy musieli je przeszukać, aby sprawdzić, czy dana klasa już występuje. Poniższa metoda znajduje węzeł zawierający podany obiekt użytkownika pod warunkiem, że istnieje on w drzewie.

```
public DefaultMutableTreeNode findUserObject(Object obj)
{
    Enumeration e = root.breadthFirstEnumeration();
    while (e.hasMoreElements())
    {
        DefaultMutableTreeNode node
            = (DefaultMutableTreeNode) e.nextElement();
        if (node.getUserObject().equals(obj))
            return node;
    }
}
```

```

    }
    return null;
}
}

```

### 6.3.3. Rysowanie węzłów

Specyfika aplikacji często wymaga zmiany sposobu prezentacji drzewa. Najczęściej polega ona na zmianie ikon folderów i liści, zmianie czcionki opisującej węzeł lub zastąpieniu opisu obrazkiem. Wszystkie te zmiany są osiągalne przez zainstalowanie nowego *obiektu rysującego komórki* danego drzewa. Domyślnie klasa `JTree` wykorzystuje obiekt klasy `DefaultTreeCellRenderer`, która jest klasą pochodną klasy `JLabel`. Etykieta węzła składa się z ikony węzła i jego opisu.



Obiekt rysujący komórki drzewa nie jest odpowiedzialny za narysowanie uchwytów węzłów umożliwiających zwijanie i rozwijanie poddrzew. Uchwyty te są częścią ogólnego wyglądu komponentów interfejsu użytkownika i nie powinny być modyfikowane.

Wygląd drzewa zmodyfikować możemy na trzy różne sposoby.

- Zmieniamy ikony, czcionkę oraz kolor tła wykorzystywany przez obiekt klasy `DefaultTreeCellRenderer`. Ustawienia te wykorzystywane będą podczas rysowania wszystkich węzłów drzewa.
- Instalujemy własny obiekt rysujący komórki drzewa, który należeć będzie do klasy pochodnej klasy `DefaultTreeCellRenderer`. Może on zmieniać ikony, czcionkę i kolor tła, rysując poszczególne węzły.
- Instalujemy własny obiekt rysujący komórki drzewa, który implementować będzie interfejs `TreeCellRenderer` i rysować dowolną reprezentację węzłów drzewa.

Przyjrzyjmy się po kolej tym sposobom. Najprostszy z nich wymaga utworzenia obiektu klasy `DefaultTreeCellRenderer`, zmiany ikony i zainstalowania obiektu dla danego drzewa:

```

DefaultTreeCellRenderer renderer = new DefaultTreeCellRenderer();
renderer.setLeafIcon(new ImageIcon("blue-ball.gif")); // ikona liści
renderer.setClosedIcon(new ImageIcon("red-ball.gif")); // ikona zwinietego węzła
renderer.setOpenIcon(new ImageIcon("yellow-ball.gif")); // ikona rozwiniętego węzła
tree.setCellRenderer(renderer);

```

Efekt takiego rozwiązania przedstawia rysunek 6.32. Użyliśmy prostych ikon węzłów; w praktyce projektant interfejsu dostarcza zwykle odpowiednich ikon dla danej aplikacji.

Nie zalecamy zmiany czcionki bądź koloru tła dla całego drzewa, ponieważ to jest zadaniem wyglądu komponentów interfejsu użytkownika.

Możemy jednak zmieniać czcionkę pojedynczych węzłów drzewa dla ich wyróżnienia. Jeśli przyjrzymy się uważnie rysunkowi 6.32, to zauważymy, że nazwy klas *abstrakcyjnych* pisane są kursywą.

Aby zmieniać wygląd poszczególnych węzłów, musimy zainstalować własny obiekt rysujący komórki drzewa. Przypomina on obiekty rysujące komórki list, które omówiliśmy wcześniej. Interfejs `TreeCellRenderer` posiada pojedynczą metodę:

```
Component getTreeCellRendererComponent(JTree tree,
    Object value, boolean selected, boolean expanded,
    boolean leaf, int row, boolean hasFocus)
```

Metoda `getTreeCellRendererComponent` klasy `DefaultTreeCellRenderer` zwraca po prostu wartość `this`, czyli innymi słowy komponent etykiety. (Przypomnijmy, że klasa `DefaultTreeCellRenderer` jest pochodną klasy `JLabel`). Aby zmodyfikować komponent, musimy sami utworzyć klasę pochodną klasy `DefaultTreeCellRenderer` i zastąpić jej metodę `getTreeCellRendererComponent` implementacją, którą:

- wywoła metodę klasy nadzędnej w celu przygotowania danych komponentu etykiety,
- zmodyfikuje właściwości etykiety,
- zwróci `this`.

```
class MyTreeCellRenderer extends DefaultTreeCellRenderer
{
    public Component getTreeCellRendererComponent(JTree tree, Object value,
        boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus)
    {
        Component comp = super.getTreeCellRendererComponent(tree, value,
            selected, expanded, leaf, row, hasFocus);
        DefaultMutableTreeNode node = (DefaultMutableTreeNode) value;
        sprawdź node.getUserObject();
        Font font = appropriate font;
        comp.setFont(font);
        return comp;
    }
};
```



Parametr `value` metody `getTreeCellRendererComponent` *nie* jest obiektem użytkownika, ale obiektem reprezentującym węzeł! Przypomnijmy w tym miejscu, że obiekty użytkownika wykorzystywane są przez klasę `DefaultMutableTreeNode`, natomiast klasa `JTree` może zawierać węzły dowolnego typu. Jeśli drzewo składa się z węzłów klasy `DefaultMutableTreeNode`, to obiekt użytkownika możemy pobrać z nich dopiero, wywołując metodę `getUserObject`, tak jak uczyniliśmy to w powyższym przykładzie.



Obiekt klasy `DefaultTreeCellRenderer` wykorzystuje ten sam obiekt klasy `JLabel` dla wszystkich węzłów, zmieniając jedynie tekst etykiety. Jeśli więc dokonamy zmiany czcionki dla danego węzła, to musimy przywrócić czcionkę domyślną, gdy metoda zostanie wywołana po raz kolejny. W przeciwnym razie wszystkie kolejne węzły zostaną opisane zmienioną czcionką! Kod programu w listingu 6.14 pokazuje sposób przywrócenia domyślnej czcionki.

Nie będziemy pokazywać osobnego przykładu obiektu rysującego komórkę drzewa dowolnej postaci. Sposób jego działania jest analogiczny do pracy obiektu rysującego komórki listy przedstawionego na listingu 6.4.

Obiekt klasy `ClassNameTreeCellRenderer` przedstawiony na listingu 6.14 prezentuje nazwę klasy czcionką prostą lub pochyłą w zależności od modyfikatora `ABSTRACT` obiektu klasy `Class`. Program korzysta z czcionki, którą dla reprezentacji etykiet drzewa przewidział bieżący

wygląd komponentów i tworzy na jej podstawie wersję pochyłą. Ponieważ wszystkie wywołania zwracają ten sam obiekt klasy `JLabel`, to kolejne wywołanie metody `getTreeCellRendererComponent` musi odtworzyć oryginalną czcionkę, jeśli wcześniej użyta była jej wersja pochyła.

Konstruktor klasy `ClassTreeFrame` zmienia dodatkowo ikony reprezentujące węzły drzewa.

#### **API javax.swing.tree.DefaultMutableTreeNode 1.2**

- `Enumeration breadthFirstEnumeration()`
- `Enumeration depthFirstEnumeration()`
- `Enumeration preOrderEnumeration()`
- `Enumeration postOrderEnumeration()`

zwracają obiekt wyliczenia umożliwiający przeglądanie wszystkich węzłów drzewa w odpowiednim porządku: w szerz, gdzie węzły podrzędne leżące bliżej korzenia odwiedzane są wcześniej, w głęb, gdzie wszystkie węzły podrzędne danego węzła są odwiedzane, zanim odwiedzone zostaną jego węzły siostrzane. Metoda `postOrderEnumeration` stanowi synonim metody `depthFirstEnumeration`. Metoda `preOrderEnumeration` przegląda drzewo podobnie do niej, z tą różnicą, że węzły nadrzędne przeglądane są przed ich węzłami podrzennymi.

#### **API javax.swing.tree.TreeCellRenderer 1.2**

- `Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus)`

zwraca komponent, którego metoda `paint` wywoływana jest w celu narysowania komórki drzewa.

<i>Parametry:</i>	<code>tree</code>	drzewo, do którego należy rysowana komórka,
	<code>value</code>	rysowany węzeł,
	<code>selected</code>	wartość <code>true</code> , jeśli węzeł jest wybrany,
	<code>expanded</code>	wartość <code>true</code> , jeśli węzły podrzędne danego węzła są widoczne,
	<code>leaf</code>	wartość <code>true</code> , jeśli węzeł jest liściem,
	<code>row</code>	numer wiersza graficznej reprezentacji drzewa zawierającej węzeł,
	<code>hasFocus</code>	wartość <code>true</code> , jeśli węzeł jest przeglądany w danym momencie przez użytkownika.

#### **API javax.swing.tree.DefaultTreeCellRenderer 1.2**

- `void setLeafIcon(Icon icon)`
- `void setOpenIcon(Icon icon)`
- `void setClosedIcon(Icon icon)`

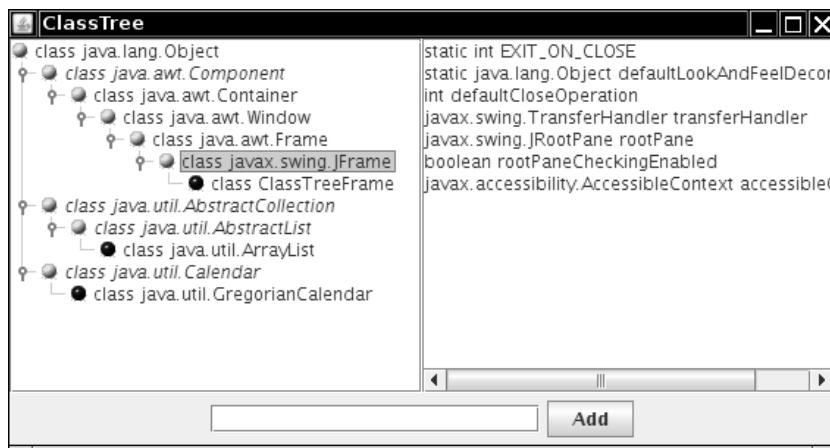
Ustalają ikonę prezentującą odpowiednio: liść, węzeł rozwinięty, węzeł zwinięty.

### 6.3.4. Nasłuchiwanie zdarzeń w drzewach

Najczęściej komponent drzewa wykorzystywany jest razem z innym komponentem interfejsu użytkownika. Gdy użytkownik wybierawęzeł drzewa, to inny komponent pokazuje pewną informację o nich, tak jak na przykład program przedstawiony na rysunku 6.33. Kiedy użytkownik wybierawęzeł drzewa reprezentujący klasę języka Java, to w polu tekstowym obok prezentowana jest informacja o jej zmiennych.

Rysunek 6.33.

Przeglądarka klas



Aby uzyskać takie działanie programu, konieczne jest zainstalowanie *obiektu nasłuchującego wyboru w drzewie*. Obiekt ten musi implementować interfejs TreeSelectionListener, który posiada tylko jedną metodę:

```
void valueChanged(TreeSelectionEvent event)
```

Jest ona wywoływana za każdym razem, gdy węzeł drzewa zostaje wybrany lub przestaje być wybrany.

Obiekt nasłuchujący dodajemy do drzewa w zwykły sposób:

```
tree.addSelectionListener(listener);
```

Możemy określić sposób wyboru węzłów drzewa przez użytkownika. Wybrany może być tylko jeden węzeł, ciągły zakres węzłów lub dowolny, potencjalnie nieciągły zbiór węzłów. Klasa JTree wykorzystuje klasę TreeSelectionModel do zarządzania wyborem węzłów. Dla modelu, który musimy najpierw pobrać, określić możemy jeden z następujących stanów wyboru: SINGLE\_TREE\_SELECTION, CONTIGUOUS\_TREE\_SELECTION lub DISCONTIGUOUS\_TREE\_SELECTION (ten ostatni jest stanem domyślnym). Nasza przeglądarka umożliwiać będzie wybór pojedynczej klasy:

```
int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
tree.getSelectionModel().setSelectionMode(mode);
```

Po określeniu sposobu wyboru na drzewie nie musimy więcej zajmować się modelem wyboru.



Sposób wyboru wielu węzłów drzewa zależy od bieżącego wyglądu komponentów interfejsu użytkownika. W przypadku wyglądu Metal wystarczy przytrzymać klawisz *Ctrl* podczas kliknięcia myszą, aby dokonać wyboru kolejnego węzła lub usunąć jego wybór, jeśli wcześniej był już wybrany. Podobnie przytrzymanie klawisza *Shift* umożliwia wybranie zakresu węzłów.

Aby uzyskać informacje o tym, które z węzłów zostały wybrane, wywołujemy metodę `getSelectionPaths()`:

```
TreePath[] selectedPaths = tree.getSelectionPaths();
```

W przypadku gdy możliwość wyboru ograniczyliśmy do jednego węzła, możemy skorzystać z metody `getSelectionPath()`, która zwróci ścieżkę do pierwszego wybranego węzła lub wartość `null`, jeśli żaden węzeł nie został wybrany.



Klasa `TreeSelectionEvent` dysponuje metodą `getPaths()`, która zwraca tablicę obiektów klasy `TreePath` reprezentujących zmiany wyboru, a nie aktualnie wybrane węzły.

Program, którego kod źródłowy zawiera listing 6.14, wykorzystuje mechanizm wyboru węzła drzewa. Program prezentuje drzewo dziedziczenia klas, wyróżniając klasy abstrakcyjne kursywą (implementację obiektu rysującego komórkę drzewa przedstawiliśmy na listingu 6.15). W polu tekstowym w dolnej części okna programu użytkownik może wpisać nazwę dowolnej klasy, a następnie wybrać klawisz *Enter* lub przycisk *Add*, aby dodać klasę i jej klasy bazowe do drzewa. Należy podać wyłącznie pełną nazwę klasy, czyli na przykład `java.util.ArrayList`.

→`ArrayList`.

#### **Listing 6.14.** *treeRenderer/ClassTreeFrame.java*

```
package treeRender;

import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;

/**
 * Ramka zawierająca drzewo klas, pole tekstowe pokazujące
 * składowe wybranej klasy oraz pole tekstowe umożliwiające
 * dodawanie nowych klas do drzewa.
 */
public class ClassTreeFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 300;

    private DefaultMutableTreeNode root;
    private DefaultTreeModel model;
    private JTree tree;
    private JTextField textField;
    private JTextArea textArea;
```

```

public ClassTreeFrame()
{
    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

    // w korzeniu drzewa znajduje się klasa Object
    root = new DefaultMutableTreeNode(java.lang.Object.class);
    model = new DefaultTreeModel(root);
    tree = new JTree(model);

    // dodaje klasę do drzewa
    addClass(getClass());

    // tworzy ikony węzłów
    ClassNameTreeCellRenderer renderer = new ClassNameTreeCellRenderer();
    renderer.setClosedIcon(new ImageIcon(getClass().getResource("red-ball.gif")));
    renderer.setOpenIcon(new ImageIcon(getClass().getResource("yellow-ball.gif")));
    renderer.setLeafIcon(new ImageIcon(getClass().getResource("blue-ball.gif")));
    tree.setCellRenderer(renderer);

    // konfiguruje sposób wyboru węzłów
    tree.addTreeSelectionListener(new TreeSelectionListener()
    {
        public void valueChanged(TreeSelectionEvent event)
        {
            // użytkownik wybrał inny węzeł drzewa i należy zaktualizować opis klasy
            TreePath path = tree.getSelectionPath();
            if (path == null) return;
            DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) path
                .getLastPathComponent();
            Class<?> c = (Class<?>) selectedNode.getUserObject();
            String description = getFieldDescription(c);
            textArea.setText(description);
        }
    });
    int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
    tree.getSelectionModel().setSelectionMode(mode);

    // obszar tekstowy zawierający opis klasy
    textArea = new JTextArea();

    // dodaje komponenty drzewa i pola tekstowego do panelu
    JPanel panel = new JPanel();
    panel.setLayout(new GridLayout(1, 2));
    panel.add(new JScrollPane(tree));
    panel.add(new JScrollPane(textArea));

    add(panel, BorderLayout.CENTER);

    addTextField();
}

<**
 * Dodaje pole tekstowe i przycisk "Add"
 * umożliwiające dodanie nowej klasy do drzewa.
 */
public void addTextField()
{
    JPanel panel = new JPanel();

```

```

        ActionListener addListener = new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
    {
        // dodaje do drzewa klasę, której nazwa znajduje się w polu tekstowym
        try
        {
            String text = textField.getText();
            addClass(Class.forName(text)); // opróżnia zawartość
            textField.setText("");
        }
        catch (ClassNotFoundException e)
        {
            JOptionPane.showMessageDialog(null, "Class not found");
        }
    }
};

// pole tekstowe, w którym wprowadzane są nazwy nowych klas
textField = new JTextField(20);
textField.addActionListener(addListener);
panel.add(textField);

JButton addButton = new JButton("Add");
addButton.addActionListener(addListener);
panel.add(addButton);

add(panel, BorderLayout.SOUTH);
}

/**
 * Wyszukuje obiekt w drzewie.
 * @param obj szukany obiekt
 * @return węzeł zawierający szukany obiekt lub null,
 * jeśli obiekt nie znajduje się w drzewie
 */
@SuppressWarnings("unchecked")
public DefaultMutableTreeNode findUserObject(Object obj)
{
    // szuka węzła zawierającego obiekt użytkownika
    Enumeration<TreeNode> e = (Enumeration<TreeNode>) root.breadthFirstEnumeration();
    while (e.hasMoreElements())
    {
        DefaultMutableTreeNode node = (DefaultMutableTreeNode) e.nextElement();
        if (node.getUserObject().equals(obj)) return node;
    }
    return null;
}

/**
 * Dodaje do drzewa klasę i jej klasy bazowe,
 * których nie ma jeszcze w drzewie.
 * @param c dodawana klasa
 * @return nowo dodany węzeł.
 */
public DefaultMutableTreeNode addClass(Class<?> c)
{

```

```

// dodaje klasę do drzewa

// pomija typy, które nie są klasami
if (c.isInterface() || c.isPrimitive()) return null;

// jeśli klasa znajduje się już w drzewie, to zwraca jej węzel
DefaultMutableTreeNode node = findUserObject(c);
if (node != null) return node;

// jeśli klasa nie znajduje się w drzewie,
// to najpierw należy dodać rekurencyjnie do drzewa jej klasy bazowe

Class<?> s = c.getSuperclass();

DefaultMutableTreeNode parent;
if (s == null) parent = root;
else parent = addClass(s);

// dodaje klasę jako węzeł podrzędny
DefaultMutableTreeNode newNode = new DefaultMutableTreeNode(c);
model.insertNodeInto(newNode, parent, parent getChildCount());

// sprawia, że węzeł jest widoczny
TreePath path = new TreePath(model.getPathToRoot(newNode));
tree.makeVisible(path);

return newNode;
}

/**
 * Zwraca opis składowych klasy.
 * @param klasa
 * @return łańcuch znaków zawierający nazwy i typy zmiennych
 */
public static String getFieldDescription(Class<?> c)
{
    // korzysta z mechanizmu refleksji
    StringBuilder r = new StringBuilder();
    Field[] fields = c.getDeclaredFields();
    for (int i = 0; i < fields.length; i++)
    {
        Field f = fields[i];
        if ((f.getModifiers() & Modifier.STATIC) != 0) r.append("static ");
        r.append(f.getType().getName());
        r.append(" ");
        r.append(f.getName());
        r.append("\n");
    }
    return r.toString();
}
}

```

**Listing 6.15.** treeRender/ClassNameTreeCellRenderer.java

```

package treeRender;

import java.awt.*;
import java.lang.reflect.*;

```

```
import javax.swing.*;
import javax.swing.tree.*;

/**
 * Klasa opisująca węzły drzewa czcionką zwykłą lub pochyloną
 * (w przypadku klas abstrakcyjnych).
 */
public class ClassNameTreeCellRenderer extends DefaultTreeCellRenderer
{
    private Font plainFont = null;
    private Font italicFont = null;

    public Component getTreeCellRendererComponent(JTree tree, Object value, boolean
        ↳selected,
        boolean expanded, boolean leaf, int row, boolean hasFocus)
    {
        super.getTreeCellRendererComponent(tree, value, selected, expanded, leaf, row,
        ↳hasFocus);
        // pobiera obiekt użytkownika
        DefaultMutableTreeNode node = (DefaultMutableTreeNode) value;
        Class<?> c = (Class<?>) node.getUserObject();

        // przy pierwszym użyciu tworzy czcionkę pochyloną odpowiadającą danej czcionce prostej
        if (plainFont == null)
        {
            plainFont = getFont();
            // obiekt rysujący komórkę drzewa wywoływany jest czasami
            // dla etykiety, która nie posiada określonej czcionki (null).
            if (plainFont != null) italicFont = plainFont.deriveFont(Font.ITALIC);
        }

        // wybiera czcionkę pochyloną, jeśli klasa jest abstrakcyjna
        if (((c.getModifiers() & Modifier.ABSTRACT) == 0) setFont(plainFont);
        else setFont(italicFont);
        return this;
    }
}
```

---

Działanie programu jest trochę skomplikowane, ponieważ wykorzystuje on przy tworzeniu drzewa refleksję klas, co odbywa się wewnątrz metody addClass. (Szczegółowe nie są w tym przypadku istotne. Przykładowy program tworzy akurat drzewo klas, ponieważ drzewo dziedziczenia jest dobrym przykładem struktury drzewiastej. Zwykle programy reprezentują jednak za pomocą drzewa inne struktury danych). Metoda addClass wywołuje metodę findUserObject, aby sprawdzić, czy klasa znajduje się już w drzewie. Metoda findUserObject przegląda drzewo wszerz. Jeśli klasa nie znajduje się jeszcze w drzewie, to program dodaje najpierw do drzewa jej klasy bazowe, a na końcu daną klasę i troszczy się o to, by jej węzeł był widoczny.

Gdy użytkownik wybiera węzeł drzewa, obszar tekstowy po prawej stronie zostaje wypełniony polami wybranej klasy. W kodzie konstruktora ramki ograniczamy możliwość wyboru do jednego węzła i dodajemy do drzewa obiekt nasłuchujący wyboru. Gdy wywołana zostaje jego metoda valueChanged, ignorujemy jej parametr i korzystamy z metody getSelectionPath. Pobieramy ostatni węzeł uzyskanej ścieżki i zawarty w nim obiekt użytkownika. Następnie wywołujemy metodę getFieldDescription, która korzysta z mechanizmu refleksji, aby utworzyć łańcuch znaków opisujący wszystkie składowe danej klasy. Otrzymany łańcuch znaków wyświetlamy w polu tekstowym okna.

**API javax.swing.JTree 1.2**

- TreePath getSelectionPath()
- TreePath[] getSelectionPaths()

Zwracają odpowiednio ścieżkę do pierwszego wybranego węzła lub tablicę ścieżek do wybranych węzłów. Jeśli żaden węzeł nie jest wybrany, obie metody zwracają null.

**API javax.swing.event.TreeSelectionListener 1.2**

- void valueChanged(TreeSelectionEvent event)
- wywoływana, gdy węzeł zostaje wybrany lub przestaje być wybrany.

**API javax.swing.event.TreeSelectionEvent 1.2**

- TreePath getPath()
- TreePath[] getPaths()

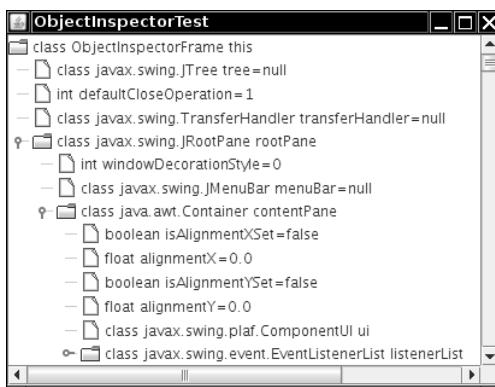
Zwracają odpowiednio ścieżkę do pierwszego obiektu lub tablicę ścieżek obiektów, których stan uległ zmianie na skutek danego zdarzenia wyboru. Jeśli interesują nas wybrane elementy, a nie zmiana ich wyboru, to powinniśmy skorzystać z metody getSelectionPaths klasy JTree.

### 6.3.5. Własne modele drzew

Ostatnim przykładem wykorzystania drzew będzie program umożliwiający inspekcję wartości pól obiektu, podobnie jak czynią to narzędzia uruchomieniowe (patrz rysunek 6.34).

**Rysunek 6.34.**

Drzewo inspekcji obiektów



Zanim rozpoczniemy omawianie programu, zalecamy, by skompilować go, uruchomić i zapoznać się z jego działaniem. Każdy węzeł utworzonego przez program drzewa odpowiada zmiennej składowej obiektu. Jeśli z kolei ta zmienna reprezentuje także obiekt, to możemy rozwinąć jej węzeł, aby sprawdzić zmienne także i tego obiektu. Program umożliwia inspekcję obiektów składających się na jego interfejs użytkownika. Jeśli rozejrzymy się trochę po drzewie,

to znajdziemy znajome obiekty odpowiadające komponentom interfejsu użytkownika. Jednocześnie nabierzemy także respektu dla złożoności mechanizmów biblioteki Swing, która nie jest zwykle widoczna dla programisty.

Istotna różnica w działaniu tego programu w stosunku do poprzednich przykładów polega na tym, że nie używa on klasy DefaultTreeModel. Jeśli program dysponuje już danymi zorganizowanymi w hierarchiczną strukturę, to nie ma sensu duplikować jej za pomocą nowego modelu i dodatkowo zajmować się jeszcze zapewnieniem synchronizacji obu struktur. Sytuacja taka występuje właśnie w przypadku naszego programu, ponieważ obiekty interfejsu użytkownika są już powiązane wzajemnymi referencjami.

Interfejs TreeModel definiuje szereg metod. Pierwsza ich grupa umożliwia klasie JTree odnalezienie węzłów drzewa przez pobranie najpierw jego korzenia, a później węzłów podległych. Klasa JTree korzysta z tych metod jedynie, gdy użytkownik rozwija węzel drzewa.

```
Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)
```

Przykład ten ukazuje, dlaczego interfejs TreeModel, podobnie jak klasa JTree, nie korzysta bezpośrednio z pojęcia węzłów. Korzeń i jego węzły podległe mogą być dowolnymi obiektami. Interfejs TreeModel umożliwia klasie JTree uzyskanie informacji o sposobie ich powiązania.

Kolejna metoda interfejsu TreeModel wykonuje operację odwrotną do metody getChild:

```
int getIndexofChild(Object parent, Object child)
```

Metoda ta może zostać zaimplementowana za pomocą wymienionych wcześniej trzech metod — patrz kod programu w listingu 6.16.

#### **Listing 6.16. treeModel/ObjectInspectorFrame.java**

```
package treeModel;

import java.awt.*;
import javax.swing.*;

/**
 * Ramka zawierająca drzewo.
 */
public class ObjectInspectorFrame extends JFrame
{
    private JTree tree;
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 300;

    public ObjectInspectorFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        // jako pierwszy inspekcji poddany jest obiekt ramki

        Variable v = new Variable(getClass(), "this", this);
        ObjectTreeModel model = new ObjectTreeModel();
        model.setRoot(v);
```

---

```
// tworzy i prezentuje drzewo

tree = new JTree(model);
add(new JScrollPane(tree), BorderLayout.CENTER);
}
}
```

---

**Listing 6.17.** treeModel/ObjectTreeModel.java

```
package treeModel;

import java.lang.reflect.*;
import java.util.*;
import javax.swing.event.*;
import javax.swing.tree.*;

/**
 * Model drzewa opisującego strukturę powiązań obiektów w języku Java.
 * Węzły podzielone reprezentują składowe obiektu.
 */
public class ObjectTreeModel implements TreeModel
{
    private Variable root;
    private EventListenerList listenerList = new EventListenerList();

    /**
     * Tworzy puste drzewo.
     */
    public ObjectTreeModel()
    {
        root = null;
    }

    /**
     * Umieszcza zmienną w korzeniu drzewa.
     * @param v zmienna opisywana przez drzewo
     */
    public void setRoot(Variable v)
    {
        Variable oldRoot = v;
        root = v;
        fireTreeStructureChanged(oldRoot);
    }

    public Object getRoot()
    {
        return root;
    }

    public int getChildCount(Object parent)
    {
        return ((Variable) parent).getFields().size();
    }

    public Object getChild(Object parent, int index)
    {
        ArrayList<Field> fields = ((Variable) parent).getFields();
```

```
Field f = (Field) fields.get(index);
Object parentValue = ((Variable) parent).getValue();
try
{
    return new Variable(f.getType(), f.getName(), f.get(parentValue));
}
catch (IllegalAccessException e)
{
    return null;
}

public int getChildCount(Object parent, Object child)
{
    int n = getChildCount(parent);
    for (int i = 0; i < n; i++)
        if (getChild(parent, i).equals(child)) return i;
    return -1;
}

public boolean isLeaf(Object node)
{
    return getChildCount(node) == 0;
}

public void valueForPathChanged(TreePath path, Object newValue)
{
}

public void addTreeModelListener(TreeModelListener l)
{
    listenerList.add(TreeModelListener.class, l);
}

public void removeTreeModelListener(TreeModelListener l)
{
    listenerList.remove(TreeModelListener.class, l);
}

protected void fireTreeStructureChanged(Object oldRoot)
{
    TreeModelEvent event = new TreeModelEvent(this, new Object[] { oldRoot });
    for (TreeModelListener l : listenerList.getListeners(TreeModelListener.class))
        l.treeStructureChanged(event);
}
```

---

**Listing 6.18.** treeModel/Variable.java

```
package treeModel;

import java.lang.reflect.*;
import java.util.*;

/**
 * Klasa reprezentująca zmienną posiadającą typ, nazwę i wartość.
 */
```

```

public class Variable
{
    private Class<?> type;
    private String name;
    private Object value;
    private ArrayList<Field> fields;

    /**
     * Tworzy obiekt reprezentujący zmienną.
     * @param aType typ zmiennej
     * @param aName nazwa zmiennej
     * @param aValue wartość zmiennej
     */
    public Variable(Class<?> aType, String aName, Object aValue)
    {
        type = aType;
        name = aName;
        value = aValue;
        fields = new ArrayList<>();

        //znajduje wszystkie pola, jeśli zmienność jest typu klasy,
        //nie rozwija jedynie łańcuchów znaków i wartości null

        if (!type.isPrimitive() && !type.isArray() && !type.equals(String.class) &&
            aValue != null)
        {
            //pobiera pola klasy i pola wszystkich jej klas bazowych
            for (Class<?> c = value.getClass(); c != null; c = c.getSuperclass())
            {
                Field[] fs = c.getDeclaredFields();
                AccessibleObject.setAccessible(fs, true);

                //pobiera wszystkie pola, które nie są statyczne
                for (Field f : fs)
                    if ((f.getModifiers() & Modifier.STATIC) == 0) fields.add(f);
            }
        }
    }

    /**
     * Zwraca wartość zmiennej.
     * @return wartość
     */
    public Object getValue()
    {
        return value;
    }

    /**
     * Zwraca wszystkie pola zmiennej, które nie są statyczne.
     * @return tablica zmiennych opisujących pola
     */
    public ArrayList<Field> getFields()
    {
        return fields;
    }

    public String toString()
    {

```

```

        String r = type + " " + name;
        if (type.isPrimitive()) r += "=" + value;
        else if (type.equals(String.class)) r += "=\"" + value;
        else if (value == null) r += "=null";
        return r;
    }
}

```

Model drzewa informuje klasę JTree o tym, które węzły powinny zostać przedstawione jako liście:

```
boolean isLeaf(Object node)
```

Jeśli w wyniku działania programu dane modelu drzewa ulegają zmianie, to drzewo musi zostać o tym powiadomione, aby dokonać aktualizacji swojego widoku. Dlatego też drzewo powinno być dodane jako obiekt nasłuchujący TreeModelListener do modelu. Model musi więc posiadać typowe metody związane z zarządzaniem obiektami nasłuchującymi:

```

void addTreeModelListener(TreeModelListener l)
void removeTreeModelListener(TreeModelListener l)

```

Implementację tych metod pokazuje listing 6.17.

Gdy zawartość modelu ulega zmianie, to wywołuje on jedną z czterech metod definiowanych przez interfejs TreeModelListener:

```

void treeNodesChanged(TreeModelEvent e)
void treeNodesInserted(TreeModelEvent e)
void treeNodesRemoved(TreeModelEvent e)
void treeStructureChanged(TreeModelEvent e)

```

Parametr tych metod opisuje miejsce wystąpienia zmian w drzewie. Szczegóły tworzenia obiektu zdarzenia opisującego wstawienie bądź usunięcie węzła są dość skomplikowane, ale musimy się nimi zajmować tylko wtedy, gdy węzły naszego drzewa są rzeczywiście dodawane bądź usuwane. Listing 6.16 okazuje konstrukcję obiektu zdarzenia w przypadku zastąpienia korzenia nowym obiektem.



Aby uprościć kod wysyłający obiekty zdarzeń, wykorzystujemy klasę javax.swing.event.EventListenerList zawierającą listę obiektów nasłuchujących. Sposób jej użycia pokazują trzy ostatnie metody na listingu 6.17.

Jeśli użytkownik zmodyfikuje węzeł drzewa, to model zostaje o tym poinformowany przez wywołanie jego metody:

```
void valueForPathChanged(TreePath path, Object newValue)
```

Gdy nie zezwolimy użytkownikowi na edycję drzewa, to metoda ta nigdy nie zostanie wywołana.

W przypadku takim konstrukcja modelu drzewa staje się łatwa. Należy zaimplementować trzy poniższe metody:

```

Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)

```

Opisują one strukturę drzewa. Następnie musimy dostarczyć jeszcze implementacji pozostałych pięciu metod, co pokazano w listingu 6.16 i będziemy gotowi do wyświetlenia własnego drzewa.

Zajmijmy się teraz implementacją naszego programu. Drzewo zawierać będzie obiekty klasy Variable.



Gdybyśmy skorzystali z klasy DefaultTreeModel, to węzły naszego drzewa byłyby obiektami klasy DefaultMutableTreeNode zawierającymi obiekty użytkownika klasy Variable.

Założymy, że inspekcji poddać chcemy zmienną:

```
Employee joe;
```

Zmienna ta posiada typ Employee.class, nazwę "joe" i wartość, którą stanowi referencja do obiektu joe. W programie na listingu 6.18 zdefiniujemy klasę Variable służącą reprezentacji zmiennych:

```
Variable v = new Variable(Employee.class, "joe", joe);
```

Jeśli zmienna jest typu podstawowego, musimy użyć dla jej wartości obiektu opakowującego.

```
new Variable(double.class, "salary", new Double(salary));
```

Jeśli typem zmiennej jest klasa, to zawierać będzie ona pola. Korzystając z mechanizmu refleksji, dokonujemy wyliczenia wszystkich pól i umieszczamy je w tablicy ArrayList. Ponieważ metoda getFields klasy Class nie zwraca pól klasy bazowej, to musimy ją dodatkowo wywołać dla wszystkich klas bazowych danej klasy. Odpowiedni kod odnajdziemy w konstruktorze klasy Variable. Metoda getFields klasy Variable zwraca tablicę pól, natomiast metoda toString — łańcuch znaków opisujący węzeł drzewa. Opis ten zawiera zawsze typ i nazwę zmiennej. Jeśli zmienna jest typu podstawowego, to opis zawiera także jej wartość.



Jeśli typem zmiennej jest tablica, to program nie wyświetla jej elementów. Nie jest to trudne zadanie i pozostawiamy je jako ćwiczenie dla czytelników.

Przejdźmy teraz do omówienia modelu drzewa. Pierwsze dwie jego metody są bardzo proste.

```
public Object getRoot()
{
    return root;
}

public int getChildCount(Object parent)
{
    return ((Variable)parent).getFields().size();
}
```

Metoda getChild zwraca nowy obiekt klasy Variable opisujący dane pole. Metody getType oraz getName klasy Field udostępniają typ i nazwę pola. Korzystając z mechanizmu refleksji, możemy odczytać wartość pola za pomocą wywołania f.get(parentValue). Metoda ta może wyrzucić wyjątek IllegalAccessException. Ponieważ w konstruktorze udostępniamy wszystkie pola, to wyjątek ten nie powinien się pojawić.

Poniżej przedstawiamy kompletny kod metody getChild.

```
public Object getChild(Object parent, int index)
{
    ArrayList fields = ((Variable) parent).getFields();
    Field f = (Field) fields.get(index);
    Object parentValue = ((Variable) parent).getValue();
    try
    {
        return new Variable(f.getType(), f.getName(),
            f.get(parentValue));
    }
    catch(IllegalAccessException e)
    {
        return null;
    }
}
```

Powyższe trzy metody udostępniają strukturę drzewa komponentowi klasy JTree. Implementacja pozostałych metod jest rutynowa (patrz listing 6.17).

Drzewo w naszym przykładzie jest strukturą *nieskończoną*. Możemy to sprawdzić, dokonując inspekcji jednego z obiektów typu WeakReference. Jeśli wybierzemy jego zmienną o nazwie referent, to powrócimy do wyjściowego obiektu. Jego poddrzewo możemy w ten sposób rozwijać w nieskończoność. Oczywiście program nie przechowuje nieskończonego zbioru węzłów, a jedynie tworzy je na żądanie, gdy użytkownik rozwija kolejne poddrzewa. Na listingu 6.16 przedstawiona została główna klasa przykładowego programu.

### javax.swing.tree.TreeModel 1.2

- `Object getRoot`  
zwraca korzeń drzewa.
- `int getChildCount(Object parent)`  
zwraca liczbę węzłów podległych węzła parent.
- `Object getChild(Object parent, int index)`  
zwraca węzeł podległy węzła parent o danym indeksie.
- `int getIndexofChild(Object parent, Object child)`  
zwraca indeks węzła child. Węzeł ten musi być węzłem podległym węzła parent.
- `boolean isLeaf(Object node)`  
zwraca wartość true, jeśli węzeł node jest koncepcyjnym liściem.
- `void addTreeModelListener(TreeModelListener l)`
- `void removeTreeModelListener(TreeModelListener l)`  
Dodają i usuwają obiekty nasłuchujące powiadamiane w momencie zmiany danych modelu.

- void valueForPathChanged(TreePath path, Object newValue)  
metoda wywoływana, gdy edytor komórki zmodyfikował węzeł.  
*Parametry:* path ścieżka do zmodyfikowanego węzła,  
newValue nowa wartość zwrócona przez edytor.

 **javax.swing.event.TreeModelListener 1.2**

- void treeNodesChanged(TreeModelEvent e)
- void treeNodesInserted(TreeModelEvent e)
- void treeNodesRemoved(TreeModelEvent e)
- void treeStructureChanged(TreeModelEvent e)

Wywoływane przez model drzewa, gdy jego dane uległy zmianie.

 **javax.swing.event.TreeModelEvent 1.2**

- TreeModelEvent(Object eventSource, TreePath node)  
tworzy model zdarzeń drzewa.  
*Parametry:* eventSource model drzewa generujący zdarzenia,  
node ścieżka do węzła, który został zmodyfikowany.

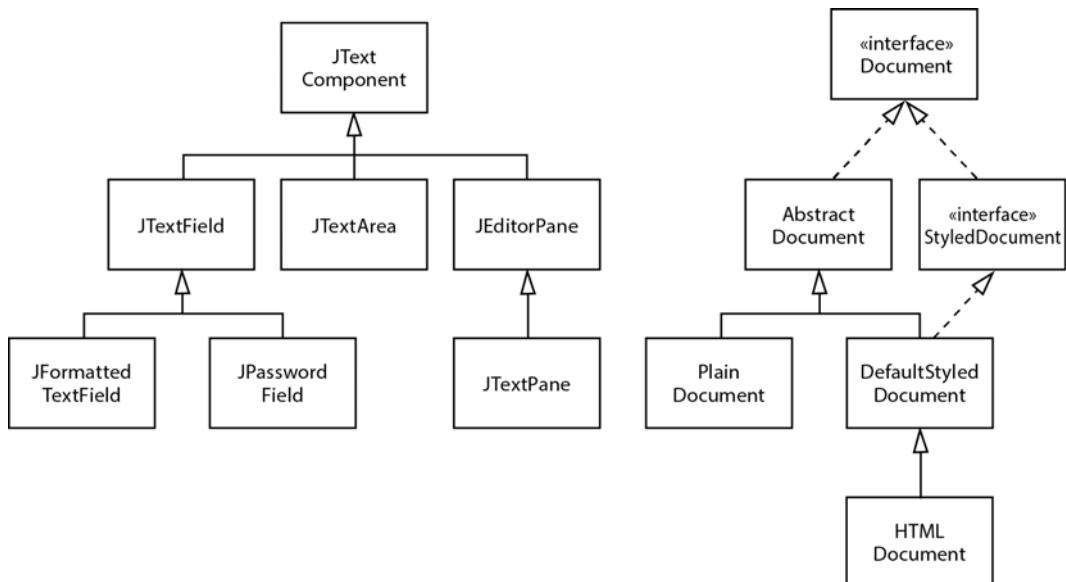
## 6.4. Komponenty tekstowe

Na rysunku 6.35 przedstawione zostały wszystkie komponenty tekstowe oferowane przez bibliotekę Swing. Trzy najczęściej używane, JTextField, JPasswordField i JTextArea, omówiliśmy w rozdziale 9. książki *Java. Podstawy*. W kolejnych częściach bieżącego rozdziału zajmiemy się pozostałymi komponentami tekstowymi. Omówimy również komponent JSpinner, który zawiera sformatowane pole tekstowe wraz z przyciskami pozwalającymi zmieniać jego zawartość.

Wszystkie komponenty tekstowe wyświetlają dane przechowywane przez obiekt modelu należący do klasy implementującej interfejs Document oraz umożliwiają ich edycję. Komponenty JTextField i JTextArea używają w tym celu obiektów PlainDocument, które po prostu przechowują sekwencję wierszy tekstu bez jakiegokolwiek formatowania.

Klasa JEditorPane umożliwia prezentację i edycję sformatowanego tekstu (różną czcionką i kolorem), najczęściej w formacie HTML. Więcej na ten temat w punkcie 6.4.4, „Prezentacja HTML za pomocą JEditorPane”. Interfejs StyledDocument opisuje dodatkowe wymagania związane ze stylami, czcionkami i kolorami tekstu. Interfejs ten implementuje klasę HTMLDocument.

Klasa JTextPane będąca klasą pochodną klasy JEditorPane również umożliwia przechowywanie sformatowanego tekstu, a także zagnieźdzonych w nim komponentów biblioteki Swing. W tej książce nie będziemy zajmować się skomplikowanym komponentem JTextPane,



**Rysunek 6.35.** Hierarchia komponentów tekstowych i dokumentów

lecz odeślemy Czytelnika do szczegółowego opisu w książce *Core Swing* autorstwa Kima Topleya. Typowy przykład zastosowania klasy `JTextPane` można znaleźć w kodzie programu StylePad dołączonym do przykładów pakietu JDK.

## 6.4.1. Śledzenie zmian zawartości komponentów tekstowych

Większość szczegółów interfejsu `Document` jest interesująca dla programisty, jeśli implementuje on własny edytor tekstu. Istnieje jednak jeszcze jedno powszechnie zastosowanie tego interfejsu: śledzenie zmian.

Często chcemy zaktualizować część interfejsu użytkownika natychmiast po tym, jak zmieniony jakiś tekst. W takiej sytuacji nie chcemy czekać, aż użytkownik sam powiadomi nas o tym, klikając jakiś przycisk. Oto prosty przykład. Okno programu będzie zawierać trzy pola tekstowe umożliwiające określenie czerwonej, niebieskiej i zielonej składowej koloru. Za każdym razem, gdy użytkownik zmieni zawartość któregoś pola, kolor powinien zostać natychmiast zaktualizowany. Na rysunku 6.36 został przedstawiony efekt działania programu z listingu 6.19.

**Rysunek 6.36.**  
Śledzenie zmian  
zawartości pola  
tekstowego



**Listing 6.19.** `textChange/ColorFrame.java`

```

package textChange;

import java.awt.*;
import javax.swing.*;
  
```

```

import javax.swing.event.*;

/**
 * Ramka zawierająca trzy pola tekstowe określające kolor tła.
 */
public class ColorFrame extends JFrame
{
    private JPanel panel;
    private JTextField redField;
    private JTextField greenField;
    private JTextField blueField;

    public ColorFrame()
    {
        DocumentListener listener = new DocumentListener()
        {
            public void insertUpdate(DocumentEvent event) { setColor(); }
            public void removeUpdate(DocumentEvent event) { setColor(); }
            public void changedUpdate(DocumentEvent event) {}
        };

        panel = new JPanel();
        panel.add(new JLabel("Red:"));
        redField = new JTextField("255", 3);
        panel.add(redField);
        redField.getDocument().addDocumentListener(listener);

        panel.add(new JLabel("Green:"));
        greenField = new JTextField("255", 3);
        panel.add(greenField);
        greenField.getDocument().addDocumentListener(listener);

        panel.add(new JLabel("Blue:"));
        blueField = new JTextField("255", 3);
        panel.add(blueField);
        blueField.getDocument().addDocumentListener(listener);

        add(panel);
        pack();
    }

    /**
     * Nadaje tłu kolor zgodnie z wartościami pól tekstowych.
     */
    public void setColor()
    {
        try
        {
            int red = Integer.parseInt(redField.getText().trim());
            int green = Integer.parseInt(greenField.getText().trim());
            int blue = Integer.parseInt(blueField.getText().trim());
            panel.setBackground(new Color(red, green, blue));
        }
        catch (NumberFormatException e)
        {
            // nie zmienia koloru, jeśli dane nie mogą być parsowane
        }
    }
}

```

Na początek zauważmy, że monitorowanie klawiszy nie jest dobrym rozwiązańiem naszego problemu. Niektóre klawisze (na przykład ze strzałkami) nie zmieniają zawartości pola tekstuowego. Co więcej, tekst może zostać zmodyfikowany również za pomocą myszy. Dlatego też poprosimy *dokument* (a nie komponent tekstowy), aby powiadamał nas za każdym razem, gdy tekst ulegnie zmianie. W tym celu zainstalujemy *obiekt nasłuchujący dokumentu*:

```
textField.getDocument().addDocumentListener(listener);
```

Gdy tekst się zmieni, wywołana zostanie jedna z metod interfejsu DocumentListener:

```
void insertUpdate(DocumentEvent event)
void removeUpdate(DocumentEvent event)
void changedUpdate(DocumentEvent event)
```

Pierwsze dwie metody są wywoływanie, gdy w tekście wstawiono lub usunięto znaki. Trzecia metoda nie jest wywoływanie dla wszystkich pól tekstowych. Jest ona wywoływanie w przypadku bardziej zaawansowanych dokumentów na skutek zmian, które zaszły w formatowaniu tekstu. Niestety nie ma jednej metody, która powiadamiałyby nas o zmianie tekstu, a przecież zwykle nie interesuje nas sposób, w jaki tekst się zmienił. Nie ma również odpowiedniego adaptera klasy. Dlatego nasz obiekt nasłuchujący dokumentu musi implementować wszystkie trzy metody:

```
DocumentListener listener = new DocumentListener()
{
    public void insertUpdate(DocumentEvent event) { setColor(); }
    public void removeUpdate(DocumentEvent event) { setColor(); }
    public void changedUpdate(DocumentEvent event) {}
}
```

Metoda `setColor` używa metody `getText` w celu pobrania łańcuchów z pól tekstowych i konfiguruje odpowiedni kolor.

Nasz program ma jedną poważną wadę. Użytkownik może wprowadzić niepoprawny tekst lub pozostawić niewypełnione pole. W tej wersji programu obsługujemy wyjątek `NumberFormatException` wyrzucany przez metodę `parseInt` i po prostu nie zmieniamy koloru, gdy zawartość pola tekstowego nie jest liczbą. W następnej wersji pokażemy, w jaki sposób można zapobiec wprowadzaniu błędnego tekstu.



 Zamiast nasłuchiwać zdarzeń dokumentu, można dodać obiekt nasłuchujący do pola tekstopowego. Będzie on powiadamiany za każdym razem, gdy użytkownik naciśnie klawisz *Enter*. Nie polecamy takiego rozwiązania, ponieważ użytkownicy często nie pamiętają o naciśnięciu klawisza *Enter* po wprowadzeniu danych. W takiej sytuacji konieczne staje się zainstalowanie również obiektu nasłuchującego, tym razem zdarzeń polegających na opuszczeniu pola tekstopowego przez użytkownika i przejęciu do innego komponentu.

API `javax.swing.JComponent` 1-2

- Dimension getPreferredSize()
  - void setPreferredSize(Dimension d)

**API javax.swing.text.Document 1.2**

- int getLength()  
zwraca bieżącą liczbę znaków dokumentu.
- String getText(int offset, int length)  
zwraca podany zakres tekstu dokumentu.  
*Parametry:* offset początek tekstu,  
length długość fragmentu.
- void addDocumentListener(DocumentListener listener)  
rejestruje obiekt nasłuchujący powiadamiany o zmianach dokumentu.

**API javax.swing.event.DocumentEvent 1.2**

- Document getDocument()  
zwraca dokument będący źródłem zdarzenia.

**API javax.swing.event.DocumentListener 1.2**

- void changedUpdate(DocumentEvent event)  
metoda wywoływana za każdym razem, gdy zmieni się atrybut lub zbiór atrybutów dokumentu.
- void insertUpdate(DocumentEvent event)  
metoda wywoływana za każdym razem, gdy do dokumentu wstawiono nowe znaki.
- void removeUpdate(DocumentEvent event)  
metoda wywoywana za każdym razem, gdy usunięto fragment dokumentu.

## 6.4.2. Sformatowane pola wejściowe

W naszym ostatnim przykładowym programie chcielibyśmy, aby użytkownik wprowadzał tylko liczby, a nie dowolne łańcuchy. Czyli należy pozwolić użytkownikowi na wpisywanie tylko cyfr od 0 do 9 oraz znaku minus. Ten ostatni, jeśli zostanie w ogóle wprowadzony, musi być zawsze *pierwszym* symbolem łańcucha.

Na pierwszy rzut oka kontrola poprawności danych wprowadzanych przez użytkownika wydaje się łatwym zadaniem. Możemy przecież zainstalować obiekty nasłuchujące klawiszy dla każdego pola tekstowego i konsumować wszystkie zdarzenia, które nie są związane z cyframi i minusem. Niestety takie proste rozwiążanie nie sprawdzi się w praktyce. Po pierwsze, nie każda kombinacja cyfr i znaku minusa jest poprawną liczbą. Na przykład kombinacje postaci --3 lub 3-3 nie są poprawne, choć składają się z poprawnych znaków. Co gorsza, istnieją przecież sposoby modyfikowania tekstu niewymagające używania klawiszy odpowiadających dozwolonym znakom. Na przykład kombinacja *Ctrl +V* (w przypadku wyglądu Metal komponentów Swing) umożliwia wklejanie zawartości bufora w polach tekstowych. Musielibyśmy

zatem monitorować również, czy użytkownik nie wkleja niepoprawnych znaków. W ten sposób kontrola zawartości pól tekstowych poprzez filtrowanie naciśnień klawiszy staje się coraz bardziej skomplikowanym zadaniem.

W wersjach wcześniejszych niż Java SE 1.4 nie było komponentów umożliwiających wprowadzanie wartości numerycznych. W pierwszym wydaniu niniejszej książki pokazaliśmy własną klasę `IntTextField` dostarczającą implementacji pola tekstowego umożliwiającego wprowadzanie poprawnych wartości całkowitych. W kolejnych wydaniach zmienialiśmy jej implementację, starając się wykorzystać różne półśrodkи w zakresie kontroli poprawności danych, wprowadzane w kolejnych edycjach języka Java. W końcu w wersji Java SE 1.4 projektanci biblioteki Swing stanęli na wysokości zadania i dostarczyli uniwersalnej klasy `JFormattedTextField`, która umożliwia nie tylko poprawne wprowadzanie danych numerycznych, ale również dat, a także bardziej abstrakcyjnych wartości, takich jak na przykład adresy IP.

#### 6.4.2.1. Wprowadzanie wartości całkowitych

Zaczniemy od najprostszego przypadku: pola tekstowego do wprowadzania wartości całkowitych.

```
JFormattedTextField intField = new
    ↪JFormattedTextField(NumberFormat.getIntegerInstance());
```

Wywołanie metody `NumberFormat.getIntegerInstance()` zwraca obiekt formatujący wartości całkowite przy użyciu bieżącego lokalizatora. W przypadku lokalizatora dla USA przecinek jest używany do oddzielania tysięcy, milionów itp. i wobec tego użytkownik może wprowadzić na przykład poprawną wartość całkowitą postaci 1,729. Szczegóły korzystania z lokalizatorów zostały omówione w rozdziale 5.

Za pomocą poniższego wywołania możemy określić liczbę kolumn pola tekstowego:

```
intField.setColumns(6);
```

Metoda `setValue()` umożliwia określenie wartości domyślnej pola. Jej parametr jest typu `Object` i wobec tego musimy obudować wartość domyślną obiektem typu `Integer`:

```
intField.setValue(new Integer(100));
```

Zwykle użytkownik wypełnia szereg pól tekstowych, a następnie wybiera przycisk, akceptując wprowadzone wartości. Po kliknięciu przycisku możemy odczytać wprowadzone wartości za pomocą metody `getValue()`. Metoda ta zwraca wynik typu `Object`, który musimy rzutować na właściwy typ. `JFormattedTextField` zwraca obiekt typu `Long`, jeśli użytkownik edytował wartość. Jeśli jednak użytkownik nie wprowadził żadnych zmian, zwrócony zostaje oryginalny obiekt `Integer`. Dlatego też musimy rzutować zwrócony obiekt na wspólną klasę bazową obu wymienionych typów, czyli klasę `Number`:

```
Number value = (Number) intField.getValue();
int v = value.intValue();
```

Sformatowane pola tekstowe same w sobie nie są szczególnie interesujące, dopóki nie zajmiemy się sytuacją, w której użytkownik wprowadził niepoprawną wartość. Zajmiemy się nią w następnej części tego rozdziału.

### 6.4.2.2. Kontrola poprawności wprowadzonych danych

Po wprowadzeniu danych użytkownik opuszcza pole tekstowe zwykle poprzez kliknięcie myszą kolejnego komponentu. Wtedy do akcji wkracza obiekt formatujący. Jeśli potrafi on zamienić wprowadzony łańcuch na obiekt, to znaczy, że użytkownik wprowadził poprawne dane. W przeciwnym razie wprowadzony tekst zostaje uznany za niepoprawny. Metoda `isValid` umożliwia sprawdzenie, czy bieżąca zawartość pola tekstowego jest poprawna.

Domyślnym zachowaniem po opuszczeniu pola przez użytkownika jest `COMMIT_OR_REVERT`. Jeśli łańcuch wprowadzony przez użytkownika jest poprawny, zostaje *zatwierdzony*. Obiekt formatujący przekształca łańcuch znaków w obiekt. Powstały obiekt staje się bieżącą wartością pola (czyli wartością zwracaną przez wspomnianą już metodę `getValue`). Wartość ta zostaje zamieniona z powrotem na łańcuch znaków, który zostaje wyświetlony w polu tekstowym. Na przykład: jeśli obiekt formatujący wartości całkowite rozpoznał łańcuch 1729 jako poprawną wartość, to bieżącą wartością pola staje się obiekt `new Long(1729)`, który zostaje zamieniony na łańcuch postaci 1,729 umieszczony z powrotem w polu tekstowym.

Jeśli łańcuch wprowadzony przez użytkownika nie jest poprawny, to bieżąca wartość pola nie zostaje zmieniona, a w polu zostaje z powrotem umieszczony łańcuch reprezentujący tę wartość. Jeśli użytkownik wprowadzi na przykład niepoprawną wartość `x1`, to po przejściu do innego komponentu w polu tekstowym zostanie przywrócona poprzednia, poprawna wartość.



Obiekt formatujący wartości całkowite uważa tekst za poprawny, jeśli rozpoczyna się on wartością całkowitą. Na przykład łańcuch `1729x` zostanie uznany za poprawny. Zostanie on przekształcony na wartość `1729`, która z kolei zostanie sformatowana jako łańcuch `1,729`.

Sposób zachowania pola po jego opuszczeniu przez użytkownika możemy określić za pomocą metody `setFocusLostBehavior`. Zachowanie `COMMIT` różni się subtelnie od domyślnego. Jeśli łańcuch jest niepoprawny, to tekst pola i jego wartość pozostają niezmienione, czyli utracona zostaje ich synchronizacja. Zachowanie `PERSIST` jest jeszcze bardziej konserwatywne. Nawet jeśli łańcuch jest poprawny, tekst pola i jego wartość pozostają niezmienione. Aby przywrócić ich synchronizację, należy użyć metod `commitEdit`, `setValue` lub `setText`. Zachowanie `REVERT` wydaje się zupełnie nieprzydatne. Dane wprowadzone przez użytkownika zostają porzucone, a w polu zostaje umieszczona poprzednia wartość.



Zachowanie `COMMIT_OR_REVERT` wydaje się najbardziej sensowne. Istnieje jednak pewien potencjalny problem. Przypuśćmy, że okno dialogowe zawiera pole tekstowe służące do wprowadzania wartości całkowitej. Użytkownik wprowadza w nim łańcuch `"1729"` (ze znakiem spacji na początku) i kliką przycisk `OK`. Spacja na początku łańcucha sprawia, że wartość zostaje uznana za niepoprawną i zostaje przywrócona poprzednia wartość pola. Obiekt nasłuchujący przycisku `OK` pobiera wartość pola i zamyka okno dialogowe. W ten sposób użytkownik nie dowie się, że wprowadzona przez niego wartość została odrzucona. W takim przypadku warto wybrać zachowanie `COMMIT` i zaimplementować obiekt nasłuchujący przycisku `OK` w taki sposób, aby przed zamknięciem okna sprawdzał, czy wartości wszystkich pól są poprawne.

### 6.4.2.3. Filtry

Podstawowa funkcjonalność sformatowanych pól tekstowych jest łatwa do opanowania przez programistę i wystarczająca w większości przypadków. Możemy jednak spróbować wprowadzić w niej trochę ulepszeń. Na przykład uniemożliwić użytkownikowi wprowadzanie znaków, które nie są cyframi. Takie zachowanie możemy osiągnąć przez zastosowanie *filtru dokumentu*. Przypomnijmy, że w architekturze model-widok-nadzorca, nadzorca przekłada zdarzenia wejścia na polecenia, które modyfikują dokument wykorzystywany przez pole tekstowe, czyli łańcuch przechowywany w obiekcie PlainDocument. Na przykład gdy nadzorca przetwarza polecenie powodujące wstawienie tekstu w dokumencie, wywołuje polecenie "insert string". Wstawiany łańcuch może być pojedynczym znakiem lub zawartością bufora klejania. Filtr dokumentu może przechwycić to polecenie i zmodyfikować wstawiany łańcuch lub nawet anulować operację wstawiania. Poniżej przedstawiamy kod metody `insertString` filtra, który analizuje wstawiany łańcuch i wstawia jedynie znaki cyfr i minusa. (Kod obsługuje uzupełniające znaki Unicode, co zostało omówione w rozdziale 3. książki *Java. Podstawy*. Patrz również omówienie klasy `StringBuilder` w rozdziale 1.).

```
public void insertString(FilterBypass fb, int offset, String string, AttributeSet
    attr)
    throws BadLocationException
{
    StringBuilder builder = new StringBuilder(string);
    for (int i = builder.length() - 1; i >= 0; i--)
    {
        int cp = builder.codePointAt(i);
        if (!Character.isDigit(cp) && cp != '-')
        {
            builder.deleteCharAt(i);
            if (Character.isSupplementaryCodePoint(cp))
            {
                i--;
                builder.deleteCharAt(i);
            }
        }
    }
    super.insertString(fb, offset, builder.toString(), attr);
}
```

Należy również dostarczyć własną wersję metody `replace` klasy `DocumentFilter`. Jest ona wywoływana, gdy tekst zostaje wybrany i następnie zastąpiony. Implementacja tej metody jest prosta — patrz listing 6.21.

Pozostaje teraz zainstalować filtr dokumentu. Niestety, nie jest to takie proste. Należy zastąpić metodę `getDocumentFilter` klasy formatującą i przekazać obiekt tej klasy polu `JFormattedTextField`. Pole umożliwiające wprowadzanie wartości całkowitych używa obiektu klasy `InternationalFormatter` inicjowanego za pomocą metody `NumberFormat.getIntegerInstance()`. Oto sposób instalacji obiektu formatującego w celu uzyskania żądanego filtra:

```
JFormattedTextField intField = new JFormattedTextField(new
    InternationalFormatter(NumberFormat.getIntegerInstance()))
{
    private DocumentFilter filter = new IntFilter();
    protected DocumentFilter getDocumentFilter()
    {
```

```

        return filter;
    }
}:

```



Dokumentacja Java SE stwierdza, że klasa DocumentFilter została stworzona, aby uniknąć tworzenia klas pochodnych. Przed wprowadzeniem Java SE 1.3 filtrowanie pól tekstowych osiągano poprzez tworzenie klas pochodnych klasy PlainDocument i zastąpienie jej metod insertString i replace. Obecnie klasa PlainDocument dysponuje możliwością dołączania filtrów. Jest to z pewnością doskonałe usprawnienie. Byłoby jeszcze lepsze, gdyby filtr można było również dołączać do klasy formatującej. Skoro jednak tak nie jest, musimy tworzyć klasy pochodne klasy formatującej.

Wypróbujmy działanie programu FormatTest zamieszczonego na końcu tego punktu. Trzecie pole tekstowe ma zainstalowany filtr. Możemy wprowadzać w nim tylko cyfry i znak minusa. Zauważmy, że nadal możemy wprowadzić niepoprawny łańcuch, na przykład "1-2-3". W ogólnym przypadku nie ma możliwości zapobieżenia wprowadzaniu niepoprawnych łańcuchów poprzez filtrowanie. Na przykład łańcuch " - " nie jest poprawny, ale filtr nie może go odrzucić, ponieważ łańcuch ten jest przedrostkiem poprawnego łańcucha "-1". Chociaż filtry nie umożliwiają pełnej kontroli nad wprowadzonymi danymi, ich zastosowanie pozwala odrzucać łańcuchy, które są niepoprawne w oczywisty sposób.



Innym zastosowaniem filtrów jest zamiana wszystkich znaków łańcucha na duże litery. Taki filtr można bardzo łatwo zaimplementować. Wystarczy w metodach `insert` → `String` i `replace` zamieniać znaki wstawianego łańcucha na duże litery i następnie wywoływać odpowiednią metodę klasy bazowej.

#### 6.4.2.4. Weryfikatory

Istnieje jeszcze jeden potencjalnie użyteczny mechanizm, który można wykorzystać do zapobiegania wprowadzaniu niepoprawnych wartości przez użytkownika. Do komponentu JComponent możemy dołączyć *weryfikator*. Weryfikator wkracza do akcji, gdy użytkownik opuszcza pole tekstowe. Jeśli uzna wartość pola za niepoprawną, kurSOR wraca natychmiast do danego pola tekstowego i użytkownik zostaje w ten sposób zmuszony do poprawienia wprowadzonej wartości.

Weryfikator musi rozszerzać klasę abstrakcyjną InputVerifier i definiować metodę `verify`. Zdefiniowanie weryfikatora, który sprawdza zawartość sformatowanego pola tekstowego, jest bardzo proste. Metoda `isValid` klasy JFormattedTextField wywołuje obiekt formatujący i zwraca wartość true, jeśli obiekt formatujący potrafi przekształcić łańcuch tekstu w obiekt. Oto kod weryfikatora dołączzonego do pola JFormattedTextField:

```

intField.setInputVerifier(new InputVerifier()
{
    public boolean verify(JComponent component)
    {
        JFormattedTextField field = (JFormattedTextField) component;
        return field.isValid();
    }
}):

```

Czwarте pole tekstowe przykładowego programu posiada dołączony weryfikator. Spróbujmy wprowadzić w nim niepoprawną wartość (na przykład x1729), a następnie nacisnąć klawisz *Tab* lub kliknąć myszą inne pole. Zauważmy, że kursor natychmiast wróci do pola zawierającego niepopawną wartość. Jeśli jednak klikniemy przycisk *OK*, to obiekt nasłuchujący przycisku wywoła metodę *getValue* i pobierze ostatnią poprawną wartość.

Jednak weryfikator nie gwarantuje całkowitej poprawności danych. Jeśli klikniemy przycisk, to obiekty nasłuchujące przycisku zostaną powiadomione, zanim kursor wróci do pola zawierającego niepopawną wartość. Obiekty nasłuchujące mogą wtedy pobrać niepopawną wartość, mimo że nie przeszła ona weryfikacji. Istnieje jednak uzasadniony powód takiego działania: dzięki temu użytkownik może zrezygnować z poprawiania wartości i wybrać przycisk *Cancel*.

#### 6.4.2.5. Inne standardowe obiekty formatujące

Klasa *JFormattedTextField* pozwala na zastosowanie jeszcze kilku innych obiektów poza obiektem formatującym wartości całkowite. Klasa *NumberFormat* udostępnia metody statyczne:

```
getNumberInstance  
getCurrencyInstance  
getPercentInstance
```

które tworzą obiekty formatujące wartości zmienoprzecinkowe, wartości wyrażone w walutach oraz procentach. Na przykład poniższe wywołanie tworzy pole tekstowe umożliwiające wprowadzanie wartości wyrażonych w walutach:

```
JFormattedTextField currencyField = new  
    ↪JFormattedTextField(NumberFormat.getCurrencyInstance());
```

Aby dokonać edycji daty i czasu, wywołujemy jedną z następujących metod statycznych klasy *DateFormat*:

```
getDateInstance  
getTimeInstance  
getDateTimeInstance
```

Na przykład:

```
JFormattedTextField dateField = new  
    ↪JFormattedTextField(DateFormat.getDateInstance());
```

Utworzane w ten sposób pole tekstowe umożliwia wprowadzenie daty w formacie domyślnym lub formacie "medium":

Aug 5, 2007

Możemy również wybrać inny format, na przykład "short":

8/5/07

wywołując

```
DateFormat.getDateInstance(DateFormat.SHORT)
```



Domyślnym formatem daty jest "lenient". W formacie tym niepoprawna data February 31, 2002 zostanie zastąpiona następną poprawną datą, to jest March 3, 2002. Takie zachowanie programu może być sporą niespodzianką dla użytkownika. Jeśli chcemy jej zapobiec, powinniśmy użyć wywołania `setLenient(false)` dla obiektu `DateFormat`.

`DefaultFormatter` może formatować obiekty dowolnej klasy, która ma konstruktor o parametrze typu `String` i metodę `toString`. Na przykład klasa `URL` dysponuje konstruktorem `URL(String)`, który umożliwia tworzenie obiektu `URL` na podstawie łańcucha znaków:

```
URL url = new URL("http://horstmann.com");
```

Wobec tego możemy zastosować klasę `DefaultFormatter` do formatowania obiektów `URL`. Obiekt formatujący wywołuje metodę `toString` w celu zainicjowania zawartości pola tekstowego. Gdy użytkownik zakończy edycję zawartości pola, obiekt formatujący tworzy nowy obiekt tej samej klasy co bieżąca wartość pola, używając w tym celu konstruktora o parametrze typu `String`. Jeśli konstruktor ten wyrzuci wyjątek, to znaczy, że użytkownik wprowadził niepoprawną wartość. Możemy przetestować tę sytuację w programie przykładowym, wprowadzając adres `URL`, który nie rozpoczyna się od przedrostka "http:".



Domyślnie obiekt `DefaultFormatter` pracuje w trybie *nadpisywania*. Tym różni się od innych obiektów formatujących i sytuacja taka nie jest zwykle korzystna. Tryb ten można wyłączyć za pomocą wywołania `setOverwriteMode(false)`.

Klasa `MaskFormatter` przydaje się w przypadku wzorców o stałym rozmiarze zawierających zarówno stałe znaki, jak i znaki zmieniane przez użytkownika. Na przykład numer katalogowy (postaci 078-05-1120) możemy sformatować za pomocą poniższego obiektu:

```
new MaskFormatter("###-##-####")
```

Symbol `#` oznacza tutaj jedną cyfrę. W tabeli 6.3 zostały przedstawione symbole, których możemy używać w obiektach klasy `MaskFormatter`.

**Tabela 6.3.** Symbole stosowane dla obiektów klasy `MaskFormatter`

Symbol	Objaśnienie
#	Cyfra
?	Litera
U	Litera, zamieniana na dużą literę
L	Litera, zamieniana na małą literę
A	Litera lub cyfra
H	Cyfra szesnastkowa [0-9A-Fa-f]
*	Dowolny znak
'	Znak sterujący pozwalający umieścić symbol we wzorcu

Znaki wprowadzane w polu tekstowym możemy ograniczyć, wywołując jedną z poniższych metod klasy `MaskFormatter`:

```
setValidCharacters
setInvalidCharacters
```

Na przykład aby umożliwić wczytywanie ocen (w skali od 2 do 6), użyjemy

```
MaskFormatter formatter = new MaskFormatter("U*");
formatter.setValidCharacters("23456+- ");
```

Nie można jednak w żaden sposób określić, że drugi znak nie powinien być cyfrą.

Zwróćmy uwagę, że łańcuch formatowany za pomocą klasy MaskFormatter ma taką samą długość jak maska. Jeśli podczas edycji użytkownik usunie jakiś znak, to zostanie on zastąpiony znakiem-wypełniaczem. Domyślnie rolę tego znaku spełnia spacja, ale można to zmienić za pomocą metody setPlaceholderCharacter, na przykład:

```
formatter.setPlaceholderCharacter('0');
```

Domyślnie MaskFormatter działa w trybie nadpisywania, co jest rozwiązaniem zgodnym z intuicją użytkownika. Zwróćmy również uwagę na to, że kursor przeskakuje przez stałe znaki maski.

Klasa MaskFormatter jest efektywnym rozwiązaniem w przypadku wzorców o stałej długości. Ponieważ nie dopuszcza jednak żadnych zmian maski, nie może być stosowana na przykład w przypadku międzynarodowych numerów telefonicznych, które różnią się pomiędzy sobą liczbą cyfr.

#### 6.4.2.6. Nietypowe obiekty formatujące

Jeśli nie odpowiada nam żaden ze standardowych obiektów formatujących, to łatwo możemy zdefiniować własny. Zastanówmy się na przykład nad 4-bajtowym adresem IP, takim jak:

```
130.65.86.66
```

Nie możemy zastosować w tym wypadku klasy MaskFormatter, ponieważ każdy bajt adresu może być reprezentowany przez jedną, dwie lub trzy cyfry. Chcemy również sprawdzić, że wartość każdego bajta nie jest większa od 255.

Aby zdefiniować własny obiekt formatujący, musimy rozszerzyć klasę DefaultFormatter i zastąpić jej metody:

```
String valueToString(Object value)
Object stringToValue(String text)
```

Pierwsza z tych metod przekształca wartość pola na łańcuch wyświetlany w tym polu tekstowym. Druga parsuje tekst wprowadzony przez użytkownika i przekształca go z powrotem na obiekt. Jeśli któraś metoda wykryje błąd, powinna wyrzucić wyjątek ParseException.

W naszym przykładowym programie przechowujemy adres IP w tablicy byte[] o rozmiarze 4. Metoda valueToString tworzy łańcuch znaków, w którym wartości poszczególnych bajtów są oddzielone kropkami. Pamiętajmy przy tym, że wartości typu byte mają znak i należą do przedziału od -128 do 127. (Na przykład dla adresu IP równego 130.65.86.66 pierwszy oktet jest w rzeczywistości wartością typu byte równą -126). Aby zamienić wartości ujemne na wartości całkowite bez znaku, dodajemy do nich liczbę 256.

```

public String valueToString(Object value) throws ParseException
{
    if (!(value instanceof byte[]))
        throw new ParseException("Not a byte[]". 0);
    byte[] a = (byte[]) value;
    if (a.length != 4)
        throw new ParseException("Length != 4". 0);
    StringBuilder builder = new StringBuilder();
    for (int i = 0; i < 4; i++)
    {
        int b = a[i];
        if (b < 0) b += 256;
        builder.append(String.valueOf(b));
        if (i < 3) builder.append('.');
    }
    return builder.toString();
}

```

Natomiast metoda `stringToValue` parsuje łańcuch i tworzy obiekt `byte[]`, jeśli łańcuch jest poprawny. W przeciwnym razie wyrzuca wyjątek `ParseException`.

```

public Object stringToValue(String text) throws ParseException
{
    StringTokenizer tokenizer = new StringTokenizer(text, ".");
    byte[] a = new byte[4];
    for (int i = 0; i < 4; i++)
    {
        int b = 0;
        try
        {
            b = Integer.parseInt(tokenizer.nextToken());
        }
        catch (NumberFormatException e)
        {
            throw new ParseException("Not an integer". 0);
        }
        if (b < 0 || b >= 256)
            throw new ParseException("Byte out of range". 0);
        a[i] = (byte) b;
    }
    return a;
}

```

Sprawdźmy działanie pola adresu IP w naszym przykładowym programie. Jeśli wprowadzimy niepoprawny adres, w polu zostanie z powrotem umieszczona poprzednia poprawna wartość. Kompletny kod obiektu formatującego przedstawiamy na listingu 6.22.

Program przedstawiony na listingu 6.20 prezentuje działanie różnych pól tekstowych (patrz rysunek 6.37). Kliknięcie przycisku *OK* powoduje pobranie bieżących wartości tych pól.

#### **Listing 6.20. *textFormat/FormatTestFrame.java***

```

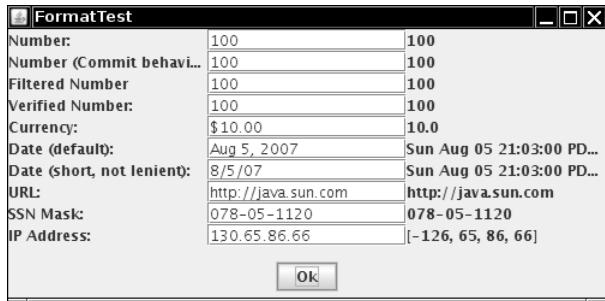
package textFormat;

import java.awt.*;
import java.awt.event.*;
import java.net.*;

```

**Rysunek 6.37.**

Program FormatTest  
w działaniu



```

import java.text.*;
import java.util.*;
import javax.swing.*;
import javax.swing.text.*;

/**
 * Ramka zawierająca kolekcję sformatowanych pól tekstowych
 * oraz przycisk wyświetlający ich wartości.
 */
public class FormatTestFrame extends JFrame
{
    private DocumentFilter filter = new IntFilter();
    private JButton okButton;
    private JPanel mainPanel;

    public FormatTestFrame()
    {
        JPanel buttonPanel = new JPanel();
        okButton = new JButton("Ok");
        buttonPanel.add(okButton);
        add(buttonPanel, BorderLayout.SOUTH);

        mainPanel = new JPanel();
        mainPanel.setLayout(new GridLayout(0, 3));
        add(mainPanel, BorderLayout.CENTER);

        JFormattedTextField intField = new
        JFormattedTextField(NumberFormat.getIntegerInstance());
        intField.setValue(new Integer(100));
        addRow("Number:", intField);

        JFormattedTextField intField2 = new
        JFormattedTextField(NumberFormat.getIntegerInstance());
        intField2.setValue(new Integer(100));
        intField2.setFocusLostBehavior(JFormattedTextField.COMMIT);
        addRow("Number (Commit behavior):", intField2);

        JFormattedTextField intField3 = new JFormattedTextField(new
        ↵InternationalFormatter(
            NumberFormat.getIntegerInstance()))
    {
        protected DocumentFilter getDocumentFilter()
        {
            return filter;
        }
    }
}

```

```
        }));  
intField3.setValue(new Integer(100));  
addRow("Filtered Number", intField3);  
  
JFormattedTextField intField4 = new  
    ↳JFormattedTextField(NumberFormat.getIntegerInstance());  
intField4.setValue(new Integer(100));  
intField4.setInputVerifier(new InputVerifier()  
{  
    public boolean verify(JComponent component)  
    {  
        JFormattedTextField field = (JFormattedTextField) component;  
        return field.isEditValid();  
    }  
});  
addRow("Verified Number:", intField4);  
  
JFormattedTextField currencyField = new JFormattedTextField(NumberFormat  
    .getCurrencyInstance());  
currencyField.setValue(new Double(10));  
addRow("Currency:", currencyField);  
  
JFormattedTextField dateField = new JFormattedTextField(DateFormat.getDateInstance());  
dateField.setValue(new Date());  
addRow("Date (default):", dateField);  
  
DateFormat format = DateFormat.getDateInstance(DateFormat.SHORT);  
format.setLenient(false);  
JFormattedTextField dateField2 = new JFormattedTextField(format);  
dateField2.setValue(new Date());  
addRow("Date (short, not lenient):", dateField2);  
  
try  
{  
    DefaultFormatter formatter = new DefaultFormatter();  
    formatter.setOverwriteMode(false);  
    JFormattedTextField urlField = new JFormattedTextField(formatter);  
    urlField.setValue(new URL("http://java.sun.com"));  
    addRow("URL:", urlField);  
}  
catch (MalformedURLException ex)  
{  
    ex.printStackTrace();  
}  
  
try  
{  
    MaskFormatter formatter = new MaskFormatter("###-##-####");  
    formatter.setPlaceholderCharacter('0');  
    JFormattedTextField ssnField = new JFormattedTextField(formatter);  
    ssnField.setValue("078-05-1120");  
    addRow("SSN Mask:", ssnField);  
}  
catch (ParseException ex)  
{  
    ex.printStackTrace();  
}
```

```
JFormattedTextField ipField = new JFormattedTextField(new IPAddressFormatter());
ipField.setValue(new byte[] { (byte) 130, 65, 86, 66 });
addRow("IP Address:", ipField);
pack();
}

/**
 * Dodaje wiersz w głównym panelu.
 * @param labelText etykieta pola
 * @param field przykładowe pole
 */
public void addRow(String labelText, final JFormattedTextField field)
{
    mainPanel.add(new JLabel(labelText));
    mainPanel.add(field);
    final JLabel valueLabel = new JLabel();
    mainPanel.add(valueLabel);
    okButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            Object value = field.getValue();
            Class<?> cl = value.getClass();
            String text = null;
            if (cl.isArray())
            {
                if (cl.getComponentType().isPrimitive())
                {
                    try
                    {
                        text = Arrays.class.getMethod("toString", cl).invoke(null, value)
                            .toString();
                    }
                    catch (ReflectiveOperationException ex)
                    {
                        // ignoruje wyjątki refleksji
                    }
                }
                else text = Arrays.toString((Object[]) value);
            }
            else text = value.toString();
            valueLabel.setText(text);
        }
    });
}
}
```

---

**Listing 6.21.** *textFormat/IntFilter.java*

```
package textFormat;

import javax.swing.text.*;

/**
 * Filtr ograniczający dane wejściowe
 * do cyfr i znaku minus.
 */
```

```

public class IntFilter extends DocumentFilter
{
    public void insertString(FilterBypass fb, int offset, String string, AttributeSet attr)
        throws BadLocationException
    {
        StringBuilder builder = new StringBuilder(string);
        for (int i = builder.length() - 1; i >= 0; i--)
        {
            int cp = builder.codePointAt(i);
            if (!Character.isDigit(cp) && cp != '-')
            {
                builder.deleteCharAt(i);
                if (Character.isSupplementaryCodePoint(cp))
                {
                    i--;
                    builder.deleteCharAt(i);
                }
            }
        }
        super.insertString(fb, offset, builder.toString(), attr);
    }

    public void replace(FilterBypass fb, int offset, int length, String string,
        AttributeSet attr)
        throws BadLocationException
    {
        if (string != null)
        {
            StringBuilder builder = new StringBuilder(string);
            for (int i = builder.length() - 1; i >= 0; i--)
            {
                int cp = builder.codePointAt(i);
                if (!Character.isDigit(cp) && cp != '-')
                {
                    builder.deleteCharAt(i);
                    if (Character.isSupplementaryCodePoint(cp))
                    {
                        i--;
                        builder.deleteCharAt(i);
                    }
                }
            }
            string = builder.toString();
        }
        super.replace(fb, offset, length, string, attr);
    }
}

```

---

**Listing 6.22.** *textFormat/IPAddressFormatter.java*

```

package textFormat;

import java.text.*;
import java.util.*;
import javax.swing.text.*;

```

```

/**
 * Obiekt formatujący 4-bajtowe adresy IP postaci a.b.c.d
 */
public class IPAddressFormatter extends DefaultFormatter
{
    public String valueToString(Object value) throws ParseException
    {
        if (!(value instanceof byte[])) throw new ParseException("Not a byte[]", 0);
        byte[] a = (byte[]) value;
        if (a.length != 4) throw new ParseException("Length != 4", 0);
        StringBuilder builder = new StringBuilder();
        for (int i = 0; i < 4; i++)
        {
            int b = a[i];
            if (b < 0) b += 256;
            builder.append(String.valueOf(b));
            if (i < 3) builder.append('.');
        }
        return builder.toString();
    }

    public Object stringToValue(String text) throws ParseException
    {
        StringTokenizer tokenizer = new StringTokenizer(text, ".");
        byte[] a = new byte[4];
        for (int i = 0; i < 4; i++)
        {
            int b = 0;
            if (!tokenizer.hasMoreTokens()) throw new ParseException("Too few bytes", 0);
            try
            {
                b = Integer.parseInt(tokenizer.nextToken());
            }
            catch (NumberFormatException e)
            {
                throw new ParseException("Not an integer", 0);
            }
            if (b < 0 || b >= 256) throw new ParseException("Byte out of range", 0);
            a[i] = (byte) b;
        }
        if (tokenizer.hasMoreTokens()) throw new ParseException("Too many bytes", 0);
        return a;
    }
}

```



Na stronach „Swing Connection” znajdziesz krótki artykuł opisujący obiekt formatujący wykorzystujący dowolne wyrażenie regularne. Patrz [www.oracle.com/technet/work/java/reftf-138955.html](http://www.oracle.com/technet/work/java/reftf-138955.html).

#### javax.swing.JFormattedTextField 1.4

- `JFormattedTextField(Format fmt)`

tworzy pole tekstowe, które używa podanego formatu.

- `JFormattedTextField(JFormattedTextField.AbstractFormatter formatter)`  
tworzy pole tekstowe, które używa podanego obiektu formatującego. Zwróćmy uwagę, że `DefaultFormatter` i `InternationalFormatter` są klasami pochodnymi klasy `JFormattedTextField.AbstractFormatter`.
- `Object getValue()`  
zwraca bieżącą, poprawną wartość pola. Zwróćmy uwagę, że wartość ta może nie odpowiadać edytowanemu łańcuchowi.
- `void setValue(Object value)`  
próbuje nadać polu wartość podanego obiektu. Próba może zakończyć się niepowodzeniem, jeśli obiekt formatujący nie zdoła zamienić obiektu na łańcuch znaków.
- `void commitEdit()`  
próbuje nadać polu poprawną wartość na podstawie edytowanego łańcucha. Próba może zakończyć się niepowodzeniem, jeśli obiekt formatujący nie zdoła przekształcić łańcucha.
- `boolean isEditValid()`  
sprawdza, czy edytowany łańcuch reprezentuje popawną wartość.
- `int getFocusLostBehavior()`
- `void setFocusLostBehavior(int behavior)`  
zwraca lub określa zachowanie pola po zakończeniu jego edycji. Wartości określające zachowanie to stałe `COMMIT_OR_REVERT`, `REVERT`, `COMMIT` i `PERSIST` zdefiniowane w klasie `JFormattedTextField`.

#### API javax.swing.JFormattedTextField.AbstractFormatter 1.4

- `abstract String valueToString(Object value)`  
zamienia wartość na łańcuch, który może być edytowany. Wyrzuca wyjątek `ParseException`, jeśli wartość nie jest odpowiednia dla tego obiektu formatującego.
- `abstract Object stringToValue(String s)`  
przekształca łańcuch na wartość. Wyrzuca wyjątek `ParseException`, jeśli s nie posiada właściwego formatu.
- `DocumentFilter getDocumentFilter()`  
metodę tę zastępujemy własną wersją, aby dostarczyć filtra dokumentu, który nakłada restrykcje na zawartość pola tekstowego. Jeśli metoda zwraca wartość null, to oznacza, że filtrowanie nie jest potrzebne.

#### API javax.swing.text.DefaultFormatter 1.3

- `boolean getOverwriteMode()`

- void setOverwriteMode(boolean mode)  
zarządzają trybem nadpisywania. Jeśli mode ma wartość true, to nowe znaki nadpisują znaki edytowanego łańcucha.

**API javax.swing.text.DocumentFilter 1.4**

- void insertString(DocumentFilter.FilterBypass bypass, int offset, String text, AttributeSet attrib)

metoda wywoływana przed wstawieniem łańcucha do dokumentu. Możemy zastąpić ją własną wersją i zmodyfikować wstawiany łańcuch. Możemy również w ogóle uniemożliwić wstawienie łańcucha, jeśli nasza wersja metody nie wywoła metody super.insertString, lub wywołać metody obiektu bypass pozwalające modyfikować dokument bez filtrowania.

*Parametry:*    bypass    obiekt umożliwiający wykonywanie poleceń edycji z pominięciem filtra;  
                     offset    pozycja dokumentu, na której należy umieścić wstawiany tekst;  
                     text     wstawiane znaki;  
                     attrib    atrybuty formatowania wstawianego tekstu.

- void replace(DocumentFilter.FilterBypass bypass, int offset, int length, String text, AttributeSet attrib)

metoda wywoływana przed zastąpieniem części dokumentu nowym łańcuchem. Możemy zastąpić ją własną wersją i zmodyfikować wstawiany łańcuch. Możemy również w ogóle uniemożliwić wstawienie łańcucha, jeśli nasza wersja metody nie wywoła metody super.insertString, lub wywołać metody obiektu bypass pozwalające modyfikować dokument bez filtrowania.

*Parametry:*    bypass    obiekt umożliwiający wykonywanie poleceń edycji z pominięciem filtra;  
                     offset    pozycja dokumentu, na której należy umieścić nowy tekst;  
                     length    rozmiar zastępowanej części dokumentu.  
                     text     wstawiane znaki;  
                     attrib    atrybuty formatowania wstawianego tekstu.

- void remove(DocumentFilter.FilterBypass bypass, int offset, int length)

metoda wywoływana przed usunięciem części dokumentu. Jeśli chcemy przeanalizować efekt usunięcia, należy pobrać dokument za pomocą wywołania bypass.getDocument().

*Parametry:*    bypass    obiekt umożliwiający wykonywanie poleceń edycji z pominięciem filtra,  
                     offset    pozycja usuwanej części dokumentu,  
                     length    rozmiar usuwanej części dokumentu.

**API** javax.swing.text.MaskFormatter 1.4

## ■ MaskFormatter(String mask)

konstruuje obiekt formatujący na podstawie podanej maski. Tabela 6.3 prezentuje symbole stosowane w maskach.

## ■ String getValidCharacters()

## ■ void setValidCharacters(String characters)

zwraca lub określa poprawne znaki. Tylko znaki podanego łańcucha są akceptowane podczas edycji modyfikowej części maski.

## ■ String getInvalidCharacters()

## ■ void setInvalidCharacters(String characters)

zwraca lub określa niepoprawne znaki. Żaden ze znaków podanego łańcucha nie zostanie zaakceptowany.

## ■ char getPlaceholderCharacter()

## ■ void setPlaceholderCharacter(char ch)

zwraca lub określa znak-wypełniacz używany dla modyfikowalnych znaków maski, których użytkownik nie zdążył jeszcze edytować. Domyślnym znakiem-wypełniaczem jest spacja.

## ■ String getPlaceholder()

## ■ void setPlaceholder(String s)

zwraca lub określa łańcuch wypełniacza. Jego końcowa część zostaje wykorzystana, jeśli użytkownik nie dostarczy wszystkich modyfikowalnych znaków maski. Jeśli łańcuch jest pusty lub krótszy od maski, to brakujące znaki zostaną uzupełnione znakiem-wypełniaczem.

## ■ boolean getValueContainsLiteralCharacters()

## ■ void setValueContainsLiteralCharacters(boolean b)

zwraca lub określa znacznik "wartość zawiera znaki literalu". Jeśli znacznik ma wartość true, to wartość pola zawiera literalne (niemodyfikowalne) części maski. Jeśli ma wartość false, to znaki literalu zostają usunięte. Domyślną wartością znacznika jest true.

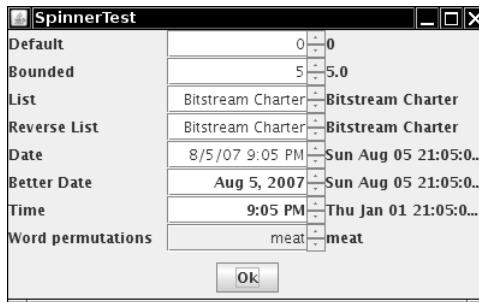
### 6.4.3. Komponent JSpinner

JSpinner jest komponentem zawierającym pole tekstowe i dwa małe przyciski. Gdy użytkownik kliknie któryś z przycisków, wartość pola tekstowego zostaje zwiększena lub zmniejszona (patrz rysunek 6.38).

Wartościami komponentu JSpinner mogą być liczby, daty, wartości z listy lub, w najogólniejszym przypadku, dowolne sekwencje wartości, dla których można ustalić wartość poprzednią i następną. Klasa JSpinner definiuje standardowe modele danych dla pierwszych trzech wymienionych przypadków. Dla pozostałych możemy zdefiniować własne modele danych.

**Rysunek 6.38.**

Kilka wariantów komponentu JSpinner



Domyślnie komponent JSpinner zarządza wartością całkowitą, a przyciski umożliwiają jej zwiększenie lub zmniejszenie o 1. Bieżącą wartość komponentu uzyskujemy, wywołując metodę `getValue`. Metoda ta zwraca wynik typu `Object`, który należy rzutować na typ `Integer`, aby otrzymać obudowaną wartość całkowitą.

```
JSpinner defaultSpinner = new JSpinner();
...
int value = (Integer) defaultSpinner.getValue();
```

Możemy zmienić krok zwiększania i zmniejszania wartości, a także określić jej dolną i górną granicę. Poniżej tworzymy komponent JSpinner o początkowej wartości równej 5, kroku 0.5 i zakresie wartości od 0 do 10.

```
JSpinner boundedSpinner = new JSpinner(new SpinnerNumberModel(5, 0, 10, 0.5));
```

Istnieją dwa konstruktory klasy `SpinnerNumberModel`; jeden posiada wyłącznie parametry typu `int`, a drugi wyłącznie typu `double`. Jeśli więc którykolwiek z parametrów jest wartością zmiennoprzecinkową, należy zastosować drugi konstruktor. Wtedy wartością komponentu JSpinner jest obiekt typu `Double`.

Wartości komponentu JSpinner nie muszą być tylko numeryczne. Komponent ten umożliwia „przeglądanie” dowolnej kolekcji wartości. Wystarczy w tym celu przekazać jego konstruktorowi odpowiedni obiekt klasy `SpinnerListModel`. Obiekt klasy `SpinnerListModel` możemy utworzyć na podstawie zwykłej tablicy lub klasy implementującej interfejs `List` (na przykład `ArrayList`). W przykładowym programie stworzyliśmy komponent JSpinner umożliwiający przeglądanie nazw wszystkich dostępnych czcionek.

```
String[] fonts = GraphicsEnvironment.getLocalGraphicsEnvironment().
    getAvailableFontFamilyNames();
JSpinner listSpinner = new JSpinner(new SpinnerListModel(fonts));
```

Jednak zwróćmy uwagę, że kierunek przeglądania nazw za pomocą komponentu JSpinner jest odwrotny niż w przypadku listy rozwijalnej. Użytkownik przyzwyczajony do korzystania z takiej listy będzie oczekiwany, że klawisz ze strzałką skierowaną w dół zaprowadzi go do „większych” wartości. Jednak JSpinner zwiększa indeks tablicy i wobec tego do tych wartości prowadzi przycisk ze strzałką skierowaną w górę. Klasa `SpinnerListModel` nie umożliwia zmiany kierunku przeglądania, ale wystarczy stworzyć w tym celu anonimową klasę pochodną:

```
JSpinner reverseListSpinner = new JSpinner(
    new SpinnerListModel(fonts)
{
    public Object getNextValue()
{
```

```

        return super.getPreviousValue();
    }
    public Object getPreviousValue()
    {
        return super.getNextValue();
    }
}:

```

Wypróbuj działanie obu wersji i sprawdź, która jest bardziej intuicyjna.

Innym typowym zastosowaniem komponentu JSpinner jest możliwość wyboru daty. Poniższy wiersz kodu tworzy komponent JSpinner zainicjowany dzisiejszą datą:

```
JSpinner dateSpinner = new JSpinner(new SpinnerDateModel());
```

Jeśli jednak przyjrzymy się uważnie rysunkowi 6.38, zauważmy, że taki komponent prezentuje oprócz daty również i czas, na przykład:

8/05/07 9:05 PM

Prezentacja czasu nie ma większego sensu w przypadku komponentu służącego do wyboru daty. Okazuje się, że jego wyeliminowanie nie jest takie proste. Oto odpowiedni do tego kod:

```
JSpinner betterDateSpinner = new JSpinner(new SpinnerDateModel());
String pattern = ((SimpleDateFormat) DateFormat.getDateInstance()).toPattern();
betterDateSpinner.setEditor(new JSpinner.DateEditor(betterDateSpinner, pattern));
```

Stosując podobne rozwiązanie, możemy stworzyć komponent umożliwiający określenie czasu:

```
JSpinner timeSpinner = new JSpinner(new SpinnerDateModel());
pattern = ((SimpleDateFormat)
    ↗DateFormat.getTimeInstance(DateFormat.SHORT)).toPattern();
timeSpinner.setEditor(new JSpinner.DateEditor(timeSpinner, pattern));
```

Jeśli zdefiniujemy własny model danych, możemy używać komponentu JSpinner do prezentacji sekwencji dowolnych wartości. W naszym przykładowym programie stworzyliśmy komponent JSpinner umożliwiający przeglądanie wszystkich permutacji łańcucha "meat". Klikając przyciski komponentu, możemy wybierać łańcuchy "mate", "meta", "team" i pozostałych 20 permutacji.

Definiując własny model, powinniśmy stworzyć klasę pochodną klasy AbstractSpinnerModel i zdefiniować cztery następujące metody:

```
Object getValue()
void setValue(Object value)
Object getNextValue()
Object getPreviousValue()
```

Metoda getValue zwraca wartość przechowywaną przez model. Metoda setValue pozwala określić nową wartość. Powinna wyrzucić wyjątek IllegalArgumentException, jeśli nowa wartość nie jest poprawna.



Metoda setValue musi wywołać metodę fireStateChanged po nadaniu nowej wartości. W przeciwnym razie zawartość pola tekstowego komponentu JSpinner nie zostanie zaktualizowana.

Metody `getNextValue` i `getPreviousValue` zwracają następną i poprzednią wartość dla bieżącej wartości lub wartość `null`, jeśli został osiągnięty koniec wartości w danym kierunku przeglądania.



Metody `getNextValue` i `getPreviousValue` *nie powinny zmieniać bieżącej wartości*. Gdy użytkownik kliknie przycisk ze strzałką skierowaną w góre, wywołana zostanie metoda `getNextValue`. Jeśli zwróci ona wartość różną od `null`, to zostanie ona nadana za pomocą metody `setValue`.

W programie przykładowym używamy standardowego algorytmu w celu określenia następnej i poprzedniej permutacji (patrz listing 6.24). Jego szczegóły nie są tutaj istotne.

Program przedstawiony na listingu 6.23 prezentuje sposoby tworzenia różnych komponentów `JSpinner`. Przycisk *OK* pozwala sprawdzić ich wartości.

### **Listing 6.23. spinner/SpinnerFrame.java**

```
package spinner;

import java.awt.*;
import java.awt.event.*;
import java.text.*;
import javax.swing.*;

/**
 * Ramka z panelem zawierającym kilka komponentów JSpinner
 * oraz przycisk wyświetlający ich wartości.
 */
public class SpinnerFrame extends JFrame
{
    private JPanel mainPanel;
    private JButton okButton;

    public SpinnerFrame()
    {
        JPanel buttonPanel = new JPanel();
        okButton = new JButton("Ok");
        buttonPanel.add(okButton);
        add(buttonPanel, BorderLayout.SOUTH);

        mainPanel = new JPanel();
        mainPanel.setLayout(new GridLayout(0, 3));
        add(mainPanel, BorderLayout.CENTER);

        JSpinner defaultSpinner = new JSpinner();
        addRow("Default", defaultSpinner);

        JSpinner boundedSpinner = new JSpinner(new SpinnerNumberModel(5, 0, 10, 0.5));
        addRow("Bounded", boundedSpinner);

        String[] fonts = GraphicsEnvironment.getLocalGraphicsEnvironment()
            .getAvailableFontFamilyNames();

        JSpinner listSpinner = new JSpinner(new SpinnerListModel(fonts));
        addRow("List", listSpinner);
    }

    void addRow(String name, JSpinner spinner)
    {
        mainPanel.add(new JLabel(name));
        mainPanel.add(spinner);
        mainPanel.add(okButton);
    }
}
```

```

JSpinner reverseListSpinner = new JSpinner(new SpinnerListModel(fonts)
{
    public Object getNextValue() { return super.getPreviousValue(); }
    public Object getPreviousValue() { return super.getNextValue(); }
});
addRow("Reverse List", reverseListSpinner);

JSpinner dateSpinner = new JSpinner(new SpinnerDateModel());
addRow("Date", dateSpinner);

JSpinner betterDateSpinner = new JSpinner(new SpinnerDateModel());
String pattern = ((SimpleDateFormat) DateFormat.getDateInstance()).toPattern();
betterDateSpinner.setEditor(new JSpinner.DateEditor(betterDateSpinner, pattern));
addRow("Better Date", betterDateSpinner);

JSpinner timeSpinner = new JSpinner(new SpinnerDateModel());
pattern = ((SimpleDateFormat)
    DateFormat.getTimeInstance(DateFormat.SHORT)).toPattern();
timeSpinner.setEditor(new JSpinner.DateEditor(timeSpinner, pattern));
addRow("Time", timeSpinner);

JSpinner permSpinner = new JSpinner(new PermutationSpinnerModel("meat"));
addRow("Word permutations", permSpinner);
pack();
}

/**
 * Dodaje wiersz do głównego panelu.
 * @param labelText etykieta komponentu JSpinner
 * @param spinner przykładowy komponent JSpinner
 */
public void addRow(String labelText, final JSpinner spinner)
{
    mainPanel.add(new JLabel(labelText));
    mainPanel.add(spinner);
    final JLabel valueLabel = new JLabel();
    mainPanel.add(valueLabel);
    okButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            Object value = spinner.getValue();
            valueLabel.setText(value.toString());
        }
    });
}
}

```

**Listing 6.24.** spinner/PermutationSpinnerModel.java

```

package spinner;

import javax.swing.*;

/**
 * Model dynamicznie generujący permutacje słowa
 */

```

```
public class PermutationSpinnerModel extends AbstractSpinnerModel
{
    private String word;

    /**
     * Tworzy model.
     * @param w permutowane słowo
     */
    public PermutationSpinnerModel(String w)
    {
        word = w;
    }

    public Object getValue()
    {
        return word;
    }

    public void setValue(Object value)
    {
        if (!(value instanceof String)) throw new IllegalArgumentException();
        word = (String) value;
        fireStateChanged();
    }

    public Object getNextValue()
    {
        int[] codePoints = toCodePointArray(word);

        for (int i = codePoints.length - 1; i > 0; i--)
        {
            if (codePoints[i - 1] < codePoints[i])
            {
                int j = codePoints.length - 1;
                while (codePoints[i - 1] > codePoints[j])
                    j--;
                swap(codePoints, i - 1, j);
                reverse(codePoints, i, codePoints.length - 1);
                return new String(codePoints, 0, codePoints.length);
            }
        }
        reverse(codePoints, 0, codePoints.length - 1);
        return new String(codePoints, 0, codePoints.length);
    }

    public Object getPreviousValue()
    {
        int[] codePoints = toCodePointArray(word);
        for (int i = codePoints.length - 1; i > 0; i--)
        {
            if (codePoints[i - 1] > codePoints[i])
            {
                int j = codePoints.length - 1;
                while (codePoints[i - 1] < codePoints[j])
                    j--;
                swap(codePoints, i - 1, j);
                reverse(codePoints, i, codePoints.length - 1);
                return new String(codePoints, 0, codePoints.length);
            }
        }
    }
}
```

```

        }
        reverse(codePoints, 0, codePoints.length - 1);
        return new String(codePoints, 0, codePoints.length);
    }

    private static int[] toCodePointArray(String str)
    {
        int[] codePoints = new int[str.codePointCount(0, str.length())];
        for (int i = 0, j = 0; i < str.length(); i++, j++)
        {
            int cp = str.codePointAt(i);
            if (Character.isSupplementaryCodePoint(cp)) i++;
            codePoints[j] = cp;
        }
        return codePoints;
    }

    private static void swap(int[] a, int i, int j)
    {
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }

    private static void reverse(int[] a, int i, int j)
    {
        while (i < j)
        {
            swap(a, i, j);
            i++;
            j--;
        }
    }
}

```

#### javax.swing.JSpinner 1.4

- **JSpinner()**

konstruuje komponent JSpinner umożliwiający edycję wartości całkowitej. Wartość początkowa komponentu wynosi 0, a krok jej zmian jest równy 1. Na wartość komponentu nie są nałożone żadne ograniczenia.

- **JSpinner(SpinnerModel model)**

konstruuje komponent JSpinner używający podanego modelu danych.

- **Object getValue()**

zwraca bieżącą wartość komponentu JSpinner.

- **void setValue(Object value)**

próbuje nadać wartość komponentowi JSpinner. Wyrzuca wyjątek IllegalArgumentException, jeśli model nie akceptuje nowej wartości.

- **void setEditor(JComponent editor)**

określa komponent używany do edycji wartości komponentu JSpinner.

### API javax.swing.SpinnerNumberModel 1.4

- SpinnerNumberModel(int initval, int minimum, int maximum, int stepSize)
- SpinnerNumberModel(double initval, double minimum, double maximum, double stepSize)

konstruktory tworzące modele numeryczne zarządzające wartością typu Integer lub Double. Stałe MIN\_VALUE i MAX\_VALUE klas Integer i Double określają najmniejszą i największą wartość.

*Parametry:*    initval        wartość początkowa,  
                    minimum        najmniejsza dozwolona wartość,  
                    maximum        największa dozwolona wartość,  
                    stepSize        krok zwiększania i zmniejszania wartości.

### API javax.swing.SpinnerListModel 1.4

- SpinnerListModel(Object[] values)
- SpinnerListModel(List values)

konstruktory tworzące modele umożliwiające wybór jednej z podanych wartości.

### API javax.swing.SpinnerDateModel 1.4

- SpinnerDateModel()  
tworzy model daty zainicjowany dzisiejszą datą, bez ograniczeń i z krokiem Calendar.DAY\_OF\_MONTH.
- SpinnerDateModel(Date initval, Comparable minimum, Comparable maximum, int step)

*Parametry:*    initval        wartość początkowa;  
                    minimum        najmniejsza dozwolona wartość lub null, jeśli dolne ograniczenie nie jest wymagane;  
                    maximum        największa dozwolona wartość lub null, jeśli górne ograniczenie nie jest wymagane;  
                    step            krok zmiany wartości daty. Jedna ze stałych ERA, YEAR, MONTH, WEEK\_OF\_YEAR, WEEK\_OF\_MONTH, DAY\_OF\_MONTH, DAY\_OF\_YEAR, DAY\_OF\_WEEK, DAY\_OF\_WEEK\_IN\_MONTH, AM\_PM, HOUR, HOUR\_OF\_DAY, MINUTE, SECOND lub MILLISECOND zdefiniowanych w klasie Calendar.

### API java.text.SimpleDateFormat 1.1

- String toPattern() 1.2

zwraca wzorzec formatowania daty dla danego obiektu formatującego. Typowym wzorcem jest "yyyy-MM-dd". Więcej informacji na temat wzorców można znaleźć w dokumentacji Java SE.

**API javax.swing.JSpinner.DateEditor 1.4**

- DateEditor(JSpinner spinner, String pattern)

tworzy edytor daty dla komponentu JSpinner.

*Parametry:* spinner komponent JSpinner, do którego należy edytor;  
 pattern wzorzec formatowania dla obiektu SimpleDateFormat  
 związanego z komponentem JSpinner.

**API javax.swing.AbstractSpinnerModel 1.4**

- Object getValue()

zwraca bieżącą wartość modelu.

- void setValue(Object value)

próbuje nadać nową wartość modelowi. Wyrzuca wyjątek IllegalArgumentException, jeśli model nie akceptuje nowej wartości. Zastępując tę metodę nową wersją, powinniśmy wywołać metodę fireStateChanged po nadaniu modelowi nowej wartości.

- Object getNextValue()

- Object getPreviousValue()

wyznacza (ale nie nadaje) następną lub poprzednią wartość w sekwencji definiowanej przez model.

#### 6.4.4. Prezentacja HTML za pomocą JEditorPane

W książce *Java. Podstawy* omówiliśmy podstawowe komponenty związane z edycją tekstu, takie jak JTextField i JTextArea. Klasy te są przydatne do pobierania tekstu wprowadzonego przez użytkownika. Istnieje także klasa JEditorPane, która wyświetla i umożliwia edycję tekstu zapisanego w formatach RTF i HTML. (Format RTF używany jest przez szereg aplikacji firmy Microsoft do wymiany dokumentów. Jest słabo udokumentowany i nawet aplikacje firmy Microsoft mają problemy z jego prawidłowym wykorzystaniem. W książce tej nie będziemy zajmować się wykorzystaniem tego formatu.

Musimy przyznać, że możliwości klasy JEditorPane są w obecnym stanie dość ograniczone. Potrafi ona wyświetlić proste strony w formacie HTML, ale ma problemy z większością stron, które możemy spotkać obecnie w Internecie. Również edytor HTML posiada ograniczone możliwości i nie jest zbyt stabilny.

Dobrym zastosowaniem klasy JEditorPane może być wyświetlanie zawartości systemu pomocy zapisanej w formacie HTML. Ponieważ korzystamy wtedy z własnych plików źródłowych HTML, to możemy unikać w nich konstrukcji, z którymi klasa JEditorPane obecnie sobie nie radzi.



Więcej informacji na temat tworzenia systemów pomocy JavaHelp dla profesjonalnych aplikacji znajdziemy pod adresem <http://javahelp.java.net>.

Program, którego kod zamieszczamy w listingu 6.25, korzysta z panelu edytora w celu wyświetlenia zawartości strony w języku HTML. W dolnej części jego okna umieszczone jest pole tekstowe, w którym wprowadzić należy adres URL. Musi on zaczynać się od http: lub file:. Wybranie przycisku *Load* spowoduje wyświetlenie w panelu edytora strony o podanym adresie (patrz rysunek 6.39).

**Rysunek 6.39.**

Panel edytora wyświetlający stronę HTML



Hiperłącza na wyświetlonej stronie są aktywne i wybranie jednego z nich spowoduje załadowanie kolejnej strony. Przycisk *Back* umożliwia powrót do poprzedniej strony.

Program ten jest właściwie prostą przeglądarką stron internetowych. Oczywiście nie posiada on możliwości komercyjnie dostępnych przeglądarek, takich jak listy zakładek czy buforowanie stron. Panel edytora nie umożliwia też wyświetlania appletów.

Jeśli zaznaczymy pole wyboru *Editable*, to panel edytora umożliwi nam edycję załadowanej strony. Możemy wpisywać w nim tekst lub usuwać go, korzystając z klawisza *Backspace*. Panel edytora obsługuje także kombinacje klawiszy *Ctrl+X*, *Ctrl+C*, *Ctrl+V* umożliwiające wycinanie, kopianie i wklejanie tekstu. Jednak umożliwienie formatowania tekstu wymagałoby jeszcze sporo pracy.

Gdy panel edytora umożliwia edycję strony, to umieszczone na niej hiperłącza nie są aktywne. Dla niektórych stron edytor pokazuje także teksty etykiet HTML, komentarze i polecenia języka Javascript (patrz rysunek 6.40). Jest to przydatne do sprawdzenia możliwości komponentu klasy *JEditorPane*, ale nie powinno być udostępniane w zwykłych programach.



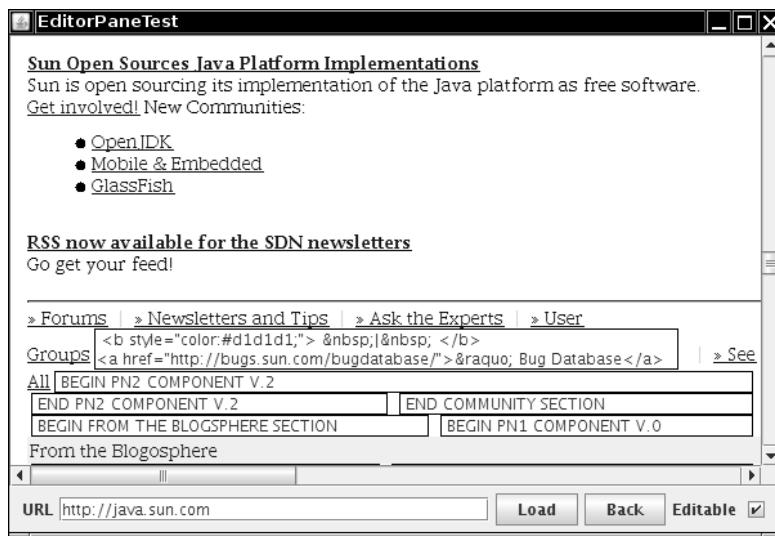
Domyślnie komponent klasy *JEditorPane* znajduje się w stanie edycji. Możemy to zmienić, wywołując metodę *editorPane.setEditable(false)*.

Przedstawione możliwości panelu edytora dają się łatwo wykorzystać. Do załadowania nowego dokumentu używamy metody *setPage*. Na przykład,

```
JEditorPane editorPane = new JEditorPane();
editorPane setPage(url);
```

**Rysunek 6.40.**

Panel edytora w trybie edycji



Jej parametrem jest łańcuch znaków bądź obiekt klasy URL. Klasa JEditorPane jest klasą pochodną klasy JTextComponent. Dlatego możemy też użyć metody setText, jeśli chcemy umieścić w panelu zwykły, a nie sformatowany tekst.



Z dokumentacji nie wynika jednoznacznie, czy metoda setPage ładowa&je nowy dokument w osobnym wątku (co byłoby pożądane, ponieważ klasa JEditorPane nie jest demonem szybkości). Ładowanie dokumentu w osobnym wątku możemy jednak wymusić w następujący sposób:

```
AbstractDocument doc= (AbstractDocument) editorPane.getDocument();
doc.setAsynchronousLoadPriority(0);
```

Aby nasłuchiwa&ć zdarzeń wyboru hiperłącza, tworzymy obiekt implementujący interfejs HyperlinkListener. Interfejs ten zawiera tylko jedną metodę, hyperlinkUpdate, która wywoływana jest, gdy użytkownik przesuwa kursor myszy ponad hiperłącze lub wybiera je. Metoda posiada parametr typu HyperlinkEvent.

Aby dowiedzieć się o rodzaju zdarzenia, musimy wywoła&ć metodę getEventType, która zwrócić może jedną z trzech poniższych wartości:

```
HyperlinkEvent.EventType.ACTIVATED
HyperlinkEvent.EventType.ENTERED
HyperlinkEvent.EventType.EXITED
```

Pierwsza z wartości oznacza, że użytkownik kliknął łączce. W takiej sytuacji zwykle będziemy chcieli załadować nową stronę. Pozostałe wartości możemy wykorzystać na przykład do wyświetlania wskazówek, gdy użytkownik przemieszcza kursor ponad łączem.



Pozostaje dla nas tajemnicą powód, dla którego w interfejsie HyperlinkListener nie zdefiniowano osobnych metod w celu obsługi różnych rodzajów zdarzeń.

Metoda `getURL` klasy `HyperlinkEvent` zwraca adres URL dla hiperłącza. Poniżej przykład instalacji obiektu nasłuchującego hiperłącza, który umożliwia przeglądanie stron wybieranych przez użytkownika przy użyciu hiperłącza.

```
editorPane.addHyperlinkListener(new
    HyperlinkListener()
{
    public void hyperlinkUpdate(HyperlinkEvent event)
    {
        if (event.getEventType()
            == HyperlinkEvent.EventType.ACTIVATED)
        {
            try
            {
                editorPane.setPage(event.getURL());
            }
            catch(IOException e)
            {
                editorPane.setText("Exception: " + e);
            }
        }
    }
});
```

Metoda obsługi zdarzenia pobiera po prostu odpowiedni adres URL i aktualizuje zawartość panelu edytora. Metoda `setPage` może wyrzucić wyjątek `IOException`. W takiej sytuacji wyświetlamy w panelu edytora informację o błędzie w postaci zwykłego tekstu.

Program, którego tekst źródłowy zawiera listing 6.25, wykorzystuje wszystkie możliwości klasy `JEditorPane`, które przydatne są do zbudowania systemu pomocy opartego o pliki w formacie HTML. Implementacja klasy `JEditorPane` jest bardziej skomplikowana niż w przypadku komponentów drzewa bądź tabeli. Jeśli jednak nie wykorzystujemy tej klasy do tworzenia własnego edytora tekstu, to szczegółami tej implementacji nie musimy się interesować.

---

**Listing 6.25.** `editorPane/EditorPaneFrame.java`

```
package editorPane;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Ramka zawierająca panel edytora, pole tekstowe i przycisk Load
 * umożliwiający wprowadzenie adresu URL i załadowanie strony,
 * a także przycisk Back umożliwiający powrót do poprzedniej strony.
 */
public class EditorPaneFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 600;
    private static final int DEFAULT_HEIGHT = 400;
```

```

public EditorPaneFrame()
{
    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

    final Stack<String> urlStack = new Stack<>();
    final JEditorPane editorPane = new JEditorPane();
    final JTextField url = new JTextField(30);

    // instaluje obiekt nasłuchujący hiperłącza

    editorPane.setEditable(false);
    editorPane.addHyperlinkListener(new HyperlinkListener()
    {
        public void hyperlinkUpdate(HyperlinkEvent event)
        {
            if (event.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
            {
                try
                {
                    // zapamiętuje adres URL dla potrzeb przycisku Back
                    urlStack.push(event.getURL().toString());
                    // prezentuje adres URL w polu tekstowym
                    url.setText(event.getURL().toString());
                    editorPane.setPage(event.getURL());
                }
                catch (IOException e)
                {
                    editorPane.setText("Exception: " + e);
                }
            }
        }
    });
}

// konfiguruje pole wyboru umożliwiające włączenie trybu edycji

final JCheckBox editable = new JCheckBox();
editable.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        editorPane.setEditable(editable.isSelected());
    }
});

// tworzy obiekt nasłuchujący przycisku Load

ActionListener listener = new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        try
        {
            // zapamiętuje adres URL dla potrzeb przycisku Back
            urlStack.push(url.getText());
            editorPane.setPage(url.getText());
        }
        catch (IOException e)
        {
            editorPane.setText("Exception: " + e);
        }
    }
};

```

```

        }
    }
};

JButton loadButton = new JButton("Load");
loadButton.addActionListener(listener);
url.addActionListener(listener);

// Konfiguruje przycisk Back i związuje z nim akcję

JButton backButton = new JButton("Back");
backButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        if (urlStack.size() <= 1) return;
        try
        {
            // zapamiętuje adres URL dla potrzeb przycisku Back
            urlStack.pop();
            // prezentuje adres URL w polu tekstowym
            String urlString = urlStack.peek();
            url.setText(urlString);
            editorPane.setPage(urlString);
        }
        catch (IOException e)
        {
            editorPane.setText("Exception: " + e);
        }
    }
});
}

add(new JScrollPane(editorPane), BorderLayout.CENTER);

// umieszcza wszystkie komponenty na głównym panelu okna

 JPanel panel = new JPanel();
panel.add(new JLabel("URL"));
panel.add(url);
panel.add(loadButton);
panel.add(backButton);
panel.add(new JLabel("Editable"));
panel.add(editable);

add(panel, BorderLayout.SOUTH);
}
}

```

### javax.swing.JEditorPane 1.2

- `void setPage(URL url)`  
Ładuje stronę o adresie `url` do panelu edytora.
- `void addHyperlinkListener(HyperlinkListener listener)`  
Instaluje obiekt nasłuchujący panelu edytora.

**API** javax.swing.event.HyperlinkListener 1.2

- void hyperlinkUpdate(HyperlinkEvent event)  
wywoływana, gdy hiperłącze zostanie wybrane.

**API** javax.swing.HyperlinkEvent 1.2

- URL getURL()  
zwraca adres URL dla wybranego hiperłącza.

## 6.5. Wskaźniki postępu

W kolejnych podrozdziałach omówione zostaną trzy klasy informujące o postępie wykonywania czasochłonnej operacji. Klasa JProgressBar jest komponentem biblioteki Swing wskażącym zaawansowanie takiej operacji. Klasa ProgressMonitor tworzy okno dialogowe zawierające pasek postępu. Klasa ProgressMonitorInputStream wyświetla okno dialogowe monitora postępu podczas odczytywania danych ze strumienia.

### 6.5.1. Paski postępu

*Pasek postępu* jest prostym komponentem interfejsu użytkownika — prostokątem stopniowo wypełnianym kolorem, obrazującym postęp wykonania pewnej operacji. Domyślnie postęp ten reprezentowany jest także przez łańcuch znaków postaci "n%". Pasek postępu widoczny jest w dolnej części okna na rysunku 6.41.

**Rysunek 6.41.**  
*Pasek postępu*



Pasek postępu tworzymy w programie podobnie jak suwak, dostarczając konstruktorowi wartości minimalnej i maksymalnej oraz, opcjonalnie, sposobu orientacji paska.

```
progressBar = new JProgressBar(0, 1000);
progressBar = new JProgressBar(SwingConstants.VERTICAL, 0, 1000);
```

Wartość maksymalną i minimalną możemy później zmienić, korzystając z metod `setMaximum` oraz `setMinimum`.

W przeciwnieństwie do suwaka, pasek postępu nie może być modyfikowany przez użytkownika, a jedynie przez program wywołujący w tym celu metodę `setValue`.

Wywołanie:

```
progressBar.setStringPainted(true);
```

powoduje, że pasek postępu oblicza odpowiednią wartość w procentach i wyświetla ją w postaci łańcucha znaków "n%". Jeśli życzymy sobie wyświetlenia innego łańcucha, to możemy skorzystać z metody `setString`, jak w poniższym przykładzie:

```
if (progressBar.getValue() > 900)
    progressBar.setString("Prawie gotowe");
```

Program, którego tekst źródłowy zawiera listing 6.26, wyświetla pasek postępu monitorujący wykonanie symulowanej, czasochłonnej operacji.

---

**Listing 6.26. *progressBar/ProgressBarFrame.java***

---

```
package progressBar;

import java.awt.*;
import java.awt.event.*;
import java.util.List;
import javax.swing.*;

/**
 * Ramka zawierająca przycisk uruchamiający
 * symulację czasochłonnej operacji oraz pasek postępu
 * i pole tekstowe.
 */
public class ProgressBarFrame extends JFrame
{
    public static final int TEXT_ROWS = 10;
    public static final int TEXT_COLUMNS = 40;

    private JButton startButton;
    private JProgressBar progressBar;
    private JCheckBox checkBox;
    private JTextArea textArea;
    private SimulatedActivity activity;

    public ProgressBarFrame()
    {
        // pole tekstowe, w którym prezentowane jest działanie wątku
        textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);

        // panel zawierający przycisk i pasek postępu

        final int MAX = 1000;
        JPanel panel = new JPanel();
        startButton = new JButton("Start");
        progressBar = new JProgressBar(0, MAX);
        progressBar.setStringPainted(true);
        panel.add(startButton);
        panel.add(progressBar);

        checkBox = new JCheckBox("indeterminate");
        checkBox.addActionListener(new ActionListener()
        {
```

```

public void actionPerformed(ActionEvent event)
{
    progressBar.setIndeterminate(checkBox.isSelected());
    progressBar.setStringPainted(!progressBar.isIndeterminate());
}
});

panel.add(checkBox);
add(new JScrollPane(textArea), BorderLayout.CENTER);
add(panel, BorderLayout.SOUTH);

// dodaje obiekt nasłuchujący przycisku

startButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        startButton.setEnabled(false);
        activity = new SimulatedActivity(MAX);
        activity.execute();
    }
});
pack();
}

class SimulatedActivity extends SwingWorker<Void, Integer>
{
    private int current;
    private int target;

    /**
     * Tworzy wątek symulowanej operacji. Zwiększa on wartość licznika
     * do momentu osiągnięcia wartości docelowej.
     * @param t wartość docelowa licznika.
     */
    public SimulatedActivity(int t)
    {
        current = 0;
        target = t;
    }

    protected Void doInBackground() throws Exception
    {
        try
        {
            while (current < target)
            {
                Thread.sleep(100);
                current++;
                publish(current);
            }
        }
        catch (InterruptedException e)
        {
        }
        return null;
    }

    protected void process(List<Integer> chunks)
    {
    }
}

```

```

    {
        for (Integer chunk : chunks)
        {
            textArea.append(chunk + "\n");
            progressBar.setValue(chunk);
        }
    }

    protected void done()
    {
        startButton.setEnabled(true);
    }
}
}

```

Klasa SimulatedActivity zwiększa wartość zmiennej current 10 razy w ciągu sekundy. Gdy wartość zmiennej osiągnie wartość docelową, kończy swoje działanie. Do zaimplementowania tej akcji oraz aktualizacji paska postępu w metodzie process użyliśmy klasy SwingWorker. Klasa SwingWorker wywołuje tę metodę w wątku rozdziału zdarzeń, co umożliwia bezpieczną aktualizację paska postępu. (Więcej informacji na temat bezpieczeństwa wątków Swing znajdziesz w rozdziale 14. książki *Java. Podstawy*).

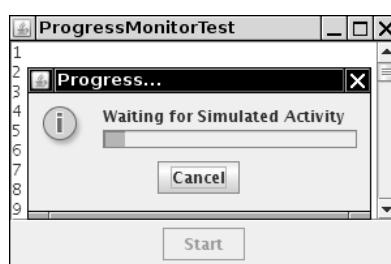
Java SE 1.4 umożliwia dodatkowo utworzenie paska postępu *nieokreślonego*. Ilustruje on postęp operacji, nie dając jednak informacji o stopniu jej wykonania. Tego typu paski postępu wykorzystują na przykład przeglądarki internetowe, pokazując, że oczekują na odpowiedź serwera i nie znają czasu, w którym ona się pojawi. Aby wyświetlić animację takiego paska postępu, należy zastosować metodę setIndeterminate.

Listing 6.26 zawiera pełny tekst źródłowy programu.

## 6.5.2. Monitory postępu

Pasek postępu jest prostym komponentem, który możemy umieścić w oknie programu. Natomiast klasa monitora postępu ProgressMonitor tworzy kompletne okno dialogowe zawierające pasek postępu (patrz rysunek 6.42). Okno to zawiera przycisk *Cancel*. Wybranie tego przycisku powoduje zamknięcie okna dialogowego monitora postępu. Dodatkowo program może sprawdzić, czy użytkownik zamknął okno dialogowe i zakończył w ten sposób monitorowaną akcję. (Zwróćmy uwagę, że nazwa klasy monitora postępu nie zaczyna się na literę J).

**Rysunek 6.42.**  
Okno dialogowe monitora postępu



Monitor postępu tworzymy, dostarczając konstruktorowi klasy informacji o:

- komponencie nadrzędnym, z którym związane jest okno dialogowe monitora postępu,
- obiekcie (łańcucha znaków, ikonie lub komponencie) wyświetlonym w oknie dialogowym,
- opcjonalnym łańcuchu znaków wyświetlonym poniżej obiektu,
- wartości minimalnej i maksymalnej.

Monitor postępu nie może niestety samodzielnie mierzyć postępu wykonania monitorowanej aktywności ani spowodować jej przerwania. Nadal musimy okresowo wywoływać metodę `setProgress` (stanowi ona odpowiednik metody `setValue` klasy `JProgressBar`). Gdy monitorowana akcja zakończy się, powinniśmy zamknąć okno dialogowe monitora, wywołując metodę `close`. Okno to możemy wykorzystać ponownie, wywołując metodę `start`.

Największym problemem podczas używania okna dialogowego monitora postępu jest obsługa przycisku *Cancel*, gdyż nie można dołączyć do niego obiektu nasłuchującego. Należy zatem cyklicznie wywoływać metodę `isCanceled`, aby sprawdzić, czy użytkownik programu kliknął ten przycisk.

Jeśli jednak wątek roboczy może zostać zablokowany na nieokreślony czas (na przykład oczekując na dane otrzymywane z połączenia sieciowego), to nie możemy użyć tego wątku do monitorowania przycisku *Cancel*. W naszym programie pokazaliśmy, jak można wykorzystać w tym celu licznik czasu. Licznik ten uczyniliśmy również odpowiedzialnym za aktualizację wartości postępu.

Jeśli uruchomimy program, którego tekst źródłowy zawiera listing 6.27, to możemy zaobserwować ciekawą właściwość okna dialogowego monitora postępu. Okno to nie pojawia się natychmiast, lecz najpierw czeka krótki okres czasu, aby sprawdzić, czy akcja nie została już zakończona lub nie zakończy się, zanim utworzone zostanie okno dialogowe.

**Listing 6.27.** *progressMonitor/ProgressMonitorFrame.java*

```
package progressMonitor;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Ramka zawierająca przycisk uruchamiający
 * symulację czasochłonnej operacji oraz pole tekstowe.
 */
class ProgressMonitorFrame extends JFrame
{
    public static final int TEXT_ROWS = 10;
    public static final int TEXT_COLUMNS = 40;

    private Timer cancelMonitor;
    private JButton startButton;
    private ProgressMonitor progressDialog;
    private JTextArea textArea;
```

```
private SimulatedActivity activity;

public ProgressMonitorFrame()
{
    // pole tekstowe, w którym prezentowane jest działanie wątku
    textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);

    // tworzy panel przycisków
    JPanel panel = new JPanel();
    startButton = new JButton("Start");
    panel.add(startButton);

    add(new JScrollPane(textArea), BorderLayout.CENTER);
    add(panel, BorderLayout.SOUTH);

    // tworzy obiekt nasłuchujący przycisku

    startButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            startButton.setEnabled(false);
            final int MAX = 1000;

            // uruchamia symulację
            activity = new SimulatedActivity(MAX);
            activity.execute();

            // uruchamia okno dialogowe monitora
            progressDialog = new ProgressMonitor(ProgressMonitorFrame.this,
                "Waiting for Simulated Activity", null, 0, MAX);
            cancelMonitor.start();
        }
    });

    // konfiguruje akcję licznika czasu

    cancelMonitor = new Timer(500, new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            if (progressDialog.isCanceled())
            {
                activity.cancel(true);
                startButton.setEnabled(true);
            }
            else if (activity.isDone())
            {
                progressDialog.close();
                startButton.setEnabled(true);
            }
            else
            {
                progressDialog.setProgress(activity.getProgress());
            }
        }
    });
    pack();
}
```

```
}

class SimulatedActivity extends SwingWorker<Void, Integer>
{
    private int current;
    private int target;

    /**
     * Tworzy wątek symulowanej operacji. Zwiększa on wartość licznika
     * do momentu osiągnięcia wartości docelowej.
     * @param t wartość docelowa licznika.
     */
    public SimulatedActivity(int t)
    {
        current = 0;
        target = t;
    }

    protected Void doInBackground() throws Exception
    {
        try
        {
            while (current < target)
            {
                Thread.sleep(100);
                current++;
                textArea.append(current + "\n");
                setProgress(current);
            }
        }
        catch (InterruptedException e)
        {
        }
    }
    return null;
}
}
```

Zachowanie to możemy kontrolować, wywołując odpowiednie metody. Metody `setMillisToPopup` i `ToDecidePopup` używamy, aby określić liczbę milisekund, która musi upływać pomiędzy konstrukcją obiektu okna a decyzją, czy w ogóle należy je pokazać na ekranie. Domyślna wartość wynosi 500 milisekund. Metoda `setMillisToPopup` umożliwia oszacowanie czasu potrzebnego na wyświetlenie okna. Projektanci biblioteki Swing zaproponowali w tym przypadku domyślną wartość 2 sekund. Orientowali się, że okna dialogowe tworzone przy użyciu biblioteki Swing często nie pojawiają się tak szybko, jak można by tego sobie życzyć. W praktyce lepiej więc nie modyfikować tej wartości.

### **6.5.3. Monitorowanie postępu strumieni wejścia**

Pakiet Swing udostępnia przydatny filtr strumieni, `ProgressMonitorInputStream`, który automatycznie tworzy okno dialogowe monitorujące proces odczytu danych ze strumienia.

Użycie tego filtra jest niezwykle proste. Filtr klasy `ProgressMonitorInputStream` umieszczamy w sekwencji filtrowanych strumieni. (Rozdział 1. zawiera wielej informacji o strumieniach).

Załóżmy, że czytamy dane z pliku tekstowego. W tym celu utworzymy strumień klasy `FileInputStream`:

```
FileInputStream in = new FileInputStream(f);
```

Zwykle dokonujemy następnie jego konwersji na obiekt klasy `InputStreamReader`.

```
InputStreamReader reader = new InputStreamReader(in);
```

Jednak aby monitorować strumień, musimy najpierw utworzyć na jego podstawie strumień wyposażony w monitor postępu:

```
ProgressMonitorInputStream progressIn  
= new ProgressMonitorInputStream(parent, caption, in);
```

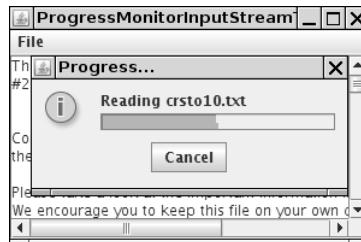
Jego konstruktorem musimy dostarczyć informacji o komponencie nadziedzonym, tekście opisu monitora oraz oczywiście monitorowanym strumieniu. Metoda `read` takiego strumienia powoduje wczytanie informacji oraz aktualizację okna dialogowego monitora postępu.

Teraz możemy dokończyć tworzenie sekwencji filtrów:

```
InputStreamReader reader = new InputStreamReader(progressIn);
```

I to wszystko. Jeśli rozpoczęmiemy wczytywanie pliku, to okno dialogowe monitora postępu pojawi się automatycznie (patrz rysunek 6.43). Rozwiążanie to stanowi udany przykład zastosowania filtrów strumieni.

**Rysunek 6.43.**  
Monitor postępu strumienia wejściowego



Strumień wyposażony w monitor postępu używa metody `available` klasy `InputStream` w celu ustalenia całkowitej liczby bajtów w strumieniu. Jednak metoda ta w rzeczywistości podaje jedynie liczbę bajtów dostępnych *bez blokowania*. Dlatego też monitory postępu pracują poprawnie w przypadku plików lub stron internetowych, których wielkość znana jest z góry, ale niestety nie dla wszystkich rodzajów strumieni.

Program, którego kod źródłowy zawiera listing 6.28, zlicza linie w pliku tekstowym. Jeśli będzie to odpowiednio duży plik (taki jak na przykład zawierający tekst *Hrabiego Monte Christo*), to pojawi się okno dialogowe monitora postępu.

**Listing 6.28.** `progressMonitorInputStream/TextFrame.java`

```
package progressMonitorInputStream;  
  
import java.awt.event.*;  
import java.io.*;  
import java.nio.file.*;  
import java.util.*;
```

```
import javax.swing.*;  
  
/**  
 * Ramka wyposażona w menu umożliwiające załadowanie  
 * pliku tekstowego i obszar tekstowy pokazujący jego zawartość.  
 * Obszar tekstowy jest tworzony i inicjowany po zakończeniu  
 * wczytywania pliku, aby uniknąć migania.  
 */  
public class TextFrame extends JFrame  
{  
    public static final int TEXT_ROWS = 10;  
    public static final int TEXT_COLUMNS = 40;  
  
    private JMenuItem openItem;  
    private JMenuItem exitItem;  
    private JTextArea textArea;  
    private JFileChooser chooser;  
  
    public TextFrame()  
    {  
        textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);  
        add(new JScrollPane(textArea));  
  
        chooser = new JFileChooser();  
        chooser.setCurrentDirectory(new File("."));  
  
        JMenuBar menuBar = new JMenuBar();  
        setJMenuBar(menuBar);  
        JMenu fileMenu = new JMenu("File");  
        menuBar.add(fileMenu);  
        openItem = new JMenuItem("Open");  
        openItem.addActionListener(new ActionListener()  
        {  
            public void actionPerformed(ActionEvent event)  
            {  
                try  
                {  
                    openFile();  
                }  
                catch (IOException exception)  
                {  
                    exception.printStackTrace();  
                }  
            }  
        });  
  
        fileMenu.add(openItem);  
        exitItem = new JMenuItem("Exit");  
        exitItem.addActionListener(new ActionListener()  
        {  
            public void actionPerformed(ActionEvent event)  
            {  
                System.exit(0);  
            }  
        });  
        fileMenu.add(exitItem);  
        pack();  
    }  
}
```

```

    /**
     * Umożliwia użytkownikowi wybranie pliku, którego zawartość
     * zostanie pokazana w obszarze tekstowym.
     */
    public void openFile() throws IOException
    {
        int r = chooser.showOpenDialog(this);
        if (r != JFileChooser.APPROVE_OPTION) return;
        final File f = chooser.getSelectedFile();

        // tworzy strumień i sekwencję filtrów odczytu

        InputStream fileIn = Files.newInputStream(f.toPath());
        final ProgressMonitorInputStream progressIn = new ProgressMonitorInputStream(
            this, "Reading " + f.getName(), fileIn);

        textArea.setText("");

        SwingWorker<Void, Void> worker = new SwingWorker<Void, Void>()
        {
            protected Void doInBackground() throws Exception
            {
                try (Scanner in = new Scanner(progressIn))
                {
                    while (in.hasNextLine())
                    {
                        String line = in.nextLine();
                        textArea.append(line);
                        textArea.append("\n");
                    }
                }
                return null;
            }
        };
        worker.execute();
    }
}

```

Jeśli użytkownik kliknie przycisk *Cancel*, strumień wejściowy zostanie zamknięty. Ponieważ kod przetwarzający dane wejściowe potrafi obsłużyć taką sytuację, to nie jest wymagana dodatkowa obsługa przycisku *Cancel*.

Zwróćmy uwagę, że program nie wypełnia pola tekstowego w zbyt efektywny sposób. Lepiej byłoby oczywiście najpierw wczytać cały plik do obiektu klasy `StringBuffer`, a następnie uczynić go źródłem zawartości pola tekstowego. Jednak w naszym przykładowym programie stosujemy wolniejszą metodę wczytywania zawartości pliku, aby dłużej oglądać okno dialogowe monitora.

W celu uniknięcia migania zawartość pola tekstowego nie jest wyświetlana do momentu całkowitego jej wczytania.

### API javax.swing.JProgressBar 1.2

- `JProgressBar()`
- `JProgressBar(int direction)`

- `JProgressBar(int min, int max)`
- `JProgressBar(int direction, int min, int max)`  
konstruuje pasek postępu o określonym położeniu i wartości minimalnej i maksymalnej.

*Parametry:* `direction` `SwingConstants.HORIZONTAL` lub `SwingConstants.VERTICAL`.

Domyślnie położenie poziome.

`min, max` wartość minimalna i maksymalna paska postępu.  
Domyślnie odpowiednio 0 i 100.

- `int getMinimum`
- `int getMaximum`
- `void setMinimum(int value)`
- `void setMaximum(int value)`

Powyższe metody umożliwiają pobranie i ustawienie wartości minimalnej i maksymalnej.

- `int getValue()`
- `void setValue(int value)`

Te pobierają i ustawiają bieżące wartości monitora.

- `String getString()`
- `void setString(String s)`

Powyższe metody umożliwiają pobranie i ustawienie łańcucha znaków wyświetlanego w pasku postępu. W przypadku gdy `s` posiada wartość `null`, zostanie wyświetlony domyślny łańcuch postaci "n%".

- `boolean isStringPainted()`
- `void setStringPainted(boolean b)`

Metody umożliwiające pobranie i ustawienie sposobu wyświetlania łańcucha znaków w pasku postępu. Gdy `b` posiada wartość `true`, to będzie on wyświetlany ponad paskiem postępu. Domyślna wartość `false` powoduje, że łańcuch nie jest wyświetlany.

- `boolean isIndeterminate() 1.4`
- `void setIndeterminate(boolean b) 1.4`

Metody umożliwiające pobranie i ustawienie sposobu prezentacji paska. Jeśli `b` posiada wartość `true`, to wyświetlany jest pasek poruszający się na przemian do przodu i do tyłu, co wskazuje na nieznany czas zakończenia operacji. Wartością domyślną jest `false`.

**API javax.swing.ProgressMonitor 1.2**

- ProgressMonitor(Component parent, Object message, String note, int min, int max)

tworzy okno dialogowe monitora postępu.

*Parametry:*

parent	komponent nadrzędny, nad którym pojawi się okno dialogowe monitora,
message	obiekt wiadomości wyświetlany przez okno dialogowe,
note	opcjonalny łańcuch znaków wyświetlany poniżej wiadomości. Jeśli wartość ta jest null, to nie jest rezerwowane miejsce na wyświetlenie łańcucha i późniejsze wywołania metody setNote nie dają efektu
min, max	wartość minimalna i maksymalna paska postępu.

- void setNote(String note)

metoda umożliwiająca zmianę łańcucha znaków wyświetlanego w oknie dialogowym monitora.

- void setProgress(int value)

aktualizuje wartość paska postępu.

- void close()

powoduje zamknięcie okna dialogowego.

- boolean isCanceled()

zwraca wartość true, jeśli użytkownik wybrał przycisk *Cancel*.

**API javax.swing.ProgressMonitorInputStream 1.2**

- ProgressMonitorInputStream(Component parent, Object message, InputStream in)

tworzy filtr strumienia wejściowego wyposażony w okno dialogowe monitora postępu.

*Parametry:*

parent	komponent nadrzędny, nad którym pojawi się okno dialogowe monitora,
message	obiekt wiadomości wyświetlany przez okno dialogowe,
in	monitorowany strumień wejściowy,

## 6.6. Organizatory komponentów i dekoratory

Omówienie zaawansowanych możliwości biblioteki Swing zakończymy przedstawieniem komponentów, które pomagają programistom w organizacji innych komponentów. Należą do nich *panele dzielone* umożliwiające podział ich obszaru na wiele części, których rozmiary

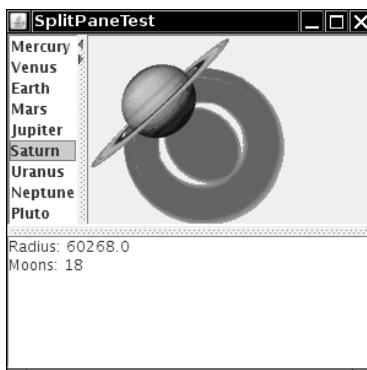
można regulować, *panele z zakładkami* pozwalające na przeglądanie wielu paneli i *panele pulpitu* ułatwiające implementację aplikacji posiadających wiele *ramek wewnętrznych*. Na koniec omówimy warstwy, którymi możemy dekorować inne komponenty.

## 6.6.1. Panele dzielone

Panele dzielone umożliwiają podział ich obszaru na dwie części. Linia podziału panelu może być zmieniana. Rysunek 6.44 pokazuje ramkę zawierającą dwa panele dzielone. Panel zewnętrzny podzielony został poziomo, w jego dolnej części umieszczono obszar tekstowy, a w górnej — kolejny panel dzielony. Ten ostatni podzielony został pionowo. W jego lewej części umieszczono listę, a w prawej — etykietę zawierającą obrazek.

**Rysunek 6.44.**

Ramka zawierająca dwa zagnieżdżone panele



Tworząc panel dzielony, musimy określić sposób jego podziału (za pomocą stałej JSplitPane.HORIZONTAL\_SPLIT lub JSplitPane.VERTICAL\_SPLIT) i dostarczyć dwóch komponentów, które umieszczone zostaną w poszczególnych częściach panelu.

```
JSplitPane innerPane =
    new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, planetList, planetImage);
```

I to wszystko. Możemy jeszcze dodać do linii podziału ikony, które umożliwią maksymalizację obszaru wybranej części panelu. Ikony te widać w górnej części linii podziału na rysunku 6.44. W przypadku wyglądu Metal są one reprezentowane za pomocą trójkątów. Wybranie jednego z nich powoduje maksymalizację obszaru części panelu w kierunku wskazywanym przez wierzchołek trójkąta.

Dodanie tej właściwości linii podziału możliwe jest za pomocą poniższej metody.

```
innerPane.setOneTouchExpandable(true);
```

Inna możliwość polega na włączeniu odrysowywania zawartości części paneli podczas przesuwania linii podziału. Jest to przydatne w niektórych sytuacjach, ale zawsze powoduje spowolnienie działania linii podziału. Możliwość tę możemy włączyć za pomocą wywołania:

```
innerPane.setContinuousLayout(true);
```

W naszym przykładowym programie dolna linia podziału posiada domyślne właściwości (brak ciągłego odrysowywania). Jej przeciąganie powoduje jedynie przesuwanie się ciemnego zarysu. Dopiero jej docelowe ustawienie powoduje odrysowanie komponentów.

Działanie programu z listingu 6.29 jest bardzo proste. Wypełnia on komponent listy nazwami planet. Wybór jednej z nich sprawia, że w prawej części panelu wyświetlany jest obrazek planety, a w dolnej — jej opis. Polecamy wypróbowanie i porównanie działania obu linii podziału.

---

**Listing 6.29.** *splitPane/SplitPaneFrame.java*

```
package splitPane;

import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Ramka zawierająca dwa zagnieżdżone panele dzielone wyświetlające
 * obrazki planet i opisy.
 */
class SplitPaneFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 300;

    private Planet[] planets = { new Planet("Mercury", 2440, 0), new Planet("Venus", 6052, 0),
        new Planet("Earth", 6378, 1), new Planet("Mars", 3397, 2),
        new Planet("Jupiter", 71492, 16), new Planet("Saturn", 60268, 18),
        new Planet("Uranus", 25559, 17), new Planet("Neptune", 24766, 8),
        new Planet("Pluto", 1137, 1), };

    public SplitPaneFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        // tworzy komponenty dla prezentacji nazw i opisu planet oraz ich obrazków

        final JList<Planet> planetList = new JList<>(planets);
        final JLabel planetImage = new JLabel();
        final JTextArea planetDescription = new JTextArea();

        planetList.addListSelectionListener(new ListSelectionListener()
        {
            public void valueChanged(ListSelectionEvent event)
            {
                Planet value = (Planet) planetList.getSelectedValue();

                // aktualizuje obrazek i opis

                planetImage.setIcon(value.getImage());
                planetDescription.setText(value.getDescription());
            }
        });
    }

    // tworzy panele dzielone

    JSplitPane innerPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, planetList,
        planetImage);

    innerPane.setContinuousLayout(true);
```

```

    innerPane.setOneTouchExpandable(true);

    JSplitPane outerPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT, innerPane,
                                         planetDescription);

    add(outerPane, BorderLayout.CENTER);
}
}

```

**API javax.swing.JSplitPane 1.2**

- JSplitPane()
- JSplitPane(int direction)
- JSplitPane(int direction, boolean continuousLayout)
- JSplitPane(int direction, Component first, Component second)
- JSplitPane(int direction, boolean continuousLayout, Component first, Component second)

Tworzą nowy panel dzielony.

<i>Parametry:</i>	direction	jedna z wartości JSplitPane.HORIZONTAL_SPLIT lub JSplitPane.VERTICAL_SPLIT,
	continuousLayout	wartość true, jeśli komponenty mają być odrysowywane podczas przesuwania linii podziału,
	first, second	komponenty, które mają być umieszczone w częściach panelu.

- boolean isOneTouchExpandable()
- void setOneTouchExpandable(boolean b)

Umożliwiają sprawdzenie oraz włączenie właściwości linii podziału polegającej na możliwości maksymalizacji części panelu.

- boolean isContinuousLayout()
- void setContinuousLayout(boolean b)

Umożliwiają sprawdzenie oraz włączenie właściwości linii podziału polegającej na odrysowywaniu zawartości komponentów panelu podczas przesuwania linii podziału.

- void setLeftComponent(Component c)
- void setTopComponent(Component c)

Obie metody dają ten sam efekt — umieszczają komponent c w pierwszej części panelu.

- void setRightComponent(Component c)
- void setBottomComponent(Component c)

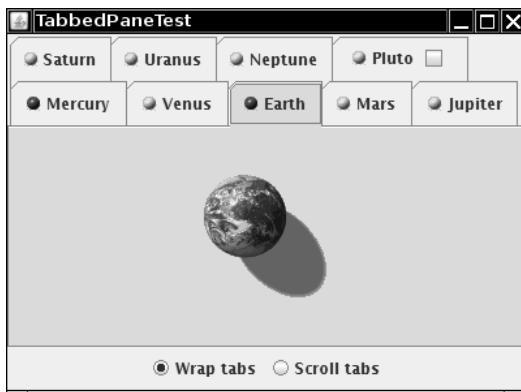
Obie metody dają ten sam efekt — umieszczają komponent c w drugiej części panelu.

## 6.6.2. Panele z zakładkami

Panele z zakładkami umożliwiają uporządkowanie zawartości złożonych okien dialogowych. Pozwalają także na przeglądanie zestawu dokumentów lub obrazów (patrz rysunek 6.45). Takie będzie też ich zastosowanie w naszym przykładowym programie.

**Rysunek 6.45.**

Panel z zakładkami



Tworząc panel z zakładkami, konstruujemy najpierw obiekt klasy `JTabbedPane`, a następnie dodajemy do niego zakładki.

```
JTabbedPane tabbedPane = new JTabbedPane();
tabbedPane.addTab(title, icon, component);
```

Ostatni parametr metody `add` jest komponentem, który umieszczony zostanie na zakładce. Jeśli chcemy, by zakładka zawierała wiele komponentów, to najpierw musimy umieścić je w kontenerze, na przykład klasy `JPanel`.

Ikona zakładki jest opcjonalna. Istnieje wersja metody `add`, która nie wymaga tego parametru:

```
tabbedPane.addTab(title, component);
```

Nową zakładkę możemy także umieścić między już istniejącymi, korzystając z metody `insertTab`:

```
tabbedPane.insertTab(title, icon, component, index);
```

Natomiast poniższe wywołanie spowoduje usunięcie zakładki:

```
tabbedPane.removeTabAt(index);
```

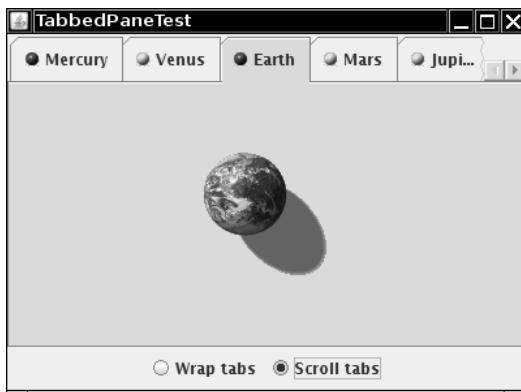
Umieszczenie w panelu nowej zakładki nie powoduje, że staje się ona automatycznie widoczna. W tym celu musimy wybrać ją za pomocą metody `setSelectedIndex`. Poniższe wywołanie pokazuje, w jaki sposób spowodować wyświetlenie dodanej właśnie zakładki:

```
tabbedPane.setSelectedIndex(tabbedPane.getTabCount() - 1);
```

Jeśli panel zawiera większą liczbę zakładek, to mogą one zajmować zbyt dużo miejsca. Dlatego też w Java SE 1.4 wprowadzono możliwość przewijania pojedynczego wiersza zakładek (patrz rysunek 6.46).

**Rysunek 6.46.**

Panel z przewijaniem zakładek



Możemy wybrać ułożenie wszystkich zakładek w kilku wierszach bądź przewijanie ich w jednym wierszu, wywołując odpowiednio:

```
tabbedPane.setTabLayoutPolicy(JTabbedPane.WRAP_TAB_LAYOUT);
```

lub

```
tabbedPane.setTabLayoutPolicy(JTabbedPane.SCROLL_TAB_LAYOUT);
```

Etykiety zakładek mogą posiadać skróty mnemoniczne, podobnie jak elementy menu. Poniższe wywołania

```
int marsIndex = tabbedPane.indexOfTab("Mars");
tabbedPane.setMnemonicAt(marsIndex, KeyEvent.VK_M);
```

sprawią, że litera *M* etykiety zakładki będzie podkreślona, a użytkownik będzie mógł wybrać zakładkę za pomocą kombinacji klawiszy *Alt+M*.

W tytułach zakładek można umieszczać dowolne komponenty. Najpierw dodajemy zakładkę, a następnie wywołujemy

```
tabbedPane.setTabComponentAt(index, component);
```

W przykładowym programie umieszczamy na zakładce Plutona (którego niektórzy astronomicznie nie uważają za planetę) pole umożliwiające zamknięcie zakładki. Efekt ten osiągamy, umieszczając na zakładce panel zawierający dwa komponenty: etykietę z ikoną oraz pole wyboru z obiektem nasłuchującym, który usuwa zakładkę.

Program przykładowy pokazuje zastosowanie pewnej techniki przydatnej w przypadku paneli z zakładkami. Polega ona na umieszczeniu komponentu na zakładce, dopiero w momencie gdy ma zostać ona pokazana. W naszym programie oznacza to, że obrazek planety zostanie umieszczony na zakładce dopiero po jej wybraniu.

Aby uzyskać powiadomienie o wyborze zakładki, musimy zainstalować obiekt nasłuchujący ChangeListener. Zwrócmy uwagę, że obiekt ten instalujemy dla panelu, a nie dla jednego z jego komponentów.

```
tabbedPane.addChangeListener(listener);
```

Gdy użytkownik wybierze zakładkę, to wywołana zostanie metoda stateChanged obiektu nasłuchującego. Korzystając z metody getSelectedIndex, możemy dowiedzieć się, która zakładka została wybrana:

```
public void stateChanged(ChangeEvent event)
{
    int n = tabbedPane.getSelectedIndex();
    loadTab(n);
}
```

W programie z listingu 6.30 wszystkie komponenty zakładek mają na początku wartość null. Kiedy zakładka jest wybierana, to sprawdzamy, czy jej komponent nadal jest wartością null. Jeśli tak, to ładowamy obrazek. (Dzieje się to, zanim wybrana zakładka zostanie pokazana i wobec tego użytkownik nie zobaczy pustej zakładki). Zmieniamy także ikonę zakładki z żółtej na czerwoną, aby zaznaczyć, które zakładki były już przeglądane.

**Listing 6.30.** tabbedPane/TabbedPaneFrame.java

```
package tabbedPane;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Ramka zawierająca panel z zakładkami oraz przyciski
 * umożliwiające przełączanie sposobu prezentacji zakładek.
 */
public class TabbedPaneFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 300;

    private JTabbedPane tabbedPane;

    public TabbedPaneFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        tabbedPane = new JTabbedPane();
        // załadowanie komponentów zakładek odkładamy
        // do momentu ich pierwszej prezentacji

        ImageIcon icon = new ImageIcon(getClass().getResource("yellow-ball.gif"));

        tabbedPane.addTab("Mercury", icon, null);
        tabbedPane.addTab("Venus", icon, null);
        tabbedPane.addTab("Earth", icon, null);
        tabbedPane.addTab("Mars", icon, null);
        tabbedPane.addTab("Jupiter", icon, null);
        tabbedPane.addTab("Saturn", icon, null);
        tabbedPane.addTab("Uranus", icon, null);
        tabbedPane.addTab("Neptune", icon, null);
        tabbedPane.addTab("Pluto", null, null);

        final int plutoIndex = tabbedPane.indexOfTab("Pluto");
```

```

JPanel plutoPanel = new JPanel();
plutoPanel.add(new JLabel("Pluto", icon, SwingConstants.LEADING));
JToggleButton plutoCheckBox = new JCheckBox();
plutoCheckBox.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        tabbedPane.remove(plutoIndex);
    }
});
plutoPanel.add(plutoCheckBox);
tabbedPane.addTabComponentAt(plutoIndex, plutoPanel);

add(tabbedPane, "Center");

tabbedPane.addChangeListener(new ChangeListener()
{
    public void stateChanged(ChangeEvent event)
    {
        // sprawdza, czy na zakładce umieszczony jest już komponent
        if (tabbedPane.getSelectedComponent() == null)
        {
            // ładuje obrazek ikony
            int n = tabbedPane.getSelectedIndex();
            loadTab(n);
        }
    }
});

loadTab(0);

JPanel buttonPanel = new JPanel();
ButtonGroup buttonGroup = new ButtonGroup();
JRadioButton wrapButton = new JRadioButton("Wrap tabs");
wrapButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        tabbedPane.setTabLayoutPolicy(JTabbedPane.WRAP_TAB_LAYOUT);
    }
});
buttonPanel.add(wrapButton);
buttonGroup.add(wrapButton);
wrapButton.setSelected(true);
JRadioButton scrollButton = new JRadioButton("Scroll tabs");
scrollButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        tabbedPane.setTabLayoutPolicy(JTabbedPane.SCROLL_TAB_LAYOUT);
    }
});
buttonPanel.add(scrollButton);
buttonGroup.add(scrollButton);
add(buttonPanel, BorderLayout.SOUTH);
}

```

```

    /**
     * Ładuje zakładkę o podanym indeksie.
     * @param n indeks ładowanej zakładki
     */
    private void loadTab(int n)
    {
        String title = tabbedPane.getTitleAt(n);
        ImageIcon planetIcon = new ImageIcon(getClass().getResource(title + ".gif"));
        tabbedPane.setComponentAt(n, new JLabel(planetIcon));

        // zaznacza, że zakładka była już przeglądana

        tabbedPane.setIconAt(n, new ImageIcon(getClass().getResource("red-
ball.gif")));
    }
}

```

### javax.swing.JTabbedPane 1.2

- `JTabbedPane()`
- `JTabbedPane(int placement)`

Tworzą panel z zakładkami.

*Parametry:* `placement` jedna z wartości `SwingConstants.TOP`,  
`SwingConstants.LEFT`, `SwingConstants.RIGHT`  
lub `SwingConstants.BOTTOM`.

- `void addTab(String title, Component component)`
- `void addTab(String title, Icon icon, Component c)`
- `void addTab(String title, Icon icon, Component c, String toolTip)`

Dodają zakładkę do panelu.

- `void insertTab(String title, Icon icon, Component c, String toolTip, int index)`

umieszcza nową zakładkę na pozycji o podanym indeksie.

- `void removeTabAt(int index)`

usuwa zakładkę o podanym indeksie.

- `void setSelectedIndex(int index)`

wybiera zakładkę o podanym indeksie.

- `int getSelectedIndex()`

zwraca indeks wybranej zakładki.

- `Component getSelectedComponent()`

zwraca komponent wybranej zakładki.

- `String getTitleAt(int index)`

- `void setTitleAt(int index, String title)`

- `Icon getIconAt(int index)`
- `void setIconAt(int index, Icon icon)`
- `Component getComponentAt(int index)`
- `void setComponentAt(int index, Component c)`

Pobierają lub ustawiają tytuł, ikonę lub komponent zakładki o danym indeksie.

- `int indexOfTab(Icon icon)`
- `int indexOfTab(String title)`
- `int indexOfTab(Component c)`

Zwracają indeks zakładki o danym tytule, ikonie bądź komponencie.

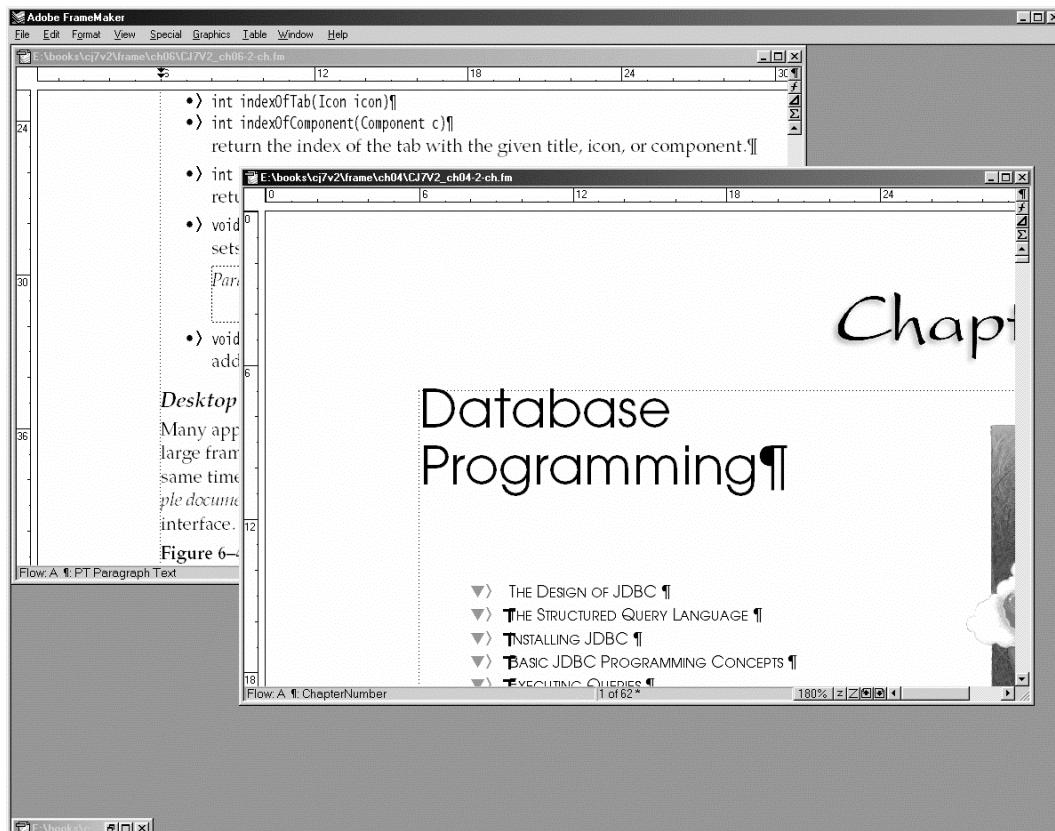
- `int getTabCount()`  
zwraca liczbę zakładek panelu.
- `int getLayoutPolicy()`
- `void setTabLayoutPolicy(int policy) 1.4`

ustala sposób prezentacji zakładek — w wielu wierszach lub jednym przewijanym. Parametr policy przyjmuje wartość `JTabbedPane.WRAP_TAB_LAYOUT` lub `JTabbedPane.SCROLL_TAB_LAYOUT`.

- `int getMnemonicAt(int index) 1.4`
- `void setMnemonicAt(int index, int mnemonic)`  
zwraca lub określa znak mnemoniczny dla zakładki o podanym indeksie. Znak ten jest określony za pomocą stałej `VK_X` zdefiniowanej w klasie `KeyEvent`. Wartość `-1` oznacza brak znaku mnemonicznego.
- `Component getTabComponentAt(int index) 6`
- `void setTabComponentAt(int index, Component c) 6`  
zwraca lub określa komponent, który rysuje tytuł zakładki o podanym indeksie. Jeśli komponent ten jest null, to rysowany jest tytuł i ikona zakładki. W przeciwnym razie tylko podany komponent zostaje wyświetlony na zakładce.
- `int indexOfTabComponent(Component c) 6`  
zwraca indeks zakładki o podanym komponencie tytułu.
- `void addChangeListener(ChangeListener listener)` instaluje obiekt nasłuchujący powiadamiany w momencie wybrania przez użytkownika zakładki.

### 6.6.3. Panele pulpitu i ramki wewnętrzne

Aplikacje często prezentują informacje, korzystając z wielu okien umieszczonych we wspólnej ramce. Zwinięcie takiej ramki do ikony równoznaczne jest z ukryciem zawartości wszystkich jej okien. W środowisku Windows ten sposób działania interfejsu użytkownika nazwany został *MDI (Multiple Document Interface)*. Rysunek 6.47 pokazuje typową aplikację korzystającą z interfejsu MDI.



Rysunek 6.47. Aplikacja korzystająca z MDI

Do niedawna był to jeden z popularniejszych sposobów tworzenia interfejsu aplikacji, ale ostatnio wykorzystywany jest rzadziej. Większość aplikacji, otwierając kolejne dokumenty, używa obecnie ramek tego samego poziomu co główna ramka programu. Który ze sposobów organizacji interfejsu użytkownika jest lepszy? Wykorzystanie MDI pozwala ograniczyć natłok okien otwieranych przez różne programy. Natomiast użycie wielu okien programu umożliwia posłużenie się przyciskami i kombinacjami klawiszy udostępnianymi przez system okienkowy do przełączania się pomiędzy oknami.

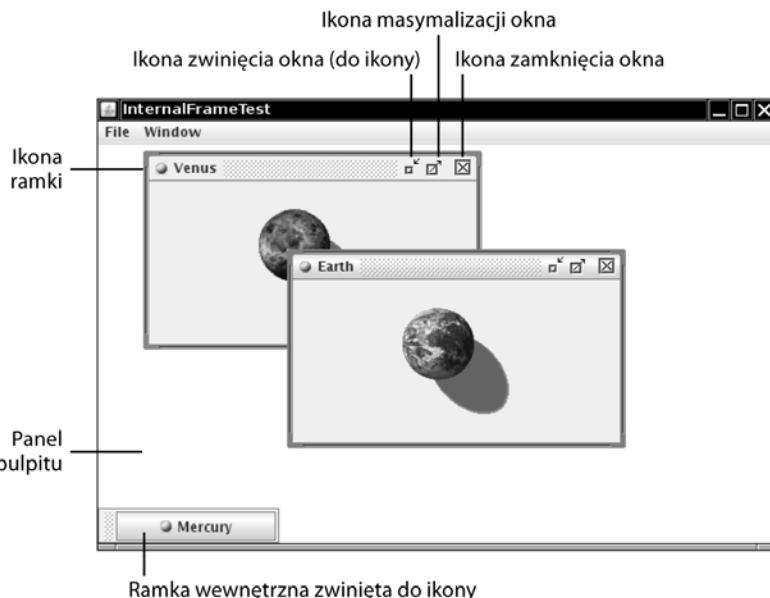
W przypadku aplikacji w języku Java, z natury niezależnych od platformy, nie możemy polegać na usługach systemu okienkowego, a więc zarządzanie własnymi oknami przez samą aplikację ma większy sens.

Rysunek 6.48 pokazuje aplikację Java, której okno zawiera trzy wewnętrzne ramki. Dwie z nich posiadają ikony umożliwiające ich maksymalizację bądź zwinięcie do ikony. Trzecia została zwinięta do ikony.

W przypadku wyglądu komponentów Metal ramki wewnętrzne posiadają wyróżniony obszar, który umożliwia ich „uchwycenie” i przesuwanie. Uchwycenie narożnika ramki umożliwia natomiast zmianę jej rozmiarów.

**Rysunek 6.48.**

Aplikacja w języku Java posiadająca trzy ramki wewnętrzne



Aby skorzystać z możliwości zarządzania wewnętrznymi ramkami należy kolejno wykonać następujące kroki.

- 1.** Tworzymy dla aplikacji zwykłą ramkę klasy JFrame.
- 2.** Umieszczamy w niej panel klasy JDesktopPane.

```
desktop = new JDesktopPane();
add(desktop, BorderLayout.CENTER);
```

- 3.** Tworzymy obiekty klasy JInternalFrame reprezentujące wewnętrzne ramki, podając przy tym, czy mają zawierać ikony zmiany rozmiarów i zamknięcia. Zwykle będziemy chcieli, by ramki posiadały wszystkie te ikony.

```
JInternalFrame iframe = new JInternalFrame(title,
    true, //zmiana rozmiarów
    true, //możliwość zamknięcia
    true, //maksymalizacja
    true); //zwinięcie do ikony
```

- 4.** Umieszczamy komponenty w ramkach wewnętrznych.

```
iframe.getContentPane().add(c);
```

- 5.** Przypisujemy ramkom wewnętrznym ikonę, która pokazywana będzie w lewym górnym rogu ramki.

```
iframe setFrameIcon(icon);
```



W obecnej wersji implementacji wyglądu Metal ikona ramki nie jest wyświetlana, gdy ramka jest zwinięta.

- 6.** Określamy rozmiary wewnętrznych ramek. Podobnie jak w przypadku zwykłych ramek, ich początkowy rozmiar wynosi 0 na 0 pikseli. Ponieważ nie chcemy, by ramki wewnętrzne przykrywały się wzajemnie, to powinniśmy wybrać dla nich także różne pozycje początkowe. Metoda reshape umożliwia określenie początkowej pozycji i rozmiarów ramki:

```
iframe.reshape(nextFrameX, nextFrameY, width, height);
```

- 7.** Podobnie jak w przypadku zwykłych ramek, musimy jeszcze je pokazać.

```
iframe.setVisible(true);
```



We wczesnych wersjach biblioteki Swing ramki wewnętrzne były pokazywane automatycznie i wywołanie metody setVisible nie było konieczne.

- 8.** Dodajemy ramki do panelu JDesktopPane:

```
desktop.add(iframe);
```

- 9.** Wybieramy jedną z dodanych ramek. W przypadku ramek wewnętrznych tylko wybrana ramka otrzymuje informacje o stanie klawiatury. Wygląd Metal wyróżnia wybraną ramkę za pomocą niebieskiego paska tytułu, podczas gdy w pozostałych ramkach ma on kolor szary. Metoda setSelectedFrame umożliwia wybranie ramki. Jednak wywołanie tej metody może zostać „zawetowane” przez aktualnie wybraną ramkę. Spowoduje to wyrzucenie przez metodę setSelectedFrame wyjątku PropertyVetoException, który musimy obsłużyć.

```
try
{
    iframe.setSelected(true);
}
catch(PropertyVetoException e)
{
    //próba wyboru została zawetowana
}
```

- 10.** Umieszczamy kolejną ramkę poniżej, tak by nie zasłaniała istniejącej ramki. Właściwą odległość będzie zwykle wysokość paska tytułowego ramki, którą możemy uzyskać w poniższy sposób:

```
int frameDistance =
    iframe.getHeight() - iframe.getContentPane().getHeight();
```

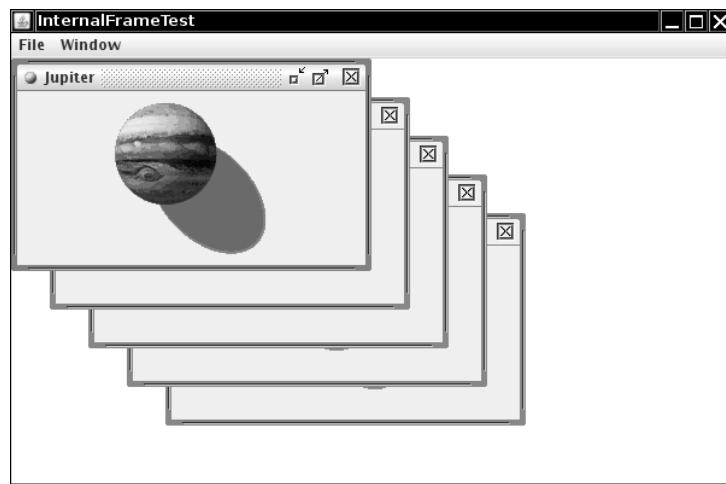
- 11.** Wykorzystujemy wyliczoną odległość w celu ustalenia pozycji kolejnej ramki.

```
nextFrameX += frameDistance;
nextFrameY += frameDistance;
if (nextFrameX + width > desktop.getWidth())
    nextFrameX = 0;
if (nextFrameY + height > desktop.getHeight())
    nextFrameY = 0;
```

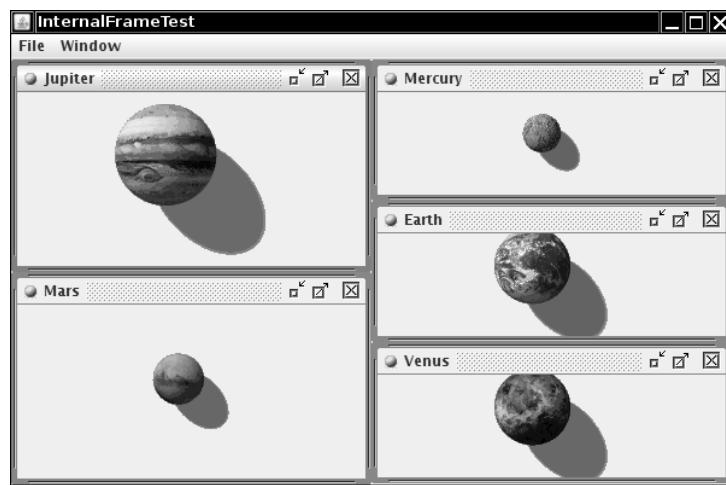
## 6.6.4. Rozmieszczenie kaskadowe i sąsiadujące

W systemie Windows istnieją standardowe komendy umożliwiające uzyskanie rozmieszczenia kaskadowego lub sąsiadującego okien (patrz rysunki 6.49 i 6.50). Klasy JDesktopPane i JInternalFrame biblioteki Swing nie udostępniają niestety odpowiednich metod. Program, którego tekst źródłowy umieściliśmy w listingu 6.31, pokazuje sposób samodzielnej implementacji takich metod.

**Rysunek 6.49.**  
Rozmieszczenie  
kaskadowe  
ramek wewnętrznych



**Rysunek 6.50.**  
Rozmieszczenie  
sąsiadujące  
ramek wewnętrznych



**Listing 6.31.** *internalFrame/DesktopFrame.java*

```
package internalFrame;

import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import javax.swing.*;
```

```
/*
 * Ramka pulpitu zawierająca panele wyświetlające zawartość plików HTML.
 */
public class DesktopFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 600;
    private static final int DEFAULT_HEIGHT = 400;
    private static final String[] planets = { "Mercury", "Venus", "Earth", "Mars",
        "Jupiter",
        "Saturn", "Uranus", "Neptune", "Pluto", };

    private JDesktopPane desktop;
    private int nextFrameX;
    private int nextFrameY;
    private int frameDistance;
    private int counter;

    public DesktopFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        desktop = new JDesktopPane();
        add(desktop, BorderLayout.CENTER);

        // tworzy menu

        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);
        JMenu fileMenu = new JMenu("File");
        menuBar.add(fileMenu);
        JMenuItem openItem = new JMenuItem("New");
        openItem.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                createInternalFrame(new JLabel(
                    new ImageIcon(getClass().getResource(planets[counter] + ".gif"))),
                    planets[counter]);
                counter = (counter + 1) % planets.length;
            }
        });
        fileMenu.add(openItem);
        JMenuItem exitItem = new JMenuItem("Exit");
        exitItem.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                System.exit(0);
            }
        });
        fileMenu.add(exitItem);
        JMenu windowMenu = new JMenu("Window");
        menuBar.add(windowMenu);
        JMenuItem nextItem = new JMenuItem("Next");
        nextItem.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
```

```

        selectNextWindow();
    }
});
windowMenu.add(nextItem);
JMenuItem cascadeItem = new JMenuItem("Cascade");
cascadeItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        cascadeWindows();
    }
});
windowMenu.add(cascadeItem);
JMenuItem tileItem = new JMenuItem("Tile");
tileItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        tileWindows();
    }
});
windowMenu.add(tileItem);
final JCheckBoxMenuItem dragOutlineItem = new JCheckBoxMenuItem("Drag
➥Outline");
dragOutlineItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        desktop.setDragMode(dragOutlineItem.isSelected() ?
➥JDesktopPane.OUTLINE_DRAG_MODE
        : JDesktopPane.LIVE_DRAG_MODE);
    }
});
windowMenu.add(dragOutlineItem);
}

/**
 * Tworzy wewnętrzną ramkę pulpitu.
 * @param c komponent wewnętrzny ramki wewnętrznej
 * @param t tytuł ramki wewnętrznej.
 */
public void createInternalFrame(Component c, String t)
{
    final JInternalFrame iframe = new JInternalFrame(t, true, //zmiana rozmiarów
    true, //możliwość zamknięcia
    true, //maksymalizacja
    true); //zwinięcie do ikony

    iframe.add(c, BorderLayout.CENTER);
    desktop.add(iframe);

    iframe setFrameIcon(new ImageIcon(getClass().getResource("document.gif")));

    //dodaje obiekt nasłuchujący, aby potwierdzić zamknięcie ramki
    iframe.addVetoableChangeListener(new VetoableChangeListener()
    {
        public void vetoableChange(PropertyChangeEvent event) throws
➥PropertyVetoException
    });
}
}

```

```

    {
        String name = event.getPropertyName();
        Object value = event.getNewValue();

        // sprawdza tylko próby zamknięcia ramki
        if (name.equals("closed") && value.equals(true))
        {
            // prosi użytkownika o potwierdzenie zamknięcia
            int result = JOptionPane.showInternalConfirmDialog(iframe, "OK to
            ↪close?",

                "Select an Option", JOptionPane.YES_NO_OPTION);

            // jeśli użytkownik się nie zgodzi, to zgłasza weto
            if (result != JOptionPane.YES_OPTION) throw new
            ↪PropertyVetoException(
                "User canceled close", event);
        }
    });

    // ustala pozycję ramki
    int width = desktop.getWidth() / 2;
    int height = desktop.getHeight() / 2;
    iframe.reshape(nextFrameX, nextFrameY, width, height);

    iframe.show();

    // wybór ramki — może zostać zawetowany
    try
    {
        iframe.setSelected(true);
    }
    catch (PropertyVetoException ex)
    {
    }

    frameDistance = iframe.getHeight() - iframe.getContentPane().getHeight();

    // oblicza odległość pomiędzy ramkami

    nextFrameX += frameDistance;
    nextFrameY += frameDistance;
    if (nextFrameX + width > desktop.getWidth()) nextFrameX = 0;
    if (nextFrameY + height > desktop.getHeight()) nextFrameY = 0;
}

/**
 * Rozmieszcza kaskadowo te ramki pulpitu, które nie są zwinięte do ikony.
 */
public void cascadeWindows()
{
    int x = 0;
    int y = 0;
    int width = desktop.getWidth() / 2;
    int height = desktop.getHeight() / 2;

    for (JInternalFrame frame : desktop.getAllFrames())
    {
}

```

```

        if (!frame.isIcon())
        {
            try
            {
                // próbuje przeprowadzić ramki do stanu pośredniego, co może zostać zawetowane
                frame.setMaximum(false);
                frame.reshape(x, y, width, height);

                x += frameDistance;
                y += frameDistance;
                // zawiąja wokół brzegu pulpitu
                if (x + width > desktop.getWidth()) x = 0;
                if (y + height > desktop.getHeight()) y = 0;
            }
            catch (PropertyVetoException ex)
            {
            }
        }
    }

    /**
     * Rozmieszcza sąsiadującą te ramki pulpitu, które nie są zwinięte do ikony.
     */
    public void tileWindows()
    {
        // zlicza ramki, które nie są zwinięte do ikony
        int frameCount = 0;
        for (JInternalFrame frame : desktop.getAllFrames())
            if (!frame.isIcon()) frameCount++;
        if (frameCount == 0) return;

        int rows = (int) Math.sqrt(frameCount);
        int cols = frameCount / rows;
        int extra = frameCount % rows;
        // liczba kolumn z dodatkowym wierszem

        int width = desktop.getWidth() / cols;
        int height = desktop.getHeight() / rows;
        int r = 0;
        int c = 0;
        for (JInternalFrame frame : desktop.getAllFrames())
        {
            if (!frame.isIcon())
            {
                try
                {
                    frame.setMaximum(false);
                    frame.reshape(c * width, r * height, width, height);
                    r++;
                    if (r == rows)
                    {
                        r = 0;
                        c++;
                        if (c == cols - extra)
                        {
                            // dodatkowy wiersz
                            rows++;
                        }
                    }
                }
            }
        }
    }
}

```

```
        height = desktop.getHeight() / rows;
    }
}
}
catch (PropertyVetoException ex)
{
}
}
}
}
}

/**
 * Wybiera ramkę.
 */
public void selectNextWindow()
{
    JInternalFrame[] frames = desktop.getAllFrames();
    for (int i = 0; i < frames.length; i++)
    {
        if (frames[i].isSelected())
        {
            //znajduje ramkę, która nie jest zwinięta do ikony i może zostać wybrana
            int next = (i + 1) % frames.length;
            while (next != i)
            {
                if (!frames[next].isIcon())
                {
                    try
                    {
                        //pozostałe ramki są zwinięte do ikon lub zgłoszyły weto do wyboru
                        frames[next].setSelected(true);
                        frames[next].toFront();
                        frames[i].toBack();
                        return;
                    }
                    catch (PropertyVetoException ex)
                    {
                    }
                }
                next = (next + 1) % frames.length;
            }
        }
    }
}
```

---

Rozmieszczenie kaskadowe charakteryzuje się jednakowym rozmiarem okien i przesunięciem ich pozycji. Metoda getAllFrames klasy JDesktopPane zwraca tablicę wszystkich wewnętrznych ramek.

```
JInternalFrame[] frames = desktop.getAllFrames();
```

Musimy jednak zwrócić uwagę na stan, w jakim one się znajdują. Ramka wewnętrzna może znajdować się w jednym z trzech stanów. Oto one:

- ikona,
- pośredni, umożliwiający zmianę rozmiarów ramki,
- w pełni rozwinięty.

Korzystając z metody `isIcon`, możemy dowiedzieć się, które ramki zwinięte są do ikony i pominąć je podczas rozmieszczania. Jeśli ramka znajduje się w stanie w pełni rozwiniętym, to musimy najpierw sprowadzić ją do stanu pośredniego, wywołując `setMaximum(false)`. Jest to kolejna metoda, której wywołanie może zostać zawetowane. Trzeba więc obsłużyć wyjątek `PropertyVetoException`.

Poniższa pętla rozmieszcza kaskadowo wszystkie wewnętrzne ramki panelu pulpitu:

```
for (JInternalFrame frame : desktop.getAllFrames())
{
    if (!frame.isIcon())
    {
        try
        {
            // próbuje przeprowadzić ramki do stanu pośredniego co może zostać zawetowane
            frame.setMaximum(false);
            frame.reshape(x, y, width, height);
            x += frameDistance;
            y += frameDistance;
            // zawią dookoła brzegu pulpitu
            if (x + width > desktop.getWidth()) x = 0;
            if (y + height > desktop.getHeight()) y = 0;
        }
        catch(PropertyVetoException e)
        {}
    }
}
```

Uzyskanie rozmieszczenia sąsiadującego okazuje się nieco bardziej skomplikowane, szczególnie jeśli liczba ramek nie jest kwadratem innej liczby. Najpierw musimy uzyskać liczbę ramek, które nie są zwinięte do ikony. Następnie obliczyć liczbę wierszy w pierwszej kolumnie jako

```
int rows = (int) Math.sqrt(frameCount);
```

oraz liczbę kolumn jako

```
int cols = frameCount / rows;
```

z tym wyjątkiem, że ostatnia kolumna

```
int extra = frameCount % rows;
```

posiadać będzie `rows + 1` wierszy.

Poniższa pętla rozmieszcza sąsiadująco wszystkie wewnętrzne ramki panelu pulpitu:

```
int width = desktop.getWidth() / cols;
int height = desktop.getHeight() / rows;
int r = 0;
int c = 0;
for (JInternalFrame frame : desktop.getAllFrames())
{
```

```

        if (!frame.isIcon())
        {
            try
            {
                frame.setMaximum(false);
                frame.reshape(c * width,
                    r * height, width, height);
                r++;
                if (r == rows)
                {
                    r = 0;
                    c++;
                    if (c == cols - extra)
                    {
                        // rozpoczyna dodatkowy wiersz
                        rows++;
                        height = desktop.getHeight() / rows;
                    }
                }
            }
            catch(PropertyVetoException e)
            {}
        }
    }
}

```

Przykładowy program prezentuje także inną typową operację związaną z ramkami: wybór kolejnych ramek, które nie są zwinięte do ikony. Klasa JDesktopPane nie udostępnia metody zwracającej wybraną ramkę. Musimy więc sami wywołać metodę isSelected dla wszystkich ramek tak długo, aż znajdziemy tę wybraną. Następnie wyszukujemy kolejną ramkę, która nie jest zwinięta do ikony i próbujemy ją wybrać.

```
frames[next].setSelected(true);
```

Także i to wywołanie może wyrzucić wyjątek PropertyVetoException. W takim przypadku musimy kontynuować poszukiwanie kolejnej ramki. Jeśli wróćmy w ten sposób do ramki wyjściowej, oznacza to, że żadna inna ramka nie mogła być wybrana. Poniżej kompletna pętla realizująca opisane działanie:

```

JInternalFrame[] frames = desktop.getAllFrames();
for (int i = 0; i < frames.length; i++)
{
    if (frames[i].isSelected())
    {
        // znajduje ramkę, która nie jest zwinięta do ikony i może zostać wybrana
        int next = (i + 1) % frames.length;
        while (next != i)
        {
            if (!frames[next].isIcon())
            {
                try
                {
                    // pozostałe ramki są zwinięte do ikon lub zgłosili weto wyboru
                    frames[next].setSelected(true);
                    frames[next].toFront();
                    frames[i].toBack();
                    return;
                }
            }
        }
    }
}

```

```
        catch (PropertyVetoException ex)
    }
}
next = (next + 1) % frames.length;
}
}
```

### **6.6.5. Zgłaszanie weta do zmiany właściwości**

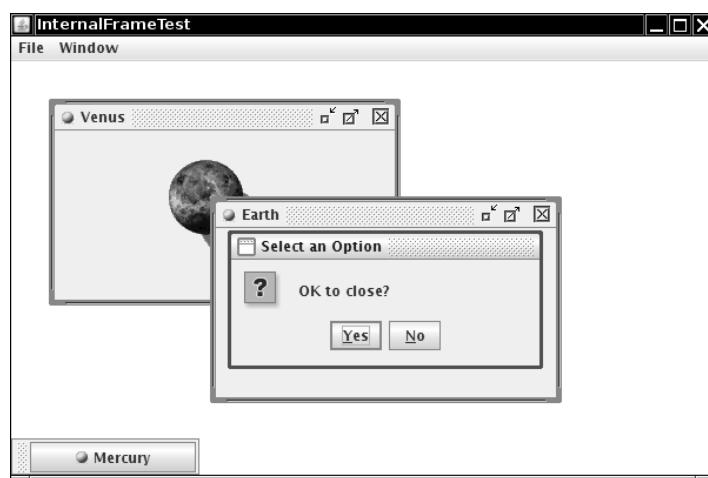
Po lekturze poprzednich przykładów możemy zastanawiać się, w jaki sposób ramka zgłasza weto. Klasa `JInternalFrame` wykorzystuje ogólny mechanizm JavaBeans w celu monitorowania zmian właściwości. Mechanizm ten omawiamy szczegółowo w rozdziale 8. Teraz będziemy chcieli jedynie pokazać, w jaki sposób ramki mogą zgłaszać weto do zmian ich właściwości.

Zwykle ramki nie zgłaszą weta, aby oprotestować zwinięcie ich do ikony bądź utratę wyboru. Typową sytuacją dla takiego zachowania będzie natomiast *zamknięcie* ramki. Ramkę zamkniemy, korzystając z metody `setClosed` klasy `JInternalFrame`. Ponieważ metoda ta może zostać zawetowana, to wywołuje najpierw wszystkie *obiekty nasłuchujące weta zmiany*. Umożliwia to tym obiektom wyrzucenie wyjątku `PropertyVetoException` i tym samym zakończenie wykonywania metody, zanim podejmie ona działania zmierzające do zamknięcia ramki.

W przykładowym programie próba zamknięcia ramki powoduje pojawienie się okna dialogowego w celu potwierdzenia zamknięcia ramki przez użytkownika (patrz rysunek 6.51). Jeśli użytkownik nie zgodzi się, to ramka pozostanie otwarta.

### Rysunek 6.51.

**Użytkownik**  
może zawetować  
zamknięcie ramki



A oto sposób uzyskania takiego powiadomienia.

- 1.** Do każdej ramki dodajemy obiekt nasłuchujący. Obiekt ten musi należeć do klasy implementującej interfejs `VetoableChangeListener`. Najlepiej dodać go zaraz po utworzeniu ramki. W przykładowym programie tworzymy go i dodajemy w klasie ramki. Inną możliwość jest wykorzystanie anonimowej klasy wewnętrznej.

```
iframe.addVetoableChangeListener(listener);
```

2. Implementujemy metodę vetoableChange, która jest jedyną metodą definiowaną przez interfejs VetoableChangeListener. Jej parametrem jest obiekt klasy PropertyChangeEvent. Korzystając z jego metody getPropertyName, uzyskujemy nazwę właściwości, która ma zostać zmieniona — na przykład "closed", jeśli wetowane jest wywołanie metody setClosed(true). Jak pokażemy w rozdziale 8., nazwa właściwości uzyskiwana jest przez usunięcie prefiksu "set" z nazwy metody i zmianę wielkości następnej litery nazwy.
3. Metodę newValue wykorzystujemy w celu uzyskania proponowanej wartości właściwości.

```
String name = event.getPropertyName();
Object value = event.getNewValue();
if (name.equals("closed") && value.equals(Boolean.TRUE))
{
    prosi użytkownika o potwierdzenie zamknięcia ramki
}
```

4. Wyrzucamy wyjątek PropertyVetoException, aby uniemożliwić zmianę właściwości lub w przeciwnym razie oddajemy sterowanie.

```
class DesktopFrame extends JFrame
    implements VetoableChangeListener
{
    .
    .
    public void vetoableChange(PropertyChangeEvent event)
        throws PropertyVetoException
    {
        .
        .
        if (not ok)
            throw new PropertyVetoException(reason, event);
        // oddaje sterowanie, jeśli ok.
    }
}
```

### 6.6.5.1. Okna dialogowe ramek wewnętrznych

W przypadku ramek wewnętrznych nie powinniśmy korzystać z klasy JDialog w celu tworzenia okien dialogowych, ponieważ:

- ich otwarcie wiąże się ze znacznym nakładem i utworzeniem nowej ramki systemu okienkowego,
- system okienkowy nie potrafi określić właściwej pozycji okna dialogowego w stosunku do ramki wewnętrznej, która je otworzyła.

Dlatego też dla prostych okien dialogowych wykorzystywać będziemy metodę showInternal ➔~~Xxx~~Dialog klasy JOptionPane. Działa ona dokładnie tak jak metoda show~~Xxx~~Dialog, ale tworzy proste okno dialogowe nad właściwą ramką wewnętrzną.

W przypadku bardziej złożonych okien dialogowych możemy skorzystać z klasy JInternalFrame, która nie umożliwia jednak tworzenia okien modalnych.

W naszym programie korzystamy z okna dialogowego w celu potwierdzenia przez użytkownika zamknięcia ramki.

```
int result
= JOptionPane.showInternalConfirmDialog(
    iframe, "OK to close?", "Select an Option", JOptionPane.YES_NO_OPTION);
```



Jeśli chcemy zostać po prostu *powiadomieni* o zamknięciu okna, to nie musimy korzystać z mechanizmu zgłaszania weta. Wystarczy jedynie zainstalować obiekt nasłuchujący klasy InternalFrameListener. Zachowuje się on podobnie do obiektu nasłuchującego klasy WindowListener. Gdy zamykana jest wewnętrzna ramka, to wywoływana jest jego metoda internalFrameClosing będąca odpowiednikiem metody windowClosing. Pozostałe sześć powiadomień o zmianach ramki wewnętrznej (otwarcie (zamknięcie), zwiększenie do ikony (rozwinięcie), aktywacja (deaktywacja)) również odpowiada znanym metodom obiektów nasłuchujących zwykłych okien.

### 6.6.5.2. Przeciąganie zarysu ramki

Często krytykowaną cechą wewnętrznych ramek jest niska efektywność odrysowywania ich zawartości. Ujawnia się ona zwłaszcza podczas przeciągania ramek o złożonej zawartości.

Podobny efekt uzyskamy także dla zwykłych okien w przypadku kiepsko zaimplementowanego sterownika ekranu. Z reguły jednak przeciąganie zwykłych okien nawet z bardzo skomplikowaną zawartością jest efektywne, ponieważ obsługiwane jest sprzętowo.

Aby poprawić działanie przeciągania ramek wewnętrznych, możemy skorzystać z ich właściwości umożliwiającej przeciąganie jedynie zarysu ramki. Zawartość ramki jest w takim przypadku odrysowywana dopiero po jej umieszczeniu na pulpicie. Podczas przeciągania odrysowywany jest jedynie zarys ramki.

Aby włączyć możliwość przeciągania zarysu, wywołujemy poniższą metodę.

```
desktop.setDragMode(JDesktopPane.OUTLINE_DRAG_MODE);
```

Możliwość ta stanowi odpowiednik odrysowywania linii podziału komponentów klasy JSplitPane.



We wczesnych wersjach biblioteki Swing odrysowywanie zawartości ramek podczas przeciągania należało wyłączyć za pomocą poniższego wywołania.

```
desktop.putClientProperty(
    "JDesktopPane.dragMode", "outline");
```

Nasz przykładowy program umożliwia zarządzanie odrysowywaniem zawartości ramek za pomocą pozycji menu *Window/Drag Outline*.



Ramki wewnętrzne pulpitu zarządzane są przez klasę DesktopManager. Instalując innego menedżera pulpitu, możemy zaimplementować odmienne zachowanie pulpitu. Możliwości tej nie będziemy jednak omawiać w naszej książce.

Program z listingu 6.31 otwiera na pulpicie ramki zawierające strony HTML. Wybranie z menu opcji *File/Open* umożliwia umieszczenie zawartości wybranego pliku HTML w nowej ramce. Wybranie hiperłącza na stronie w ramce powoduje otwarcie nowej strony w osobnej ramce. Pozycje menu *Window/Cascade* i *Window/Tile* umożliwiają uzyskanie różnych rozmieszczeń ramek na pulpicie.

#### javax.swing.JDesktopPane 1.2

- `JInternalFrame[] getAllFrames()`  
zwraca wszystkie ramki wewnętrzne panelu pulpitu.
- `void setDragMode(int mode)`  
określa sposób zachowania ramek wewnętrznych panelu podczas przeciągania (tylko zarys bądź także zawartość ramki).  
*Parametry:* mode jedna z wartości `JDesktopPane.LIVE_DRAG_MODE`  
lub `JDesktopPane.OUTLINE_DRAG_MODE`.

#### javax.swing.JInternalFrame 1.2

- `JInternalFrame()`
- `JInternalFrame(String title)`
- `JInternalFrame(String title, boolean resizable)`
- `JInternalFrame(String title, boolean resizable, boolean closable)`
- `JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable)`
- `JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable, boolean iconifiable)`  
tworzą nową ramkę wewnętrzną.  
*Parametry:* title tytuł ramki,  
resizable wartość true, jeśli rozmiary ramki mogą być zmieniane,  
closable wartość true, jeśli ramka może być zamknięta,  
maximizable wartość true, jeśli ramka może być maksymalizowana,  
iconifiable wartość true, jeśli ramka może być zwijana do ikony,
- `boolean isResizable()`
- `void setResizable(boolean b)`
- `boolean isClosable()`
- `void setClosable(boolean b)`
- `boolean isMaximizable()`

- void setMaximizable(boolean b)
- boolean isIconifiable()
- void setIconifiable(boolean b)

sprawdzają lub konfigurują odpowiednie właściwości ramki. Jeśli właściwość posiada wartość true, oznacza to także obecność odpowiedniej ikony w pasku tytułu ramki.

- boolean isIcon()
- void setIcon(boolean b)
- boolean isMaximum()
- void setMaximum(boolean b)
- boolean isClosed()
- void setClosed(boolean b)

sprawdzają lub ustawiają właściwości ramki. Jeśli właściwość posiada wartość true, oznacza to, że ramka jest zwinięta do ikony, zmaksymalizowana bądź zamknięta.

- boolean isSelected()
- void setSelected(boolean b)

sprawdza lub ustawia właściwość wyboru ramki. Jeśli właściwość posiada wartość true, oznacza to, że ramka jest wybraną ramką pulpitu.

- void moveToFront()
  - void moveToBack()
- umieszcza ramkę na wierzchu lub spodzie pulpitu.
- void reshape(int x, int y, int width, int height)
- przesuwa ramkę i zmienia jej rozmiar.

*Parametry:* x, y                  nowe współrzędne lewego górnego narożnika ramki,  
width, height      szerokość i wysokość ramki.

- Container getContentPane()
  - void setContentPane(Container c)
- pobierają i zwracają panel ramki wewnętrznej.
- JDesktopPane getDesktopPane()
- pobiera pulpit dla danej ramki wewnętrznej.
- Icon getFrameIcon()
  - void setFrameIcon(Icon icon)
- pobierają i nadają ikonę ramki umieszczoną w jej pasku tytułowym.

- boolean isVisible()
- void setVisible(boolean b)  
sprawdzają i ustawiają właściwość „widoczności” ramki.
- void show()  
sprawia, że ramka staje się widoczna i pojawia się na wierzchu pulpitu.

**API javax.swing.JComponent 1.2**

- void addVetoableChangeListener(VetoableChangeListener listener)  
instaluje obiekt nasłuchujący zmiany, która może zostać zawetowana. Jest on zawiadamiany, gdy ma miejsce próba zmiany ograniczonej właściwości.

**API java.beans.VetoableChangeListener 1.1**

- void vetoableChange(PropertyChangeEvent event)  
metoda wywoływana, gdy metoda set ograniczonej właściwości zawiadamia obiekt nasłuchujący zmiany, która może być zawetowana.

**API java.beans.PropertyChangeEvent 1.1**

- String getPropertyName()  
zwraca nazwę zmienianej właściwości.
- Object getNewValue()  
zwraca proponowaną nową wartość właściwości.

**API java.beans.PropertyVetoException 1.1**

- PropertyVetoException(String reason, PropertyChangeEvent event)  
tworzy wyjątek weta zmiany właściwości.  
*Parametry:* reason powód weta,  
event wetowane zdarzenie.

### 6.6.5.3. Warstwy

W Java SE 1.7 wprowadzono opcję polegającą na umieszczaniu dodatkowej warstwy nad innym komponentem. Na warstwie tej możemy rysować i jednocześnie nasłuchiwać zdarzeń dla komponentu znajdującego się pod spodem. Stosowanie warstw pozwala wzbogacić interfejs użytkownika o podpowiedzi. Możemy w ten sposób na przykład dekorować wprowadzane dane, sygnalizować niepoprawne dane czy nieaktywne komponenty.

Klasa JLayer wiąże komponent z obiektem LayerUI, który zajmuje się rysowaniem i obsługą zdarzeń. Klasa LayerUI ma parametr typu, który musi zgadzać się z typem powiązanego z nią komponentu. Oto w jaki sposób możemy dodać warstwę do komponentu JPanel:

```
JPanel panel = new JPanel();
LayerUI<JPanel> layerUI = new PanelLayer();
JLayer layer = new JLayer(panel, layerUI);
frame.add(layer);
```

Zwrócmy uwagę, że do komponentu nadzawanego (ramki) dodajemy potem właśnie warstwę, a nie panel. PanelLayer jest klasą pochodną zdefiniowaną jak poniżej:

```
class PanelLayer extends LayerUI<Panel>
{
    public void paint(Graphics g, JComponent c)
    {
        . . .
    }
    . . .
}
```

Metoda paint pozwala nam dowolnie modyfikować wygląd komponentu. Musimy jednak najpierw pamiętać o wywołaniu super.paint, które odrysuje komponent. Poniżej przykład metody paint pokrywającej komponent przezroczystym kolorem:

```
public void paint(Graphics g, JComponent c)
{
    super.paint(g, c);

    Graphics2D g2 = (Graphics2D) g.create();
    g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, .3f));
    g2.setPaint(color);
    g2.fillRect(0, 0, c.getWidth(), c.getHeight());
    g2.dispose();
}
```

Aby nasłuchiwać zdarzeń powiązanego komponentu lub jego komponentów podrzędnych, obiekt klasy LayerUI musi skonfigurować maskę zdarzeń warstwy. Powinno to odbyć się w kodzie metody installUI:

```
class PanelLayer extends LayerUI<JPanel>
{
    public void installUI(JComponent c)
    {
        super.installUI(c);
        ((JLayer<?>) c).setLayerEventMask(AWTEvent.KEY_EVENT_MASK |
AWTEvent.FOCUS_EVENT_MASK);
    }

    public void uninstallUI(JComponent c)
    {
        ((JLayer<?>) c).setLayerEventMask(0);
        super.uninstallUI(c);
    }
    . . .
}
```

Teraz metody o nazwie processXxxEvent będą już odbierać odpowiednie zdarzenia. W naszym przykładowym programie będziemy odrysowywać warstwę po każdym naciśnięciu klawisza:

```
public class PanelLayer extends LayerUI<JPanel>
{
    protected void processKeyEvent(KeyEvent e, JLayer<? extends JPanel> l)
    {
        l.repaint();
    }
}
```

Program przedstawiony na listingu 6.32 ma trzy pola tekstowe umożliwiające wprowadzenie składowych RGB. Gdy użytkownik zmienia te wartości, odpowiadający im kolor jest pokazywany przezrocznie ponad panelem. Dodatkowo obsługujemy również zdarzenia przejścia do pola tekstopowego i wyświetlamy jego tekst pogrubioną czcionką.

**Listing 6.32.** *layer/ColorFrame.java*

---

```
package layer;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.plaf.*;

/**
 * Ramka zawierająca trzy pola tekstowe określające kolor.
 */
public class ColorFrame extends JFrame
{
    private JPanel panel;
    private JTextField redField;
    private JTextField greenField;
    private JTextField blueField;

    public ColorFrame()
    {
        panel = new JPanel();

        panel.add(new JLabel("Red:"));
        redField = new JTextField("255", 3);
        panel.add(redField);

        panel.add(new JLabel("Green:"));
        greenField = new JTextField("255", 3);
        panel.add(greenField);

        panel.add(new JLabel("Blue:"));
        blueField = new JTextField("255", 3);
        panel.add(blueField);

        LayerUI<JPanel> layerUI = new PanelLayer();
        JLayer<JPanel> layer = new JLayer<JPanel>(panel, layerUI);

        add(layer);
        pack();
    }

    class PanelLayer extends LayerUI<JPanel>
    {
```

```

public void installUI(JComponent c)
{
    super.installUI(c);
    ((JLayer<?>) c).setLayerEventMask(AWTEvent.KEY_EVENT_MASK |
        AWTEvent.FOCUS_EVENT_MASK);
}

public void uninstallUI(JComponent c)
{
    ((JLayer<?>) c).setLayerEventMask(0);
    super.uninstallUI(c);
}

protected void processKeyEvent(KeyEvent e, JLayer<? extends JPanel> l)
{
    l.repaint();
}

protected void processFocusEvent(FocusEvent e, JLayer<? extends JPanel> l)
{
    if (e.getID() == FocusEvent.FOCUS_GAINED)
    {
        Component c = e.getComponent();
        c.setFont(getFont().deriveFont(Font.BOLD));
    }
    if (e.getID() == FocusEvent.FOCUS_LOST)
    {
        Component c = e.getComponent();
        c.setFont(getFont().deriveFont(Font.PLAIN));
    }
}

public void paint(Graphics g, JComponent c)
{
    super.paint(g, c);

    Graphics2D g2 = (Graphics2D) g.create();
    g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, .3f));
    int red = Integer.parseInt(redField.getText().trim());
    int green = Integer.parseInt(greenField.getText().trim());
    int blue = Integer.parseInt(blueField.getText().trim());
    g2.setPaint(new Color(red, green, blue));
    g2.fillRect(0, 0, c.getWidth(), c.getHeight());
    g2.dispose();
}
}
}

```

**API** javax.swing.JLayer<V extends Component> 7

- **JLayer(V view, LayerUI<V> ui)**

tworzy warstwę ponad widokiem `view` i deleguje rysowanie oraz obsługę zdarzeń do obiektu `ui`.

- void setLayerEventMask(long layerEventMask)

dla zdarzeń zgodnych z maską layerEventMask dotyczących komponentu lub jego komponentów podlegających włącza wysyłanie tych zdarzeń do powiązanego obiektu LayerUI. Maska zdarzeń stanowi kombinację następujących stałych:

```
COMPONENT_EVENT_MASK
FOCUS_EVENT_MASK
HIERARCHY_BOUNDS_EVENT_MASK
HIERARCHY_EVENT_MASK
INPUT_METHOD_EVENT_MASK
KEY_EVENT_MASK
MOUSE_EVENT_MASK
MOUSE_MOTION_EVENT_MASK
MOUSE_WHEEL_EVENT_MASK
```

należących do klasy AWTEvent.

#### javax.swing.plaf.LayerUI<V extends Component> 7

- void installUI(JComponent c)
- void uninstallUI(JComponent c)

wywoływane, gdy obiekt LayerUI dla komponentu c jest instalowany lub odinstalowywany. Przesłaniamy je własną implementacją, aby skonfigurować lub skasować maskę zdarzeń.

- void paint(Graphics g, JComponent c)  
wywoływana, gdy dekorowany komponent wymaga odrysowania. Implementacja powinna wywołać super.paint i narysować dekoracje.
- void processComponentEvent(ComponentEvent e, JLayer<? extends V> 1)
- void processFocusEvent(FocusEvent e, JLayer<? extends V> 1)
- void processHierarchyBoundsEvent(HierarchyEvent e, JLayer<? extends V> 1)
- void processHierarchyEvent(HierarchyEvent e, JLayer<? extends V> 1)
- void processInputMethodEvent(InputMethodEvent e, JLayer<? extends V> 1)
- void processKeyEvent(KeyEvent e, JLayer<? extends V> 1)
- void processMouseEvent(MouseEvent e, JLayer<? extends V> 1)
- void processMouseMotionEvent(MouseEvent e, JLayer<? extends V> 1)
- void processMouseWheelEvent(MouseWheelEvent e, JLayer<? extends V> 1)

wywoływane, gdy określone zdarzenie zostaje wysłane do tego obiektu LayerUI.

W tym rozdziale omówiliśmy sposoby użycia skomplikowanych komponentów biblioteki Swing. W kolejnym rozdziale zajmiemy się zaawansowanymi zastosowaniami biblioteki AWT: złożonymi operacjami rysowania, obróbki obrazu, drukowaniem i interfejsem z macierzystym systemem operacyjnym.

# 7

## Zaawansowane możliwości biblioteki AWT

W tym rozdziale:

- Potokowe tworzenie grafiki.
- Figury.
- Pola.
- Ślad pędzla.
- Wypełnianie.
- Przekształcenia układu współrzędnych.
- Przycinanie.
- Przezroczystość i składanie obrazów.
- Wskazówki operacji graficznych.
- Czytanie i zapisywanie plików graficznych.
- Operacje na obrazach.
- Drukowanie.
- Schowek.
- Mechanizm „przeciagnij i upuść”.
- Integracja z macierzystą platformą.

Do tworzenia prostych rysunków możemy wykorzystać metody klasy `Graphics`. Metody te są wystarczające w przypadku podstawowych aplikacji, ale nie nadają się do zastosowań wymagających tworzenia skomplikowanych rysunków lub potrzebujących daleko idącej kontroli nad tworzoną grafiką. Do takich zastosowań przeznaczona jest biblioteka Java 2D, której możliwości przedstawiemy w bieżącym rozdziale.

Zajmiemy się także omówieniem implementacji drukowania w aplikacjach Java.

Omówimy też dwa sposoby przekazywania danych pomiędzy programami: schowek systemowy oraz mechanizm „przeciagnij i upuść”. Mogą one być wykorzystywane do przekazywania danych między dwiema aplikacjami Java lub pomiędzy aplikacją Java i programem rodzimym danej platformy. Na koniec przedstawimy techniki pozwalające zbliżyć zachowanie aplikacji tworzonych w języku Java do aplikacji macierzystych platformy: tworzenie ekranu powitania i korzystanie z zasobnika systemowego.

## 7.1. Potokowe tworzenie grafiki

JDK 1.0 dysponował bardzo prostym mechanizmem tworzenia grafiki. Wystarczyło wybrać jedynie kolor i tryb rysowania, a następnie wywołać odpowiednią metodę klasy Graphics, na przykład `drawRect` lub `filloval`. Tworzenie grafiki za pomocą Java 2D udostępnia wiele więcej możliwości:

- tworzenie różnorodnych *figur*,
- kontrolę nad sposobem tworzenia *obrysu* figur,
- *wypełnianie* figur różnymi kolorami i ich odcieniami, a także wzorami wypełnień,
- metody *przekształceń* figur — przesunięcia, skalowania, obrotu i zmian proporcji,
- *przycinanie* figur do dowolnie wybranych obszarów,
- wybór *zasad składania* obrazów opisujących sposób tworzenia kombinacji pikseli nowej figury z istniejącymi już pikselami tła,
- *wskazówki tworzenia grafiki* umożliwiające uzyskanie kompromisu pomiędzy szybkością tworzenia grafiki a jej jakością.

Aby narysować figurę, należy kolejno wykonać następujące kroki.

- 1 Musimy uzyskać obiekt klasy `Graphics2D`, która jest klasą pochodną klasy `Graphics`. Począwszy od Java SE 1.2 metody, takie jak `paint` czy `paintComponent`, automatycznie otrzymują jako parametr obiekty klasy `Graphics2D`. Wystarczy więc wykonać następujące rzutowanie:

```
public void paintComponent(Graphics g)
{
    Graphics2D g2 = (Graphics2D)g;
    ...
}
```

- 2 Korzystamy z metody `setRenderingHints` w celu ustalenia kompromisu między szybkością tworzenia grafiki a jej jakością.

```
RenderingHints hints = . . . ;
g2.setRenderingHints(hints);
```

- 3.** Wywołujemy metodę `setStroke` w celu określenia rodzaju *śladu pędzla*, który zostanie użyty do narysowania obrysu figury. Musimy wybrać odpowiednią grubość śladu pędzla oraz sposób jego pozostawiania (ciągły, przerywany).

```
Stroke stroke = . . . ;
g2.setStroke(stroke);
```

- 4.** Wywołujemy metodę `setPaint` w celu określenia sposobu *wypełnienia* obszaru ograniczonego obrysem. Należy wybrać wypełnienie stałym kolorem, kolorem o zmieniających się odcieniach lub wzorem wypełnienia.

```
Paint paint = . . . ;
g2.setPaint(paint);
```

- 5.** Korzystamy z metody `clip` w celu określenia *obszaru przycięcia*.

```
Shape clip = . . . ;
g2.clip(clip);
```

- 6.** Wywołujemy metodę `transform`, aby *przekształcić* współrzędne użytkownika na współrzędne urządzenia. Przekształcenie takie stosujemy, jeśli w programie łatwiej nam posługiwać się własnym układem współrzędnych niż współrzędnymi pikseli.

```
AffineTransform transform = . . . ;
g2.transform(transform);
```

- 7.** Wywołujemy metodę `setComposite` dla określenia *zasady składania* obrazów opisującej tworzenie kombinacji nowych pikseli z już istniejącymi.

```
Composite composite = . . . ;
g2.setComposite(composite);
```

- 8.** Tworzymy figurę. Java 2D udostępnia w tym celu wiele obiektów i metod umożliwiających tworzenie kombinacji figur.

```
Shape shape = . . . ;
```

- 9.** Rysujemy i (lub) wypełniamy figurę. Jeśli narysujemy figurę, to powstanie jedynie jej obrys, który następnie możemy wypełnić.

```
g2.draw(shape);
g2.fill(shape);
```

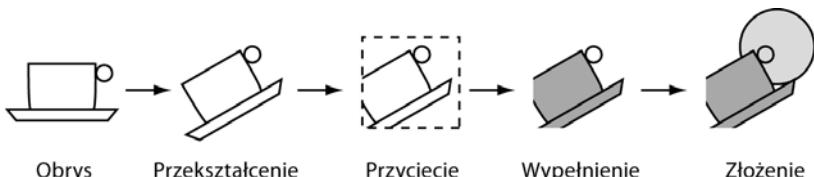
Oczywiście nie zawsze musimy realizować wszystkie wymienione etapy. Parametry kontekstu graficznego posiadają wartości domyślne wystarczające w wielu przypadkach.

W kolejnych podrozdziałach przedstawimy sposób tworzenia figur, definiowania śladów pędzla i wypełnień, przekształcenia i zasady składania obrazów.

Metody `set` nie wywołują żadnych operacji graficznych, a powodują jedynie zmiany stanu kontekstu graficznego. Podobnie utworzenie obiektu reprezentującego figurę nie powoduje narysowania jej reprezentacji. Operacje graficzne wykonywane są jedynie na skutek wywołania metod `draw` lub `fill`. W ich rezultacie reprezentacja graficzna figury tworzona jest w *potoku rysowania* (patrz rysunek 7.1).

**Rysunek 7.1.**

Potok rysowania



W potoku tym wykonywane są następujące operacje prowadzące do utworzenia reprezentacji graficznej obiektu.

- 1 Tworzony jest obrrys figury.
- 2 Obrrys figury jest przekształcany.
3. Obrrys figury jest przycinany. Jeśli figura i obszar przycięcia nie mają części wspólnej, to przetwarzanie w potoku kończy się na tym etapie.
4. Wypełniany jest obszar figury powstały po przycięciu.
5. Piksele wypełnionej figury składane są z istniejącymi pikselami tła. (Na rysunku 7.1 koło stanowi fragment tła, na które nakładana jest figura filiżanki).

W kolejnym podrozdziale przedstawimy sposób definiowania figur, a następnie przejdziemy do omówienia kontekstów graficznych.

#### API `java.awt.Graphics2D 1.2`

- `void draw(Shape s)` rysuje obrrys figury zgodnie z bieżącymi parametrami kontekstu graficznego.
- `void fill(Shape s)` wypełnia obrrys figury zgodnie z bieżącymi parametrami kontekstu graficznego.

## 7.2. Figury

Klasa `Graphics` udostępnia szereg metod umożliwiających rysowanie figur:

```
drawLine
drawRectangle
drawRoundRect
draw3DRect
drawPolygon
drawPolyline
drawOval
drawArc
```

Posiada także odpowiadające im metody `fill` tworzenia *wypełnień* figur. Wszystkie te metody dostępne są w klasie `Graphics` już od wersji JDK 1.0. Natomiast Java 2D proponuje inne, bardziej obiektywe podejście. Zamiast metod udostępnia odpowiednie klasy:

```
Line2D
Rectangle2D
RoundRectangle2D
```

```

Ellipse2D
Arc2D
QuadCurve2D
CubicCurve2D
GeneralPath

```

Wszystkie wymienione klasy implementują interfejs Shape.

Zdefiniowana jest także klasa Point2D, która choć reprezentuje punkt o współrzędnych *x* i *y*, a nie figurę, to używana jest podczas definiowania figur.

Aby uzyskać reprezentację graficzną figury, musimy więc najpierw utworzyć obiekt klasy implementującej interfejs Shape, a następnie wywołać metodę draw klasy Graphics2D.

Jak łatwo zauważyc, klasy Line2D, Rectangle2D, RoundRectangle2D, Ellipse2D i Arc2D odpowiadają metodom drawLine, drawRectangle, drawRoundRectangle, drawOval i drawArc klasy Graphics. (Koncept „prostokąta w trzech wymiarach” zarzucono i dlatego metoda draw3DRect nie posiada swego odpowiednika w postaci klasy). Java 2D udostępnia dodatkowo klasy QuadCurve2D i CubicCurve2D reprezentujące krzywe drugiego i trzeciego stopnia. Omówimy je w dalszej części tego podrozdziału. Nie istnieje natomiast bezpośredni odpowiednik metody drawPolygon. Możemy jednak do tworzenia wielokątów wykorzystać klasę GeneralPath, która opisuje ścieżki złożone z odcinków, a także krzywych drugiego i trzeciego stopnia.

## Klasy

```

Rectangle2D
RoundRectangle2D
Ellipse2D
Arc2D

```

są pochodnymi klasy *RectangularShape*. Choć elipsy i łuki nie są prostokątami, to jednak podczas ich definiowania wykorzystywany jest *prostokąt ograniczający* (patrz rysunek 7.2).

**Rysunek 7.2.**

Prostokąt ograniczający elipsę i łuk



Każda z klas, której nazwa zakończona jest przyrostkiem 2D, posiada dwie klasy pochodne umożliwiające określanie współrzędnych za pomocą wartości typu float lub double. W książce *Java. Podstawy* spotkaliśmy już klasy Rectangle2D.Float i Rectangle2D.Double.

Ten sam schemat stosowany jest w przypadku pozostałych klas, na przykład Arc2D.Float i Arc2D.Double.

Implementacja wszystkich klas graficznych wykorzystuje w rzeczywistości współrzędne typu float, ponieważ reprezentacja tego typu zajmuje mniej miejsca i posiada wystarczającą dokładność dla operacji graficznych. Ponieważ jednak operowanie wartościami typu float jest w języku Java mało wygodne, to większość metod klas graficznych posiada parametry i zwraca wartości typu double. Zwykle więc jedynie podczas tworzenia obiektu klasy graficznej wybierać możemy pomiędzy konstruktorem o parametrach typu float i konstruktorem o parametrach typu double, na przykład:

```

Rectangle2D floatRect
    = new Rectangle2D.Float(5F, 10F, 7.5F, 15F);
Rectangle2D doubleRect
    = new Rectangle2D.Double(5, 10, 7.5, 15);

```

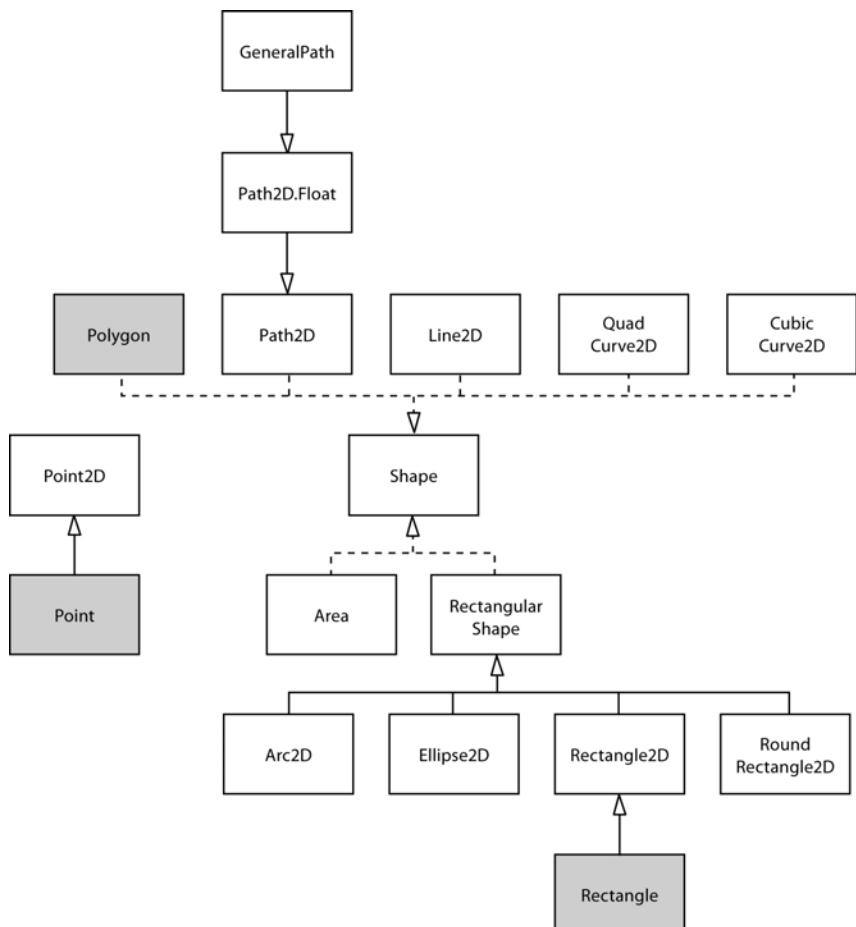
Klasy `xxx2D.Float` i `xxx2D.Double` są klasami pochodnymi klas `xxx2D`. Po utworzeniu obiektu nie ma już znaczenia, do której z klas pochodnych on należy. Do przechowywania referencji obiektu możemy wtedy używać zmiennej o typie klasy bazowej, tak jak pokazaliśmy to w powyższym przykładzie.

Jak łatwo domyślić się na podstawie nazw klas `xxx2D.Float` i `xxx2D.Double`, są one klasami wewnętrznymi klas `xxx2D`. Rozwiążanie takie ma na celu uniknięcie nadmiaru nazw klas zewnętrznych.

Rysunek 7.3 przedstawia zależności pomiędzy klasami reprezentującymi figury. Pominięto na nim podklasy `xxx2D.Float` i `xxx2D.Double`. Tradycyjne klasy dostępne zanim pojawiła się Java 2D zaznaczono na nim za pomocą prostokątów wypełnionych kolorem szarym.

**Rysunek 7.3.**

Zależności  
pomiędzy klasami  
reprezentującymi  
figury



## 7.2.1. Wykorzystanie klas obiektów graficznych

W rozdziale 7. książki *Java. Podstawy* pokazaliśmy już sposób wykorzystania klas `Rectangle2D`, `Ellipse2D` i `Line2D`. Teraz zajmiemy się więc pozostałymi klasami.

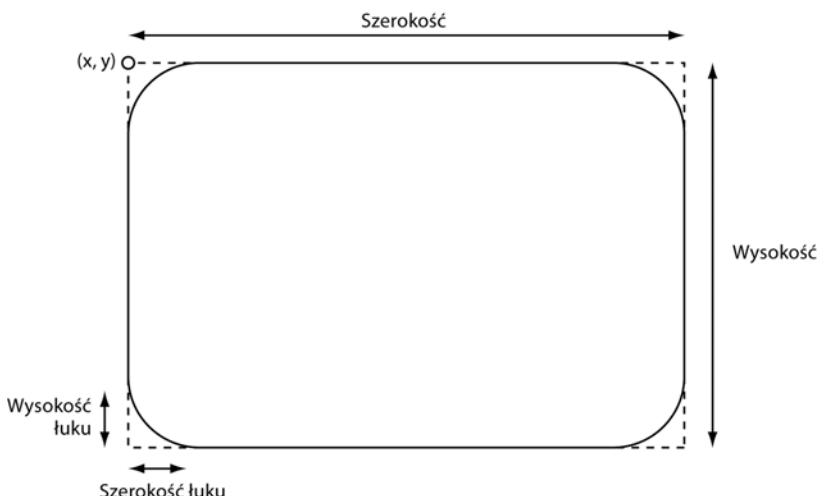
W przypadku klasy `RoundRectangle2D` musimy określić współrzędne lewego górnego wierzchołka figury, jej szerokość i wysokość, a także szerokość i wysokość obszaru narożnika, który powinien zostać zaokrąglony (patrz rysunek 7.4). Na przykład wywołanie

```
RoundRectangle2D r = new RoundRectangle2D.Double(150, 200, 100, 50, 20, 20)
```

utworzy obiekt reprezentujący prostokąt, którego narożniki będą zaokrąglone za pomocą łuków okręgu o promieniu 20 pikseli.

**Rysunek 7.4.**

Parametry prostokątów klasy `RoundRectangle2D`

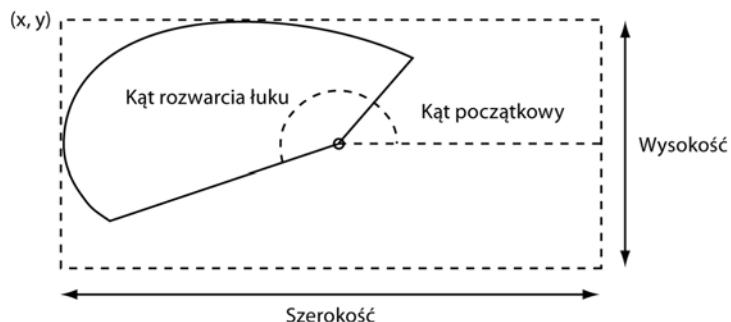


Tworząc łuk, musimy określić ograniczający go prostokąt, kąt początkowy, kąt rozpinający łuku (patrz rysunek 7.5) oraz sposób jego zamknięcia jako jedną z wartości `Arc2D.OPEN`, `Arc2D.PIE`, `Arc2D.CHORD`.

```
Arc2D a = new Arc2D(x, y, width, height, startAngle, arcAngle, closureType);
```

**Rysunek 7.5.**

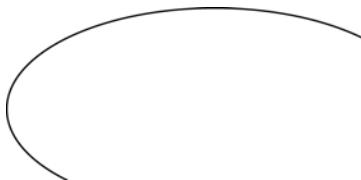
Parametry łuków klasy `Arc2D`



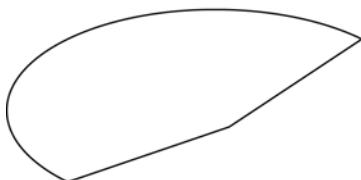
Rysunek 7.6 pokazuje różne rodzaje zamknięcia łuku.

**Rysunek 7.6.**

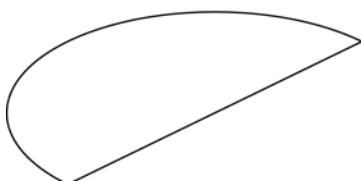
Rodzaje łuków



Arc2D.OPEN



Arc2D.PIE



Arc2D.CHORD



Jeśli łuk jest eliptyczny, to wyznaczenie kątów łuku nie jest trywialne. Dokumentacja mówi na ten temat: „Kąty podaje się względem prostokąta ograniczającego w taki sposób, że kąt 45 stopni zawsze wypada na linii biegnącej ze środka elipsy do prawego górnego wierzchołka prostokąta ograniczającego. W rezultacie, jeśli prostokąt ograniczający jest wyraźnie dłuższy wzdłuż jednej osi, kąty do początku i końca segmentu łuku będą zniekształcone wzdłuż dłuższej osi ograniczenia”. Niestety dokumentacja milczy na temat sposobu wyznaczenia tego zniekształcenia. A oto szczegóły:

Załóżmy, że środkiem łuku jest początek układu współrzędnych, a punkt  $(x, y)$  leży na łuku. Wartość kąta zniekształconego wyznaczymy wtedy w sposób podany poniżej.

```
skewedAngle = Math.toDegrees(Math.atan2(-y * height, x * width));
```

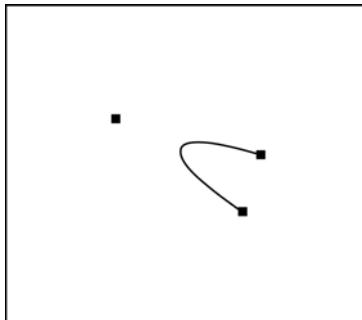
Otrzymana wartość będzie należeć do przedziału od  $-180$  do  $180$  stopni. W ten sposób musimy obliczyć wartość początkową i końcową zniekształconego kąta łuku, a następnie wyznaczyć ich różnicę. Jeśli początkowa wartość kąta bądź różnica wartości kąta jest ujemna, to należy dodać do niej  $360$ . Uzyskaną wartość początkową kąta oraz różnicę wartości kąta przekazujemy konstruktorowi łuku.

Uruchamiając przykładowy program zamieszczony na końcu tego podrozdziału, możemy przekonać się, że opisana powyżej metoda daje właściwe wyniki (patrz rysunek 7.9).

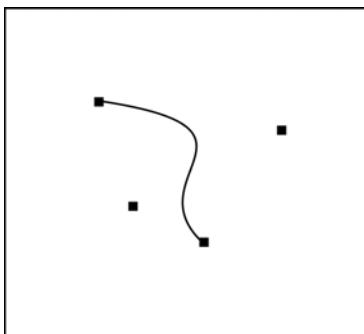
Java 2D udostępnia krzywe *drugiego* i *trzeciego* stopnia. W rozdziale tym nie będziemy zajmować się matematycznymi podstawami ich działania. Sugerujemy praktyczne zapoznanie się z ich możliwościami za pomocą programu z listingu 7.1. Rysunki 7.7 i 7.8 przedstawiają krzywe drugiego i trzeciego stopnia, które tworzone są przez określenie ich *końców* oraz jednego lub dwu *punktów kontrolnych*. Zmiana położenia punktów kontrolnych umożliwia zmianę kształtu krzywych.

**Rysunek 7.7.**

Krzywa drugiego stopnia

**Rysunek 7.8.**

Krzywa trzeciego stopnia



Parametrami konstruktorów krzywych drugiego i trzeciego stopnia są współrzędne ich końców i punktów kontrolnych:

```
QuadCurve2D q = new QuadCurve2D.Double(startX, startY,
    control1X, control1Y, endX, endY);
CubicCurve2D c = new CubicCurve2D.Double(startX, startY,
    control1X, control1Y, control2X, control2Y, endX, endY);
```

Krzywe drugiego stopnia nie są zbyt uniwersalne i dlatego w praktyce stosowane są rzadko. Natomiast krzywe trzeciego stopnia (takie jak krzywe Beziera reprezentowane przez klasę CubicCurve2D) wykorzystywane są powszechnie. Łącząc je w łańcuchy, w których koniec jednej krzywej jest początkiem następnej, możemy tworzyć skomplikowane kształty o dowolnym obrysie. Więcej informacji na ten temat zawiera książka *Computer Graphics: Principles and Practice, Second Edition in C* autorstwa Jamesa D. Foley'a, Andriesa van Dama, Stevena K. Feinera i in. (Addison-Wesley, 1995).

Klasa GeneralPath umożliwia tworzenie dowolnych sekwencji złożonych z odcinków prostych oraz krzywych drugiego i trzeciego stopnia. Współrzędne początku takiego obiektu określamy za pomocą metody `moveTo`, co pokazano poniżej.

```
GeneralPath path = new GeneralPath();
path.moveTo(10, 20);
```

Następnie rozbudowujemy sekwencję o odcinek linii prostej, krzywej drugiego lub trzeciego stopnia, korzystając odpowiednio z metod `lineTo`, `quadTo` bądź `curveTo`. W przypadku metody `lineTo` musimy określić jedynie koniec odcinka. W pozostałych wypadkach najpierw przekazujemy metodom współrzędne punktów kontrolnych, a następnie współrzędne końca segmentu, na przykład:

```
path.lineTo(20, 30);
path.curveTo(control1X, control1Y, control2X, control2Y, endX, endY);
```

Tworzoną sekwencję możemy zamknąć, korzystając z metody `closePath`. Rysuje ona odcinek do punktu, którego współrzędne zawierało ostatnio wywołanie metody `moveTo`.

Aby, korzystając z obiektu klasy `GeneralPath`, utworzyć wielokąt, należy wywołać metodę `moveTo`, przekazując jej współrzędne pierwszego wierzchołka, a następnie wywoływać metodę `lineTo`, podając współrzędne kolejnych wierzchołków. Wywołanie metody `closePath` spowoduje połączenie ostatniego z wierzchołków z pierwszym. Tekst źródłowy programu z listingu 7.1 pokazuje szczegóły tych operacji.

**Listing 7.1.** *shape/ShapeTest.java*

---

```
package shape;

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.util.*;
import javax.swing.*;

/**
 * Program demonstrujący tworzenie figur za pomocą Java 2D.
 * @version 1.02 2007-08-16
 * @author Cay Horstmann
 */
public class ShapeTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new ShapeTestFrame();
                frame.setTitle("ShapeTest");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

/**
 * Ramka zawierająca listę rozwijalną wyboru figury
 * oraz panel, na którym rysowana jest jej reprezentacja.
 */
class ShapeTestFrame extends JFrame
{
    public ShapeTestFrame()
    {
        final ShapeComponent comp = new ShapeComponent();
        add(comp, BorderLayout.CENTER);
        final JComboBox<ShapeMaker> comboBox = new JComboBox<>();
        comboBox.addItem(new LineMaker());
        comboBox.addItem(new RectangleMaker());
```

```

comboBox.addItem(new RoundRectangleMaker());
comboBox.addItem(new EllipseMaker());
comboBox.addItem(new ArcMaker());
comboBox.addItem(new PolygonMaker());
comboBox.addItem(new QuadCurveMaker());
comboBox.addItem(new CubicCurveMaker());
comboBox.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        ShapeMaker shapeMaker = comboBox.getItemAt(comboBox.getSelectedIndex());
        comp.setShapeMaker(shapeMaker);
    }
});
add(comboBox, BorderLayout.NORTH);
comp.setShapeMaker((ShapeMaker) comboBox.getItemAt(0));
pack();
}

/**
 * Komponent rysujący figurę
 * i umożliwiający przesuwanie definiujących ją punktów.
 */
class ShapeComponent extends JComponent
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;

    private Point2D[] points;
    private static Random generator = new Random();
    private static int SIZE = 10;
    private int current;
    private ShapeMaker shapeMaker;

    public ShapeComponent()
    {
        addMouseListener(new MouseAdapter()
        {
            public void mousePressed(MouseEvent event)
            {
                Point p = event.getPoint();
                for (int i = 0; i < points.length; i++)
                {
                    double x = points[i].getX() - SIZE / 2;
                    double y = points[i].getY() - SIZE / 2;
                    Rectangle2D r = new Rectangle2D.Double(x, y, SIZE, SIZE);
                    if (r.contains(p))
                    {
                        current = i;
                        return;
                    }
                }
            }
        });
    }

    public void mouseReleased(MouseEvent event)
    {
}

```

```
        current = -1;
    }
});
addMouseMotionListener(new MouseMotionAdapter()
{
    public void mouseDragged(MouseEvent event)
    {
        if (current == -1) return;
        points[current] = event.getPoint();
        repaint();
    }
});
current = -1;
}

/**
 * Inicjuje obiekt ShapeMaker losowo wybranym zbiorem punktów kontrolnych.
 * @param aShapeMaker obiekt klasy ShapeMaker definiujący figurę
 * za pomocą zbioru punktów kontrolnych.
 */
public void setShapeMaker(ShapeMaker aShapeMaker)
{
    shapeMaker = aShapeMaker;
    int n = shapeMaker.getPointCount();
    points = new Point2D[n];
    for (int i = 0; i < n; i++)
    {
        double x = generator.nextDouble() * getWidth();
        double y = generator.nextDouble() * getHeight();
        points[i] = new Point2D.Double(x, y);
    }
    repaint();
}

public void paintComponent(Graphics g)
{
    if (points == null) return;
    Graphics2D g2 = (Graphics2D) g;
    for (int i = 0; i < points.length; i++)
    {
        double x = points[i].getX() - SIZE / 2;
        double y = points[i].getY() - SIZE / 2;
        g2.fill(new Rectangle2D.Double(x, y, SIZE, SIZE));
    }

    g2.draw(shapeMaker.makeShape(points));
}

public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH,
    DEFAULT_HEIGHT); }

}

/**
 * Klasa abstrakcyjna reprezentująca figurę
 * za pomocą zbioru punktów kontrolnych.
 * Jej konkretne klasy pochodne muszą implementować
 * metodę makeShape zwracającą dany rodzaj figury.
 */
```

```

abstract class ShapeMaker
{
    public abstract Shape makeShape(Point2D[] p);
    private int pointCount;

    /**
     * Tworzy obiekt klasy ShapeMaker.
     * @param aPointCount liczba punktów kontrolnych definiujących figurę.
     */
    public ShapeMaker(int aPointCount)
    {
        pointCount = aPointCount;
    }

    /**
     * Zwraca liczbę punktów kontrolnych definiujących figurę.
     * @return liczba punktów kontrolnych
     */
    public int getPointCount()
    {
        return pointCount;
    }

    /**
     * Tworzy figurę na podstawie zbioru punktów.
     * @param p punkty definiujące figurę
     * @return zdefiniowana figura
     */
    public String toString()
    {
        return getClass().getName();
    }
}

/**
 * Tworzy odcinek linii prostej łączący dwa punkty.
 */
class LineMaker extends ShapeMaker
{
    public LineMaker()
    {
        super(2);
    }

    public Shape makeShape(Point2D[] p)
    {
        return new Line2D.Double(p[0], p[1]);
    }
}

/**
 * Tworzy prostokąt rozpięty za pomocą dwóch narożników.
 */
class RectangleMaker extends ShapeMaker
{
    public RectangleMaker()
    {
}

```

```
        super(2);
    }

    public Shape makeShape(Point2D[] p)
    {
        Rectangle2D s = new Rectangle2D.Double();
        s setFrameFromDiagonal(p[0], p[1]);
        return s;
    }

}

< /**
 * Tworzy zaokrąglony prostokąt rozpięty za pomocą dwóch narożników.
 */

class RoundRectangleMaker extends ShapeMaker
{
    public RoundRectangleMaker()
    {
        super(2);
    }

    public Shape makeShape(Point2D[] p)
    {
        RoundRectangle2D s = new RoundRectangle2D.Double(0, 0, 0, 0, 20, 20);
        s setFrameFromDiagonal(p[0], p[1]);
        return s;
    }

}

< /**
 * Tworzy elipsę zawartą w prostokącie ograniczającym
 * rozpiętym za pomocą dwóch narożników.
 */

class EllipseMaker extends ShapeMaker
{
    public EllipseMaker()
    {
        super(2);
    }

    public Shape makeShape(Point2D[] p)
    {
        Ellipse2D s = new Ellipse2D.Double();
        s setFrameFromDiagonal(p[0], p[1]);
        return s;
    }

}

< /**
 * Tworzy łuk zawarty w prostokącie ograniczającym
 * rozpiętym za pomocą dwóch narożników. Dodatkowe dwa punkty
 * kontrolne umożliwiają określenie wartości początkowej i końcowej
 * kąta łuku. Aby zilustrować poprawność obliczeń kątów łuku, metoda makeShape
 * zwraca figurę złożoną z łuku, prostokąta ograniczającego i linii
 * łączących środek łuku z punktami kontrolnymi kąta.
 */

class ArcMaker extends ShapeMaker
{
```

```

public ArcMaker()
{
    super(4);
}

public Shape makeShape(Point2D[] p)
{
    double centerX = (p[0].getX() + p[1].getX()) / 2;
    double centerY = (p[0].getY() + p[1].getY()) / 2;
    double width = Math.abs(p[1].getX() - p[0].getX());
    double height = Math.abs(p[1].getY() - p[0].getY());

    double skewedStartAngle = Math.toDegrees(Math.atan2(-(p[2].getY() - centerY) *
        width, (p[2].getX() - centerX) * height));
    double skewedEndAngle = Math.toDegrees(Math.atan2(-(p[3].getY() - centerY) *
        width, (p[3].getX() - centerX) * height));
    double skewedAngleDifference = skewedEndAngle - skewedStartAngle;
    if (skewedStartAngle < 0) skewedStartAngle += 360;
    if (skewedAngleDifference < 0) skewedAngleDifference += 360;

    Arc2D s = new Arc2D.Double(0, 0, 0, 0, skewedStartAngle,
        skewedAngleDifference, Arc2D.OPEN);
    s.setFrameFromDiagonal(p[0], p[1]);

    GeneralPath g = new GeneralPath();
    g.append(s, false);
    Rectangle2D r = new Rectangle2D.Double();
    r.setFrameFromDiagonal(p[0], p[1]);
    g.append(r, false);
    Point2D center = new Point2D.Double(centerX, centerY);
    g.append(new Line2D.Double(center, p[2]), false);
    g.append(new Line2D.Double(center, p[3]), false);
    return g;
}

}

<**
 * Tworzy wielokąt o sześciu wierzchołkach.
 */
class PolygonMaker extends ShapeMaker
{
    public PolygonMaker()
    {
        super(6);
    }

    public Shape makeShape(Point2D[] p)
    {
        GeneralPath s = new GeneralPath();
        s.moveTo((float) p[0].getX(), (float) p[0].getY());
        for (int i = 1; i < p.length; i++)
            s.lineTo((float) p[i].getX(), (float) p[i].getY());
        s.closePath();
        return s;
    }
}

```

```
/*
 * Tworzy krzywą drugiego stopnia zdefiniowaną przez jej końce
 * i pojedynczy punkt kontrolny.
 */
class QuadCurveMaker extends ShapeMaker
{
    public QuadCurveMaker()
    {
        super(3);
    }

    public Shape makeShape(Point2D[] p)
    {
        return new QuadCurve2D.Double(p[0].getX(), p[0].getY(), p[1].getX(),
            ↳p[1].getY(), p[2].getX(), p[2].getY());
    }
}

/*
 * Tworzy krzywą trzeciego stopnia zdefiniowaną przez jej końce
 * i dwa punkty kontrolne.
 */
class CubicCurveMaker extends ShapeMaker
{
    public CubicCurveMaker()
    {
        super(4);
    }

    public Shape makeShape(Point2D[] p)
    {
        return new CubicCurve2D.Double(p[0].getX(), p[0].getY(), p[1].getX(),
            ↳p[1].getY(), p[2].getX(), p[2].getY(), p[3].getX(), p[3].getY());
    }
}
```

---

W ogólności obiekt klasy GeneralPath nie musi reprezentować sekwencji połączonych ze sobą odcinków. Metodę moveTo możemy wywoływać dla niego dowolną ilość razy, rozpoczynając za każdym razem nowy fragment sekwencji.

Do sekwencji możemy także dodać obiekt dowolnej klasy implementującej interfejs Shape. Jego obrrys zostanie dołączony na końcu sekwencji za pomocą metody append. Jej drugi parametr zadecyduje o tym, czy obrrys powinien zostać połączony z ostatnim punktem sekwencji. Na przykład wywołanie

```
Rectangle2D r = . . . ;
path.append(r, false);
```

spowoduje dodanie obrysu prostokąta do sekwencji, ale bez połączenia. Natomiast wywołanie  
path.append(r, true);

połączy dodatkowo za pomocą odcinka początkowy wierzchołek prostokąta z ostatnim punktem sekwencji.

Program, którego tekst źródłowy zawiera listing 7.1, pozwala tworzyć proste sekwencje na przykład, takie jak pokazano na rysunkach 7.7. i 7.8. Z listy rozwijalnej można wybrać następujące rodzaje tworzonych figur:

- odcinki linii prostych,
- prostokąty, zwykłe i zaokrąglone, elipsy,
- łuki (tworzone przez zmianę rozmiarów prostokąta ograniczającego oraz kąta początkowego i końcowego łuku),
- wielokąty (przy użyciu klasy GeneralPath),
- krzywe drugiego i trzeciego stopnia.

Punkty kontrolne wszystkich figur można zmieniać za pomocą myszy. Podczas ich przesuwania reprezentacja graficzna figury odrysowywana jest na bieżąco.

Kod programu jest dość skomplikowany, głównie za sprawą obsługi wielu rodzajów figur i przesuwania punktów kontrolnych.

Bazowa klasa abstrakcyjna ShapeMaker hermetyzuje wspólne właściwości klas umożliwiających tworzenie różnych rodzajów figur. Każda z tych klas definiuje określona liczbę punktów kontrolnych, którymi może manipulować użytkownik. Metoda getPointCount zwraca liczbę tych punktów. Metoda abstrakcyjna

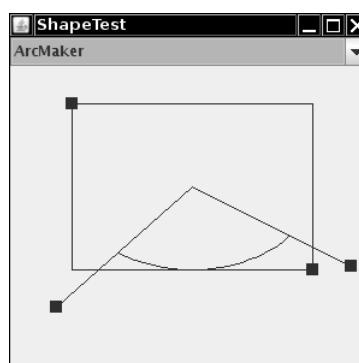
```
Shape makeShape(Point2D[] points)
```

tworzy reprezentację figury, korzystając z aktualnego położenia punktów kontrolnych. Metoda `toString` zwraca po prostu nazwę klasy, umożliwiając w ten sposób wypełnienie listy rozwijalnej klasy JComboBox nazwami specjalizowanych obiektów ShapeMaker.

Aby umożliwić przesuwanie punktów kontrolnych, klasa ShapePanel musi obsługiwać zdarzenia związane zarówno z wyborem za pomocą myszy, jak i jej ruchem. Jeśli klawisz myszy przyciski zostanie, gdy jej kurSOR znajduje się ponad prostokątem reprezentującym punkt kontrolny, to przesunięcie kursora spowoduje także przesunięcie punktu kontrolnego.

Implementacja większości klas umożliwiających tworzenie figur jest stosunkowo prosta. Ich metody `makeShape` tworzą figury na podstawie bieżącego położenia punktów kontrolnych. Jedynie klasa ArcMaker musi dodatkowo wyznaczać zniekształcone wartości początkowe i końcowe kąta łuku. Dodatkowo aby pokazać, że obliczenia te są prawidłowe, metoda `makeShape` klasy ArcMaker zwraca obiekt klasy GeneralPath złożony z łuku, ale i ograniczającego go prostokąta oraz linii łączących środek łuku z punktami kontrolnymi kątów (patrz rysunek 7.9).

**Rysunek 7.9.**  
Program ShapeTest  
w działaniu



**API java.awt.geom.RoundRectangle2D.Double 1.2**

- RoundRectangle2D.Double(double x, double y, double w, double h, double arcWidth, double arcHeight)  
tworzy prostokąt o zaokrąglonych narożnikach o podanej pozycji i wymiarach. Objasnienie parametrów arcWidth i arcHeight znajdziesz na rysunku 7.4.

**API java.awt.geom.Arc2D.Double 1.2**

- Arc2D.Double(double x, double y, double w, double h, double startAngle, double arcAngle, int type)  
tworzy łuk ograniczony prostokątem o określonych wartościach początkowej i kącie rozwarcia, a także typie zamknięcia. Parametry startAngle i arcAngle zostały omówione na rysunku 7.5. Parametr type może przyjmować jedną z wartości Arc2D.OPEN, Arc2D.PIE i Arc2D.CHORD.

**API java.awt.geom.QuadCurve2D.Double 1.2**

- QuadCurve2D.Double(double x1, double y1, double ctrlx, double ctrly, double x2, double y2)  
tworzy krzywą drugiego stopnia na podstawie współrzędnych początku krzywej, punktu kontrolnego i końca krzywej.

**API java.awt.geom.CubicCurve2D 1.2**

- CubicCurve2D.Double(double x1, double y1, double ctrlx1, double ctrly1, double ctrlx2, double ctrly2, double x2, double y2)  
tworzy krzywą trzeciego stopnia na podstawie współrzędnych początku krzywej, dwu punktów kontrolnych i końca krzywej.

**API java.awt.geom.GeneralPath 1.2**

- GeneralPath()  
tworzy pustą sekwencję.

**API java.awt.geom.Path2D.Float 6**

- void moveTo(float x, float y)  
przyjmuje punkt (x, y) za punkt bieżący nowego segmentu.
- void lineTo(float x, float y)
- void quadTo(float ctrlx, float ctrly, float x, float y)
- void curveTo(float ctrlx1, float ctrly1, float ctrlx2, float ctrly2, float x, float y)

Rysując odcinek linii prostej, krzywej drugiego bądź trzeciego stopnia łączący bieżący punkt z punktem (x, y), który staje się nowym punktem bieżącym segmentu.

**API** java.awt.geom.Path2D 6

- void append(Shape s, boolean connect)

dodaje obrys figury do sekwencji. Jeśli connect posiada wartość true, to bieżący punkt sekwencjiłączony jest za pomocą odcinka linii prostej z początkowym wierzchołkiem figury.

- void closePath()

zamyka sekwencję wierzchołków, łącząc bieżący punkt z pierwszym z wierzchołków sekwencji.

## 7.3. Pola

W poprzednim podrozdziale pokazaliśmy, w jaki sposób można tworzyć skomplikowane figury, konstruując je z odcinków prostych i krzywych. Stosując odpowiednio dużą liczbę takich odcinków, możemy narysować dowolnie skomplikowaną figurę. Nawet czcionki, które widzimy na ekranie bądź wydruku, tworzone są jako kombinacje odcinków prostych i krzywych trzeciego stopnia.

Czasami prościej jednak opisać figurę jako kombinację *pól* ograniczonych prostokątami, wielokątami czy elipsami. Java 2D udostępnia cztery operacje *geometrii pól*, z których każda tworzy nowe pole jako kombinację dwóch innych pól:

- add — utworzone pole zawiera wszystkie punkty, które należały do jednego z pól wyjściowych,
- subtract — utworzone pole zawiera wszystkie punkty, które należały do pierwszego pola, ale nie do drugiego,
- intersect — utworzone pole zawiera wszystkie punkty, które należały jednocześnie do obu pól,
- exclusiveOr — utworzone pole zawiera wszystkie punkty, które należały do jednego z pól wyjściowych, ale nie do obu naraz.

Rysunek 7.10 przedstawia wymienione operacje.

Tworzenie skomplikowanego pola rozpoczynamy od utworzenia domyślnego obiektu pola.

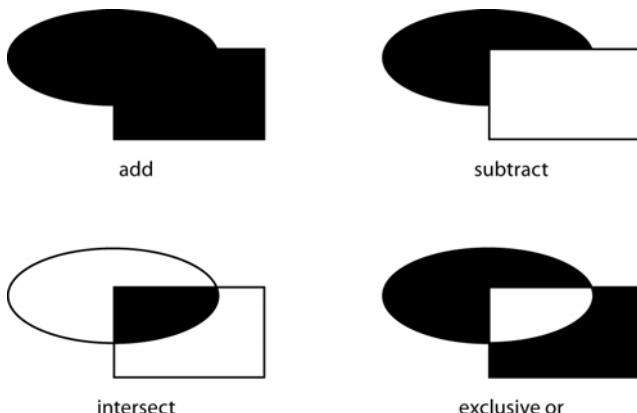
```
Area a = new Area();
```

Następnie tworzymy jego kombinację z dowolnym innym polem:

```
a.add(new Rectangle2D.Double(. . .));
a.subtract(path);
. . .
```

Klasa Area implementuje interfejs Shape. Możemy więc narysować obrys pola, korzystając z metody draw klasy Graphics2D i wypełnić go za pomocą metody fill.

**Rysunek 7.10.**  
Operacje na polach



#### API java.awt.geom.Area

- void add(Area other)
- void subtract(Area other)
- void intersect(Area other)
- void exclusiveOr(Area other)

Wykonują operacje geometrii pól na danym obiekcie oraz obiekcie będącym parametrem metody. Obiekt, na rzecz którego wywołano metody, staje się automatycznie wynikiem operacji.

## 7.4. Ślad pędzla

Metoda `draw` klasy `Graphics2D` rysuje obrrys figury, korzystając z aktualnego śladu pędzla. Domyślnie ślad pędzla posiada postać ciągłej linii o szerokości jednego piksela. Możemy go zmienić, używając metody `setStroke`, której parametrem jest obiekt klasy implementującej interfejs `Stroke`. Java 2D definiuje tylko jedną taką klasę o nazwie `BasicStroke`. W tym podrozdziale przedstawimy jej możliwości.

Stosując klasę `BasicStroke`, możemy utworzyć ślad pędzla dowolnej szerokości. Poniższy przykład tworzy ślad pędzla o szerokości 10 pikseli.

```
g2.setStroke(new BasicStroke(10.0F));
g2.draw(new Line2D.Double(...));
```

Gdy ślad pędzla posiada szerokość większą niż jeden piksel, to dostępne są różne jego *zakończenia*. Rysunek 7.11 przedstawia następujące rodzaje zakończeń:

- *proste* — urywa ślad pędzla w punkcie końcowym,
- *zaokrąglone* — dodaje półkole za punktem końcowym,
- *kwadratowe* — dodaje półkwadrat za punktem końcowym.

**Rysunek 7.11.**

Rodzaje zakończeń  
śladu pędzla

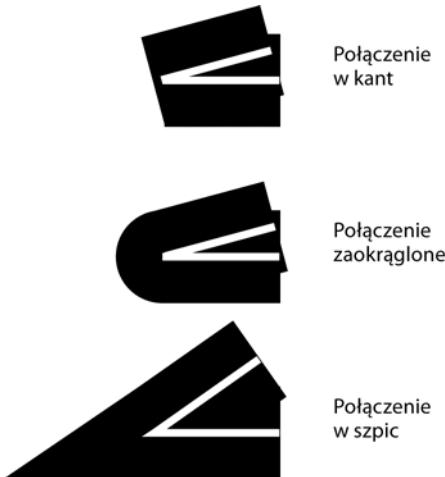


Podobnie możemy określić też trzy rodzaje *połączeń* dwu śladów pędzla (patrz rysunek 7.12):

- *kant* — kończy połączenie śladów pędzla prostym odcinkiem prostopadlym do dwusiecznej kąta pomiędzy śladami,
- *okrągłe* — dodaje zaokrąglone połączenie śladów,
- *szpic* — przedłuża ślady, tak by tworzyły szpicaste zakończenie.

**Rysunek 7.12.**

Rodzaje połączeń



Połączenie w szpic nie może być zastosowane w przypadku śladów łączących się pod niewielkim kątem. Zapobiega to tworzeniu wyjątkowo długich zakończeń połączeń. Jeśli kąt jest mniejszy od pewnej *wartości granicznej* kąta ustalonej dla połączenia w szpic, to automatycznie zastępowane jest ono kantem, co zapobiega powstawaniu zbyt długich „szpiców”. Domyślna wartość graniczna kąta wynosi 10 stopni.

Rodzaj zakończenia oraz połączenia możemy określić, tworząc obiekt klasy `BasicStroke`, na przykład jak poniżej:

```
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_ROUND,
    BasicStroke.JOIN_ROUND));
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_BUTT,
    BasicStroke.JOIN_MITER, 15.0F /* kąt graniczny połączenia w szpic */));
```

Możemy określić także dowolny *przerywany wzór śladu pędzla*. Program z listingu 7.2 umożliwia wybranie śladu pędzla odpowiadającego sygnałowi SOS w alfabetie Morse'a. Przerywany wzór śladu pędzla opisujemy za pomocą tablicy float[], której kolejne pozycje określają na przemian długości odcinków ciągłych i przerw wzoru (patrz rysunek 7.13).

**Rysunek 7.13.**

*Przerywany wzór śladu pędzla*

**Listing 7.2.** stroke/StrokeTest.java

```
package stroke;

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * Program demonstrujący różne ślady pędzla.
 * @version 1.03 2007-08-16
 * @author Cay Horstmann
 */
public class StrokeTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new StrokeTestFrame();
                frame.setTitle("StrokeTest");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }

    /**
     * Ramka zawierająca przyciski wyboru parametrów śladu pędzla
     * i pokazująca efekt zastosowania śladu pędzla.
     */
    class StrokeTestFrame extends JFrame
    {
        private StrokeComponent canvas;
        private JPanel buttonPanel;

        public StrokeTestFrame()
        {
            canvas = new StrokeComponent();
            add(canvas, BorderLayout.CENTER);

            buttonPanel = new JPanel();
            buttonPanel.setLayout(new GridLayout(3, 3));
        }
    }
}
```

```

        add(buttonPanel, BorderLayout.NORTH);

        ButtonGroup group1 = new ButtonGroup();
        makeCapButton("Butt Cap", BasicStroke.CAP_BUTT, group1);
        makeCapButton("Round Cap", BasicStroke.CAP_ROUND, group1);
        makeCapButton("Square Cap", BasicStroke.CAP_SQUARE, group1);

        ButtonGroup group2 = new ButtonGroup();
        makeJoinButton("Miter Join", BasicStroke.JOIN_MITER, group2);
        makeJoinButton("Bevel Join", BasicStroke.JOIN_BEVEL, group2);
        makeJoinButton("Round Join", BasicStroke.JOIN_ROUND, group2);

        ButtonGroup group3 = new ButtonGroup();
        makeDashButton("Solid Line", false, group3);
        makeDashButton("Dashed Line", true, group3);
    }

    /**
     * Tworzy przycisk wyboru zakończenia śladu pędzla.
     * @param label etykieta przycisku
     * @param style rodzaj zakończenia
     * @param group grupa przycisków wyboru
     */
    private void makeCapButton(String label, final int style, ButtonGroup group)
    {
        // wybiera pierwszy przycisk grupy
        boolean selected = group.getButtonCount() == 0;
        JRadioButton button = new JRadioButton(label, selected);
        buttonPanel.add(button);
        group.add(button);
        button.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                canvas.setCap(style);
            }
        });
        pack();
    }

    /**
     * Tworzy przycisk wyboru połączenia śladów pędzla.
     * @param label etykieta przycisku
     * @param style rodzaj połączenia
     * @param group grupa przycisków wyboru
     */
    private void makeJoinButton(String label, final int style, ButtonGroup group)
    {
        // wybiera pierwszy przycisk grupy
        boolean selected = group.getButtonCount() == 0;
        JRadioButton button = new JRadioButton(label, selected);
        buttonPanel.add(button);
        group.add(button);
        button.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                canvas.setJoin(style);
            }
        });
    }
}

```

```
        }
    });
}

/**
 * Tworzy przycisk wyboru wzoru śladu pędzla.
 * @param label etykieta przycisku
 * @param style false dla linii ciągłej, true dla przerwanej
 * @param group grupa przycisków wyboru
 */
private void makeDashButton(String label, final boolean style, ButtonGroup group)
{
    // wybiera pierwszy przycisk grupy
    boolean selected = group.getButtonCount() == 0;
    JRadioButton button = new JRadioButton(label, selected);
    buttonPanel.add(button);
    group.add(button);
    button.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            canvas.setDash(style);
        }
    });
}

/**
 * Komponent rysujący dwa połączone odcinki
 * za pomocą różnych śladów pędzla i umożliwiający
 * użytkownikowi manipulację końcami odcinków.
 */
class StrokeComponent extends JComponent
{
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 400;
    private static int SIZE = 10;

    private Point2D[] points;
    private int current;
    private float width;
    private int cap;
    private int join;
    private boolean dash;

    public StrokeComponent()
    {
        addMouseListener(new MouseAdapter()
        {
            public void mousePressed(MouseEvent event)
            {
                Point p = event.getPoint();
                for (int i = 0; i < points.length; i++)
                {
                    double x = points[i].getX() - SIZE / 2;
                    double y = points[i].getY() - SIZE / 2;
                    Rectangle2D r = new Rectangle2D.Double(x, y, SIZE, SIZE);
                    if (r.contains(p))

```

```

        {
            current = i;
            return;
        }
    }

    public void mouseReleased(MouseEvent event)
    {
        current = -1;
    }
});

addMouseMotionListener(new MouseMotionAdapter()
{
    public void mouseDragged(MouseEvent event)
    {
        if (current == -1) return;
        points[current] = event.getPoint();
        repaint();
    }
});

points = new Point2D[3];
points[0] = new Point2D.Double(200, 100);
points[1] = new Point2D.Double(100, 200);
points[2] = new Point2D.Double(200, 200);
current = -1;
width = 8.0F;
}

public void paintComponent(Graphics g)
{
    Graphics2D g2 = (Graphics2D) g;
    GeneralPath path = new GeneralPath();
    path.moveTo((float) points[0].getX(), (float) points[0].getY());
    for (int i = 1; i < points.length; i++)
        path.lineTo((float) points[i].getX(), (float) points[i].getY());
    BasicStroke stroke;
    if (dash)
    {
        float miterLimit = 10.0F;
        float[] dashPattern = { 10F, 10F, 10F, 10F, 10F, 10F, 30F, 10F, 30F, 10F,
            30F, 10F, 10F, 10F, 10F, 10F, 10F, 30F };
        float dashPhase = 0;
        stroke = new BasicStroke(width, cap, join, miterLimit, dashPattern, dashPhase);
    }
    else stroke = new BasicStroke(width, cap, join);
    g2.setStroke(stroke);
    g2.draw(path);
}

/**
 * Określa rodzaj połączenia.
 * @param j rodzaj połączenia
 */
public void setJoin(int j)
{
}

```

```

        join = j;
        repaint();
    }

    /**
     * Określa rodzaj zakończenia.
     * @param c rodzaj zakończenia
     */
    public void setCap(int c)
    {
        cap = c;
        repaint();
    }

    /**
     * * Określa rodzaj linii.
     * @param d false dla ciąglej, true dla przerwanej
     */
    public void setDash(boolean d)
    {
        dash = d;
        repaint();
    }

    public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH,
        DEFAULT_HEIGHT); }
}

```

Tworząc obiekt klasy `BasicStroke`, możemy określić nie tylko wzór śladu pędzla, ale także jego fazę. Określa ona miejsce wzoru pędzla, od którego powinno rozpoczynać się rysowanie linii. Najczęściej wartość ta wynosi 0 i oznacza rozpoczęcie rysowania od początku wzoru.

```

float[] dashpattern
= { 10, 10, 10, 10, 10, 10, 30, 10, 30, ...};
g2.setStroke(new BasicStroke(10.0F, BasicStroke.CAP_BUTT,
    BasicStroke.JOIN_MITER, 15.0F /*kąt graniczny połączenia w szpic */,
    dashPattern, 0 /*faza */));

```



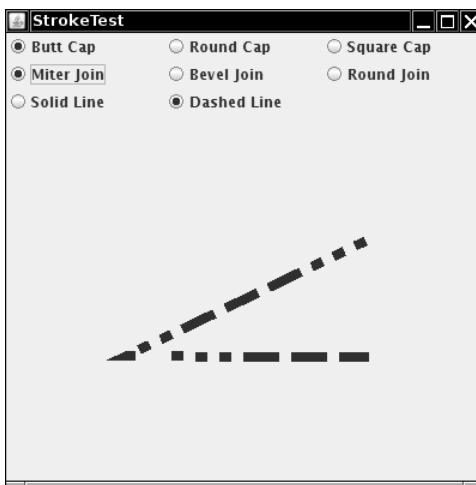
Wybrany rodzaj zakończenia używany jest także dla poszczególnych odcinków przerwanego wzoru pędzla.

Program z listingu 7.2 pozwala określić rodzaj zakończeń i połączeń oraz wybierać ciągły lub przerwany wzór pędzla (patrz rysunek 7.14). Dodatkowo pozwala on na zmianę położenia niepołączonych końców odcinków. Dzięki temu możliwe jest przetestowanie granicznej wartości kąta dla połączenia w szpic. Jeśli wybierzemy połączenie w szpic, a następnie będziemy zmniejszać kąt pomiędzy odcinkami, to po przekroczeniu wartości granicznej kąta połączenie w szpic zostanie automatycznie zastąpione kantem.

Program ten działa podobnie jak program z listingu 7.1. Obiekt nasłuchujący zdarzenia wyboru myszą zapamiętuje sytuację wyboru końca odcinka, a obiekt nasłuchujący ruchu myszy monitoruje przesuwanie tego punktu. Zestaw przycisków wyboru służy określeniu rodzaju śladu pędzla. Metoda `paintComponent` klasy `StrokePanel` tworzy obiekt klasy `GeneralPath` złożony z trzech punktów połączonych dwoma odcinkami, którymi może manipulować użytkownik. Następnie kreuje obiekt klasy `BasicStroke` zgodnie z wyborem jego parametrów dokonanym przez użytkownika i rysuje obiekt `GeneralPath`.

**Rysunek 7.14.**

Program *StrokeTest*  
w działaniu

**API** `java.awt.Graphics2D 1.2`

- `void setStroke(Stroke s)`

sprawia, że obiekt implementujący interfejs `Stroke` tworzyć będzie ślad pędzla dla danego kontekstu graficznego.

**API** `java.awt.BasicStroke 1.2`

- `BasicStroke(float width)`
- `BasicStroke(float width, int cap, int join)`
- `BasicStroke(float width, int cap, int join, float miterLimit)`
- `BasicStroke(float width, int cap, int join, float miterLimit, float[] dash, float dashPhase)`

Tworzą ślad pędzla o określonych atrybutach.

<i>Parametry:</i>	<i>width</i>	szerokość pędzla,
	<i>cap</i>	zakończenie śladu pędzla, jedna z wartości <code>CAP_BUTT</code> , <code>CAP_ROUND</code> i <code>CAP_SQUARE</code> ,
	<i>join</i>	połączenie śladów pędzla, jedna z wartości <code>JOIN_BEVEL</code> , <code>JOIN_MITER</code> i <code>JOIN_ROUND</code> ,
	<i>miterLimit</i>	wartość graniczna kąta wyrażona w stopniach, poniżej której połączenie w szpic jest automatycznie zastępowane kantem,
	<i>dash</i>	tablica długości występujących na przemian odcinków ciągłych i odstępów przerywanego śladu pędzla,
	<i>dashPhase</i>	„faza” wzoru przerywanego pędzla. Określa długość segmentu wzoru pędzla, która powinna zostać pominięta za każdym razem, gdy rozpoczynane jest rysowanie.

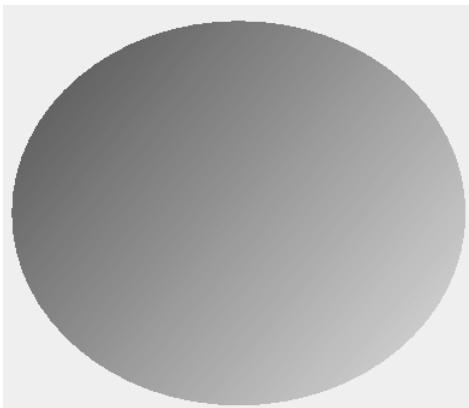
## 7.5. Wypełnienia

Wywołanie metody `fill` powoduje *wypełnienie* obszaru figury. Metoda `setPaint` umożliwia określenie rodzaju wypełnienia za pomocą obiektu implementującego interfejs `Paint`. Java 2D udostępnia trzy klasy implementujące ten interfejs:

- `Color` — umożliwia wypełnienie figury pojedynczym kolorem, na przykład:  
`g2.setPaint(Color.RED);`
- `GradientPaint` — zmienia stopniowo kolor wypełnienia pomiędzy dwiema określonymi wartościami (patrz rysunek 7.15),

**Rysunek 7.15.**

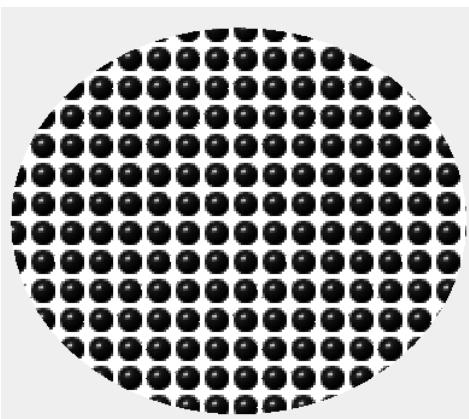
Przykład zastosowania  
klasy `GradientPaint`



- `TexturePaint` — wypełnia obszar figury, powtarzając zadany wzorzec (patrz rysunek 7.16).

**Rysunek 7.16.**

Przykład zastosowania  
klasy `TexturePaint`



Obiekt klasy `GradientPaint` tworzymy, określając kolor w dwóch wybranych punktach obszaru figury.

```
g2.setPaint(new GradientPaint(p1, Color.RED, p2, Color.YELLOW));
```

Kolory są zmieniane wzdłuż linii łączącej te punkty, natomiast pozostają stałe wzdłuż prostopadłych. Punktom, które znajdują się poza punktami końcowymi odcinka, przyporządkowany zostaje taki sam kolor jak dla punktów końcowych.

Alternatywnie, możemy także wykorzystać inną wersję konstruktora, w której dodatkowy parametr `cyclic` określa, czy kolor zmieniać ma się także dla punktów leżących na zewnątrz odcinka.

```
g2.setPaint(new GradientPaint(p1, Color.RED, p2, Color.YELLOW, true));
```

Aby utworzyć obiekt klasy `TexturePaint`, musimy przekazać mu obiekt klasy `BufferedImage` zawierający obrazek wzorca wypełnienia i określić prostokąt *zakotwiczenia*.

```
g2.setPaint(new TexturePaint(bufferedImage, anchorRectangle));
```

Klasę `BufferedImage` omówimy dokładniej podczas przedstawiania operacji na obrazach. Najprościej możemy utworzyć obiekt klasy `BufferedImage` wczytując plik zawierający obrazek:

```
bufferedImage = ImageIO.read(new File("blue-ball.gif "));
```

Obrazek wzorca jest skalowany, tak by mieścił się w prostokącie zakotwiczenia, a prostokąt ten powielany jest następnie w kierunkach osi *x* i *y* tak, by pokryć obszar całej figury.

#### **API** java.awt.Graphics2D 1.2

- `void setPaint(Paint s)`

sprawia, że obiekt implementujący interfejs `Paint` tworzyć będzie wypełnienie dla danego kontekstu graficznego.

#### **API** java.awt.GradientPaint 1.2

- `GradientPaint(float x1, float y1, Color color1, float x2, float y2, Color color2)`
- `GradientPaint(float x1, float y1, Color color1, float x2, float y2, Color color2, boolean cyclic)`
- `GradientPaint(Point2D p1, Color color1, Point2D p2, Color color2)`
- `GradientPaint(Point2D p1, Color color1, Point2D p2, Color color2, boolean cyclic)`

Tworzą obiekt, który wypełnia obszar figury w taki sposób, że punkt początkowy posiada kolor `color1`, a punkt końcowy kolor `color2`. Punkty leżące na odcinku pomiędzy punktem początkowym i końcowym posiadają zmieniający się stopniowo kolor. Punkty leżące na prostopadłych do odcinka posiadają ten sam kolor. Domyślnie wzór zmiany koloru nie jest cykliczny. Oznacza to, że punkty leżące poza odcinkiem posiadają taki sam kolor jak punkt początkowy lub końcowy. Jeśli wzór zmiany koloru jest cykliczny, oznacza to, że powtarzany będzie także dla punktów leżących poza odcinkiem.

**API** java.awt.TexturePaint 1.2

- `TexturePaint(BufferedImage texture, Rectangle2D anchor)`

tworzy wypełnienie za pomocą wzorca. Prostokąt kotwiczący definiuje sposób wypełnienia obszaru figury wzorcem. Prostokąt ten jest powtarzany w kierunkach osi  $x$  i  $y$ , a wymiary wzorca są dopasowywane do jego wymiarów.

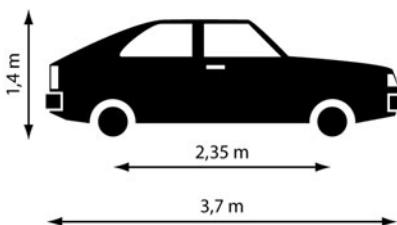
## 7.6. Przekształcenia układu współrzędnych

Założymy, że naszym zadaniem jest przedstawienie na rysunku pewnego modelu samochodu. Z jego specyfikacji technicznej znamy wymiary, takie jak wysokość, rozstaw osi i długość całkowita. Możemy oczywiście sami wyznaczyć rozmiary rysunku, przyjmując pewną liczbę pikseli na metr. Jest jednak łatwiejszy sposób. Wykonanie odpowiednich przekształceń możemy zlecić kontekstowi graficznemu.

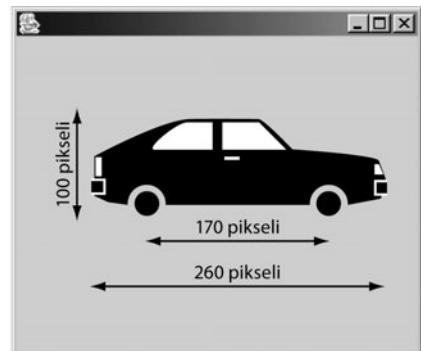
```
g2.scale(pixelsPerMeter, pixelsPerMeter);
g2.draw(new Line2D.Double(współrzędne wyrażone w metrach));
// przekształcenie metrów na piksele i narysowanie linii w odpowiedniej skali
```

Metoda `scale` klasy `Graphics2D` określa w tym przypadku *przekształcenie układu współrzędnych* polegające na jego przeskalowaniu. Przekształcenie to zamienia *współrzędne użytkownika* (w określonych przez niego jednostkach) na *współrzędne ekranowe* (piksele). Rysunek 7.17 przedstawia zasadę takiego przekształcenia.

**Rysunek 7.17.**  
Współrzędne użytkownika i współrzędne ekranowe



Współrzędne użytkownika



Współrzędne ekranowe

Przekształcenia układu współrzędnych okazują się bardzo przydatne w praktyce. Pozwalają pisać programy z wykorzystaniem najwygodniejszego w danym momencie układu współrzędnych. Przekształceniem do układu współrzędnych ekranowych zajmuje się kontekst graficzny.

Wyróżniamy cztery podstawowe rodzaje przekształceń układu współrzędnych:

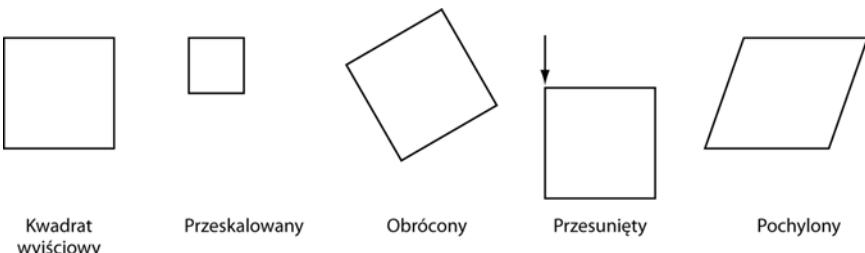
- *skalowanie* polegające na wydłużeniu lub skróceniu wszystkich odległości mierzonych od pewnego punktu,
- *obrót* wszystkich punktów rysunku wokół środka obrotu,

- *przesunięcie* wszystkich punktów o pewną odległość,
- *pochylanie* polegające na przesunięciu wszystkich linii rysunku równoległych do wybranej linii o wartość proporcjonalną do ich odległości od tej linii.

Rysunek 7.18 pokazuje efekt zastosowania tych przekształceń na przykładzie kwadratu.

**Rysunek 7.18.**

Podstawowe przekształcenia



Metody `scale`, `rotate`, `translate` i `shear` klasy `Graphics2D` pozwalają określić jeden z wymienionych rodzajów przekształcenia dla danego kontekstu graficznego.

Możliwe jest *składanie* przekształceń. Możemy na przykład obrócić daną figurę i następnie powiększyć ją dwukrotnie. W tym celu musimy określić oba przekształcenia dla kontekstu graficznego.

```
g2.rotate(angle);
g2.scale(2, 2);
g2.draw(...);
```

W powyższym przypadku kolejność określenia przekształceń nie ma znaczenia. Jednak w większości przypadków złożeniu przekształceń kolejność jest istotna. Na przykład kiedy składamy obrót i pochylenie, uzyskany efekt zależy od tego, które z przekształceń zostanie wykonane jako pierwsze. Kontekst graficzny wykonuje przekształcenia w porządku odwrotnym do tego, w którym je określiliśmy. Tak więc ostatnie przekształcenie określone dla danego kontekstu wykonywane jest jako pierwsze.

Dla danego kontekstu graficznego możemy określić dowolnie wiele przekształceń. Rozważmy na przykład poniższą sekwencję:

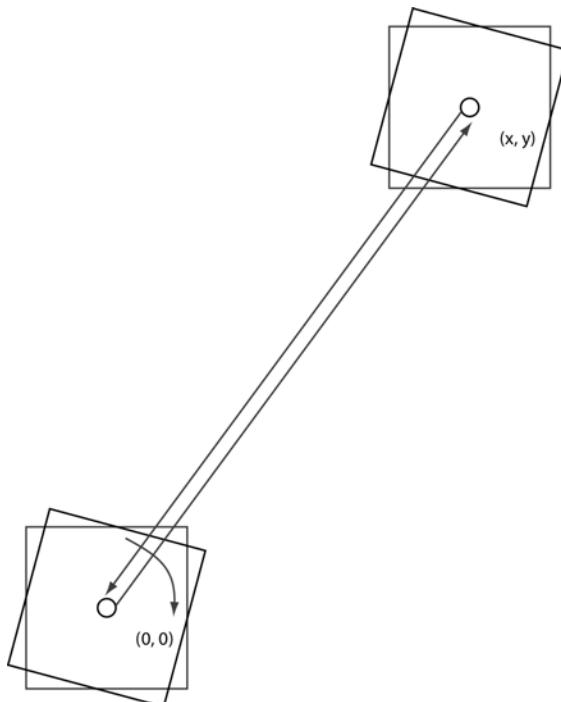
```
g2.translate(x, y);
g2.rotate(a);
g2.translate(-x, -y);
```

Ostatnie z przekształceń (wykonywane jako pierwsze) przesuwa punkt  $(x, y)$  do początku układu współrzędnych. Kolejne wykonuje obrót o kąt  $a$  dookoła początku układu współrzędnych. Ostatnie z przekształceń umieszcza początek układu współrzędnych na powrót w punkcie  $(x, y)$ . Efektem złożenia tych trzech przekształceń jest obrót dookoła punktu  $(x, y)$ , co możemy zobaczyć na rysunku 7.19. Ponieważ obroty względem punktu innego niż początek układu współrzędnych są często spotykanym rodzajem przekształcenia, to Java 2D udostępnia dla nich osobną metodę:

```
g2.rotate(a, x, y);
```

**Rysunek 7.19.**

*Składanie przekształceń*



Wszystkie rodzaje przekształceń oraz ich złożenia opisane mogą być za pomocą działania na macierzach poniższej postaci:

$$\begin{bmatrix} x_{nowe} \\ y_{nowe} \\ 1 \end{bmatrix} = \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Takie przekształcenie macierzy nazywane jest *przekształceniem afinycznym*. W Java 2D jest ono reprezentowane przez klasę `AffineTransform`. Jeśli znamy elementy macierzy przekształceń, to możemy utworzyć obiekt tej klasy w poniższy sposób:

```
AffineTransform t = new AffineTransform(a, b, c, d, e, f);
```

Dostępne są metody fabryki `getRotateInstance`, `getScaleInstance`, `getTranslateInstance` oraz `getShearInstance`, które tworzą macierze dla odpowiednich rodzajów przekształceń. Na przykład wywołanie:

```
t = AffineTransform.getScaleInstance(2.0F, 0.5F);
```

zwraca przekształcenie reprezentowane przez następującą macierz:

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Metody `setToRotation`, `setToScale`, `setToTranslation` i `setToShear` umożliwiają zmianę przekształcenia reprezentowanego przez obiekt, na przykład:

```
t.setToRotation(angle); // przekształcenie t będzie obrotem
```

Za pomocą obiektu klasy `AffineTransform` możemy także określić przekształcenie układu współrzędnych dla danego kontekstu graficznego.

```
g2.setTransform(t); // zastępuje dotychczasowe przekształcenie układu współrzędnych
```

W praktyce jednak metoda ta nie jest zalecana, ponieważ zastępuje także obszar przycięcia danego kontekstu graficznego. W praktyce może zdarzyć się, że na przykład kontekst graficzny dla drukowania na arkuszu o orientacji poziomej zawiera już odpowiedni obrót o 90 stopni. Jeśli wywołamy metodę `setTransform`, to zastapimy ten obrót własnym przekształceniem. W takim przypadku lepiej więc skorzystać z metody `transform`, która złoży istniejące przekształcenie z reprezentowanym przez obiekt klasy `AffineTransform`.

```
g2.transform(t); // składa bieżące przekształcenie z reprezentowanym przez t
```

Jeśli chcemy zastosować pewne przekształcenie tylko jednorazowo, to powinniśmy najpierw zachować istniejące przekształcenie, następnie dokonać złożenia przekształceń, wykonać operacje graficzne i przywrócić wyjściowe przekształcenie.

```
AffineTransform oldTransform = g2.getTransform();
// przechowuje bieżące przekształcenie
g2.transform(t); // dodaje nowe przekształcenie
operacje graficzne z wykorzystaniem kontekstu g2
g2.setTransform(oldTransform); // przywraca wyjściowe przekształcenie
```

#### **java.awt.geom.AffineTransform 1.2**

- `AffineTransform(double a, double b, double c, double d, double e, double f)`
- `AffineTransform(float a, float b, float c, float d, float e, float f)`

Tworzą macierz przekształcenia afiničnego postaci.

$$\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix}$$

- `AffineTransform(double[] m)`
- `AffineTransform(float[] m)`

Tworzą macierz przekształcenia afiničnego postaci.

$$\begin{bmatrix} m[0] & m[2] & m[4] \\ m[1] & m[3] & m[5] \\ 0 & 0 & 1 \end{bmatrix}$$

- `static AffineTransform getRotateInstance(double a)`

tworzy obrót dookoła początku układu współrzędnych o kąt a (wyrażony w radianach). Macierz przekształcenia ma postać:

$$\begin{bmatrix} \cos(a) & -\sin(a) & 0 \\ \sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Jeśli  $a$  jest z przedziału od 0 do  $\pi/2$ , to dodatnia część osi  $x$  zostaje obrócona w kierunku dodatniej części osi  $y$ .

- static `AffineTransform getRotateInstance(double a, double x, double y)`  
tworzy obrót dookoła początku punktu  $(x, y)$  o kąt  $a$  (wyrażony w radianach).
- static `AffineTransform getScaleInstance(double sx, double sy)`  
tworzy przekształcenie skalowania o współczynnik  $sx$  dla osi  $x$  i współczynnik  $sy$  dla osi  $y$ . Macierz przekształcenia ma postać:

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- static `AffineTransform getShearInstance(double shx, double shy)`  
tworzy pochylenie o współczynnik  $shx$  dla osi  $x$  i współczynnik  $shy$  dla osi  $y$ . Macierz przekształcenia ma postać:

$$\begin{bmatrix} 1 & shx & 0 \\ shy & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- static `AffineTransform getTranslateInstance(double tx, double ty)`  
tworzy przesunięcie o współczynnik na odległość  $tx$  w osi  $x$  i odległość  $ty$  w osi  $y$ . Macierz przekształcenia ma postać:

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$$

- void `setToRotation(double a)`
- void `setToRotation(double a, double x, double y)`
- void `setToScale(double sx, double sy)`
- void `setToShear(double shx, double shy)`
- void `setToTranslation(double tx, double ty)`

Określają rodzaj danego przekształcenia afinicznego za pomocą jednego z czterech podstawowych przekształceń. Opis przekształceń i ich parametrów zawiera opis metod `getXXXInstanc`.

**API** java.awt.Graphics2D 1.2

- void setTransform(AffineTransform t)  
zastępuje istniejące przekształcenie układu współrzędnych danego kontekstu graficznego za pomocą przekształcenia t.
  - void transform(AffineTransform t)  
składa istniejące przekształcenie układu współrzędnych danego kontekstu graficznego z przekształceniem t.
  - void rotate(double a)
  - void rotate(double a, double x, double y)
  - void scale (double sx, double sy)
  - void shear (double shx, double shy)
  - void translate(double tx, double ty)
- Składają istniejące przekształcenie układu współrzędnych danego kontekstu graficznego z podstawowym przekształceniem o podanych parametrach. Opis przekształceń i ich parametrów zawiera opis metody `AffineTransform.getXXXInstance`.

## 7.7. Przycinanie

Przez określenie obszaru przycięcia ograniczamy wszystkie operacje graficzne do jego wnętrza.

```
g2.setClip(clipShape); // efekt dla metody poniżej
g2.draw(shape); // rysuje jedynie tę część figury, która mieści się w obszarze przycięcia
```

Zwykle jednak nie będziemy korzystać z metody `setClip`, ponieważ zastępuje ona istniejący już obszar przycięcia kontekstu graficznego, na przykład, jak pokażemy w dalszej części tego rozdziału, kontekst graficzny drukowania zawiera już obszar przycięcia uniemożliwiający drukowanie na marginesach. Dlatego też lepiej w takim przypadku skorzystać z metody `clip`.

```
g2.clip(clipShape); // lepiej
```

Metoda `clip` powoduje utworzenie nowego obszaru przycięcia, który jest częścią wspólną istniejącego dotąd obszaru i obszaru przekazanego jako jej parametr.

Jeśli chcemy wykorzystać dany obszar przycięcia jednorazowo, to powinniśmy zachować najpierw istniejący obszar przycięcia, następnie dodać nowy obszar, wykonać operacje graficzne kontekstu i odtworzyć wyjściowy obszar przycięcia:

```
Shape oldClip = g2.getClip(); // przechowuje obszar przycięcia
g2.clip(clipShape); // dodaje nowy obszar przycięcia
operacje graficzne kontekstu g2
g2.setClip(oldClip); // przywraca wyjściowy obszar przycięcia
```

Rysunek 7.20 demonstruje sposób przycięcia rysunku złożonego z wielu linii za pomocą złożonego obszaru wyznaczonego przez obrys znaków tekstu.

**Rysunek 7.20.**

Przycięcie za pomocą  
obrysu liter



Abytrzymać obrys czcionek, musimy uzyskać najpierw *kontekst tworzenia czcionki*. Użyjemy w tym celu metody `getFontRenderContext` klasy `Graphics2D`.

```
FontRenderContext context = g2.getFontRenderContext();
```

Następnie, używając łańcucha znaków, czcionki i kontekstu tworzenia czcionki, tworzymy obiekt klasy `TextLayout`:

```
TextLayout layout = new TextLayout("Hello", font, context);
```

Obiekt ten opisuje wygląd ciągu znaków utworzonych za pomocą określonego kontekstu tworzenia czcionki. Wygląd ten zależy od zastosowanego kontekstu tworzenia czcionki i będzie różny w przypadku ekranu i drukarki.

Jego metoda `getOutline` zwraca obiekt implementujący interfejs `Shape` opisujący obrys czcionek tekstu. Obrys ten umieszczony jest w początku układu współrzędnych, co nie jest odpowiednie dla większości przypadków. Dlatego też operacji `getOutline` musimy dostarczyć odpowiednio zdefiniowane przekształcenie afimiczne, które spowoduje umieszczenie obrysu tekstu w żądanym miejscu. W naszym przypadku przekształceniem tym będzie przesunięcie do punktu (0, 100).

```
AffineTransform transform
    = AffineTransform.getTranslateInstance(0, 100);
Shape outline = layout.getOutline(transform);
```

Uzyskany w ten sposób obrys tekstułączamy do obszaru przycięcia.

```
GeneralPath clipShape = new GeneralPath();
clipShape.append(outline, false);
```

Następnie określamy obszar przycięcia dla kontekstu graficznego i rysujemy zbiór linii, które widoczne są jedynie wewnątrz obszaru znaków tekstu.

```
g2.setClip(clipShape);
Point2D p = new Point2D.Double(0, 0);
for(int i = 0; i < NLINES; i++)
{
    double x = . . . ;
    double y = . . . ;
    Point2D q = new Point2D.Double(x, y);
    g2.draw(new Line2D.Double(p, q)); //linie są przycinane
}
```

**API** java.awt.Graphics 1.0

- void setClip(Shape s) 1.2  
sprawia, że obszar przycięcia określony jest przez obiekt s.
- Shape getClip() 1.2  
zwraca bieżący obszar przycięcia

**API** java.awt.Graphics2D 1.2

- void clip(Shape s)  
tworzy część wspólną bieżącego obszaru przycięcia z obszarem definiowanym przez s.
- FontRendererContext getFontRendererContext()  
zwraca kontekst tworzenia czcionki niezbędny do utworzenia obiektów klasy TextLayout.

**API** java.awt.font.TextLayout 1.2

- TextLayout(String s, Font f, FontRendererContext context)  
zwraca obiekt opisujący obrys łańcucha znaków s zapisanego czcionką f w kontekście graficznym context.
- float getAdvance()  
zwraca szerokość obrysu tekstu.
- float getAscent()  
■ float getDescent()  
Zwracają wysokość obrysu tekstu odpowiednio nad i pod linią bazową tekstu.
- float getLeading()  
zwraca odległość między kolejnymi liniami czcionki używanej do utworzenia obrysu tekstu.

## 7.8. Przezroczystość i składanie obrazów

W modelu kolorów RGB każdy kolor opisany jest przez wartość jego czerwonej, zielonej i niebieskiej składowej. Czasami przydatna okazuje się jednak możliwość określenia także stopnia przejrzystości obrazu. Gdy nakładamy obraz na już istniejący, to przezroczyste piksele nie zastępują zupełnie istniejących wcześniej pikseli, lecz tworzona jest ich kombinacja. Rysunek 7.21 pokazuje efekt nałożenia częściowo przezroczystego prostokąta na obrazek. Jak widać, nadal możemy rozróżnić szczegóły obrazka przesłonięte częściowo przez prostokąt.

**Rysunek 7.21.**

Nałożenie częściowo przejrzystego prostokąta na obrazek



Java 2D opisuje przejrzystość za pomocą wartości alfa. Każdy piksel oprócz wartości składowej czerwonej, zielonej i niebieskiej jest scharakteryzowany także przez wartość alfa z przedziału od 0 (całkowita przejrzystość) do 1 (całkowite przesłanianie). Prostokąt na pokazany rysunku 7.21 został wypełniony kolorem żółtym o przejrzystości 50%:

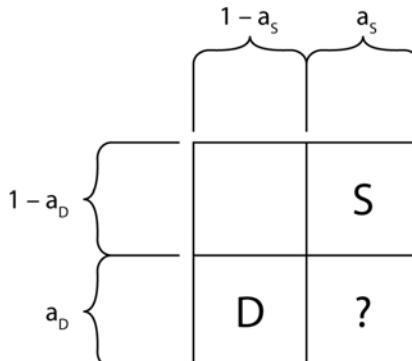
```
new Color(0.7F, 0.7F, 0.0F, 0.5F);
```

Podczas składania obrazów musimy utworzyć nie tylko kombinację składowych kolorów, ale także i wartości alfa. Dwóch badaczy z dziedziny grafiki komputerowej, Porter i Duff, podało dwanaście możliwych reguł składania. Java 2D implementuje wszystkie te reguły. Zanim przejdziemy do ich omówienia, należy zaznaczyć, że jedynie dwie z tych reguł posiadają znaczenie praktyczne. Jeśli reguły składania obrazów wydają się mało czytelne, to polecamy korzystanie jedynie z reguły SRC\_OVER. Jest ona domyślną regułą składania obrazów dla obiektów klasy Graphics2D i daje rezultaty zgodne z naszą intuicją.

Przejdźmy zatem do teorii reguł składania obrazów. Założymy w tym celu, że piksel źródłowy posiada wartość alfa równą  $a_S$ . Natomiast istniejący już obraz posiada piksel docelowy o wartości alfa równej  $a_D$ . Będziemy chcieli utworzyć kombinację obu pikseli. Sposób tworzenia reguły ich złożenia pokazuje diagram na rysunku 7.22.

**Rysunek 7.22.**

Projektowanie reguły składania obrazów



Porter i Duff przez wartość alfa rozumieją prawdopodobieństwo, że zostanie użyty kolor piksela. Z perspektywy piksela źródłowego prawdopodobieństwo to wynosi  $a_S$ , natomiast prawdopodobieństwo, że jego kolor nie zostanie użyty, wynosi  $1-a_S$ . Analogicznie wartości te wyglądają dla piksela docelowego. Tworząc kombinację pikseli, musimy założyć, że są to prawdopodobieństwa zdarzeń niezależnych. W rezultacie otrzymujemy cztery przypadki pokazane na rysunku 7.22. Jeśli kolor piksela źródłowego ma być użyty, a docelowego nie, to oczywiście zastosujemy kolor piksela źródłowego. Dlatego też prawe górnego pole diagramu oznaczamy literą S. Prawdopodobieństwo tego zdarzenia wynosi  $a_S * (1-a_D)$ . Analogicznie lewe dolne pole oznaczamy literą D. A co w przypadku, gdy każdy z pikseli będzie chciał

zastosować własny kolor? Właśnie tu znajdują zastosowanie reguły Portera-Duffa. Jeśli zdecydujemy, że piksel źródłowy jest „ważniejszy”, to także lewe dolne pole oznaczmy literą  $S$ . Na tym polega właśnie domyślna reguła SRC\_OVER. Tworzy ona kombinację koloru piksela źródłowego o wadze  $a_S$  z kolorem piksela docelowego o wadze  $(1-a_S)*a_D$ .

Efektem graficznym zastosowania tej reguły jest złożenie kolorów piksela źródłowego i docelowego z przewagą koloru źródłowego. W szczególności jeśli  $a_S$  wynosi 1, to kolor piksela źródłowego w ogóle nie jest brany pod uwagę. Natomiast w przypadku wartości 0 piksel źródłowy jest zupełnie przezroczysty i zachowany zostaje niezmieniony kolor piksela docelowego.

Możliwe są inne reguły składania obrazów różniące się sposobem umieszczenia liter na diagramie. Tabela 7.1 oraz rysunek 7.23 prezentują reguły Portera-Duffa. Rysunek 7.23 pokazuje wyniki reguł złożenia źródłowego obszaru prostokąta o wartości alfa równej 0,75 z docelowym obszarem wycinka elipsy o wartości alfa równej 1.

**Tabela 7.1.** Zasady składania obrazów Portera-Duffa

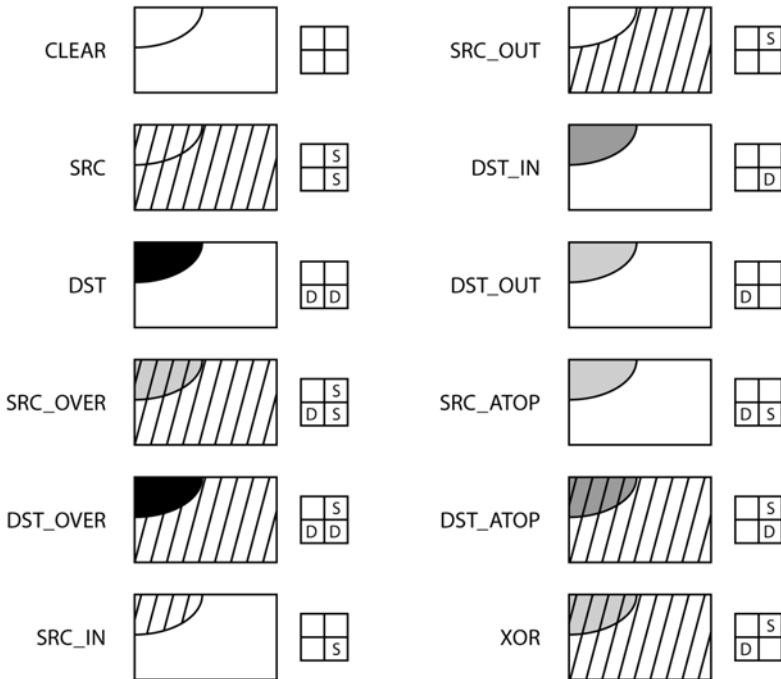
Zasada	Opis
CLEAR	Piksele obrazu źródłowego usuwają piksele obrazu docelowego.
SRC	Piksele obrazu źródłowego zastępują piksele obrazu docelowego oraz puste piksele.
DST	Piksele obrazu źródłowego nie mają wpływu na piksele obrazu docelowego.
SRC_OVER	Piksele obrazu źródłowego tworzą kombinację z pikselami obrazu docelowego i zastępują puste piksele.
DST_OVER	Piksele obrazu źródłowego nie mają wpływu na piksele obrazu docelowego, ale zastępują puste piksele.
SRC_IN	Piksele obrazu źródłowego zastępują piksele obrazu docelowego.
SRC_OUT	Piksele obrazu źródłowego usuwają piksele obrazu docelowego i zastępują puste piksele.
DST_IN	Wartość alfa pikseli obrazu źródłowego modyfikuje piksele obrazu docelowego.
DST_OUT	Uzupełnienie wartości alfa pikseli obrazu źródłowego modyfikuje piksele obrazu docelowego.
SRC_ATOP	Piksele obrazu źródłowego tworzą kombinację z pikselami obrazu docelowego.
DST_ATOP	Wartość alfa pikseli obrazu źródłowego modyfikuje piksele obrazu docelowego. Piksele obrazu źródłowego zastępują puste piksele.
XOR	Uzupełnienie wartości alfa pikseli obrazu źródłowego modyfikuje piksele obrazu docelowego. Piksele obrazu źródłowego zastępują puste piksele.

Jak łatwo zauważać większość reguł jest mało przydatna. Skrajnym przypadkiem jest reguła DST\_IN. Nie bierze ona w ogóle pod uwagę koloru źródła, ale używa jego wartości alfa do modyfikacji obrazu docelowego. Natomiast reguła SRC jest przynajmniej potencjalnie przydatna — wymusza ona użycie koloru źródła, nie tworząc jego kombinacji z kolorem obrazu docelowego.

Więcej informacji o regułach Portera-Duffa odnajdziemy na przykład we wspomnianej już książce autorstwa Foley'ego, van Dama, Feinera et al.

**Rysunek 7.23.**

Reguły składania obrazów Portera-Duffa



Metody `setComposite` klasy `Graphics2D` używamy w celu instalacji obiektu implementującego interfejs `Composite`. Java 2D dostarcza jedną klasę o nazwie `AlphaComposite`, która implementuje reguły Portera-Duffa przedstawione na rysunku 7.23.

Metoda fabryki `getInstance` klasy `AlphaComposite` udostępnia obiekt klasy `AlphaComposite` na podstawie podanej reguły i wartości alfa.

```
int rule = AlphaComposite.SRC_OVER;
float alpha = 0.5;
g2.setComposite(AlphaComposite.getInstance(rule, alpha));
g2.setPaint(Color.blue);
g2.fill(rectangle);
```

Powyższy fragment kodu rysuje prostokąt wypełniony kolorem niebieskim. Ponieważ wartość alfa wynosi dla niego 0,5, a regułą złożenia jest `SRC_OVER`, to nakrywa on półprzezroczystie już istniejący rysunek.

Program z listingu 7.3 umożliwia zapoznanie się w praktyce z działaniem reguł składania obrazów. Lista rozwijalna ułatwia wybór jednej z reguł złożenia, a suwak wartości alfa — nakładanego obrazu.

Dodatkowo program wyświetla krótkie objaśnienie sposobu działania wybranej reguły. Zwróćmy uwagę, że opisy te tworzone są na podstawie zawartości diagramu składania obrazów.

Z działaniem programu składającego obrazy wiąże się pewien problem. Nie ma żadnej gwarancji, że kontekst graficzny związany z ekranem wyposażony jest w kanał wartości alfa. (W rzeczywistości najczęściej nie jest). Gdy piksele umieszczone są bez wykorzystania kanału alfa, to wartości ich kolorów są przemnażane przez wartość alfa, która następnie jest pominięta.

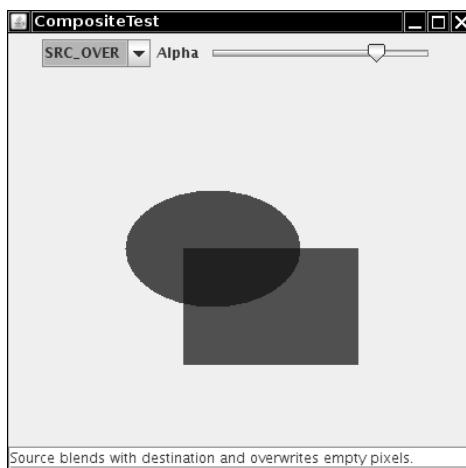
jana. Ponieważ kilka reguł Portera-Duffa korzysta jednak z wartości alfa obrazu docelowego, to istotne jest, by posiadał on kanał alfa. Z tego też powodu do składania kolorów wykorzystujemy obiekt klasy `BufferedImage` wyposażony w model kolorów ARGB. Dopiero po złożeniu obrazów rysujemy je na ekranie.

```
BufferedImage image = new BufferedImage(getWidth(),
    getHeight(), BufferedImage.TYPE_INT_ARGB);
Graphics2D gImage = image.createGraphics();
// teraz możemy narysować obraz
g2.drawImage(image, null, 0, 0);
```

Na listingach 7.3 i 7.4 przedstawiamy kod klasy ramki i klasy komponentu. Klasa `Rule` pokazana na listingu 7.5 dostarcza krótkiego objaśnienia dla każdej z reguł złożenia, co ilustruje rysunek 7.24. Po wybraniu reguły złożenia obrazów warto przesunąć suwak, aby zaobserwować wpływ wartości alfa. W szczególności warto też zwrócić uwagę, że jedyną różnicą między regułami `DST_IN` i `DST_OUT` jest sposób zmiany koloru obrazu docelowego (!) w odpowiedzi na zmianę wartości alfa obrazu źródłowego.

**Rysunek 7.24.**

Program  
*CompositeTest*  
w działaniu



**Listing 7.3.** composite/CompositeTestFrame.java

```
package composite;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Ramka zawierająca listę rozwijalną wyboru reguły,
 * suwak zmiany wartości źródłowej alfa
 * oraz komponent prezentujący efekt złożenia.
 */
class CompositeTestFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 400;

    private CompositeComponent canvas;
```

```
private JComboBox<Rule> ruleCombo;
private JSlider alphaSlider;
private JTextField explanation;

public CompositeTestFrame()
{
    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

    canvas = new CompositeComponent();
    add(canvas, BorderLayout.CENTER);

    ruleCombo = new JComboBox<new Rule[] { new Rule("CLEAR", " ", " "),  

        new Rule("SRC", "S", "S"), new Rule("DST", " ", "DD"),  

        new Rule("SRC_OVER", "S", "DS"), new Rule("DST_OVER", "S", "DD"),  

        new Rule("SRC_IN", " ", "S"), new Rule("SRC_OUT", "S", " "),  

        new Rule("DST_IN", " ", "D"), new Rule("DST_OUT", " ", "D"),  

        new Rule("SRC_ATOP", " ", "DS"), new Rule("DST_ATOP", "S", "D"),  

        new Rule("XOR", "S", "D") } );
    ruleCombo.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            Rule r = (Rule) ruleCombo.getSelectedItem();
            canvas.setRule(r.getValue());
            explanation.setText(r.getExplanation());
        }
    });
}

alphaSlider = new JSlider(0, 100, 75);
alphaSlider.addChangeListener(new ChangeListener()
{
    public void stateChanged(ChangeEvent event)
    {
        canvas.setAlpha(alphaSlider.getValue());
    }
});
JPanel panel = new JPanel();
panel.add(ruleCombo);
panel.add(new JLabel("Alpha"));
panel.add(alphaSlider);
add(panel, BorderLayout.NORTH);

explanation = new JTextField();
add(explanation, BorderLayout.SOUTH);

canvas.setAlpha(alphaSlider.getValue());
Rule r = ruleCombo.getItemAt(ruleCombo.getSelectedIndex());
canvas.setRule(r.getValue());
explanation.setText(r.getExplanation());
}
```

---

**Listing 7.4.** composite/CompositeComponent.java

```
package composite;  
import java.awt.*;
```

```

import java.awt.geom.*;
import java.awt.image.*;
import javax.swing.*;

/**
 * Komponent prezentujący złożenie dwu figur.
 */
class CompositeComponent extends JComponent
{
    private int rule;
    private Shape shape1;
    private Shape shape2;
    private float alpha;

    public CompositeComponent()
    {
        shape1 = new Ellipse2D.Double(100, 100, 150, 100);
        shape2 = new Rectangle2D.Double(150, 150, 150, 100);
    }

    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;

        BufferedImage image = new BufferedImage(getWidth(), getHeight(),
        BufferedImage.TYPE_INT_ARGB);
        Graphics2D gImage = image.createGraphics();
        gImage.setPaint(Color.red);
        gImage.fill(shape1);
        AlphaComposite composite = AlphaComposite.getInstance(rule, alpha);
        gImage.setComposite(composite);
        gImage.setPaint(Color.blue);
        gImage.fill(shape2);
        g2.drawImage(image, null, 0, 0);
    }

    /**
     * Określa regułę złożenia.
     * @param r reguła (jako stała AlphaComposite)
     */
    public void setRule(int r)
    {
        rule = r;
        repaint();
    }

    /**
     * Określa wartość alfa dla źródła
     * @param a wartość alfa z przedziału od 0 do 100
     */
    public void setAlpha(int a)
    {
        alpha = (float) a / 100.0F;
        repaint();
    }
}

```

**Listing 7.5.** composite/Rule.java

```

package composite;

import java.awt.*;

/**
 * Klasa reprezentująca regułę Portera-Duffa.
 */
class Rule
{
    private String name;
    private String porterDuff1;
    private String porterDuff2;

    /**
     * Tworzy obiekt reprezentujący regułę Portera-Duffa
     * @param n nazwa reguły
     * @param pd1 pierwszy wiersz diagramu Portera-Duffa
     * @param pd2 drugi wiersz diagramu Portera-Duffa
     */
    public Rule(String n, String pd1, String pd2)
    {
        name = n;
        porterDuff1 = pd1;
        porterDuff2 = pd2;
    }

    /**
     * Zwraca objaśnienie sposobu działania reguły.
     * @return objaśnienie
     */
    public String getExplanation()
    {
        StringBuilder r = new StringBuilder("Source ");
        if (porterDuff2.equals(" ")) r.append("clears");
        if (porterDuff2.equals(" S")) r.append("overwrites");
        if (porterDuff2.equals("DS")) r.append("blends with");
        if (porterDuff2.equals(" D")) r.append("alpha modifies");
        if (porterDuff2.equals("D ")) r.append("alpha complement modifies");
        if (porterDuff2.equals("DD")) r.append("does not affect");
        r.append(" destination");
        if (porterDuff1.equals(" S")) r.append(" and overwrites empty pixels");
        r.append(".");
        return r.toString();
    }

    public String toString()
    {
        return name;
    }

    /**
     * Zwraca wartość wyznaczoną przez regułę
     * jako obiekt klasy AlphaComposite
     * @return stała AlphaComposite lub -1, jeśli nie istnieje odpowiednia stała.
     */
    public int getValue()
    {

```

```

try
{
    return (Integer) AlphaComposite.class.getField(name).get(null);
}
catch (Exception e)
{
    return -1;
}
}
}

```

**API** java.awt.Graphics2D 1.2

- void setComposite(Composite s)

sprawia, że sposób składania obrazów dla danego kontekstu graficznego określony jest przez obiekt s implementujący interfejs Composite.

**API** java.awt.AlphaComposite 1.2

- static AlphaComposite getInstance(int rule)
- static AlphaComposite getInstance(int rule, float alpha)

Tworzą obiekt klasy AlphaComposite. Parametr rule przyjmuje jedną z wartości CLEAR, SRC, SRC\_OVER, DST\_OVER, SRC\_IN, SRC\_OUT, DST\_IN, DST\_OUT, DST, DST\_ATOP, SRC\_ATOP lub XOR.

## 7.9. Wskazówki operacji graficznych

Na podstawie lektury poprzednich podrozdziałów można przekonać się, że proces tworzenia obrazów może być bardzo złożony. Choć Java 2D okazuje się w większości przypadków zaskakująco efektywna, to jednak zdarzają się sytuacje, w których przydatna jest możliwość określania kompromisu między szybkością operacji graficznych a jakością otrzymanych obrazów. Java 2D udostępnia w tym celu *wskazówki operacji graficznych*. Metoda setRenderingHint klasy Graphics2D umożliwia określenie pojedynczej wskazówki. Klucze i wartości dostępnych wskazówek zadeklarowane są w klasie RenderingHints. Tabela 7.2 prezentuje wszystkie możliwości. Wartości, których nazwa kończy się przyrostkiem \_DEFAULT, oznaczają wartości domyślne wybrane przez określoną implementację jako najlepszy kompromis pomiędzy szybkością działania i jakością obrazu.

Jedna z bardziej przydatnych wskazówek związana jest z techniką *antialiasingu*. Technika ta służy wygładzeniu pochyłych linii prostych oraz krzywych. Jak pokazano na rysunku 7.25, krzywa rysowana jest za pomocą pikseli układających się w „schodki”. Zwłaszcza w przypadku prezentacji za pomocą urządzeń o niskiej rozdzielczości daje to niezbyt elegancki efekt. Okazuje się, że reprezentację krzywej można wygładzić, rysując dodatkowo piksele krzywej o „częściowym” wypełnieniu. Efekt ten uzyskujemy, nadając pikselom wartość koloru proporcjonalną do stopnia ich pokrycia przez krzywą. Jednak ceną, jaką musimy zapłacić za wygładzenie krzywej, są oczywiście obliczenia związane z wyznaczeniem kolorów dodatkowych pikseli.

**Tabela 7.2.** Wskazówki operacji graficznych

Klucz	Wartości	Objaśnienie
KEY_ANTIALIASING	VALUE_ANTIALIAS_ON VALUE_ANTIALIAS_OFF VALUE_ANTIALIAS_DEFAULT	Włącza (wyłącza) antialiasing dla rysowania figur.
KEY_TEXT_ANTIALIASING	VALUE_TEXT_ANTIALIAS_ON VALUE_TEXT_ANTIALIAS_OFF VALUE_TEXT_ANTIALIAS_DEFAULT VALUE_TEXT_ANTIALIAS_GASP 6 VALUE_TEXT_ANTIALIAS_LCD_HRGB 6 VALUE_TEXT_ANTIALIAS_LCD_HRGR 6 VALUE_TEXT_ANTIALIAS_LCD_VRGB 6 VALUE_TEXT_ANTIALIAS_LCD_VBGR 6	Włącza (wyłącza) antialiasing dla czcionek. W przypadku wartości VALUE_TEXT_ANTIALIAS_GASP decyzja o zastosowaniu antialiasingu dla określonego rozmiaru czcionki jest podejmowana na podstawie tabeli opisującej rasteryzację czcionki.
KEY_FRACTIONAL_METRICS	VALUE_FRACTIONALMETRICS_ON VALUE_FRACTIONALMETRICS_OFF VALUE_FRACTIONALMETRICS_DEFAULT	Zwiększa lub zmniejsza dokładność obliczeń rozmiarów czcionek.
KEY_RENDERING	VALUE_RENDER_QUALITY VALUE_RENDER_SPEED VALUE_RENDER_DEFAULT	Wybiera algorytmy rysowania o większej efektywności lub dokładności (jeśli są dostępne).
KEY_STROKE_CONTROL 1.3	VALUE_STROKE_NORMALIZE VALUE_STROKE_PURE VALUE_STROKE_DEFAULT	Wybiera tworzenia śladów pędzla kontrolowane przez akcelerator graficzny (który może przesuwać ślad pędzla o nie więcej niż pół piksela) lub przez „czystą” regułę dopuszczającą jedynie przebieg śladu przez „środek” pikseli.
KEY_DITHERING	VALUE_DITHER_ENABLE VALUE_DITHER_DISABLE VALUE_DITHER_DEFAULT	Włącza (wyłącza) symulowanie wartości kolorów za pomocą grup pikseli o różnych kolorach. (Sposób działania może zależeć od tego, czy stosowany jest również antialiasing).
KEY_ALPHA_INTERPOLATION	VALUE_ALPHA_INTERPOLATION_QUALITY VALUE_ALPHA_INTERPOLATION_SPEED VALUE_ALPHA_INTERPOLATION_DEFAULT	Włącza (wyłącza) precyzyjne obliczenia wartości alfa.
KEY_COLOR_RENDERING	VALUE_COLOR_RENDER_QUALITY VALUE_COLOR_RENDER_SPEED VALUE_COLOR_RENDER_DEFAULT	Zwiększa dokładność lub efektywność wyznaczania kolorów.
KEY_INTERPOLATION	VALUE_INTERPOLATION_NEAREST_NEIGHBOR VALUE_INTERPOLATION_BILINEAR VALUE_INTERPOLATION_BICUBIC	Wybiera regułę interpolacji pikseli podczas skalowania obrazów.

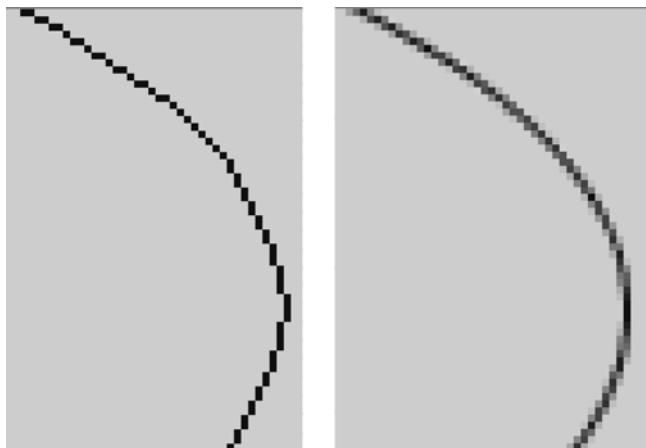
Na przykład wykorzystania antialiasingu podczas rysowania obrysów figur zażądamy w następujący sposób:

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                    RenderingHints.VALUE_ANTIALIAS_ON);
```

Technika antialiasingu jest także przydatna w przypadku rysowania czcionek:

**Rysunek 7.25.**

Antialiasing



```
g2.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
    RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
```

Pozostałe wskazówki operacji graficznych używane są rzadziej.

Zbiór par klucz-wartość wskazówek możemy także umieścić na mapie i wyegzekwować je wszystkie za pomocą pojedynczego wywołania metody `setRenderHints`. Wykorzystać możemy w tym celu dowolną klasę kolekcji implementującą interfejs `Map`, ale także i samą klasę `RenderingHints`. Implementuje ona także interfejs `Map` i dostarcza domyślnej implementacji mapy, jeśli jej konstruktowemu przekażemy wartość `null`, na przykład:

```
RenderingHints hints = new RenderingHints(null);
hints.put(RenderingHints.KEY_ANTIALIASING,
    RenderingHints.VALUE_ANTIALIAS_ON);
hints.put(RenderingHints.KEY_TEXT_ANTIALIASING,
    RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
g2.setRenderHints(hints);
```

Technikę tę wykorzystujemy w programie z listingu 7.6. Tworzy on obrazek, korzystając na początku z wartości wskazówek, które uznaliśmy za potencjalnie przydatne w tym przypadku. Zauważmy, że:

- Antialiasing wygładza elipsę.
- Antialiasing wygładza tekst.
- Na niektórych platformach zastosowanie dokładniejszych obliczeń dla czcionek (`VALUE_FRACTIONALMETRICS_ON`) powoduje ich nieco ciasniejsze wyświetlanie.
- Zastosowanie wskazówki `VALUE_RENDER_QUALITY` wygładza przeskalowany obrazek (ten sam efekt uzyskamy, stosując wartość `VALUE_INTERPOLATION_BICUBIC` dla wskazówki `KEY_INTERPOLATION`).
- Jeśli wyłączymy antialiasing, to wybór wartości `VALUE_STROKE_NORMALIZE` spowoduje zmianę wyglądu elipsy oraz zmianę położenia przekątnej kwadratu.

**Listing 7.6.** renderQuality/RenderQualityTestFrame.java

```
package renderQuality;

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;

/**
 * Ramka zawierająca przyciski wyboru wskazówek
 * oraz obrazek tworzony przy ich zastosowaniu.
 */
public class RenderQualityTestFrame extends JFrame
{
    private RenderQualityComponent canvas;
    private JPanel buttonBox;
    private RenderingHints hints;
    private int r;

    public RenderQualityTestFrame()
    {
        buttonBox = new JPanel();
        buttonBox.setLayout(new GridBagLayout());
        hints = new RenderingHints(null);

        makeButtons("KEY_ANTIALIASING", "VALUE_ANTIALIAS_OFF", "VALUE_ANTIALIAS_ON");
        makeButtons("KEY_TEXT_ANTIALIASING", "VALUE_TEXT_ANTIALIAS_OFF",
                   "VALUE_TEXT_ANTIALIAS_ON");
        makeButtons("KEY_FRACTIONALMETRICS", "VALUE_FRACTIONALMETRICS_OFF",
                   "VALUE_FRACTIONALMETRICS_ON");
        makeButtons("KEY_RENDERING", "VALUE_RENDER_SPEED", "VALUE_RENDER_QUALITY");
        makeButtons("KEY_STROKE_CONTROL", "VALUE_STROKE_PURE", "VALUE_STROKE_NORMALIZE");
        canvas = new RenderQualityComponent();
        canvas.setRenderingHints(hints);

        add(canvas, BorderLayout.CENTER);
        add(buttonBox, BorderLayout.NORTH);
        pack();
    }

    /**
     * Tworzy zestaw przycisków wyboru wskazówek
     * @param key nazwa klucza
     * @param value1 nazwa pierwszej wartości dla klucza
     * @param value2 nazwa drugiej wartości dla klucza
     */
    void makeButtons(String key, String value1, String value2)
    {
        try
        {
            final RenderingHints.Key k = (RenderingHints.Key)
                RenderingHints.class.getField(key).get(
                    null);
            final Object v1 = RenderingHints.class.getField(value1).get(null);
            final Object v2 = RenderingHints.class.getField(value2).get(null);
            JLabel label = new JLabel(key);
```

```

buttonBox.add(label, new GBC(0, r).setAnchor(GBC.WEST));
ButtonGroup group = new ButtonGroup();
JRadioButton b1 = new JRadioButton(value1, true);

buttonBox.add(b1, new GBC(1, r).setAnchor(GBC.WEST));
group.add(b1);
b1.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        hints.put(k, v1);
        canvas.setRenderingHints(hints);
    }
});
JRadioButton b2 = new JRadioButton(value2, false);

buttonBox.add(b2, new GBC(2, r).setAnchor(GBC.WEST));
group.add(b2);
b2.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        hints.put(k, v2);
        canvas.setRenderingHints(hints);
    }
});
hints.put(k, v1);
r++;
}
catch (Exception e)
{
    e.printStackTrace();
}
}

}

/***
 * Komponent tworzący rysunek z zastosowaniem wskazówek.
 */
class RenderQualityComponent extends JComponent
{
    private static final int DEFAULT_WIDTH = 750;
    private static final int DEFAULT_HEIGHT = 150;

    private RenderingHints hints = new RenderingHints(null);
    private Image image;

    public RenderQualityComponent()
    {
        image = new ImageIcon(getClass().getResource("face.gif")).getImage();
    }

    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;
        g2.setRenderingHints(hints);

        g2.draw(new Ellipse2D.Double(10, 10, 60, 50));
    }
}

```

```
g2.setFont(new Font("Serif", Font.ITALIC, 40));
g2.drawString("Hello", 75, 50);

g2.draw(new Rectangle2D.Double(200, 10, 40, 40));
g2.draw(new Line2D.Double(201, 11, 239, 49));

g2.drawImage(image, 250, 10, 100, 100, null);
}

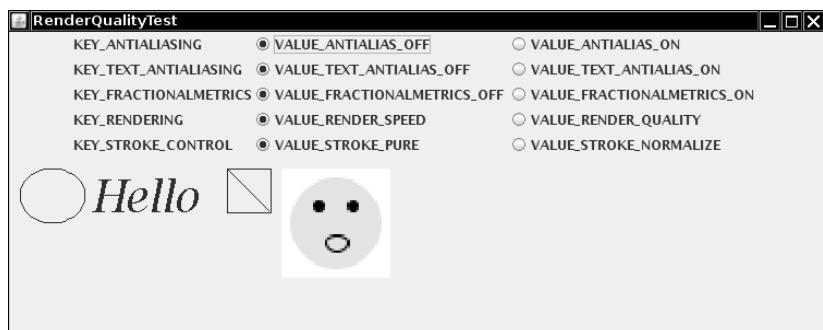
/**
 * Konfiguruje wskazówki i powoduje odrysowanie obrazka.
 * @param h wskazówki
 */
public void setRenderingHints(RenderingHints h)
{
    hints = h;
    repaint();
}

public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH,
DEFAULT_HEIGHT); }
```

Na rysunku 7.26 przedstawiony został efekt działania programu.

**Rysunek 7.26.**

Program  
*RenderingHints*  
w działaniu



### API java.awt.Graphics2D 1.2

- `void setRenderingHint(RenderingHints.Key key, Object value)`  
stosuje wskazówkę do danego kontekstu graficznego.
- `void setRenderingHints(Map m)`  
stosuje do danego kontekstu graficznego wszystkie wskazówki, których pary klucz-wartość przechowuje mapa.

### API java.awt.RenderingHints 1.2

- `RenderingHints(Map<RenderingHints.Key, ?> m)`  
tworzy mapę wskazówek operacji graficznych. Jeśli `m` posiada wartość `null`, to wykorzystywana jest domyślna implementacja mapy.

## 7.10. Czytanie i zapisywanie plików graficznych

W wersjach wcześniejszych niż 1.4, Java SE posiadała bardzo ograniczone możliwości odczytu i zapisu plików graficznych. Możliwy był odczyt plików zapisanych w formatach GIF oraz JPEG, ale zapis obrazów nie był możliwy w żadnym z formatów.

Sytuacja ta uległa poprawie z wprowadzeniem Java SE 1.4. Pakiet `javax.imageio` dostarcza gotowej implementacji operacji odczytu i zapisu plików graficznych w kilku powszechnie stosowanych formatach, a także szkielet umożliwiający tworzenie własnych implementacji dla innych formatów plików graficznych. W Java SE 6 możliwy jest odczyt i zapis plików zapisanych w formatach GIF, JPEG, PNG, BMP (Windows bitmap) i WBMP (wireless bitmap). Brak możliwości zapisu plików w formacie GIF we wcześniejszych wersjach był spowodowany ograniczeniami patentowymi.

Sposób korzystania z podstawowych operacji na plikach graficznych jest bardzo prosty. Aby załadować zawartość pliku graficznego, korzystamy z metody statycznej `read` klasy `ImageIO`:

```
File f = . . . ;
BufferedImage image = ImageIO.read(f);
```

Klasa `ImageIO` sama dobiera odpowiedni obiekt do odczytu danego formatu na podstawie rozszerzenia pliku oraz identyfikatora formatu zapisanego na początku pliku. Jeśli obiekt odczytu dla danego formatu jest niedostępny, to metoda `read` zwraca wartość `null`.

Równie łatwo można zapisać obraz w pliku:

```
File f = . . . ;
String format = . . . ;
ImageIO.write(image, format, f);
```

Parametr `format` metody `write` identyfikuje docelowy format pliku za pomocą łańcucha znaków postaci "JPEG" lub "PNG". Na tej podstawie klasa `ImageIO` dobiera odpowiedni obiekt zapisu pliku.

### 7.10.1. Wykorzystanie obiektów zapisu i odczytu plików graficznych

W przypadku bardziej zaawansowanych operacji na plikach graficznych metody statyczne `read` i `write` klasy `ImageIO` mogą okazać się niewystarczające. W takich sytuacjach musimy najpierw uzyskać odpowiedni dla danego formatu obiekt odczytu `ImageReader` lub obiekt zapisu `ImageWriter`. Klasa `ImageIO` tworzy wyliczenie dostępnych obiektów na podstawie jednej z poniższych informacji:

- formatu pliku (na przykład "JPEG"),
- rozszerzenia nazwy pliku (na przykład "jpg"),
- typu określonego w standardzie MIME (na przykład "image/jpeg").



Standard MIME (*Multipurpose Internet Mail Extensions*) definiuje formaty wymiany danych, na przykład "image/jpeg" czy "application/pdf". Wersję dokumentu RFC w formacie HTML definującego standard MIME można odnaleźć pod adresem <http://www.oac.uci.edu/indiv/ehood/MIME>.

Obiekt odczytu pliku w formacie JPEG możemy uzyskać w poniższy sposób:

```
ImageReader reader = null;
Iterator<ImageReader> iter =
    ImageIO.getImageReadersByFormatName("JPEG");
if (iter.hasNext()) reader = iter.next();
```

Można też wykorzystać w tym celu metody `getImageReadersBySuffix` lub `getImageReadersByMIMEType`, które tworzą wyliczenie na podstawie odpowiednio rozszerzenia pliku lub typu MIME.

Może zdarzyć się, że klasa `ImageIO` zlokalizuje więcej niż jeden obiekt umożliwiający odczyt danego formatu pliku. Należy wtedy wybrać jeden z nich. Aby uzyskać więcej informacji o dostępnych obiektach odczytu, musimy utworzyć dla nich *interfejs dostawcy*:

```
ImageReaderSpi spi = reader.getOriginatingProvider();
```

Korzystając z niego, możemy pobrać informację o nazwie producenta (dostawcy) i numerze wersji:

```
String vendor = spi.getVendor();
String version = spi.getVersion();
```

Informacja ta może okazać się pomocna w wyborze odpowiedniego obiektu lub zostać wykorzystana do utworzenia listy, która pozwoli wybrać obiekt użytkownikowi programu. Dla dalszych rozważań przyjmiemy, że korzystać będziemy z pierwszego ze znalezionych obiektów.

Program zamieszczony na listingu 7.7 znajduje wszystkie rozszerzenia plików dla wszystkich dostępnych obiektów odczytu plików graficznych. Rozszerzenia te wykorzystuje następnie do utworzenia filtra nazw plików. W tym celu używamy metody statycznej `ImageIO.getReaderFileSuffixes`:

```
String[] extensions = ImageIO.getWriterFileSuffixes();
chooser.setFileFilter(new FileNameExtensionFilter("Image files", extensions));
```

---

**Listing 7.7. *imageIO/ImageIOFrame.java***

```
package imageIO;

import java.awt.event.*;
import java.awt.image.*;
import java.io.*;
import java.util.*;
import javax.imageio.*;
import javax.imageio.stream.*;
import javax.swing.*;
import javax.swing.filechooser.*;

/**
 * Ramka wyświetlająca zawartość pliku graficznego.
 * Jej menu umożliwia odczyt i zapis plików.
 */
public class ImageIOFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 400;
```

```

private static Set<String> writerFormats = getWriterFormats();

private BufferedImage[] images;

public ImageIOFrame()
{
    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

    JMenu fileMenu = new JMenu("File");
    JMenuItem openItem = new JMenuItem("Open");
    openItem.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            openFile();
        }
    });
    fileMenu.add(openItem);

    JMenu saveMenu = new JMenu("Save");
    fileMenu.add(saveMenu);
    Iterator<String> iter = writerFormats.iterator();
    while (iter.hasNext())
    {
        final String formatName = iter.next();
        JMenuItem formatItem = new JMenuItem(formatName);
        saveMenu.add(formatItem);
        formatItem.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                saveFile(formatName);
            }
        });
    }
}

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

JMenuBar menuBar = new JMenuBar();
menuBar.add(fileMenu);
setJMenuBar(menuBar);
}

/**
 * Otwiera plik i ładuje obrazek.
 */
public void openFile()
{

```

```
JFileChooser chooser = new JFileChooser();
chooser.setCurrentDirectory(new File("."));
String[] extensions = ImageIO.getReaderFileSuffixes();
chooser.setFileFilter(new FileNameExtensionFilter("Image files", extensions));
int r = chooser.showOpenDialog(this);
if (r != JFileChooser.APPROVE_OPTION) return;
File f = chooser.getSelectedFile();
Box box = Box.createVerticalBox();
try
{
    String name = f.getName();
    String suffix = name.substring(name.lastIndexOf('.') + 1);
    Iterator<ImageReader> iter = ImageIO.getImageReadersBySuffix(suffix);
    ImageReader reader = iter.next();
    ImageInputStream imageIn = ImageIO.createImageInputStream(f);
    reader.setInput(imageIn);
    int count = reader.getNumImages(true);
    images = new BufferedImage[count];
    for (int i = 0; i < count; i++)
    {
        images[i] = reader.read(i);
        box.add(new JLabel(new ImageIcon(images[i])));
    }
}
catch (IOException e)
{
    JOptionPane.showMessageDialog(this, e);
}
setContentPane(new JScrollPane(box));
validate();
}

/**
 * Zapisuje bieżący obrazek w pliku.
 * @param formatName format pliku
 */
public void saveFile(final String formatName)
{
    if (images == null) return;
    Iterator<ImageWriter> iter = ImageIO.getImageWritersByFormatName(formatName);
    ImageWriter writer = iter.next();
    JFileChooser chooser = new JFileChooser();
    chooser.setCurrentDirectory(new File("."));
    String[] extensions = writer.getOriginatingProvider().getFileSuffixes();
    chooser.setFileFilter(new FileNameExtensionFilter("Image files", extensions));

    int r = chooser.showSaveDialog(this);
    if (r != JFileChooser.APPROVE_OPTION) return;
    File f = chooser.getSelectedFile();
    try
    {
        ImageOutputStream imageOut = ImageIO.createImageOutputStream(f);
        writer.setOutput(imageOut);

        writer.write(new IIOImage(images[0], null, null));
        for (int i = 1; i < images.length; i++)
        {

```

```

        IIOImage iioImage = new IIOImage(images[i], null, null);
        if (writer.canInsertImage(i)) writer.writeInsert(i, iioImage, null);
    }
}
catch (IOException e)
{
    JOptionPane.showMessageDialog(this, e);
}
}

/**
 * Tworzy zbiór "preferowanych" nazw formatów plików graficznych,
 * dla których istnieją obiekty odczytu. Preferowaną nazwą formatu
 * zostaje pierwsza nazwa dla danego obiektu odczytu.
 * @return zbiór nazw formatów
*/
public static Set<String> getWriterFormats()
{
    Set<String> writerFormats = new TreeSet<>();
    Set<String> formatNames = new TreeSet<>(Arrays.asList(ImageIO
        .getWriterFormatNames()));
    while (formatNames.size() > 0)
    {
        String name = formatNames.iterator().next();
        Iterator<ImageWriter> iter = ImageIO.getImageWritersByFormatName(name);
        ImageWriter writer = iter.next();
        String[] names = writer.getOriginatingProvider().getFormatNames();
        String format = names[0];
        if (format.equals(format.toLowerCase())) format = format.toUpperCase();
        writerFormats.add(format);
        formatNames.removeAll(Arrays.asList(names));
    }
    return writerFormats;
}
}

```

Zapis plików wymaga trochę więcej pracy. Chcemy zaprezentować użytkownikowi menu formatów graficznych do wyboru, ale, niestety, metoda `getWriterFormatNames` klasy `ImageIO` tworzy mało przydatną listę powtarzających się nazw:

jpg, BMP, bmp, JPG, jpeg, wbmp, png, JPEG, PNG, WBMP, GIF, gif

Na pewno menu wyboru nie powinno wyglądać w ten sposób. Musimy raczej stworzyć listę „preferowanych” nazw formatów. W tym celu tworzymy metodę pomocniczą `getWriterFormats` (patrz listing 7.7). Przeglądając listę, możemy pobrać pierwszą nazwę formatu i wyszukać związaną z nią obiekt zapisu. Następnie pobrać dla niego nazwę formatu w nadziei, że będzie to najbardziej popularna wersja nazwy. Dla obiektu zapisu formatu JPEG rozwiązywanie takie sprawdza się: otrzymujemy nazwę „JPEG”. (Jednak już dla obiektu zapisu formatu PNG jako pierwszą uzyskamy nazwę „png”. Mamy nadzieję, że sytuacja ta zostanie uporządkowana w kolejnej wersji biblioteki. Tymczasem rozwiążujemy problem, zamieniając wszystkie nazwy na duże litery). Po wybraniu preferowanej nazwy usuwamy wszystkie alternatywne nazwy z listy wyjściowej. W ten sposób postępujemy dla wszystkich formatów.

## 7.10.2. Odczyt i zapis plików zawierających sekwencje obrazów

Niektóre pliki, w szczególności pliki animacji w formacie GIF, mogą zawierać wiele obrazów. Metoda `read` klasy `ImageIO` umożliwia odczyt pojedynczego obrazu. Aby załadować wiele obrazów, musimy przekształcić źródło wejścia (na przykład strumień wejściowy lub plik) w obiekt klasy `InputStream`.

```
InputStream in = . . . ;
ImageInputStream imageIn = ImageIO.createImageInputStream(in);
```

A następnie dołączyć uzyskany obiekt do obiektu odczytu pliku:

```
reader.setInput(imageIn, true);
```

Drugi z parametrów metody `setInput` wskazuje, że źródło umożliwiać będzie tylko odczyt w trybie „do przodu”. W przeciwnym razie możliwy będzie swobodny dostęp do danych źródła albo przez buforowanie ich w czasie odczytu, albo za pomocą pliku o dostępie swobodnym. Swobodny dostęp do pliku graficznego bywa przydatny w pewnych sytuacjach. Na przykład jeśli chcemy uzyskać informacje o liczbie obrazów w pliku GIF, musimy przeczytać cały plik. Jeśli zechcemy następnie pobrać jeden z obrazów, to plik nie będzie musiał być odczytany ponownie.

Powyższa uwaga jest istotna jedynie w przypadku odczytu za pośrednictwem strumienia, pliku, który zawiera wiele obrazów i jego format nie udostępnia w nagłówku pliku informacji o liczbie zapisanych obrazów. Natomiast kiedy czytamy obrazy bezpośrednio z pliku, wystarczy nam poniższy fragment kodu:

```
File f = . . . ;
ImageInputStream imageIn = ImageIO.createImageInputStream(f);
reader.setInput(imageIn);
```

Dysponując obiektem odczytu pliku, możemy wczytać określony obraz za pomocą poniższego wywołania:

```
BufferedImage image = reader.read(index);
```

gdzie `index` jest numerem obrazu (numeracja rozpoczyna się od zera).

Jeśli źródło obrazów jest w trybie odczytu „do przodu”, to metodę `read` wywołujemy tak długo, aż nie wyrzuci ona wyjątku `IndexOutOfBoundsException`. W przeciwnym razie możemy skorzystać z metody `getNumImages`:

```
int n = reader.getNumImages(true);
```

Jej parametr zezwala na przeszukanie zawartości pliku w celu ustalenia liczby obrazów. Jeśli ich źródło jest jednak w trybie odczytu „do przodu”, to metoda ta wyrzuci wyjątek `IllegalStateException`. Jeśli podając metodzie parametr o wartości `false`, zabronimy przeszukiwania pliku, to zwróci ona wartość `-1`, w przypadku gdy ustalenie liczby obrazów bez przeszukania pliku nie jest możliwe. W takiej sytuacji musimy odczytywać kolejne obrazy aż do wyrzucenia wyjątku `IndexOutOfBoundsException`.

Nagłówki niektórych plików graficznych mogą zawierać miniatury umożliwiające podgląd ich zawartości. Liczbę miniaturow obrazu o danym indeksie możemy określić, wywołując poniższą metodę:

```
int count = reader.getNumThumbnails(index);
```

Określona miniaturę danego obrazu wczytujemy, tak jak poniżej:

```
BufferedImage thumbnail = reader.readThumbnail(index,
    thumbnailIndex);
```

Często przydatna jest możliwość uzyskania informacji o rozmiarze obrazu — na przykład zanim zostanie załadowany za pośrednictwem sieci o niskiej przepustowości. Następujące wywołania

```
int width = reader.getWidth(index);
int height = reader.getHeight(index);
```

umożliwiają uzyskanie informacji o rozmiarach obrazu o danym indeksie.

Aby zapisać w pliku sekwencję obrazów, musimy najpierw uzyskać odpowiedni obiekt klasy `ImageWriter`. Klasa `ImageIO` umożliwia wyliczenie obiektów zapisu dla określonego formatu graficznego:

```
String format = . . . ;
ImageWriter writer = null;
Iterator<ImageWriter> iter =
    IOImage.getImageWriterByFormatName(format);
if (iter.hasNext()) writer = iter.next();
```

Następnie musimy przekształcić strumień wyjścia lub plik w obiekt klasy `ImageOutputStream`, który udostępnimy obiektowi zapisu. Możemy to zrobić w sposób przedstawiony poniżej:

```
File f = . . . ;
ImageOutputStream imageOut = ImageIO.createImageOutputStream(f);
writer.setOutput(imageOut);
```

Każdy z zapisywanych obrazów musimy obudować jeszcze za pomocą obiektu klasy `IIIOImage`. Opcjonalnie możemy dostarczyć także listę miniatur oraz metadane obrazów (opisujące na przykład algorytm kompresji czy sposób kodowania kolorów). W naszym przypadku użyjemy wartości `null` dla obu opcjonalnych parametrów. Więcej informacji na temat ich wykorzystania znajdziemy w dokumentacji Java 2D.

```
IIIOImage iioImage = new IIIOImage(images[i], null, null);
```

*Pierwszy z obrazów zapiszemy, korzystając z metody `write`:*

```
writer.write(new IIIOImage(images[0], null, null));
```

Natomiast kolejne w następujący sposób:

```
if (writer.canInsertImage(i))
    writer.writeInsert(i, iioImage, null);
```

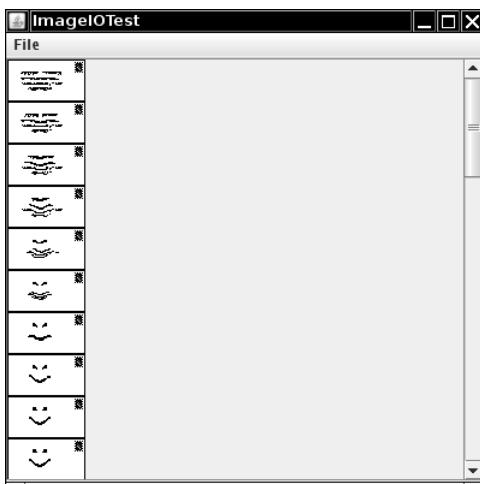
Trzeci z parametrów metody `writeInsert` może być obiektem klasy `ImageWriteParam` opisującym szczegóły zapisu obrazu, takie jak na przykład sposób kompresji. Jeśli chcemy skorzystać z domyślnych sposobów zapisu obrazów, to podajemy po prostu wartość `null`.

Nie wszystkie formaty graficzne umożliwiają zapis wielu obrazów w jednym pliku. W takim przypadku metoda `canInsertImage` zwróci dla wartości indeksu większych od zera wartość `false`.

Program, którego tekst źródłowy zawiera listing 7.7, umożliwia odczyt i zapis plików graficznych w formatach, dla których biblioteka języka Java dostarcza implementacji obiektów odczytu i zapisu. Program wyświetla wszystkie obrazy zawarte w pliku (patrz rysunek 7.27), ale nie wyświetla miniatur.

**Rysunek 7.27.**

Plik animacji  
w formacie GIF



#### API javax.imageio.ImageIO 1.4

- static BufferedImage read(File input)
- static BufferedImage read(InputStream input)
- static BufferedImage read(URL input)

Czytają obraz ze źródła input.

- static boolean write(RenderedImage image, String formatName, File output)
- static boolean write(RenderedImage image, String formatName, OutputStream output)

Zapisują obraz w określonym formacie do wyjścia output. Zwraca wartość false, jeśli nie został znaleziony obiekt zapisu odpowiedni dla danego formatu.

- static Iterator<ImageReader> getImageReadersByFormatName(String formatName)
- static Iterator<ImageReader> getImageReadersBySuffix(String fileSuffix)
- static Iterator<ImageReader> getImageReadersByMIMEType(String mimeType)
- static Iterator<ImageWriter> getImageWritersByFormatName(String formatName)
- static Iterator<ImageWriter> getImageWritersBySuffix(String fileSuffix)
- static Iterator<ImageWriter> getImageWritersByMIMEType(String mimeType)

Tworzą wyliczenie wszystkich obiektów odczytu lub zapisu odpowiednich dla danego formatu (określonego za pomocą nazwy — na przykład "JPEG", rozszerzenia nazwy pliku — na przykład "jpg" bądź typu zgodnego ze standardem MIME — na przykład "image/jpeg").

- static String[] getReaderFormatNames()
- static String[] getReaderMIMETypes()
- static String[] getWriterFormatNames()
- static String[] getWriterMIMETypes()
- static String[] getReaderFileSuffixes() 6
- static String[] getWriterFileSuffixes() 6

Zwracają tablicę nazw formatów, nazw typów MIME lub rozszerzeń nazw, dla których istnieją obiekty odczytu lub zapisu.

- ImageInputStream createImageInputStream(Object input)
- ImageOutputStream createImageOutputStream(Object output)

Przekształca podany obiekt na obiekt klasy ImageInputStream lub ImageOutputStream. Parametrem może być plik, strumień, obiekt klasy RandomAccessFile lub inny obiekt, dla którego istnieje dostawca usługi. Zwraca wartość null, jeśli żaden z zarejestrowanych dostawców usług nie potrafi dokonać konwersji.

#### javax.imageio.ImageReader 1.4

- void setInput(Object input)
- void setInput(Object input, boolean seekForwardOnly)

Okręślają źródło danych dla obiektu odczytu.

Parametry:

input  
seekForwardOnly

obiekt klasy ImageInputStream lub inny obiekt akceptowany przez obiekt odczytu,

wartość true oznacza, że obiekt może czytać jedynie dane „do przodu”. Domyślnie obiekt odczytu wykorzystuje swobodny dostęp do danych, które buforuje, jeśli zachodzi taka potrzeba.

- BufferedImage read(int index)

wczytuje obraz o danym indeksie (wartości indeksu rozpoczynają się od 0). Wyrzuca wyjątek IndexOutOfBoundsException, jeśli obraz taki nie jest dostępny.

- int getNumImages(boolean allowSearch)

pobiera liczbę obrazów dostępnych za pomocą danego obiektu odczytu. Jeśli parametr allowSearch ma wartość false, a liczba obrazów nie może być ustalona bez wcześniejszego odczytania danych, to metoda zwraca wartość -1. Jeśli parametr allowSearch ma wartość true, a obiekt odczytu znajduje się w trybie odczytu „do przodu”, to wyrzucany jest wyjątek IllegalStateException.

- int getNumThumbnails(int index)

zwraca liczbę miniatur obrazu o podanym indeksie.

- BufferedImage readThumbnail(int index, int thumbnailIndex)

zwraca miniaturę o indeksie thumbnailIndex dla obrazu o indeksie index.

- int getWidth(int index)
- int getHeight(int index)

Zwracają szerokość i wysokość obrazu. Wyrzucają wyjątek IndexOutOfBoundsException, jeśli obraz taki nie jest dostępny.

- ImageReaderSpi getOriginatingProvider()
- zwraca dostawcę usługi dla danego obiektu odczytu.

**API javax.imageio.spi.IIOServiceProvider 1.4**

- String getVendorName()
- String getVersion()

Zwracają nazwę producenta i wersję danego dostawcy usługi.

**API javax.imageio.spi.ImageReaderWriterSpi 1.4**

- String[] getFormatNames()
- String[] getFileSuffixes()
- String[] getMIMETypes()

Zwracają nazwy formatów, rozszerzenia nazw plików lub typy MIME obsługiwane przez obiekty odczytu i zapisu danego dostawcy usługi.

**API javax.imageio.ImageWriter 1.4**

- void setOutput(Object output)

określa cel danych dla obiektu zapisu.

*Parametry:* output obiekt klasy FileOutputStream lub inny obiekt akceptowany przez obiekt zapisu.

- void write(IIOImage image)

- void write(RenderedImage image)

Zapisują pojedynczy obraz.

- void writeInsert(int index, IIOImage image, ImageWriteParam param)

zapisuje obraz do pliku zawierającego sekwencję obrazów.

*Parametry:* index indeks obrazu,

image zapisywany obraz,

param parametry zapisu lub wartość null.

- boolean canInsertImage(int index)

zwraca wartość true, jeśli obraz może zostać zapisany na pozycji określonej wartością indeksu.

- `ImageWriterSpi getOriginatingProvider()`  
zwraca dostawcę usługi dla danego obiektu zapisu.

**API javax.imageio.IIOImage 1.4**

- `IIOImage(RenderedImage image, List thumbnails, IIOMetadata metadata)`  
obudowuje obraz, jego miniatury i metadane za pomocą obiektu klasy `IIOImage`.  
 Parametry: `image` obraz,  
`thumbnails` lista obiektów klasy `BufferedImage` lub wartość null,  
`metadata` metadane opisujące obraz lub wartość null.

## 7.11. Operacje na obrazach

W praktyce często pojawia się potrzeba dostosowania wczytanego z pliku obrazu do własnych potrzeb przez modyfikację części jego pikseli, a nawet utworzenia własnego obrazu od podstaw — na przykład w celu prezentacji wyników pomiarów lub obliczeń. Klasa `BufferedImage` umożliwia pełną kontrolę nad pikselami obrazu, a klasy implementujące interfejs `BufferedImageOp` pozwalają wykonywać operacje na obrazach.



JDK 1.0 dysponowało zupełnie innym, znacznie bardziej skomplikowanym szkieletem przetwarzania obrazów, zoptymalizowanym pod kątem przyrostowego tworzenia obrazów ładowanych z serwerów internetowych linia po linii i prezentowanych użytkownikowi stopniowo w miarę napełniania kolejnych danych. Wykorzystanie tego rozwiązania było w praktyce dość trudne. W tej książce nie będziemy zajmować się wspomnianym szkieletem.

### 7.11.1. Dostęp do danych obrazu

W większości przypadków obrazy, na których operują nasze programy, wczytywane są z pliku — zapisanego przez cyfrowy aparat fotograficzny, skaner bądź program graficzny. W bieżącym podrozdziale zajmiemy się tworzeniem obrazów od podstaw — piksel po pikselu.

Aby utworzyć nowy obraz, musimy najpierw skonstruować obiekt klasy `BufferedImage`.

```
image = new BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB);
```

Korzystając z metody `getRaster`, uzyskamy obiekt typu `WritableRaster`, który umożliwi modyfikacje pikseli obrazu.

```
WritableRaster raster = image.getRaster();
```

Metoda `setPixel` tego obiektu pozwala modyfikować pojedynczy piksel. Nie wystarczy jednak w tym celu przypisać mu po prostu odpowiedniej wartości typu `Color`. Musimy wiedzieć, w jaki sposób obraz przechowuje informacje o kolorach, co zależy od jego *typu*. W przypadku typu `TYPE_INT_ARGB` każdy z pikseli opisany jest przez cztery wartości z przedziału od 0 do 255

reprezentujące wartość koloru czerwonego, zielonego, niebieskiego oraz wartość alfa. Przekażemy je metodzie `setPixel` za pomocą tablicy.

```
int[] black = { 0, 0, 0, 255 };
raster.setPixel(i, j, black);
```

W żargonie Java 2D wartości te określa się mianem *wartości próbki* piksela.



Istnieją także wersje metody `setPixel`, których parametrami są tablice typów `float[]` bądź `double[]`. Wartości, które należy umieścić w tych tablicach, *nie są jednak* znormalizowanymi wartościami kolorów z przedziału od 0 do 1.

```
float[] red = { 1.0F, 0.0F, 0.0F, 1.0F };
raster.setPixel(i, j, red); // ŹLE
```

Niezależnie od typu tablicy zawsze musimy dostarczyć w niej wartości z przedziału od 0 do 255.

Z pomocą metody `setPixels` możemy zmodyfikować od razu prostokątny obszar pikseli. Metodzie musimy przekazać pozycję pierwszego z pikseli, szerokość i wysokość obszaru prostokąta oraz tablicę zawierającą wartości próbek dla wszystkich pikseli obszaru. Jeśli obraz jest typu `TYPE_INT_ARGB`, to w tablicy tej umieścimy wartości koloru czerwonego, zielonego, niebieskiego i wartość alfa dla pierwszego piksela, następnie wartości dla drugiego piksela itd.

```
int[] pixels = new int [4 * width * height];
pixels[0] = . . . // wartość koloru czerwonego dla pierwszego piksela
pixels[1] = . . . // wartość koloru zielonego dla pierwszego piksela
pixels[2] = . . . // wartość koloru niebieskiego dla pierwszego piksela
pixels[3] = . . . // wartość alfa dla pierwszego piksela
.
.
.
raster.setPixels(x, y, width, height, pixels);
```

Aby odczytać wartości piksela, korzystamy z metody `getPixel`, której musimy dostarczyć tablicę dla wartości próbki.

```
int[] sample = new int[4];
raster.getPixel(x, y, sample);
Color c = new Color(sample[0], sample[1], sample[2], sample[3]);
```

Z pomocą metody `getPixels` możemy odczytać wartości próbek dla wielu pikseli.

```
raster.getPixels(x, y, width, height, samples);
```

Metod `setPixel` i `getPixel` możemy oczywiście używać także dla innych typów obrazów niż `TYPE_INT_ARGB` pod warunkiem, że znamy sposób reprezentacji wartości pikseli dla danego typu obrazu.

Jeśli natomiast musimy operować na obrazie dowolnego typu, to zadanie jest nieco bardziej skomplikowane. Każdy typ obrazu posiada określony *model kolorów*, który umożliwia przekształcenie wartości próbek na model kolorów RGB.

Metoda `getColorModel` zwraca model kolorów dla danego obrazu.

```
ColorModel model = image.getColorModel();
```



Kolor zapisany przy użyciu modelu RGB może wyglądać różnie w zależności od charakterystyki konkretnego urządzenia. International Color Consortium ([www.color.org](http://www.color.org)) zaleca więc, aby danym o kolorach towarzyszył zawsze *profil ICC* określający sposób odwzorowania kolorów w stosunku do standardowej specyfikacji kolorów 1931 CIE XYZ. Specyfikacja ta została zaproponowana przez międzynarodową organizację CIE (*Commission Internationale de l'Eclairage* — [www.cie.co.at/cie](http://www.cie.co.at/cie)) zajmującą się tworzeniem technicznych standardów związanych z kolorami. Specyfikacja ta określa kolory rozróżniane przez ludzkie oko za pomocą trzech współrzędnych X, Y, Z. (Więcej informacji na ten temat znajdziemy w rozdziale 13. książki Foley, van Damma, Feinera, et al.)

Profile ICC są dość skomplikowane i dlatego zaproponowano prostszy standard sRGB (patrz <http://www.w3.org/Graphics/Color/sRGB.html>) określający odwzorowanie pomiędzy wartościami modelu RGB a wartościami specyfikacji 1931 CIE XYZ, które sprawdza się w przypadku typowych monitorów kolorowych. Java 2D korzysta z tego odwzorowania podczas konwersji pomiędzy RGB i innymi modelami kolorów.

Aby uzyskać informacje o kolorze piksela, wywołujemy metodę `getDataElements` klasy `Raster`. Zwraca ona obiekt zawierający opis wartości koloru piksela w danym modelu kolorów.

```
Object data = raster.getDataElements(x, y, null);
```



Obiekt zwracany przez metodę `getDataElements` jest w rzeczywistości tablicą wartości próbki. Oczywiście przetwarzając go, nie musimy tego wiedzieć, ale wyjaśnia to nazwę wybraną przez projektantów dla tej metody.

Model kolorów umożliwia zamianę wartości próbki zawartych w obiekcie na standardowe wartości ARGB. Metoda `getRGB` zwraca wartość typu `int`, która zawiera wartości koloru czerwonego, zielonego, niebieskiego i wartość alfa upakowane w czterech blokach po osiem bitów każdy. Na podstawie tej wartości możemy utworzyć obiekt klasy `Color`, korzystając z konstruktora `Color(int argb, boolean hasAlpha)`.

```
int argb = model.getRGB(data);
Color color = new Color(argb, true);
```

Jeśli chcemy nadać pikselowi pewien kolor, to musimy wykonać opisane kroki w odwrotnej kolejności. Metoda `getRGB` zwróci wartość typu `int` zawierającą upakowane wartości koloru czerwonego, zielonego, niebieskiego i wartość alfa. Wartość tę musimy przekazać metodzie `getDataElements` klasy `ColorModel`. Zwróci ona obiekt zawierający wartości próbki w danym modelu. Obiekt ten przekażemy następnie metodzie `setDataElements` klasy `WritableRaster`.

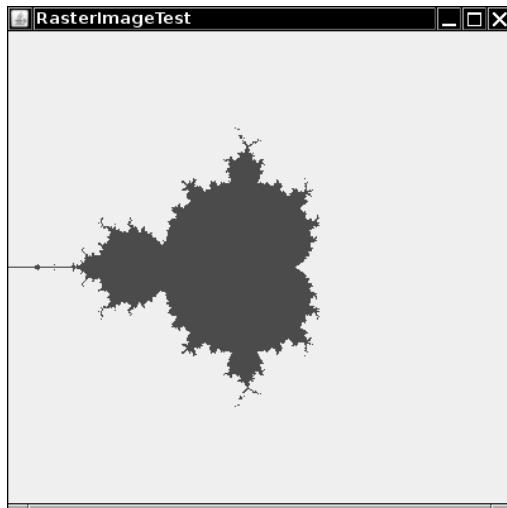
```
int argb = color.getRGB();
Object data = model.getDataElements(argb, null);
raster.setDataElements(x, y, data);
```

Ilustracją omówionych metod tworzenia obrazu z pojedynczych pikseli zbiór Mandelbrota pokazany na rysunku 7.28.

Idea zbioru Mandelbrota opiera się na związaniu z każdym punktem płaszczyzny ciągu liczb. Jeśli ciąg ten posiada skończoną granicę, to zaznaczamy odpowiadający mu punkt. Jeśli dąży do nieskończoności, to związaną z ciągiem punkt płaszczyzny pozostawiamy bez zmian.

A oto sposób konstrukcji najprostszego zbioru Mandelbrota. Dla każdego punktu (a, b) tworzenie ciągu rozpoczęmy od  $(x, y) = (0, 0)$  i następnie wykonujemy iteracje, korzystając z poniższych wzorów:

**Rysunek 7.28.**  
Zbiór Mandelbrota



$$x_{\text{new}} = x^2 - y^2 + a$$

$$y_{\text{new}} = 2*x*y + b$$

Następnie należy sprawdzić, czy uzyskany ciąg posiada granicę, czy dąży do nieskończoności. W praktyce okazuje się, że jeśli  $x$  i  $y$  będą większe od 2, do ciągu będzie dążył do nieskończoności. Zaznaczamy jedynie te piksele, dla których ciąg posiada skończoną granicę. (Formuła tworzenia ciągu liczb wywodzi się z matematyki liczb zespolonych. Nie będziemy jej wyprowadzać, lecz skorzystamy z gotowego wzoru.Więcej informacji na temat fraktali znajdziesz na stronie <http://classes.yale.edu/fractals>).

Listing 7.8 zawiera kompletny tekst źródłowy programu. Program pokazuje między innymi, w jaki sposób należy wykorzystać klasę `ColorModel` do przekształcenia wartości typu `Color` na dane opisujące piksel. Sposób ten jest niezależny od typu obrazu. Możemy sprawdzić jego działanie, zmieniając typ obrazu na przykład na `TYPE_BYTE_GRAY`. Nie wymaga to żadnych innych modyfikacji kodu, ponieważ przekształceniem wartości kolorów na wartości próbek zajmie się automatycznie model kolorów.

**Listing 7.8.** `rasterImage/RasterImageFrame.java`

```
package rasterImage;

import java.awt.*;
import java.awt.image.*;
import javax.swing.*;

/**
 * Ramka wyświetlająca zbiór Mandelbrota.
 */
public class RasterImageFrame extends JFrame
{
    private static final double XMIN = -2;
    private static final double XMAX = 2;
    private static final double YMIN = -2;
```

```

private static final double YMAX = 2;
private static final int MAX_ITERATIONS = 16;
private static final int IMAGE_WIDTH = 400;
private static final int IMAGE_HEIGHT = 400;

public RasterImageFrame()
{
    BufferedImage image = makeMandelbrot(IMAGE_WIDTH, IMAGE_HEIGHT);
    add(new JLabel(new ImageIcon(image)));
    pack();
}

/**
 * Tworzy obraz zbioru Mandelbrota.
 * @param width szerokość
 * @param height wysokość
 * @return obraz
 */
public BufferedImage makeMandelbrot(int width, int height)
{
    BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB);
    WritableRaster raster = image.getRaster();
    ColorModel model = image.getColorModel();

    Color fractalColor = Color.red;
    int argb = fractalColor.getRGB();
    Object colorData = model.getDataElements(argb, null);

    for (int i = 0; i < width; i++)
        for (int j = 0; j < height; j++)
    {
        double a = XMIN + i * (XMAX - XMIN) / width;
        double b = YMIN + j * (YMAX - YMIN) / height;
        if (!escapesToInfinity(a, b)) raster.setDataElements(i, j, colorData);
    }
    return image;
}

private boolean escapesToInfinity(double a, double b)
{
    double x = 0.0;
    double y = 0.0;
    int iterations = 0;
    while (x <= 2 && y <= 2 && iterations < MAX_ITERATIONS)
    {
        double xnew = x * x - y * y + a;
        double ynew = 2 * x * y + b;
        x = xnew;
        y = ynew;
        iterations++;
    }
    return x > 2 || y > 2;
}

```

**API java.awt.image.BufferedImage 1.2**

- `BufferedImage(int width, int height, int imageType)`

tworzy obiekt klasy `BufferedImage`.

*Parametry:* `width, height` szerokość i wysokość obrazu

`imageType` jedna z wartości `TYPE_INT_RGB`, `TYPE_INT_ARGB`,  
`TYPE_BYTE_GRAY`, `TYPE_BYTE_INDEXED`,  
`TYPE USHORT_555_RGB` itd.

- `ColorModel getColorModel()`

zwraca model kolorów dla danego obrazu.

- `WritableRaster getRaster()`

pobiera raster umożliwiający operacje na pikselach obrazu.

**API java.awt.image.Raster 1.2**

- `Object getDataElements(int x, int y, Object data)`

zwraca wartości próbki piksela w postaci tablicy, której typ i liczba elementów zależy od modelu kolorów. Jeśli parametr `data` różny jest od `null`, to metoda przyjmuje, że reprezentuje on tablicę odpowiednią do umieszczenia wartości próbek w danym modelu. W przeciwnym razie tworzona i wypełniana jest nowa tablica. Typ i liczba elementów tablicy zależą od modelu kolorów.

- `int[] getPixel(int x, int y, int[] sampleValues)`
- `float[] getPixel(int x, int y, float[] sampleValues)`
- `double[] getPixel(int x, int y, double[] sampleValues)`
- `int[] getPixels(int x, int y, int w, int h, int[] sampleValues)`
- `float[] getPixels(int x, int y, int w, int h, float[] sampleValues)`
- `double[] getPixels(int x, int y, int w, int h, double[] sampleValues)`

Zwracają wartości próbki dla pojedynczego piksela lub wielu pikseli leżących w obszarze prostokąta. Wartości te zwracane są w postaci tablicy, której długość zależy od modelu kolorów. Jeśli parametr `sampleValues` jest różny od `null`, to przyjmuje się, że reprezentuje on tablicę o odpowiedniej długości do umieszczenia w niej wartości próbek i wypełnia ją. W przeciwnym razie tworzona jest nowa tablica. Metody te są przydatne pod warunkiem, że znamy sposób interpretacji wartości próbek w danym modelu kolorów.

**API java.awt.image.WritableRaster 1.2**

- `void setDataElements(int x, int y, Object data)`

nadaje pikselowi wartość próbki. Argument `data` jest tablicą wypełnioną wartościami próbki w danym modelu kolorów. Typ i liczba elementów tablicy zależą od modelu kolorów.

- void setPixel(int x, int y, int[] sampleValues)
- void setPixel(int x, int y, float[] sampleValues)
- void setPixel(int x, int y, double[] sampleValues)
- void setPixels(int x, int y, int width, int height, int[] sampleValues)
- void setPixels(int x, int y, int width, int height, float[] sampleValues)
- void setPixels(int x, int y, int width, int height, double[] sampleValues)

nadają wartości próbki pojedynczemu pikselowi lub prostokątnemu obszarowi pikseli. Metody te są przydatne pod warunkiem, że znamy sposób interpretacji wartości próbek w danym modelu kolorów.

#### **java.awt.image.ColorModel 1.2**

- int getRGB(Object data)  
zwraca wartość ARGB odpowiadającą wartości próbki przekazanej za pomocą tablicy data. Typ i liczba elementów tablicy zależą od modelu kolorów.
- Object getDataElements(int argb, Object data)  
zwraca wartości próbki dla danej wartości koloru. Jeśli parametr data różny jest od null, to metoda przyjmuje, że reprezentuje on tablicę odpowiednią do umieszczenia wartości próbek w danym modelu i wypełnia ją. W przeciwnym razie tworzona i wypełniana jest nowa tablica. Typ i liczba elementów tablicy zależą od modelu kolorów.

#### **java.awt.Color 1.0**

- Color(int argb, boolean hasAlpha) 1.2  
tworzy kolor o podanej wartości ARGB, gdy parametr hasAlpha ma wartość true lub o podanej wartości RGB, gdy parametr hasAlpha ma wartość false.
- int getRGB()  
zwraca wartość koloru ARGB odpowiadającą danemu kolorowi.

## 7.11.2. Filtrowanie obrazów

W poprzednim podrozdziale pokazaliśmy, w jaki sposób utworzyć obraz od podstaw. Często jednak dysponujemy już gotowym obrazem, który następnie należy poddać obróbce.

Oczywiście możemy w tym celu użyć przedstawionych już metod `getPixel/getDataElements`. Jednak Java 2D dostarcza także gotowych *filtrów* obrazów, które implementują najczęściej spotykane operacje.

Klasy operacji na obrazach implementują interfejs `BufferedImageOp`. Po utworzeniu obiektu takiej klasy wywołujemy jej metodę `filter`, aby wykonać operację na obrazie.

```
BufferedImageOp op = . . . ;
BufferedImage filteredImage
    = new BufferedImage(image.getWidth(), image.getHeight(), image.getType());
op.filter(image, filteredImage);
```

Jedynie niektóre operacje mogą być wykonane bez konieczności utworzenia wynikowego obrazu (czyli przez wywołanie `op.filter(image, image)`).

Interfejs `BufferedImageOp` implementuje pięć następujących klas:

```
AffineTransformOp
RescaleOp
LookupOp
ColorConvertOp
ConvolveOp
```

Klasa `AffineTransformOp` umożliwia afinczne przekształcenie pikseli. Poniżej przykład obrotu obrazu dookoła jego środka.

```
AffineTransform transform
    = AffineTransform.getRotateInstance(Math.toRadians(angle), image.getWidth() /
        2, image.getHeight() / 2);
AffineTransformOp op
    = new AffineTransformOp(transform, interpolation);
op.filter(image, filteredImage);
```

Konstruktorowi klasy `AffineTransformOp` musimy przekazać przekształcenie afinczne oraz strategię *interpolacji*. Interpolacja wykorzystywana jest do tworzenia pikseli obrazu wynikowego, w przypadku gdy nie istnieje jednoznaczne odwzorowanie pikseli obrazu źródłowego na piksele obrazu wynikowego. Sytuacja taka występuje na przykład podczas obrotu obrazu. Dostępne są dwie strategie interpolacji: `AffineTransformOp.TYPE_BILINEAR` oraz `AffineTransformOp.TYPE_NEAREST_NEIGHBOR`. Pierwsza z nich wymaga nieco więcej obliczeń, ale daje lepszy efekt.

Program z listingu 7.9 pozwala wykonywać obrót obrazu o kąt 5 stopni (patrz rysunek 7.29).

**Rysunek 7.29.**  
*Obrót obrazu*



**Listing 7.9.** *imageProcessing/ImageProcessingFrame.java*

```
package imageProcessing;

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.awt.image.*;
import java.io.*;
import javax.imageio.*;
import javax.swing.*;
import javax.swing.filechooser.*;

/**
 * Ramka posiadająca menu umożliwiające załadowanie obrazu z pliku
 * i wybór przekształcenia oraz komponent przedstawiający wynik przekształcenia.
 */
public class ImageProcessingFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 400;

    private BufferedImage image;

    public ImageProcessingFrame()
    {
        setTitle("ImageProcessingTest");
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        add(new JComponent()
        {
            public void paintComponent(Graphics g)
            {
                if (image != null) g.drawImage(image, 0, 0, null);
            }
        });
    }

    JMenu fileMenu = new JMenu("File");
    JMenuItem openItem = new JMenuItem("Open");
    openItem.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            openFile();
        }
    });
    fileMenu.add(openItem);

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

```
JMenu editMenu = new JMenu("Edit");
JMenuItem blurItem = new JMenuItem("Blur");
blurItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        float weight = 1.0f / 9.0f;
        float[] elements = new float[9];
        for (int i = 0; i < 9; i++)
            elements[i] = weight;
        convolve(elements);
    }
});
editMenu.add(blurItem);

JMenuItem sharpenItem = new JMenuItem("Sharpen");
sharpenItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        float[] elements = { 0.0f, -1.0f, 0.0f, -1.0f, 5.0f, -1.0f, 0.0f,
                           -1.0f, 0.0f };
        convolve(elements);
    }
});
editMenu.add(sharpenItem);

JMenuItem brightenItem = new JMenuItem("Brighten");
brightenItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        float a = 1.0f;
        // float b = 20.0f;
        float b = 0;
        RescaleOp op = new RescaleOp(a, b, null);
        filter(op);
    }
});
editMenu.add(brightenItem);

JMenuItem edgeDetectItem = new JMenuItem("Edge detect");
edgeDetectItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        float[] elements = { 0.0f, -1.0f, 0.0f, -1.0f, 4.0f, -1.0f, 0.0f,
                           -1.0f, 0.0f };
        convolve(elements);
    }
});
editMenu.add(edgeDetectItem);

JMenuItem negativeItem = new JMenuItem("Negative");
negativeItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
```

```

    {
        short[] negative = new short[256 * 1];
        for (int i = 0; i < 256; i++)
            negative[i] = (short) (255 - i);
        ShortLookupTable table = new ShortLookupTable(0, negative);
        LookupOp op = new LookupOp(table, null);
        filter(op);
    }
});

editMenu.add(negativeItem);

JMenuItem rotateItem = new JMenuItem("Rotate");
rotateItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        if (image == null) return;
        AffineTransform transform =
            AffineTransform.getRotateInstance(Math.toRadians(5),
                image.getWidth() / 2, image.getHeight() / 2);
        AffineTransformOp op = new AffineTransformOp(transform,
            AffineTransformOp.TYPE_BICUBIC);
        filter(op);
    }
});
editMenu.add(rotateItem);

JMenuBar menuBar = new JMenuBar();
menuBar.add(fileMenu);
menuBar.add(editMenu);
setJMenuBar(menuBar);
}

<**
 * Otwiera plik i ładuje obrazek.
 */
public void openFile()
{
    JFileChooser chooser = new JFileChooser(".");
    chooser.setCurrentDirectory(new File(getClass().getPackage().getName()));
    String[] extensions = ImageIO.getReaderFileSuffixes();
    chooser.setFileFilter(new FileNameExtensionFilter("Image files", extensions));
    int r = chooser.showOpenDialog(this);
    if (r != JFileChooser.APPROVE_OPTION) return;

    try
    {
        Image img = ImageIO.read(chooser.getSelectedFile());
        image = new BufferedImage(img.getWidth(null), img.getHeight(null),
            BufferedImage.TYPE_INT_RGB);
        image.getGraphics().drawImage(img, 0, 0, null);
    }
    catch (IOException e)
    {
        JOptionPane.showMessageDialog(this, e);
    }
    repaint();
}
}

```

```
/**  
 * Stosuje filtr obrazu i odrysowuje obraz.  
 * @param op rodzaj przekształcenia  
 */  
private void filter(BufferedImageOp op)  
{  
    if (image == null) return;  
    image = op.filter(image, null);  
    repaint();  
}  
  
/**  
 * Wykonuje operację splotu i odrysowuje obraz.  
 * @param elements jądro splotu (tablica zawierająca 9 elementów macierzy)  
 */  
private void convolve(float[] elements)  
{  
    Kernel kernel = new Kernel(3, 3, elements);  
    ConvolveOp op = new ConvolveOp(kernel);  
    filter(op);  
}  
}
```

---

Klasa RescaleOp umożliwia wykonanie na pikselach operacji określonej wzorem:

$$x_{\text{nowe}} = a \cdot x + b$$

gdzie  $x$  jest wartością próbki piksela. W wyniku operacji, dla której  $a > 1$ , otrzymujemy rozjaśniony obraz. Obiekt klasy RescaleOp tworzymy, określając parametry skalowania oraz opcjonalne wskazówki. W programie z listingu 7.9 używamy następujących parametrów:

```
float a = 1.1f;  
float b = 20.0f;  
RescaleOp op = new RescaleOp(a, b, null);
```

Można również dostarczyć osobny parametr skalowania dla każdej składowej koloru — patrz omówienie metod klasy RescaleOp na końcu tego podrozdziału.

Klasa LookupOp pozwala określić dowolne odwzorowanie wartości próbek. W tym celu musimy dostarczyć jej tablicy opisującej sposób odwzorowania. Nasz przykładowy program wykorzystuje klasę LookupOp do utworzenia negatywu obrazu zamieniając kolor  $c$  na 255 —  $c$ .

Konstruktor klasy LookupOp musi otrzymać jako parametr obiekt klasy LookupTable oraz opcjonalnie mapę wskazówek. Klasa LookupTable jest klasą abstrakcyjną. Dostępne są dwie jej konkretne klasy pochodne: ByteLookupTable i ShortLookupTable. Ponieważ wartości RGB zapisywane są za pomocą bajtów, powinna nam wystarczyć klasa ByteLookupTable. Jednak ze względu na błąd opisany na stronie [http://bugs.sun.com/bugdatabase/view\\_bug.do?bug\\_id=6183251](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6183251) użyjemy zamiast niej klasy ShortLookupTable. Oto sposób, w jaki tworzymy obiekt LookupOp w programie przykładowym:

```
short negative[] = new short[256];  
for (int i = 0; i < 256; i++) negative[i] = (short) (255 - i);  
ShortLookupTable table = new ShortLookupTable(0, negative);  
LookupOp op = new LookupOp(table, null);
```

Odwzorowywanie wykonywane jest dla wartości każdego z kolorów oddzielnie, lecz nie dla wartości alfa. Można również dostarczyć osobne tablice odwzorowań dla każdej składowej koloru — patrz omówienie klas na końcu podrozdziału.



Klasy `LookupOp` nie możemy zastosować w przypadku obrazów posiadających kolor indeksowany. (W ich przypadku wartości próbki są indeksami palety kolorów).

Klasa `ColorConvertOp` przydatna jest do przekształceń przestrzeni kolorów. Nie będziemy jej tutaj omawiać.

Największych możliwości dostarcza klasa `ConvolveOp`, która umożliwia realizację operacji *splotu*. Nie będziemy wnikać głębiej w jej matematyczne podstawy. Podstawowa idea tej operacji jest prosta. Rozważmy na przykład *filtrowanie* (patrz rysunek 7.30).

**Rysunek 7.30.**

Rozmycie obrazu



Efekt rozmycia uzyskujemy, zastępując każdy piksel wartością *średnią* danego piksela i jego ośmiu sąsiadów. Intuicyjnie rozumiemy sposób działania takiego przekształcenia. Natomiast matematyka pozwala nam je opisać jako splot o następującym *jadrze*:

$$\begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}$$

Jądro splotu jest macierzą przedstawiającą wagę, które zastosowane mają być do sąsiednich wartości. Efektem splotu o powyższym jadrze jest rozmycie obrazu. Natomiast jądro postaci

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

pozwala wykryć krawędzie, na których dokonuje się zmiana koloru (patrz rysunek 7.31). Wykrywanie krawędzi jest ważną techniką stosowaną w przetwarzaniu obrazów fotograficznych.

**Rysunek 7.31.**

Wykrywanie krawędzi



Aby wykonać operację splotu, musimy najpierw utworzyć jej jądro jako obiekt klasy `Kernel`. Dla utworzonego na jego podstawie obiektu klasy `ConvolveOp` wykonujemy jak zwykle metodę `filter`.

```
float[] elements =
{
    0.0f, -1.0f, 0.0f,
    -1.0f, 4.0f, -1.0f,
    0.0f, -1.0f, 0.0f
};
Kernel kernel = new Kernel(3, 3, elements);
ConvolveOp op = new ConvolveOp(kernel);
op.filter(image, filteredImage);
```

Program z listingu 7.9 pozwala załadować obraz zapisany w pliku GIF lub JPEG i wykonać na nim omówione dotąd operacje. Dzięki wykorzystaniu implementacji tych operacji dostępnych w postaci klas Java 2D program jest bardzo prosty.

#### API `java.awt.image.BufferedImageOp 1.2`

- `BufferedImage filter(BufferedImage source, BufferedImage dest)`  
wykonuje operację na obrazie `source`, której wynikiem jest obraz `dest`. Jeśli `dest` posiada wartość `null`, to tworzony jest i zwracany nowy obraz.

#### API `java.awt.image.AffineTransformOp 1.2`

- `AffineTransformOp(AffineTransform t, int interpolationType)`  
tworzy obiekt przekształcenia afinicznego obrazu. Parametr `interpolationType` przyjmuje jedną z wartości `TYPE_BILINEAR` lub `TYPE_NEAREST_NEIGHBOR`.

**API** `java.awt.image.RescaleOp 1.2`

- `RescaleOp(float a, float b, RenderingHints hints)`
- `RescaleOp(float[] as, float[] bs, RenderingHints hints)`

tworzą obiekt klasy `RescaleOp` umożliwiający wykonanie operacji skalowania określonej wzorem  $x_{\text{nowe}} = a*x+b$ . W przypadku zastosowania pierwszego konstruktora podczas skalowania wszystkich składowych (ale nie wartości alfa) zastosowany zostaje ten sam współczynnik. W przypadku drugiego konstruktora dostarczamy osobnej wartości współczynników dla każdej składowej koloru (wtedy wartość alfa nie jest modyfikowana) lub dla każdej składowej koloru i wartości alfa.

**API** `java.awt.image.LookupOp 1.2`

- `LookupOp(LookupTable table, RenderingHints hints)`
- tworzy obiekt odwzorowania wartości próbek obrazu.

**API** `java.awt.image.ByteLookupTable 1.2`

- `ByteLookupTable(int offset, byte[] data)`
- `ByteLookupTable(int offset, byte[][] data)`

tworzą tablicę odwzorowania złożoną z bajtów. Wartość parametru `offset` zostaje odjęta od danych wejściowych przed zastosowaniem wartości `data`. W przypadku zastosowania pierwszego konstruktora podczas skalowania wszystkich składowych (ale nie wartości alfa) zastosowany zostaje ten sam współczynnik. W przypadku drugiego konstruktora dostarczamy osobnej wartości współczynników dla każdej składowej koloru (wtedy wartość alfa nie jest modyfikowana) lub dla każdej składowej koloru i wartości alfa.

**API** `java.awt.image.ShortLookupTable 1.2`

- `ByteLookupTable(int offset, short[] data)`
- `ByteLookupTable(int offset, short[][] data)`

tworzą tablicę odwzorowania złożoną z wartości typu `short`. Wartość parametru `offset` zostaje odjęta od danych wejściowych przed zastosowaniem wartości `data`. W przypadku zastosowania pierwszego konstruktora podczas skalowania wszystkich składowych (ale nie wartości alfa) zastosowany zostaje ten sam współczynnik. W przypadku drugiego konstruktora dostarczamy osobnej wartości współczynników dla każdej składowej koloru (wtedy wartość alfa nie jest modyfikowana) lub dla każdej składowej koloru i wartości alfa.

**API** `java.awt.ConvolveOp 1.2`

- `ConvolveOp(Kernel kernel)`

- ConvolveOp(Kernel kernel, int edgeCondition, RenderingHints hints)

Tworzy obiekt splotu. Warunki brzegowe zostają określone za pomocą stałej EDGE\_NO\_OP lub EDGE\_ZERO\_FILL. Wartości brzegowe wymagają specjalnego traktowania, ponieważ nie posiadają wystarczającej liczby wartości sąsiednich dla wyznaczenia splotu. Wartością domyślną jest EDGE\_ZERO\_FILL.

#### java.awt.image.Kernel 1.2

- Kernel(int width, int height, float[] matrixElements)
- tworzy jądro splotu dla podanej macierzy.

## 7.12. Drukowanie

Początkowo Java Development Kit nie zawierał obsługi operacji drukowania. Drukowanie z appletów nie było w ogóle możliwe, natomiast drukowanie z aplikacji wymagało wykorzystania bibliotek dostarczanych przez innych producentów. JDK 1.1 umożliwiał już tworzenie prostych wydruków o niskiej jakości. Model drukowania dostępny w tej wersji JDK służyć miał przede wszystkim producentom przeglądarek stron internetowych do wydruku appletów pojawiających się na stronach WWW. Nie zyskał sobie jednak sympatii ani producentów przeglądarek, ani pozostałych programistów.

Java SE 1.2 wprowadziła nowy model drukowania zintegrowany z możliwościami graficznymi platformy Java 2D. Java SE 1.4 przyniosła istotne rozszerzenia jego możliwości, takie jak wykrywanie charakterystyki drukarek czy wysyłanie prac do serwerów wydruków.

Omawiając możliwości drukowania na platformie Java, przedstawimy sposób tworzenia podstawowych wydruków, wydruków wielostronowych oraz wykorzystania możliwości graficznych Java 2D do tworzenia podglądu wydruku.

### 7.12.1. Drukowanie grafiki

W podrozdziale tym przedstawimy sposób drukowania grafiki 2D. Oczywiście grafika taka może zawierać także tekst tworzony za pomocą różnych czcionek, a nawet składać się wyłącznie z samego tekstu.

Aby utworzyć wydruk, musimy:

- dostarczyć obiekt implementujący interfejs Printable,
- uruchomić zadanie drukowania.

Interfejs Printable posiada tylko jedną metodę:

```
int print(Graphics g, PageFormat format, int page)
```

Metoda ta jest wywoływana przez podsystem drukowania, gdy chce on uzyskać kolejną sformatowaną stronę do wydruku. Nasza implementacja tej metody musi więc utworzyć obraz

wydruku, korzystając z dostarczonego kontekstu graficznego. Drugi z parametrów metody opisuje rozmiary strony i marginesów, a trzeci parametr określa numer drukowanej strony.

Aby uruchomić zadanie drukowania, korzystamy z klasy PrinterJob. Najpierw musimy wywołać jej metodę statyczną getPrinterJob, która zwróci obiekt tej klasy. Następnie musimy przekazać otrzymanemu obiekowi klasy PrinterJob nasz obiekt implementujący interfejs Printable.

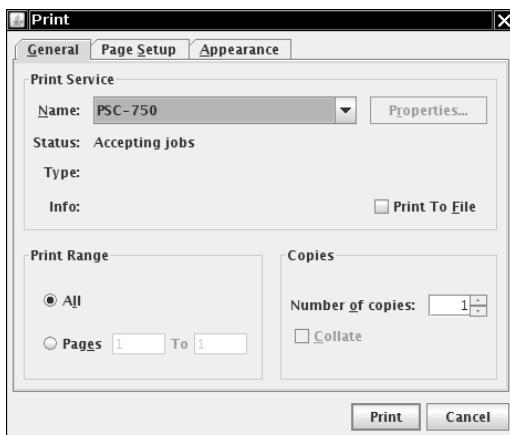
```
Printable canvas = . . . ;
PrinterJob job = PrinterJob.getPrinterJob();
job.setPrintable(canvas);
```



W JDK 1.1 dostępna jest klasa PrintJob wykorzystywana przez przestarzały model drukowania. Nie należy jej mylić z klasą PrinterJob.

Zanim uruchomimy zadanie drukowania, powinniśmy najpierw wywołać metodę printDialog wyświetlającą okno dialogowe drukowania (patrz rysunek 7.32). Umożliwia ono użytkownikowi wybór drukarki (jeśli dostępna jest więcej niż jedna), określenie zakresu drukowanych stron i innych opcji drukowania.

**Rysunek 7.32.**  
Okno dialogowe  
wydruku  
dla dowolnej  
platformy



Parametry wydruku zbieramy za pomocą obiektu implementującego interfejs PrintRequestAttributeSet. Dostępna jest gotowa klasa HashPrintRequestAttributeSet implementująca ten interfejs.

```
HashPrintRequestAttributeSet attributes = new HashPrintRequestAttributeSet();
```

Po dodaniu odpowiednich atrybutów obiekt attributes przekazujemy metodzie printDialog.

Metoda ta zwraca wartość true, jeśli użytkownik zaakceptował wydruk, wybierając przycisk OK lub wartość false — w przeciwnym razie. Jeśli użytkownik zaakceptował operację drukowania, to wywołujemy metodę print klasy PrinterJob uruchamiającą proces drukowania. Metoda print może wyrzucić wyjątek PrinterException. Poniżej przedstawiamy ogólną postać kodu wydruku.

```
if (job.printDialog(attributes))
{
    try
```

```
    {
        job.print(attributes);
    }
    catch (PrinterException exception)
    {
        ...
    }
}
```



 W wersjach wcześniejszych niż JDK 1.4 używano okna dialogowego wydruku dostarczanego przez konkretny system. Jeśli chcemy skorzystać z tego okna w nowszych wersjach JDK, należy wywołać metodę `printDialog` bez parametrów (nie można wtedy zebrać ustawień użytkownika w postaci zbiuru atrybutów).

Podczas drukowania metoda `print` klasy `PrinterJob` wywołuje wielokrotnie metodę `print` obiektu implementującego interfejs `Printable`.

Ponieważ obiekt klasy PrinterJob nie posiada informacji o liczbie stron, które będą wydrukowane, to wywołuje po prostu cyklicznie metodę print. Tak długo, jak zwraca ona wartość Printable.PAGE\_EXISTS, obiekt PrinterJob tworzy kolejne strony wydruku. Swoje działanie kończy, gdy metoda print zwróci wartość Printable.NO SUCH PAGE.



Numery stron, które obiekt klasy `PrinterJob` przekazuje metodzie `print`, rozpoczynają się od 0.

Obiekt klasy `PrinterJob` nie dysponuje więc informacją o liczbie drukowanych stron, dopóki wydruk nie zakończy się. Z tego powodu okno dialogowe drukowania nie potrafi wyświetlić odpowiedniego zakresu drukowanych stron, a jedynie zakres od 1 do 1. Pokażemy później, w jaki sposób można to poprawić, dostarczając obiekty klasy `PrinterJob` obiektu klasy `Book`.

Jak już wspomnieliśmy w procesie drukowania metoda print obiektu typu Printable wywoływana jest wielokrotnie. Obiekt klasy PrinterJob może wywołać ją wielokrotnie dla tej samej strony wydruku. Dlatego też metoda print nie powinna sama zliczać drukowanych stron, a jedynie bazować na przekazywanym jej parametrze określającym numer strony. Metoda print wywoływaną jest wielokrotnie dla tej samej strony, zwłaszcza w przypadku wydruku za pomocą drukarek igłowych lub atramentowych, które stopniowo tworzą wydruk linia po linii. Także w przypadku drukarek laserowych, które posiadają techniczną możliwość wydrukowania od razu całości strony, metoda print może być wywoływana wielokrotnie, umożliwiając w ten sposób obiektowi klasy PrinterJob lepsze zarządzanie buforem wydruku.

Jeśli obiekt klasy `PrinterJob` chce wydrukować kolejny fragment strony, to określa odpowiednio granicę obszaru przycięcia kontekstu graficznego i wywołuje metodę `print`. Utworzony przez nią wydruk zostanie automatycznie przycięty do obszaru odpowiadającego bieżącemu fragmentowi strony. Metoda `print` nie musi być nawet tego świadoma z jednym wyjątkiem: *nie powinna ona zmieniać obszaru przycięcia*.

Parametr klasy `PageFormat` przekazuje metodzie `print` informacje o formacie drukowanej strony. Metody `getWidth` i `getHeight` zwracają rozmiar strony w *punktach*. Jeden punkt odpowiada  $\frac{1}{72}$  cala. (Jeden cal to 25,4 milimetra). Strona formatu *A4* posiada w przybliżeniu wymiary 595 na 842 punkty, a amerykańskiego formatu *letter* 612 na 792 punkty.



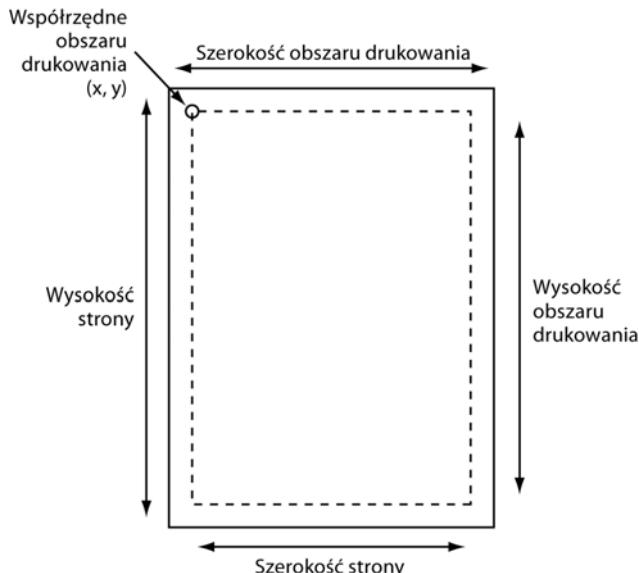
Kontekst graficzny otrzymywany przez metodę `print` posiada obszar przycięcia odpowiadający marginesom strony. Jeśli metoda `print` zmieni obszar przycięcia, to może uzyskać możliwość drukowania także w obszarze marginesów. Metoda `print` powinna jednak przestrzegać obszaru przycięcia przekazanego jej kontekstu graficznego. Operując na obszarze przycięcia, powinna korzystać z metody `clip` zamiast `setClip`. Jeśli z konkretnych powodów musi ona usunąć przekazany jej obszar przycięcia, to powinna go najpierw zachować, korzystając z metody `getClip` i odtworzyć po zakończeniu operacji z własnym obszarem przycięcia.

Punkty stosowane są nie tylko jako jednostka wymiarów strony i marginesów, ale także jako domyślna jednostka dla wszystkich kontekstów graficznych związanych z drukowaniem. Można sprawdzić to za pomocą przykładowego programu zamieszczonego na końcu podrozdziału. Drukuje on dwa wiersze tekstu w odległości 72 jednostek od siebie. Jeśli zmierzmy odległość między liniami bazowymi obu wierszy, to okaże się, że wynosi ona dokładnie 1 cal czyli 25,4 milimetra.

Metody `getWidth` i `getHeight` pozwalają określić wymiary całej strony. Jednak wydruk z reguły nie odbywa się na całym obszarze strony, ponieważ ograniczony jest marginesami. Nawet jeśli zrezygnujemy z marginesów, to i tak dla większości drukarek pozostałą pewne niewielkie obszary strony, na których nie możemy drukować, co związane jest z koniecznością uchwytcenia arkusza papieru przez mechanizm przesuwu papieru drukarki.

Metody `getImageableWidth` i `getImageableHeight` umożliwiają uzyskanie rozmiarów obszaru, na którym możemy drukować. Ponieważ marginesy nie muszą być symetryczne, to istotne są też współrzędne lewego górnego wierzchołka tego obszaru (patrz rysunek 7.33), które możemy uzyskać, korzystając z metod `getImageableX` i `getImageableY`.

**Rysunek 7.33.**  
Rozmiary strony



Jeśli chcemy umożliwić użytkownikowi zmianę rozmiarów marginesów lub orientacji strony, to wywołujemy metodę `pageDialog` klasy `PrinterJob`:

```
PageFormat format = job.pageDialog(attributes);
```



Kontekst graficzny przekazywany metodzie `print` posiada obszar przycięcia wyznaczony przez marginesy strony. Jednak układ współrzędnych strony nadal ma swój początek w lewym górnym narożniku arkusza papieru. Wygodnie więc dokonać przekształcenia układu współrzędnych, tak by jego początek znajdował się w lewym górnym narożniku obszaru strony, na którym możemy drukować. W tym celu na początku metody `print` najlepiej wykonać poniższe wywołanie:

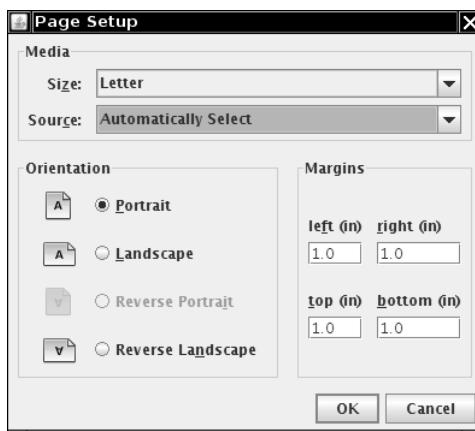
```
g.translate(pf.getImageableX(), pf.getImageableY());
```



Jedna z zakładek okna dialogowego wydruku także umożliwia określenie parametrów strony (rysunek 7.34). Metoda `pageDialog` zwraca obiekt `PageFormat` zawierający atrybuty strony określone przez użytkownika.

**Rysunek 7.34.**

Okno dialogowe ustawień strony dla dowolnej platformy



Program przedstawiony na listingach 7.10 i 7.11 tworzy ten sam rysunek na ekranie i na wydruku. Klasa pochodna klasy `JPanel` implementuje interfejs `Printable`. Jej metody `paintComponent` oraz `Print` korzystają z jednej i tej samej metody tworzenia rysunku.

```
class PrintPanel extends JPanel implements Printable
{
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D)g;
        drawPage(g2);
    }

    public int print(Graphics g, PageFormat pf, int page)
        throws PrinterException
    {
        if (page >= 1) return Printable.NO_SUCH_PAGE;
        Graphics2D g2 = (Graphics2D)g;
        g2.translate(pf.getImageableX(), pf.getImageableY());
        drawPage(g2);
        return Printable.PAGE_EXISTS;
    }

    public void drawPage(Graphics2D g2)
```

```
{
    // kod tworzenia rysunku tak na ekranie, jak i na wydruku
    ...
}
```

...

**Listing 7.10.** print/PrintTestFrame.java

```
package print;

import java.awt.*;
import java.awt.event.*;
import java.awt.print.*;
import javax.print.attribute.*;
import javax.swing.*;

/**
 * Ramka zawierająca panel z grafiką 2D
 * i przyciski umożliwiające wydruk grafiki
 * oraz określenie formatu strony.
 */
public class PrintTestFrame extends JFrame
{
    private PrintComponent canvas;
    private PrintRequestAttributeSet attributes;

    public PrintTestFrame()
    {
        canvas = new PrintComponent();
        add(canvas, BorderLayout.CENTER);

        attributes = new HashPrintRequestAttributeSet();

        JPanel buttonPanel = new JPanel();
        JButton printButton = new JButton("Print");
        buttonPanel.add(printButton);
        printButton.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                try
                {
                    PrinterJob job = PrinterJob.getPrinterJob();
                    job.setPrintable(canvas);
                    if (job.printDialog(attributes)) job.print(attributes);
                }
                catch (PrinterException e)
                {
                    JOptionPane.showMessageDialog(PrintTestFrame.this, e);
                }
            }
        });
        buttonPanel.add(pageSetupButton);
        pageSetupButton.addActionListener(new ActionListener()
        {

```

```

        public void actionPerformed(ActionEvent event)
    {
        PrinterJob job = PrinterJob.getPrinterJob();
        job.pageDialog(attributes);
    }
});

add(buttonPanel, BorderLayout.NORTH);
pack();
}
}

```

**Listing 7.11.** print/PrintComponent.java

```

package print;

import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.awt.print.*;
import javax.swing.*;

/**
 * Panel tworzący grafikę 2D na potrzeby prezentacji na ekranie
 * i wydruku.
 */
public class PrintComponent extends JComponent implements Printable
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 300;

    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;
        drawPage(g2);
    }

    public int print(Graphics g, PageFormat pf, int page) throws PrinterException
    {
        if (page >= 1) return Printable.NO_SUCH_PAGE;
        Graphics2D g2 = (Graphics2D) g;
        g2.translate(pf.getImageableX(), pf.getImageableY());
        g2.draw(new Rectangle2D.Double(0, 0, pf.getImageableWidth(),
            pf.getImageableHeight()));

        drawPage(g2);
        return Printable.PAGE_EXISTS;
    }

    /**
     * Metoda tworząca obraz strony, wykorzystującą kontekst
     * graficzny ekranu oraz drukarki.
     * @param g2 kontekst graficzny
     */
    public void drawPage(Graphics2D g2)
    {
        FontRenderContext context = g2.getFontRenderContext();

```

```

Font f = new Font("Serif", Font.PLAIN, 72);
GeneralPath clipShape = new GeneralPath();

TextLayout layout = new TextLayout("Hello", f, context);
AffineTransform transform = AffineTransform.getTranslateInstance(0, 72);
Shape outline = layout.getOutline(transform);
clipShape.append(outline, false);

layout = new TextLayout("World", f, context);
transform = AffineTransform.getTranslateInstance(0, 144);
outline = layout.getOutline(transform);
clipShape.append(outline, false);

g2.draw(clipShape);
g2.clip(clipShape);

final int NLINES = 50;
Point2D p = new Point2D.Double(0, 0);
for (int i = 0; i < NLINES; i++)
{
    double x = (2 * getWidth() * i) / NLINES;
    double y = (2 * getHeight() * (NLINES - 1 - i)) / NLINES;
    Point2D q = new Point2D.Double(x, y);
    g2.draw(new Line2D.Double(p, q));
}
}

public Dimension getPreferredSize() { return new Dimension(DEFAULT_WIDTH,
DEFAULT_HEIGHT); }
}

```

Program wyświetla i drukuje obrazek pokazany na rysunku 7.20, czyli zbiór linii przycięty obrzeżem tekstu „Hello, World”.

Wybranie przycisku *Print* umożliwia wydrukowanie rysunku, a przycisku *Page Setup* — określenie parametrów strony. Listing 7.10 zawiera pełen tekst źródłowy programu.



Aby uzyskać okno dialogowe ustawień strony właściwe danemu systemowi, przekazujemy domyślny obiekt klasy `PageFormat` metodzie `pageDialog`. Metoda ta klonuje uzyskany obiekt, modyfikuje go zgodnie z wyborem dokonanym przez użytkownika i zwraca klon obiektu.

```

PageFormat defaultFormat = printJob.defaultPage();
PageFormat selectedFormat = printJob.pageDialog(defaultFormat);

```

### `java.awt.print.Printable 1.2`

■ `int print(Graphics g, PageFormat format, int pageNumber)`

tworzy stronę i zwraca wartość `PAGE_EXISTS` lub `NO_SUCH_PAGE`.

*Parametry:* `g` kontekst graficzny,  
`format` format tworzonej strony,  
`numer` tworzonej strony.

**API** java.awt.print.PrinterJob 1.2

- static PrinterJob getPrinterJob()  
zwraca obiekt klasy PrinterJob.
- PageFormat defaultPage()  
zwraca domyślny format strony dla danej drukarki.
- boolean printDialog(PrintRequestAttributeSet attributes)
- boolean printDialog()

Wyświetlają okno dialogowe wydruku umożliwiające użytkownikowi wybór drukowanych stron i opcji drukowania. Pierwsza z metod wyświetla okno niezależne od platformy, na której pracuje aplikacja, natomiast druga — okno specyficzne dla danej platformy. Pierwsza z metod modyfikuje obiekt attributes, tak by odzwierciedlała wybór dokonany przez użytkownika. Obie metody zwracają wartość true, jeśli użytkownik zaakceptował wydruk.

- PageFormat pageDialog(PrintRequestAttributeSet attributes)
- PageFormat pageDialog()

Wyświetlają okno dialogowe ustawień strony. Pierwsza z nich wyświetla okno niezależne od platformy, natomiast druga — okno specyficzne dla danej platformy. Obie metody zwracają obiekt klasy PageFormat określający format wybrany przez użytkownika. Pierwsza z metod modyfikuje obiekt attributes, tak by odzwierciedlała wybór dokonany przez użytkownika. Druga z metod nie modyfikuje obiektu defaults.

- void setPrintable(Printable p)
- void setPrintable(Printable p, PageFormat format)

Okręślają obiekt typu Printable wykorzystywany przez dany obiekt klasy PrinterJob i opcjonalnie format strony.

- void print()
- void print(PrintRequestAttributeSet attributes)

wywołuje cyklicznie metodę print obiektu typu Printable i wysyła uzyskane obrazy stron na drukarkę tak długo, dopóki kolejna strona nie jest dostępna.

**API** java.awt.print.PageFormat 1.2

- double getWidth()
  - double getHeight()
- Zwracają szerokość i wysokość strony.
- double getImageableWidth()
  - double getImageableHeight()

Zwracają szerokość i wysokość obszaru strony, na którym możliwe jest drukowanie.

- double getImageableX()
- double getImageableY()

Zwracają współrzędne lewego górnego wierzchołka obszaru strony, na którym możliwe jest drukowanie.

- int getOrientation()

zwraca jedną z wartości PORTRAIT, LANDSCAPE, REVERSE\_LANDSCAPE. Programista nie musi zajmować się opisaniem orientacji strony, ponieważ jest ona określana automatycznie dla formatu strony i kontekstu graficznego drukowania.

## 7.12.2. Drukowanie wielu stron

W praktyce obiektowi klasy PrinterJob nie przekazujemy zwykle obiektu implementującego jedynie interfejs Printable, a wykorzystujemy raczej obiekt implementujący interfejs Pageable. Java dostarcza klasę o nazwie Book implementującą interfejs Pageable. Obiekt klasy Book składa się z rozdziałów, z których każdy implementuje interfejs Printable. Obiekt klasy Book tworzymy, dodając do niego kolejne rozdziały i liczniki ich stron.

```
Book book = new Book();
Printable coverPage = . . . ;
Printable bodyPages = . . . ;
book.append(coverPage, pageFormat); //dodaje pierwszą stronę
book.append(bodyPages, pageCount);
```

Aby przekazać obiektowi klasy PrinterJob obiekt klasy Book, korzystamy z metody setPageable.

```
printJob.setPageable(book);
```

W ten sposób obiekt klasy PrinterJob zna zawsze liczbę stron, które będą wydrukowane. Dzięki temu okno dialogowe wydruku może zaprezentować użytkownikowi właściwy zakres drukowanych stron.



Gdy obiekt klasy PrinterJob wywołuje metodę print rozdziału reprezentowanego przez obiekt typu Printable, to przekazuje jej numer strony w obrębie całego wydruku, a nie danego rozdziału. Jest to poważny problem, ponieważ jeśli metoda print chce wykorzystać przekazany jej numer strony, to musi znać także wartości liczników stron poprzednich rozdziałów.

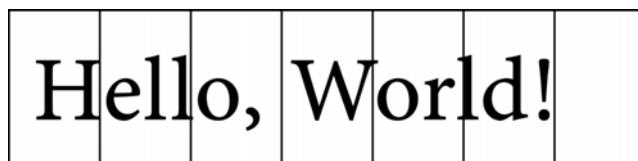
Z punktu widzenia programisty największy problem związany z wykorzystaniem obiektów klasy Book polega na określeniu liczby stron dla każdego z rozdziałów. Klasa implementująca interfejs Printable dysponować musi w tym celu *algorytmem rozmieszczenia* tworzonego materiału na kolejnych stronach. Algorytm ten musi zostać wykonany przed rozpoczęciem drukowania w celu ustalenia początku kolejnych stron i ich liczby. Informację tę warto zachować i wykorzystać w procesie tworzenia wydruku.

Pamiętać przy tym należy, że jeśli użytkownik zmieni format strony, to algorytm musi zostać wykonany od nowa, nawet jeśli drukowana informacja nie uległa zmianie.

Program z listingu 7.13 pokazuje sposób tworzenia wydruku wielostronicowego. Drukuje on tekst komunikatu bardzo dużą czcionką, tak że zajmuje on kilka stron (patrz rysunek 7.35). Możemy obciąć marginesy wydrukowanych stron i skleić je razem w transparent.

**Rysunek 7.35.**

*Transparent*



Metoda `layoutPages` klasy `Banner` wyznacza rozmieszczenie tekstu na stronach, korzystając z czcionki o rozmiarze 72 punktów. Obliczamy wysokość tekstu i porównujemy ją z wysokością obszaru strony, na którym można drukować. Z porównania tego uzyskujemy współczynnik przeskalowania tekstu, który wykorzystujemy do powiększenia tekstu podczas wydruku.



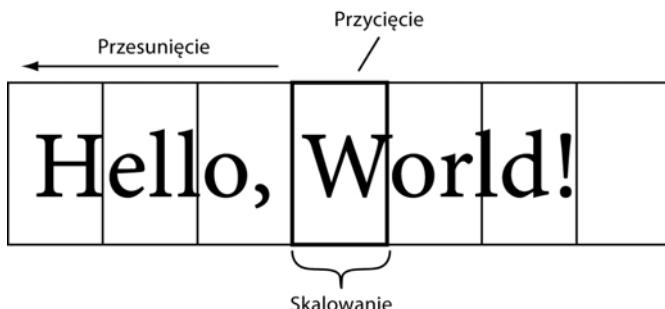
Aby precyjnie wyznaczyć rozmieszczenie materiału na stronach, z reguły potrzebne są informacje, które możemy uzyskać z kontekstu graficznego drukowania. Jednak dostęp do niego nie jest możliwy przed rozpoczęciem drukowania. Nasz przykładowy program wyznacza sposób rozmieszczenia tekstu, korzystając z ekranowego kontekstu graficznego w nadziei, że rozmiary czcionek ekranowych i na wydruku nie będą się różnić.

Metoda `getPageCount` klasy `Banner` wywołuje najpierw metodę wyznaczającą rozmieszczenie wydruku. Następnie przeskalałatwko szerokość łańcucha znaków i dzieli ją przez szerokość zadrukowywanego obszaru strony. Uzyskany wynik po zaokrągleniu w góre jest liczbą stron wydruku.

Drukowanie transparentu wydawać się może kłopotliwe ze względu na to, że poszczególne znaki mogą być dzielone pomiędzy strony. Jednak gdy korzystamy z Java 2D, problem ten nie występuje. Kiedy obiekt klasy `PrinterJob` żąda utworzenia wydruku kolejnej strony, posługujemy się po prostu metodą `translate` klasy `Graphics2D`, aby przesunąć lewy górny wierzchołek łańcucha znaków w lewo. Następnie określamy obszar przycięcia jako obszar bieżącej strony (patrz rysunek 7.36) i skalujemy kontekst graficzny za pomocą współczynnika uzyskanego przez metodę wyznaczającą rozmieszczenie tekstu.

**Rysunek 7.36.**

*Drukowanie strony transparentu*

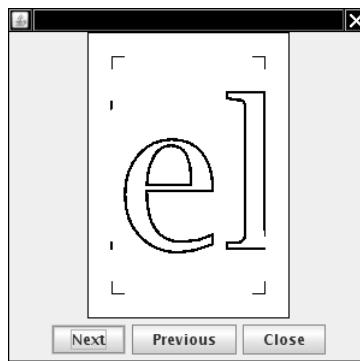


Przykład ten stanowi doskonałą ilustrację praktycznej przydatności przekształceń. Dzięki ich wykorzystaniu do umieszczania rysunku w żądanym położeniu kod programu pozostaje prosty i przejrzysty. Operacja przycięcia umożliwia łatwe usunięcie fragmentów rysunku, które nie mieścią się na danej stronie. W następnym podrozdziale przedstawimy kolejny przykład wykorzystania przekształceń do tworzenia podglądu wydruku.

### 7.12.3. Podgląd wydruku

Większość profesjonalnych aplikacji udostępnia mechanizm podglądu wydruku, który pozwala obejrzeć jego strony przed ich wydrukowaniem i uniknąć marnowania papieru, jeśli efekt nie jest zadowalający. Klasy języka Java nie dostarczają wprawdzie gotowego okna dialogowego dla podglądu wydruku, ale jego zaimplementowanie okazuje się dość proste (patrz rysunek 7.37). Program z listingu 7.14 pokazuje, w jaki sposób można to osiągnąć. Co ważne, zdefiniowana w nim klasa PrintPreviewDialog jest zupełnie ogólna i może być wykorzystywana do podglądu dowolnych wydruków.

**Rysunek 7.37.**  
Okno dialogowe  
podglądu wydruku



**Listing 7.12.** book/BookTestFrame.java

```
package book;

import java.awt.*;
import java.awt.event.*;
import java.awt.print.*;
import javax.print.attribute.*;
import javax.swing.*;

/**
 * Ramka zawierająca pole tekstowe umożliwiające wprowadzenie
 * tekstu transparentu oraz przyciski wydruku, formatu strony i podglądu wydruku.
 */
public class BookTestFrame extends JFrame
{
    private JTextField text;
    private PageFormat pageFormat;
    private PrintRequestAttributeSet attributes;

    public BookTestFrame()
    {
        text = new JTextField();
        add(text, BorderLayout.NORTH);

        attributes = new HashPrintRequestAttributeSet();

        JPanel buttonPanel = new JPanel();
        JButton printButton = new JButton("Print");
        buttonPanel.add(printButton);
    }
}
```

```
buttonPanel.add(printButton);
printButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        try
        {
            PrinterJob job = PrinterJob.getPrinterJob();
            job.setPageable(makeBook());
            if (job.printDialog(attributes))
            {
                job.print(attributes);
            }
        }
        catch (PrinterException e)
        {
            JOptionPane.showMessageDialog(BookTestFrame.this, e);
        }
    }
}):
JButton pageSetupButton = new JButton("Page setup");
buttonPanel.add(pageSetupButton);
pageSetupButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        PrinterJob job = PrinterJob.getPrinterJob();
        pageFormat = job.pageDialog(attributes);
    }
}):
JButton printPreviewButton = new JButton("Print preview");
buttonPanel.add(printPreviewButton);
printPreviewButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        PrintPreviewDialog dialog = new PrintPreviewDialog(makeBook());
        dialog.setVisible(true);
    }
}):
add(buttonPanel, BorderLayout.SOUTH);
pack();
}

/**
 * Tworzy obiekt klasy Book zawierający stronę okładki
 * i kolejne strony transparentu.
 */
public Book makeBook()
{
    if (pageFormat == null)
    {
        PrinterJob job = PrinterJob.getPrinterJob();
        pageFormat = job.defaultPage();
    }
}
```

```

        Book book = new Book();
        String message = text.getText();
        Banner banner = new Banner(message);
        int pageCount = banner.getPageCount((Graphics2D) getGraphics(), pageFormat);
        book.append(new CoverPage(message + " (" + pageCount + " pages)", pageFormat));
        book.append(banner, pageFormat, pageCount);
        return book;
    }
}

```

**Listing 7.13.** book/Banner.java

```

package book;

import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.awt.print.*;

/**
 * Klasa reprezentująca transparent drukowany na wielu stronach.
 */
public class Banner implements Printable
{
    private String message;
    private double scale;

    /**
     * Tworzy obiekt reprezentujący transparent.
     * @param m tekst transparentu
     */
    public Banner(String m)
    {
        message = m;
    }

    /**
     * Zwraca liczbę stron danego rozdziału.
     * @param g2 kontekst graficzny
     * @param pf format strony
     * @return liczba stron
     */
    public int getPageCount(Graphics2D g2, PageFormat pf)
    {
        if (message.equals("")) return 0;
        FontRenderContext context = g2.getFontRenderContext();
        Font f = new Font("Serif", Font.PLAIN, 72);
        Rectangle2D bounds = f.getStringBounds(message, context);
        scale = pf.getImageableHeight() / bounds.getHeight();
        double width = scale * bounds.getWidth();
        int pages = (int) Math.ceil(width / pf.getImageableWidth());
        return pages;
    }

    public int print(Graphics g, PageFormat pf, int page) throws PrinterException
    {
        Graphics2D g2 = (Graphics2D) g;
        if (page > getPageCount(g2, pf)) return Printable.NO_SUCH_PAGE;
    }
}

```

```

g2.translate(pf.getImageableX(), pf.getImageableY());

drawPage(g2, pf, page);
return Printable.PAGE_EXISTS;
}

public void drawPage(Graphics2D g2, PageFormat pf, int page)
{
    if (message.equals("")) return;
    page--; //uwzględnia okładkę

    drawCropMarks(g2, pf);
    g2.clip(new Rectangle2D.Double(0, 0, pf.getImageableWidth()-
        pf.getImageableHeight()));
    g2.translate(-page * pf.getImageableWidth(), 0);
    g2.scale(scale, scale);
    FontRenderContext context = g2.getFontRenderContext();
    Font f = new Font("Serif", Font.PLAIN, 72);
    TextLayout layout = new TextLayout(message, f, context);
    AffineTransform transform = AffineTransform.getTranslateInstance(0,
        -layout.getAscent());
    Shape outline = layout.getOutline(transform);
    g2.draw(outline);
}

/**
 * Rysuje znaki wyznaczające obszar drukowania
 * w odległości ½ cala od narożników strony.
 * @param g2 kontekst graficzny
 * @param pfformat strony
 */
public void drawCropMarks(Graphics2D g2, PageFormat pf)
{
    final double C = 36; //znak obszaru drukowania = 1/2 cala
    double w = pf.getImageableWidth();
    double h = pf.getImageableHeight();
    g2.draw(new Line2D.Double(0, 0, 0, C));
    g2.draw(new Line2D.Double(0, 0, C, 0));
    g2.draw(new Line2D.Double(w, 0, w, C));
    g2.draw(new Line2D.Double(w, 0, w - C, 0));
    g2.draw(new Line2D.Double(0, h, 0, h - C));
    g2.draw(new Line2D.Double(0, h, C, h));
    g2.draw(new Line2D.Double(w, h, w, h - C));
    g2.draw(new Line2D.Double(w, h, w - C, h));
}
}

/**
 * Klasa drukująca stronę okładki zawierającą tytuł.
 */
public class CoverPage implements Printable
{
    private String title;

    /**
     * Tworzy stronę okładki.
     * @param t tytuł
     */

```

```

public CoverPage(String t)
{
    title = t;
}

public int print(Graphics g, PageFormat pf, int page) throws PrinterException
{
    if (page >= 1) return Printable.NO_SUCH_PAGE;
    Graphics2D g2 = (Graphics2D) g;
    g2.setPaint(Color.black);
    g2.translate(pf.getImageableX(), pf.getImageableY());
    FontRenderContext context = g2.getFontRenderContext();
    Font f = g2.getFont();
    TextLayout layout = new TextLayout(title, f, context);
    float ascent = layout.getAscent();
    g2.drawString(title, 0, ascent);
    return Printable.PAGE_EXISTS;
}
}

```

**Listing 7.14.** book/PrintPreviewDialog.java

```

package book;

import java.awt.*;
import java.awt.event.*;
import java.awt.font.*;
import java.awt.print.*;
import javax.swing.*;

/**
 * Klasa implementująca uniwersalne okno dialogowe podglądu wydruku.
 */
class PrintPreviewDialog extends JDialog
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 300;

    private PrintPreviewCanvas canvas;

    /**
     * Tworzy okno dialogowe podglądu wydruku.
     * @param p obiekt typu Printable
     * @param pf format strony
     * @param pages liczba stron obiektu p
     */
    public PrintPreviewDialog(Printable p, PageFormat pf, int pages)
    {
        Book book = new Book();
        book.append(p, pf, pages);
        layoutUI(book);
    }

    /**
     * Tworzy okno podglądu wydruku.
     * @param b obiekt klasy Book
     */
    public PrintPreviewDialog(Book b)

```

```
{  
    layoutUI(b);  
}  
  
/**  
 * Rozmieszcza komponenty okna dialogowego.  
 * @param book obiekt klasy Book, którego podgląd będzie prezentowany  
 */  
public void layoutUI(Book book)  
{  
    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);  
  
    canvas = new PrintPreviewCanvas(book);  
    add(canvas, BorderLayout.CENTER);  
  
    JPanel buttonPanel = new JPanel();  
  
    JButton nextButton = new JButton("Next");  
    buttonPanel.add(nextButton);  
    nextButton.addActionListener(new ActionListener()  
    {  
        public void actionPerformed(ActionEvent event)  
        {  
            canvas.flipPage(1);  
        }  
    });  
  
    JButton previousButton = new JButton("Previous");  
    buttonPanel.add(previousButton);  
    previousButton.addActionListener(new ActionListener()  
    {  
        public void actionPerformed(ActionEvent event)  
        {  
            canvas.flipPage(-1);  
        }  
    });  
  
    JButton closeButton = new JButton("Close");  
    buttonPanel.add(closeButton);  
    closeButton.addActionListener(new ActionListener()  
    {  
        public void actionPerformed(ActionEvent event)  
        {  
            setVisible(false);  
        }  
    });  
  
    add(buttonPanel, BorderLayout.SOUTH);  
}
```

---

**Listing 7.15. book/PrintPreviewCanvas.java**

---

```
package book;  
  
import java.awt.*;  
import java.awt.geom.*;
```

```

import java.awt.print.*;
import javax.swing.*;

/**
 * Komponent prezentacji podglądu wydruku.
 */
class PrintPreviewCanvas extends JComponent
{
    private Book book;
    private int currentPage;

    /**
     * Tworzy obiekt prezentacji podglądu wydruku.
     * @param b obiekt klasy Book, którego podgląd będzie prezentowany
     */
    public PrintPreviewCanvas(Book b)
    {
        book = b;
        currentPage = 0;
    }

    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;
        PageFormat pageFormat = book.getPageFormat(currentPage);

        double xoff; // przesunięcie wzdłuż osi x początku strony w oknie
        double yoff; // przesunięcie wzdłuż osi y początku strony w oknie
        double scale; // współczynnik przeskalowania strony w oknie
        double px = pageFormat.getWidth();
        double py = pageFormat.getHeight();
        double sx = getWidth() - 1;
        double sy = getHeight() - 1;
        if (px / py < sx / sy) // centruje w poziomie
        {
            scale = sy / py;
            xoff = 0.5 * (sx - scale * px);
            yoff = 0;
        }
        else
        // centruje w pionie
        {
            scale = sx / px;
            xoff = 0;
            yoff = 0.5 * (sy - scale * py);
        }
        g2.translate((float) xoff, (float) yoff);
        g2.scale((float) scale, (float) scale);

        // rysuje obraz strony (ignorując marginesy)
        Rectangle2D page = new Rectangle2D.Double(0, 0, px, py);
        g2.setPaint(Color.white);
        g2.fill(page);
        g2.setPaint(Color.black);
        g2.draw(page);

        Printable printable = book.getPrintable(currentPage);
        try

```

```

    {
        printable.print(g2, pageFormat, currentPage);
    }
    catch (PrinterException e)
    {
        g2.draw(new Line2D.Double(0, 0, px, py));
        g2.draw(new Line2D.Double(px, 0, 0, py));
    }
}

/**
 * Przesuwa się o zadaną liczbę stron.
 * @param liczba stron. Wartość ujemna oznacza kierunek wstecz.
 */
public void flipPage(int by)
{
    int nextPage = currentPage + by;
    if (0 <= nextPage && nextPage < book.getNumberOfPages())
    {
        currentPage = nextPage;
        repaint();
    }
}
}

```

Aby utworzyć obiekt klasy PrintPreviewDialog, należy dostarczyć obiekt typu Printable lub klasy Book wraz z obiektem klasy PageFormat. Okno dialogowe klasy PrintPreviewDialog zawiera komponent klasy PrintPreviewCanvas (patrz listing 7.15). Gdy użytkownik wybiera przyciski podglądu poprzedniej lub następnej strony, to metoda paintComponent wywołuje metodę print obiektu Printable dla żądanej strony.

Zwykle metoda print tworzy rysunek strony, korzystając z kontekstu graficznego drukarki. Jednak tym razem dostarczamy jej kontekst graficzny ekranu przeskalowany, tak by obraz strony znalazł się w prostokącie okna dialogowego.

```

float xoff = . . . ; // współrzędna x lewego górnego wierzchołka strony
float yoff = . . . ; // współrzędna y lewego górnego wierzchołka strony
float scale = . . . ; // współczynnik skalowania strony do okna dialogowego
g2.translate(xoff, yoff);
g2.scale(scale, scale);
Printable printable = book.getPrintable(currentPage);
printable.print(g2, pageFormat, currentPage);

```

W ten sposób metoda print nie wie nawet, czy tworzy stronę wydruku, czy też jej podgląd na ekranie. Jest to kolejny przykład przemyślanej konstrukcji modelu tworzenia obrazów zastosowanej przez projektantów Java 2D.

Listing 7.12 zawiera kod programu drukującego transparent wyposażonego w okno podglądu wydruku. Umożliwia on wpisanie dowolnej treści transparentu, następnie podejrzenie jego rozkładu na stronach i na tej podstawie zdecydowanie o jego wydruku.

### **java.awt.print.PrinterJob 1.2**

■ **void setPageable(Pageable p)**

instaluje drukowany obiekt typu Pageable (na przykład klasy Book).

**API** `java.awt.print.Book` 1.2

- `void append(Printable p, PageFormat format)`
- `void append(Printable p, PageFormat format, int pageCount)`

Dołączają kolejny rozdział do obiektu klasy Book. Jeśli parametr pageCount jest pominięty, to dodawana jest jedna strona.

- `Printable getPrintable(int page)`  
pobiera obiekt typu Printable dla podanej strony.

## 7.12.4. Usługi drukowania

Dotychczas pokazaliśmy sposoby drukowania grafiki Java 2D. Jednak Java SE 1.4 posiada o wiele szersze możliwości. Definiuje szereg typów danych, dla których możliwe jest odnalezienie odpowiedniej usługi drukowania. Należą do nich między innymi:

- obrazy w formatach GIF, JPEG i PNG,
- dokumenty w formacie tekstowym, HTML, PostScript i PDF,
- dane przekazywane bez interpretacji wprost do drukarki,
- obiekty klas implementujących interfejsy `Printable`, `Pageable` i `RenderableImage`.

Źródłem tych danych może być strumień wejściowy, lokalizacja o podanym adresie URL bądź tablica. *Rodzaj dokumentu* opisuje kombinację źródła danych i ich typu. Klasa `DocFlavor` definiuje szereg klas wewnętrznych dla różnych źródeł danych. Każda z tych klas definiuje z kolei stałe określające rodzaj dokumentu. Na przykład stała

`DocFlavor.INPUT_STREAM.GIF`

opisuje obraz w formacie GIF dostępny za pomocą strumienia wejścia. Tabela 7.3 przedstawia pozostałe kombinacje.

**Tabela 7.3.** Rodzaje dokumentów dla usług drukowania

Źródło danych	Typ danych	Typ MIME
INPUT_STREAM	GIF	image/gif
URL	JPEG	image/jpeg
BYTE_ARRAY	PNG	image/png
	POSTSCRIPT	application/postscript
	PDF	application/pdf
	TEXT_HTML_HOST	text/html (wykorzystujący kodowanie znaków hosta)
	TEXT_HTML_US_ASCII	text/html; charset=us-ascii

**Tabela 7.3.** Rodzaje dokumentów dla usług drukowania — ciąg dalszy

Źródło danych	Typ danych	Typ MIME
	TEXT_HTML_UTF_8	text/html; charset=utf-8
	TEXT_HTML_UTF_16	text/html; charset=utf-16
	TEXT_HTML_UTF_16LE	text/html; charset=utf-16le (najpierw bajt mniej znaczący)
	TEXT_HTML_UTF_16BE	text/html; charset=utf-16be (najpierw bajt bardziej znaczący)
	TEXT_PLAIN_HOST	text/plain (wykorzystujący kodowanie znaków hosta)
	TEXT_PLAIN_US_ASCII	text/plain; charset=us-ascii
	TEXT_PLAIN_UTF_8	text/plain; charset=utf-8
	TEXT_PLAIN_UTF_16	text/plain; charset=utf-16
	TEXT_PLAIN_UTF_16LE	text/plain; charset=utf-16le (najpierw bajt mniej znaczący)
	TEXT_PLAIN_UTF_16BE	text/plain; charset=utf-16be (najpierw bajt bardziej znaczący)
	PCL	application/vnd.hp_PCL (Hewlett Packard Printer Control Language)
	AUTOSENSE	application/octet-stream (dane wysyłane bez interpretacji wprost do drukarki)
READER	TEXT_HTML	text/html; charset=utf-16
STRING	TEXT_PLAIN	text/plain; charset=utf-16
CHAR_ARRAY		
SERVICE_FORMATTED	PRINTABLE	nie dotyczy
	PAGEABLE	nie dotyczy
	RENDERABLE_IMAGE	nie dotyczy

Załóżmy, że chcemy wydrukować obraz w formacie GIF zapisany w pliku. Musimy najpierw sprawdzić, czy istnieje usługa drukowania, która wykona to zadanie. Metoda statyczna `lookupPrintServices` klasy `PrintServiceLookup` zwraca tablicę obiektów typu `PrintService`, które umożliwiają wydruk danego rodzaju dokumentu.

```
DocFlavor flavor = DocFlavor.INPUT_STREAM.GIF;
PrintService[] services
    = PrintServiceLookup.lookupPrintServices(flavor, null);
```

Drugi z parametrów metody `lookupPrintServices` jest wartością `null`, co wskazuje, że nie chcemy zawęzać zakresu poszukiwania przez wskazanie atrybutów drukarki, które omówimy w kolejnym podrozdziale.

Jeśli tablica zwrocona przez metodę `lookupPrintServices` zawiera więcej niż jeden element, musimy zdecydować, którą z usług chcemy wykorzystać. Metoda `getName` klasy `PrintService` zwraca nazwę drukarki, która posłużyć może jako kryterium wyboru.

Następnie uzyskujemy dla danej usługi obiekt klasy `DocPrintJob`:

```
DocPrintJob job = services[i].createPrintJob();
```

Do drukowania musimy dostarczyć obiekt implementujący interfejs `Doc`. Java udostępnia klasę `SimpleDoc` implementującą ten interfejs. Konstruktor klasy `SimpleDoc` wymaga, aby jako parametrów obiektu reprezentującego źródło danych użyć rodzaju dokumentu i, opcjonalnie, zbioru atrybutów. Na przykład:

```
InputStream in = new FileInputStream(fileName);
Doc doc = new SimpleDoc(in, flavor, null);
```

Możemy teraz uruchomić proces drukowania:

```
job.print(doc, null)
```

Podobnie jak wcześniej, wartość `null` może zostać zastąpiona zbiorem atrybutów.

Zwróćmy uwagę, że proces drukowania przebiega teraz inaczej, niż zostało to opisane w poprzednim podrozdziale, ponieważ nie ma miejsca żadna interakcja z użytkownikiem wykorzystująca okna dialogowe. Możemy za to zaimplementować na przykład mechanizm wydruku na serwerze, któremu użytkownicy będą przekazywać prace do wydruku za pośrednictwem formularza na stronie internetowej.

Program z listingu 7.16 demonstruje sposób wykorzystania usług drukowania na przykładzie wydruku pliku graficznego.

#### **Listing 7.16.** printService/PrintServiceTest.java

```
package printService;

import java.io.*;
import java.nio.file.*;
import javax.print.*;

/**
 * Program demonstrujący wykorzystanie usług drukowania.
 * Drukuję obrazek w formacie GIF za pomocą jednej z usług
 * dostępnych dla tego rodzaju dokumentu (wybranej przez użytkownika).
 * @version 1.10 2007-08-16
 * @author Cay Horstmann
 */
public class PrintServiceTest
{
    public static void main(String[] args)
    {
        DocFlavor flavor = DocFlavor.URL.GIF;
```

```
PrintService[] services = PrintServiceLookup.lookupPrintServices(flavor, null);
if (args.length == 0)
{
    if (services.length == 0) System.out.println("No printer for flavor " + flavor);
    else
    {
        System.out.println("Specify a file of flavor " + flavor
                           + "\nand optionally the number of the desired printer.");
        for (int i = 0; i < services.length; i++)
            System.out.println((i + 1) + ": " + services[i].getName());
    }
    System.exit(0);
}
String fileName = args[0];
int p = 1;
if (args.length > 1) p = Integer.parseInt(args[1]);
if (fileName == null) return;
try (InputStream in = Files.newInputStream(Paths.get(fileName)))
{
    Doc doc = new SimpleDoc(in, flavor, null);
    DocPrintJob job = services[p - 1].createPrintJob();
    job.print(doc, null);
}
catch (Exception ex)
{
    ex.printStackTrace();
}
```

API javax.print.PrintServiceLookup 1-4

- PrintService[] lookupPrintServices(DocFlavor flavor, AttributeSet attributes)  
wyszukuje usługi drukowania dla danego rodzaju dokumentu i zbioru atrybutów.  
*Parametry:* flavor rodzaj dokumentu,  
                  attributes wymagane atrybuty drukowania lub wartość null.

API `javax.print.PrintService` 1-4

- DocPrintJob createPrintJob()  
zwraca obiekt DocPrintJob umożliwiający wydruk obiektu implementującego interfejs Doc, na przykład klasy SimpleDoc.

API *javax.print.DocPrintJob* 1-4

- void print(Doc doc, PrintRequestAttributeSet attributes)  
drukuje dokument o podanych atrybutach.  
*Parametry:* doc drukowany dokument,  
                  attributes wymagane atrybuty drukowania lub wartość null.

**API javax.print.SimpleDoc 1.4**

- **SimpleDoc(Object data, DocFlavor flavor, DocAttributeSet attributes)**  
tworzy obiekt klasy SimpleDoc, który może być wydrukowany za pomocą obiektu klasy DocPrintJob.
- Parametry:*
- |            |   |
|------------|---|
| data       | obiekt zawierający dane do wydruku, na przykład strumień wejściowy lub obiekt typu Printable, |
| flavor     | rodzaj dokumentu,   |
| attributes | wymagane atrybuty drukowania lub wartość null.  |

## 7.12.5. Usługi drukowania za pośrednictwem strumienia

Usługa drukowania tworzy wydruk, wysyłając dane do drukarki. Usługa drukowania za pośrednictwem strumienia tworzy te same dane, ale zamiast wysyłać je do drukarki, umieszcza je w strumieniu, aby odroczyć proces drukowania lub umożliwić zinterpretowanie danych wydruku innemu programowi. Platforma Java udostępnia usługę drukowania za pośrednictwem strumienia, która umożliwia utworzenie danych w formacie PostScript dla obrazów i grafiki 2D. Usługa ta dostępna jest w każdym systemie, nawet jeśli nie posiada on zainstalowanej żadnej drukarki.

Znalezienie odpowiedniej usługi drukowania za pośrednictwem strumienia jest nieco bardziej pracochłonne niż znalezienie zwykłej usługi drukowania. Musimy określić rodzaj drukowanego dokumentu i typ MIME dla strumienia wyjściowego. W rezultacie otrzymamy tablicę StreamPrintServiceFactory zawierającą fabryki.

```
DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINTABLE;
String mimeType = "application/postscript";
StreamPrintServiceFactory[] factories =
    StreamPrintServiceFactory.lookupStreamPrintServiceFactories(flavor, mimeType);
```

Klasa StreamPrintServiceFactory nie posiada żadnej metody, która umożliwiłaby rozróżnienie poszczególnych fabryk, dlatego wybierzemy po prostu pierwszą z nich czyli factories[0]. Wywołamy następnie jej metodę getPrintService z parametrem w postaci strumienia wyjściowego, aby uzyskać obiekt klasy StreamPrintService.

```
OutputStream out = new FileOutputStream(fileName);
StreamPrintService service = factories[0].getPrintService(out);
```

Klasa StreamPrintService jest klasą pochodną klasy PrintService. Aby uzyskać wydruk, należy następnie powtórzyć kroki podane w poprzednim rozdziale.

**API javax.print.StreamPrintServiceFactory 1.4**

- **StreamPrintServiceFactory[] lookupStreamPrintServiceFactories(DocFlavor flavor, String mimeType)**  
wyszukuje fabryki usług drukowania umożliwiające wydruk danego rodzaju dokumentów za pomocą strumienia wyjściowego podanego typu MIME.

- StreamPrintService getPrintService(OutputStream out)  
zwraca usługę drukowania, która wysyła dane do podanego strumienia wyjściowego.

## 7.12.6. Atrybuty drukowania

Usługi drukowania dostarczają bogatego zestawu interfejsów i klas umożliwiających wyspecyfikowanie różnego rodzaju atrybutów. Możemy wyróżnić cztery najistotniejsze grupy atrybutów. Dwie pierwsze pozwalają wyspecyfikować żądania wysypane do drukarki.

- *Atrybuty żądań wydruku* wykorzystywane są do określenie opcji wydruku (takich jak na przykład druk dwustronny czy format papieru) dla wszystkich dokumentów danego wydruku.
- *Atrybuty dokumentów* dotyczące poszczególnych dokumentów w ramach danego wydruku.

Pozostałe dwie grupy atrybutów dotyczą informacji o drukarce i stanie wydruku.

- *Atrybuty usługi drukowania* zawierają informacje o usłudze drukowania, takie jak marka i model drukarki i określenie, czy przyjmuje ona zlecenia wydruku.
- *Atrybuty wydruku* zawierają informacje o stanie określonego wydruku (na przykład, czy został on wykonany).

Do opisu różnych atrybutów służy interfejs Attribute wraz z następującymi interfejsami pochodnymi:

```
PrintRequestAttribute
DocAttribute
PrintServiceAttribute
PrintJobAttribute
SupportedValuesAttribute
```

Klasy poszczególnych atrybutów implementują jeden lub więcej wymienionych interfejsów. Na przykład obiekt klasy Copies używany do opisu liczby kopii wydruku implementuje interfejsy PrintRequestAttribute i PrintJobAttribute. W przypadku pierwszego z nich oznacza to, że żądanie wydruku może określać liczbę kopii. Natomiast implementacja drugiego z interfejsów oznacza, że wydruk zawiera określoną liczbę kopii, która może być mniejsza od żądanej ze względu na ograniczone możliwości drukarki lub brak papieru.

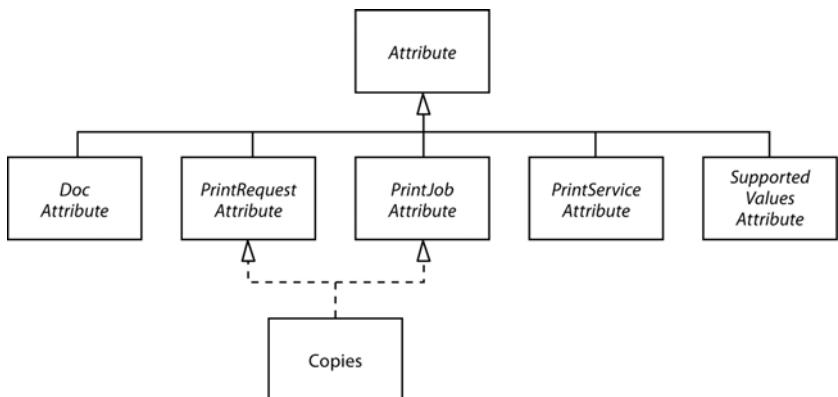
Implementacja przez atrybut interfejsu SupportedValuesAttribute wskazuje, że wartość atrybutu nie jest związana z określonym żądaniem lub stanem, ale określa potencjalne możliwości usługi. Istnieje na przykład klasa CopiesSupported implementująca interfejs SupportedValuesAttribute. Obiekt tej klasy podaje, ile kopii może wydrukować dana drukarka.

Rysunek 7.38 prezentuje diagram hierarchii atrybutów.

Oprócz klas i interfejsów reprezentujących pojedyncze atrybuty zdefiniowano także klasy i interfejsy dla zbiorów atrybutów. Interfejs AttributeSet posiada cztery interfejsy pochodne:

```
PrintRequestAttributeSet
DocAttributeSet
PrintServiceAttributeSet
PrintJobAttributeSet
```

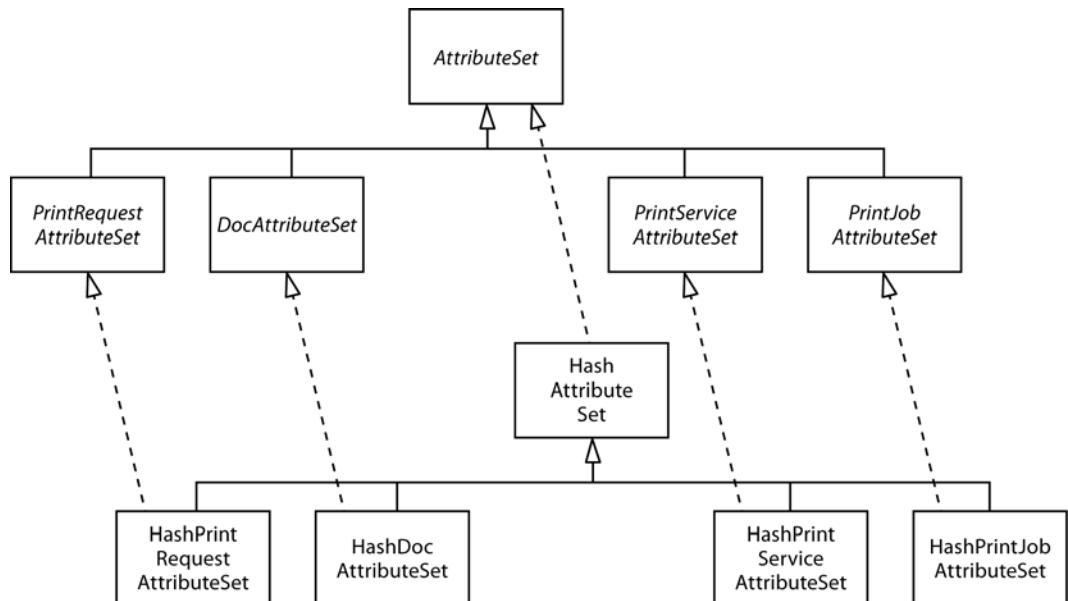
**Rysunek 7.38.**  
Hierarchia atrybutów



Dla każdego z tych interfejsów dostępna jest klasa implementacji:

`HashAttributeSet`  
`HashPrintRequestAttributeSet`  
`HashDocAttributeSet`  
`HashPrintServiceAttributeSet`  
`HashPrintJobAttributeSet`

Na rysunku 7.39 przedstawiony został diagram klas dla hierarchii zbiorów atrybutów.



**Rysunek 7.39.** Hierarchia zbiorów atrybutów

Na przykład zbiór atrybutów żądania wydruku tworzymy w następujący sposób:

```

PrintRequestAttributeSet attributes
= new HashPrintRequestAttributeSet();
  
```

Po utworzeniu zbioru atrybutów nie musimy już posługiwać się przedrostkiem Hash.

W jakim celu utworzono tyle interfejsów? Umożliwiają one kontrolę poprawności użycia atrybutów. Na przykład interfejs DocAttributeSet akceptuje jedynie obiekty implementujące interfejs DocAttribute. Próba dodania innego atrybutu kończy się błędem wykonania programu.

Zbiór atrybutów jest specjalizowaną odmianą mapy, której klucze są typu Class, a wartości należą do klasy implementującej interfejs Attribute. Na przykład jeśli w zbiorze atrybutów umieścimy obiekt

```
new Copies(10)
```

to jego kluczem będzie obiekt Copies.class. Klucz ten nazywany jest *kategorią* atrybutu. Interfejs Attribute definiuje metodę

```
Class getCategory()
```

która zwraca kategorię atrybutu. Klasa Copies implementuje tę metodę, tak by zwracała ona obiekt Copies.class. Nie jest jednak wymagane, by kategoria atrybutu była równoznaczna z jego klasą.

Gdy umieszczamy atrybut w zbiorze atrybutów, to informacja o jego kategorii pobierana jest automatycznie. Wystarczy więc jedynie umieścić atrybut w zbiorze, jak pokazano poniżej:

```
attributes.add(new Copies(10));
```

Jeśli następnie dodamy do zbioru inny atrybut tej samej kategorii, to zastąpi on poprzedni atrybut.

Aby pobrać atrybut, musimy użyć jego kategorii jako klucza, na przykład:

```
AttributeSet attributes = job.getAttributes();
Copies copies = (Copies)attribute.get(Copies.class);
```

Atrybuty są także zorganizowane wokół wartości, które mogą przyjmować. Atrybut klasy Copies może posiadać dowolną wartość całkowitą i dlatego jego klasa jest pochodną klasy IntegerSyntax, podobnie jak klasy innych atrybutów przyjmujących wartości całkowite. Metoda getValue umożliwia pobranie wartości atrybutu:

```
int n = copies.getValue();
```

Istnieją także klasy

```
TextSyntax
DateTimeSyntax
URISyntax
```

hermetyzujące odpowiednio łańcuch znaków, datę i czas lub identyfikator URI (*Uniform Resource Identifier*).

Istnieje także wiele atrybutów, które mogą przyjmować określona liczbę wartości. Na przykład atrybut PrintQuality umożliwia określenie jakości wydruku jako roboczej, zwykłej i wysokości za pomocą następujących stałych:

```
PrintQuality.DRAFT
PrintQuality.NORMAL
PrintQuality.HIGH
```

Klasy atrybutów przyjmujących skończoną liczbę wartości są pochodnymi klasy EnumSyntax, która posiada szereg metod umożliwiających zorganizowanie wyliczeń wartości atrybutów. Dzięki temu, korzystając z atrybutu, nie musimy sami tworzyć odpowiedniego mechanizmu do posługiwanego się jego wartościami. Wystarczy gdy podamy jedynie nazwę wartości, umieszczając atrybut w zbiorze:

```
attributes.add(PrintQuality.HIGH);
```

Poniżej przedstawiamy sposób sprawdzenia wartości atrybutu.

```
if (attribute.get(PrintQuality.class) == PrintQuality.HIGH)
```

Tabela 7.4 prezentuje atrybuty drukowania. Jej druga kolumna zawiera nazwę klasy bazowej dla klasy atrybutu (na przykład IntegerSyntax dla Copies) lub zbiór wartości w przypadku atrybutów, dla których jest on skończony. Ostatnie cztery kolumny pokazują, czy klasa atrybutu implementuje interfejs DocAttribute (DA), PrintJobAttribute (PJA), PrintRequestAttribute (PRA) i PrintServiceAttribute (PSA).

**Tabela 7.4. Atrybuty drukowania**

Atrybut	Klasa bazowa lub wyliczenie wartości atrybutu	DA	PJA	PRA	PSA
Chromaticity	MONOCHROME, COLOR	✓	✓	✓	
ColorSupported	SUPPORTED, NOT_SUPPORTED				✓
Compression	COMPRESS, DEFLATE, GZIP, NONE	✓			
Copies	IntegerSyntax	✓	✓		
DateTimeAtCompleted	DateTimeSyntax	✓			
DateTimeAtCreation	DateTimeSyntax	✓			
DateTimeAtProcessing	DateTimeSyntax	✓			
Destination	UriSyntax	✓	✓		
DocumentName	TextSyntax	✓			
Fidelity	FIDELITY_TRUE, FIDELITY_FALSE	✓	✓		
Finishings	NONE, STAPLE, EDGE_STITCH, BIND, SADDLE_STITCH, COVER, . . .	✓	✓	✓	
JobHoldUntil	DateTimeSyntax	✓	✓		
JobImpressions	IntegerSyntax	✓	✓		
JobImpressionsCompleted	IntegerSyntax	✓			
JobKOctets	IntegerSyntax	✓	✓		
JobKOctetsProcessed	IntegerSyntax	✓			
JobMediaSheets	IntegerSyntax	✓	✓		
JobMediaSheetsCompleted	IntegerSyntax	✓			
JobMessageFromOperator	TextSyntax	✓			
JobName	TextSyntax	✓	✓		

**Tabela 7.4.** Atrybuty drukowania — ciąg dalszy

Atrybut	Klasa bazowa lub wyliczenie wartości atrybutu	DA	PJA	PRA	PSA
JobOriginatingUserName	TextSyntax		✓		
JobPriority	IntegerSyntax	✓	✓		
JobSheets	STANDARD, NONE	✓	✓		
JobState	ABORTED, CANCELED, COMPLETED, PENDING, PENDING_HELD, PROCESSING, PROCESSING_STOPPED		✓		
JobStateReason	ABORTED_BY_SYSTEM, DOCUMENT_FORMAT_ERROR i wiele innych				
JobStateReasons	HashSet		✓		
MediaName	ISO_A4_WHITE, ISO_A4_TRANSPARENT, NA_LETTER_WHITE, NA_LETTER_TRANSPARENT	✓	✓	✓	
MediaSize	ISO.A0 – ISO.A10, ISO.B0 – ISO.B10, ISO.CO – ISO.C10, NA.LETTER, NA.LEGAL oraz inne rozmiary papieru i kopert				
MediaSizeName	ISO_A0 – ISO_A10, ISO_B0 – ISO_B10, ISO_CO – ISO_C10, NA_LETTER, NA_LEGAL oraz inne nazwy rozmiarów papieru i kopert	✓	✓	✓	
MediaTray	TOP, MIDDLE, BOTTOM, SIDE, ENVELOPE, LARGE_CAPACITY, MAIN, MANUAL	✓	✓	✓	
MultipleDocumentHandling	SINGLE_DOCUMENT, SINGLE_DOCUMENT_NEW_SHEET, SEPARATE_DOCUMENTS_COLLATED_COPIES, SEPARATE_DOCUMENTS_UNCOLLATED_COPIES		✓	✓	
NumberOfDocuments	IntegerSyntax		✓		
NumberOfInterveningJobs	IntegerSyntax		✓		
NumberUp	IntegerSyntax	✓	✓	✓	
OrientationRequested	PORTRAIT, LANDSCAPE, REVERSE_PORTRAIT, REVERSE_LANDSCAPE	✓	✓	✓	
OutputDeviceAssigned	TextSyntax		✓		
PageRanges	SetOfInteger	✓	✓	✓	
PagesPerMinute	IntegerSyntax				✓
PagesPerMinuteColor	IntegerSyntax				✓
PDLOverrideSupported	ATTEMPTED, NOT_ATTEMPTED				✓
PresentationDirection	TORIGHT_TOBOTTOM, TORIGHT_TOTOP, TOBOTTOM_TORIGHT, TOBOTTOM_TOLEFT, TOLEFT_TOBOTTOM, TOLEFT_TOTOP, TOTOP_TORIGHT, TOTOP_TOLEFT	✓	✓		
PrinterInfo	TextSyntax				✓
PrinterIsAcceptingJobs	ACCEPTING_JOBS, NOT_ACCEPTING_JOBS				✓

**Tabela 7.4.** Atrybuty drukowania — ciąg dalszy

Atrybut	Klasa bazowa lub wyliczenie wartości atrybutu	DA	PJA	PRA	PSA
PrinterLocation	TextSyntax				✓
PrinterMakeAndModel	TextSyntax				✓
PrinterMessageFromOperator	TextSyntax				✓
PrinterMoreInfo	UriSyntax				✓
PrinterMoreInfoManufacturer	UriSyntax				✓
PrinterName	TextSyntax				✓
PrinterResolution	ResolutionSyntax	✓	✓	✓	
PrinterState	PROCESSING, IDLE, STOPPED, UNKNOWN				✓
PrinterStateReason	COVER_OPEN, FUSER_OVER_TEMP, MEDIA_JAM i wiele innych				
PrinterStateReasons	HashMap				
PrinterURI	UriSyntax				✓
PrintQuality	DRAFT, NORMAL, HIGH	✓	✓	✓	
QueuedJobCount	IntegerSyntax				✓
ReferenceUriSchemesSupported	FILE, FTP, GOPHER, HTTP, HTTPS, NEWS, NNTP, WAIS				
RequestingUserName	TextSyntax				✓
Severity	ERROR, REPORT, WARNING				
SheetCollate	COLLATED, UNCOLLATED	✓	✓	✓	
Sides	ONE_SIDED, DUPLEX (=TWO_SIDED_LONG_EDGE), TUMBLE (=TWO_SIDED_SHORT_EDGE)	✓	✓	✓	



Istnieje wiele wyspecjalizowanych atrybutów drukowania. Większość z nich związana jest z protokołem Internet Printing Protocol 1.1 (RFC 2911).



We wcześniejszych wersjach JDK wprowadzono klasy JobAttributes i PageAttribu  
→tes, których przeznaczenie podobne jest do atrybutów omówionych w tym rozdziale.  
Obecnie klasy te są uważane za przestarzałe.

#### API `javax.print.attribute.Attribute 1.4`

- `Class getCategory()`  
zwraca kategorię atrybutu.
- `String getName()`  
zwraca nazwę atrybutu.

**API *javax.print.attribute.AttributeSet 1.4***■ `boolean add(Attribute attr)`

dodaje atrybut do zbioru atrybutów. Jeśli zbiór zawiera już atrybut tej samej kategorii, to zostanie on zastąpiony. Zwraca wartość true, jeśli zbiór uległ zmianie na skutek wykonania metody.

■ `Attribute get(Class category)`

pobiera atrybut, którego kluczem jest podana kategoria lub zwraca wartość null, jeśli taki atrybut nie istnieje w zbiorze.

■ `boolean remove(Attribute attr)`■ `boolean remove(Class category)`

usuwają ze zbioru atrybut lub atrybut danej kategorii. Zwraca wartość true, jeśli zbiór uległ zmianie na skutek wykonania metody.

■ `Attribute[] toArray()`

zwraca tablicę zawierającą wszystkie atrybuty danego zbioru.

**API *javax.print.PrintService 1.4***■ `PrintServiceAttributeSet getAttributes()`

zwraca atrybuty danej usługi drukowania.

**API *javax.print.DocPrintJob 1.4***■ `PrintJobAttributeSet getAttributes()`

zwraca atrybuty danego wydruku.

Na tym kończymy omówienie zagadnień związanych z drukowaniem. Przedstawiliśmy sposób drukowania grafiki 2D i innych rodzajów dokumentów, system wyszukiwania drukarek i usług drukowania za pośrednictwem strumieni oraz metody korzystania z atrybutów drukowania. W dalszej części rozdziału omówimy zagadnienia związane z wykorzystaniem schowka oraz mechanizmu „przeciagnij i upuść”.

## 7.13. Schowek

Mechanizm „wytnij i wklej” jest jednym z najwygodniejszych i bardziej przydatnych elementów graficznych interfejsów użytkownika (takich jak X Window czy Windows). Dzięki niemu możemy zaznaczyć pewne dane programu i wyciąć je lub skopiować do schowka. Następnie zawartość schowka możemy wkleić w zupełnie innym programie. Korzystając z pośrednictwa schowka, możemy przekazywać tekst, grafikę i inne dane z jednego dokumentu do innego, a także z jednego miejsca do innego miejsca w obrębie tego samego dokumentu. Mechanizm ten jest tak naturalny, że większość użytkowników nie zastanawia się nawet nad sposobem jego działania.

Mimo że koncepcja schowka jest bardzo prosta, to jego implementacja jest trudniejsza, niż mogłoby się to wydawać. Założymy, że do schowka skopiowaliśmy tekst z edytora. Jeśli będziemy chcieli skopiować go do innego dokumentu, to oczywiście informacja o sposobie sformatowania tekstu powinna zostać zachowana. Natomiast umieszczając tekst ze schowka w zwykłym pliku tekstowym, oczekujemy jedynie wstawienia samych znaków tekstu bez dodatkowych kodów formatowania. Aby takie różne zachowania schowka były możliwe, dane muszą być do niego dostarczane w wielu formatach, z których odbiorca wybiera jeden.

Implementacje schowka w systemach Microsoft Windows i Macintosh posiadają podobne możliwości, ale występują też między nimi pewne różnice. Natomiast schowek systemu X Window posiada zdecydowanie mniejsze możliwości — wycinanie i wklejanie danych innych niż zwykły tekst jest rzadko dostępne. Ograniczenia te należy uwzględnić, wypróbowując działanie przykładowych programów zamieszczonych w dalszej części rozdziału.



Informacje o tym, jakiego rodzaju obiekty mogą być przekazywane między schowkiem systemowym a aplikacjami Java, można odnaleźć w pliku *jre/lib/flavormap.properties*.

Często zdarza się, że programy muszą umożliwiać kopowanie i wklejanie takich typów danych, których nie obsługuje schowek systemowy. Java umożliwia przekazywanie referencji do dowolnych lokalnych obiektów w obrębie tej samej maszyny wirtualnej. Pomiędzy różnymi maszynami wirtualnymi można przekazywać serializowane obiekty oraz referencje obiektów zdalnych.

Tabela 7.5 podsumowuje możliwości przekazywania danych za pośrednictwem schowka.

**Tabela 7.5. Möglichosci przekazywania danych na platformie Java**

Rodzaj transferu danych	Typ danych
Pomiędzy programem w języku Java a programem w kodzie macierzystym maszyny.	Tekst, grafika, listy plików, ... (w zależności od platformy).
Pomiędzy dwoma programami w języku Java.	Obiekty serializowane i obiekty zdalne.
W obrębie jednego programu w języku Java.	Dowolne obiekty.

### 7.13.1. Klasa i interfejsy umożliwiające przekazywanie danych

Pakiet `java.awt.datatransfer` zawiera implementację mechanizmu przekazywania danych. Poniżej zamieszczamy przegląd najważniejszych dostępnych w nim klas i interfejsów.

- Obiekty przekazywane za pomocą schowka muszą implementować interfejs `Transferable`.
- Klasa `Clipboard` reprezentuje schowek. W schowku mogą być umieszczone lub pobierane jedynie obiekty implementujące interfejs `Transferable`. Schowek systemowy reprezentowany jest przez obiekt klasy `Clipboard`.
- Klasa `DataFlavor` opisuje rodzaje danych, które mogą być umieszczone w schowku.

- Klasa StringSelection jest klasą konkretną implementującą interfejs Transferable. Jest ona wykorzystywana do przekazywania łańcuchów znaków.
- Klasa musi implementować interfejs ClipboardOwner, jeśli chce być powiadamiana o tym, że zawartość schowka została nadpisana przez inny obiekt. Posiadanie schowka umożliwia odroczenie formatowania skomplikowanych danych. Jeśli program wysyła proste dane, na przykład łańcuch danych, to wystarczy, że umieści go w schowku i może przejść do wykonywania innych zadań. Jeśli jednak program przekazuje za pomocą schowka skomplikowane dane, które mogą być dostępne w różnych formatach, to nie ma sensu, by umieszczać je wszystkie w schowku, dopóki nie będą rzeczywiście potrzebne. Musi wtedy jednak nasłuchiwać żądań pobrania zawartości schowka. O zmianie zawartości schowka zostaje powiadomiony przez wywołanie metody lostOwnership, co oznacza, że dane nie będą już potrzebne. W naszych programach ilustrujących działanie schowka nie będziemy wykorzystywać możliwości posiadania schowka.

### 7.13.2. Przekazywanie tekstu

Najlepszym sposobem na zapoznanie się ze sposobem korzystania z klas mechanizmu „wytnij-wklej” jest najprostszy przypadek przekazywania tekstu do schowka i z niego. Na początek musimy uzyskać referencję schowka systemowego.

```
Clipboard clipboard =
    Toolkit.getDefaultToolkit().getSystemClipboard();
```

Aby przekazywać łańcuchy znaków za pomocą schowka, należy je obudować obiektem klasy StringSelection. Konstruktorowi tej klasy przekazujemy obudowywany łańcuch.

```
String text = . . . ;
StringSelection selection = new StringSelection(text);
```

Przekazanie obiektu do schowka odbywa się za pomocą metody setContents, której parametrami są obiekty klasy StringSelection i ClipboardOwner. Jeśli nie chcemy wyznaczać posiadacza schowka, to jako drugi z parametrów metody przekazujemy wartość null.

```
clipboard.setContents(selection, null);
```

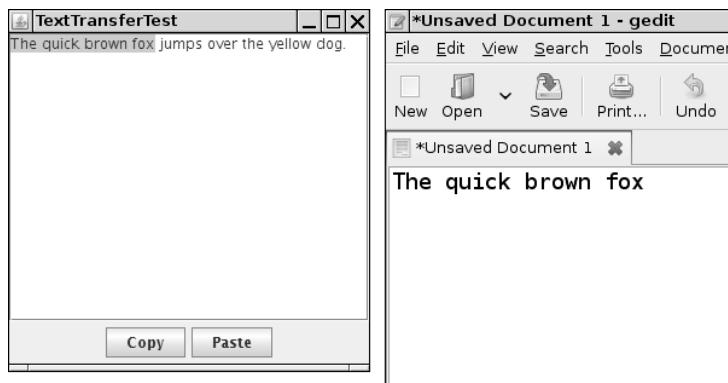
Przyjrzyjmy się operacji odwrotnej, polegającej na odczycie łańcucha znaków ze schowka:

```
DataFlavor flavor = DataFlavor.stringFlavor;
if (clipboard.isDataFlavorAvailable(flavor))
    String text = (String) clipboard.getData(flavor);
```

Program, którego tekst źródłowy zawiera listing 7.17, demonstruje sposób korzystania ze schowka systemowego w aplikacjach Java. Wybranie tekstu w oknie programu, a następnie przycisku *Copy* powoduje umieszczenie tekstu w schowku. Rysunek 7.40 pokazuje, że uzy- skaną w ten sposób zawartość schowka możemy następnie skopiować do dowolnego edytora tekstu. I na odwrót, jeśli skopujemy tekst z edytora, możemy wkleić go w naszym przykładowym programie.

**Rysunek 7.40.**

Program  
*TextTransferTest*  
w działaniu

**Listing 7.17.** transferText/TextTransferFrame.java

```
package transferText;

import java.awt.*;
import java.awt.datatransfer.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;

/**
 * Ramka zawierająca obszar tekstowy
 * oraz przyciski umożliwiające kopiowanie do schowka
 * i wklejanie ze schowka.
 */
public class TextTransferFrame extends JFrame
{
    private JTextArea textArea;
    private static final int TEXT_ROWS = 20;
    private static final int TEXT_COLUMNS = 60;

    public TextTransferFrame()
    {
        textArea = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
        add(new JScrollPane(textArea), BorderLayout.CENTER);
        JPanel panel = new JPanel();

        JButton copyButton = new JButton("Copy");
        panel.add(copyButton);
        copyButton.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                copy();
            }
        });
        JButton pasteButton = new JButton("Paste");
        panel.add(pasteButton);
        pasteButton.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                paste();
            }
        });
    }

    void copy()
    {
        StringSelection selection = new StringSelection(textArea.getText());
        Toolkit.getDefaultToolkit().getSystemClipboard().setContents(selection, null);
    }

    void paste()
    {
        Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
        Transferable contents = clipboard.getContents(null);
        if (contents.isText())
        {
            String text = (String) contents.getTransferData(DataFlavor.stringFlavor);
            textArea.setText(text);
        }
    }
}
```

```
{  
    paste();  
}  
});  
  
add(panel, BorderLayout.SOUTH);  
pack();  
}  
  
/**  
 * Kopiuje wybrany tekst do schowka systemowego.  
 */  
private void copy()  
{  
    Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();  
    String text = textArea.getSelectedText();  
    if (text == null) text = textArea.getText();  
    StringSelection selection = new StringSelection(text);  
    clipboard.setContents(selection, null);  
}  
  
/**  
 * Wkleja tekst ze schowka systemowego.  
 */  
private void paste()  
{  
    Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();  
    DataFlavor flavor = DataFlavor.stringFlavor;  
    if (clipboard.isDataFlavorAvailable(flavor))  
    {  
        try  
        {  
            String text = (String) clipboard.getData(flavor);  
            textArea.replaceSelection(text);  
        }  
        catch (UnsupportedFlavorException e)  
        {  
            JOptionPane.showMessageDialog(this, e);  
        }  
        catch (IOException e)  
        {  
            JOptionPane.showMessageDialog(this, e);  
        }  
    }  
}
```

---

**API** **java.awt.Toolkit 1.0**

- **Clipboard getSystemClipboard() 1.1**

zwraca obiekt reprezentujący schowek systemowy.

**API** **java.awt.datatransfer.Clipboard 1.1**

- **Transferable getContents(Object requester)**

zwraca zawartość schowka.

*Parametry:* requestor obiekt pobierający zawartość schowka.  
W rzeczywistości nie jest używany.

- void setContents(Transferable contents, ClipboardOwner owner)  
umieszcza dane w schowku.

*Parametry:* contents obiekt typu Transferable hermetyzujący dane umieszczone w schowku,  
owner obiekt, który będzie powiadamiany (przez wywołanie jego metody lostOwnership), gdy w schowku umieszczone zostaną nowe dane lub wartość null, jeśli powiadamianie nie jest potrzebne.

- boolean isDataFlavorAvailable(DataFlavor flavor) **5.0**  
zwraca wartość true, jeśli schowek zawiera dane w podanym formacie.
- Object getData(DataFlavor flavor) **5.0**  
zwraca dane w określonym formacie lub zgłasza wyjątek UnsupportedFlavorException, gdy dane w tym formacie nie są dostępne.

#### **java.awt.datatransfer.ClipboardOwner 1.1**

- void lostOwnership(Clipboard clipboard, Transferable contents)  
powiadamia obiekt, że nie jest on już posiadaczem zawartości schowka.
- Parametry:* clipboard schowek, w którym obiekt umieścił dane,  
contents obiekt, który został umieszczony w schowku.

#### **java.awt.datatransfer.Transferable 1.1**

- boolean isDataFlavorSupported(DataFlavor flavor)  
zwraca wartość true, jeśli możliwe jest przekazywanie danego formatu danych, wartość false — w przeciwnym razie.
- Object getTransferData(DataFlavor flavor)  
zwraca dane w żadanym formacie. Wyrzuca wyjątek UnsupportedFlavor, jeśli dane nie są dostępne w danym formacie.

### 7.13.3. Interfejs Transferable i formaty danych

Format danych reprezentowany przez obiekt klasy DataFlavor określony jest przez:

- nazwę typu MIME (na przykład "image/gif"),
- klasę reprezentującą dane i umożliwiającą dostęp do nich (na przykład java.awt.Image).

Dodatkowo każdy format danych posiada czytelną nazwę (na przykład "GIF Image").

Klasa reprezentująca dane może być podana jako parametr `class` typu MIME, na przykład  
`image/gif;class=java.awt.Image`



Powyższy przykład ilustruje jedynie składnię. Nie istnieje standardowy format umożliwiający przekazywanie danych w formacie GIF.

Jeśli nie podamy żadnego parametru `class`, to domyślną klasą reprezentacji będzie `InputStream`.

Zdefiniowano trzy typy MIME dla transferu lokalnych, serializowanych i zdalnych obiektów Java.

```
application/x-java-jvm-local-objectref  
application/x-java-serialized-object  
application/x-java-remote-object
```



Prefiks `x-` sygnalizuje, że są to tzw. nazwy eksperymentalne, czyli nie są oficjalnie zaakceptowane przez IANA — organizację zajmującą się przydziałem nazw typów MIME.

Standardowy format danych `StringFlavor` opisany jest przez następujący typ MIME:

```
application/x-java-serialized-object;class=java.lang.String
```

Istnieje możliwość odpytania wszystkich dostępnych formatów danych schowka:

```
DataFlavor[] flavors = clipboard.getAvailableDataFlavors();
```

Możemy również zainstalować obiekt nasłuchujący `FlavorListener` schowka, który będzie powiadamiany o zmianach formatów danych schowka. Więcej szczegółów na ten temat można odnaleźć w dokumentacji klas i interfejsów przedstawionej poniżej.

### **java.awt.datatransfer.DataFlavor 1.1**

- `DataFlavor(String mimeType, String humanPresentableName)`  
tworzy obiekt reprezentujący format strumienia danych opisany podanym typem MIME.  
*Parametry:* `mimeType` łańcuch znaków opisujący typ MIME,  
`humanPresentableName` czytelna nazwa formatu.
- `DataFlavor(Class class, String humanPresentableName)`  
tworzy obiekt reprezentujący format danych reprezentowany przez klasę Java.  
Jego typem MIME jest `application/x-java-serialized-object;class=className`.  
*Parametry:* `class` klasa obiektu zawartego w obiekcie `Transferable`,  
`humanPresentableName` czytelna nazwa formatu.
- `String getMimeType()`  
zwraca łańcuch znaków reprezentujący typ MIME dla danego formatu.

- boolean isMIMETypeEqual(String mimeType)  
sprawdza, czy format danych jest określonego typu MIME.
- String getHumanPresentableName()  
zwraca czytelną nazwę formatu danych.
- Class getRepresentationClass()  
zwraca obiekt klasy Class reprezentujący klasę obiektu, który zwróci obiekt Transferable, jeśli zażądamy danego formatu danych. Klasa ta będzie odpowiadać parametrowi class typu MIME lub będzie klasą InputStream.

**API java.awt.datatransfer.Clipboard 1.1**

- DataFlavor[] getAvailableDataFlavors() 5.0  
zwraca tablicę dostępnych formatów danych.
- void addFlavorListener(FlavorListener listener) 5.0  
instaluje obiekt nasłuchujący schowka powiadamiany o zmianach w zbiorze formatów danych.

**API java.awt.datatransfer.Transferable 1.1**

- DataFlavor[] getTransferDataFlavors()  
zwraca tablicę dostępnych formatów.

**API java.awt.datatransfer.FlavorListener 5.0**

- void flavorsChanged(FlavorEvent event)  
metoda wywoływana, gdy zmianie ulega zbiór formatów danych schowka.

### 7.13.4. Przekazywanie obrazów za pomocą schowka

Obiekty przekazywane za pomocą schowka muszą implementować interfejs Transferable. Klasa StringSelection jest obecnie jedyną publicznie dostępną klasą w standardowej bibliotece Java implementującą ten interfejs. W bieżącym podrozdziale pokażemy, w jaki sposób przekazywać obrazy za pomocą schowka. Ponieważ Java nie dostarcza klasy do tego odpowiedniej, musimy zaimplementować ją sami.

Implementacja takiej klasy jest banalna. Przechowuje ona obiekt klasy Image i informuje, że jedynym dostępnym formatem danych jest DataFlavor.imageFlavor.

```
class ImageSelection implements Transferable
{
    private Image theImage;

    public ImageSelection(Image image)
    {
```

```

        theImage = image;
    }

    public DataFlavor[] getTransferDataFlavors()
    {
        return new DataFlavor[] { DataFlavor.imageFlavor };
    }

    public boolean isDataFlavorSupported(DataFlavor flavor)
    {
        return flavor.equals(DataFlavor.imageFlavor);
    }

    public Object getTransferData(DataFlavor flavor)
        throws UnsupportedFlavorException
    {
        if(flavor.equals(DataFlavor.imageFlavor))
        {
            return theImage;
        }
        else
        {
            throw new UnsupportedFlavorException(flavor);
        }
    }
}
}

```

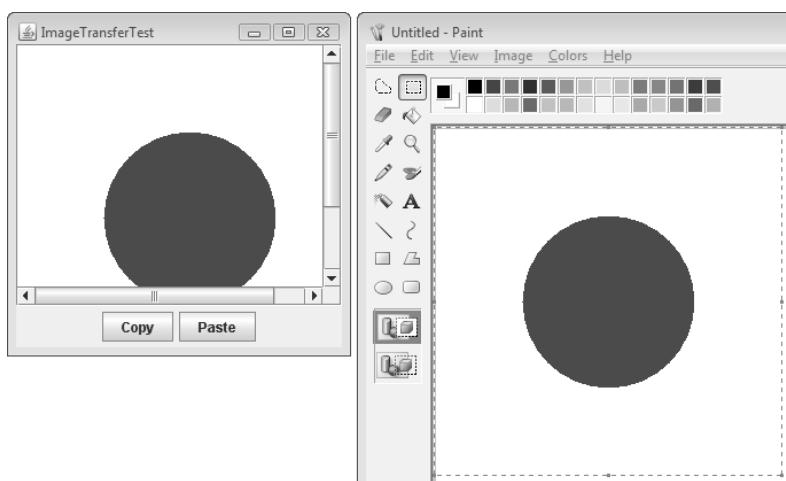


Java SE definiuje stałą DataFlavor.imageFlavor i wykonuje konwersję między formatem obrazów w języku Java a formatem schowka. Nie dostarcza jednak klasy obsługującej, która umożliwiałaby umieszczanie obrazów Java w schowku.

Program z listingu 7.18 demonstruje przekazywanie obrazów między aplikacją Java a schowkiem systemowym. Po uruchomieniu tworzy on obraz zawierający czerwone koło. Wybranie przycisku *Copy* powoduje umieszczenie obrazu w schowku, skąd może zostać wklejony do innych aplikacji (patrz rysunek 7.41). Możemy także umieścić w schowku obraz pochodzący z dowolnej innej aplikacji, a następnie wkleić go do okna naszego przykładowego programu, wybierając przycisk *Paste* (patrz rysunek 7.42).

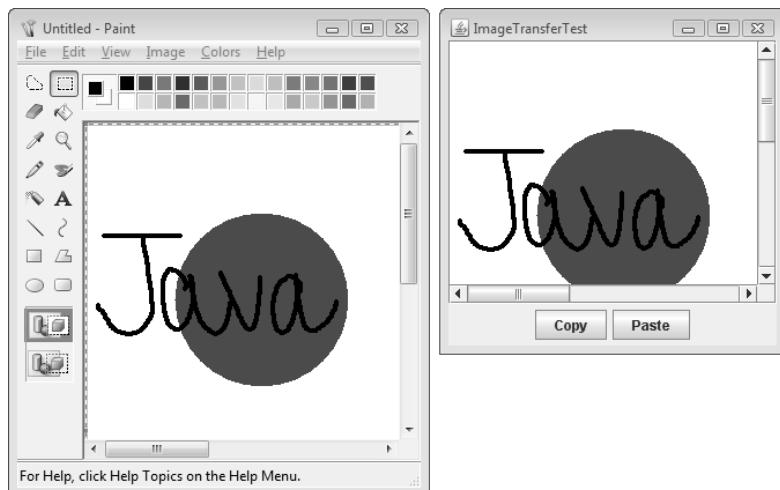
**Rysunek 7.41.**

Kopiowanie obrazu z aplikacji Java do macierzystej aplikacji systemu



**Rysunek 7.42.**

Kopiowanie obrazu z macierzystej aplikacji systemu do aplikacji Java

**Listing 7.18.** *imageTransfer/ImageTransferFrame.java*

```
package imageTransfer;

import java.awt.*;
import java.awt.datatransfer.*;
import java.awt.event.*;
import java.awt.image.*;
import java.io.*;
import javax.swing.*;

/**
 * Ramka zawierająca etykietę wyświetlającą obraz
 * i przyciski umożliwiające jego kopiowanie i wklejanie
 * za pośrednictwem schowka systemowego.
 */
class ImageTransferFrame extends JFrame
{
    private JLabel label;
    private Image image;
    private static final int IMAGE_WIDTH = 300;
    private static final int IMAGE_HEIGHT = 300;

    public ImageTransferFrame()
    {
        label = new JLabel();
        image = new BufferedImage(IMAGE_WIDTH, IMAGE_HEIGHT, BufferedImage.TYPE_INT_ARGB);
        Graphics g = image.getGraphics();
        g.setColor(Color.WHITE);
        g.fillRect(0, 0, IMAGE_WIDTH, IMAGE_HEIGHT);
        g.setColor(Color.RED);
        g.fillOval(IMAGE_WIDTH / 4, IMAGE_WIDTH / 4, IMAGE_WIDTH / 2, IMAGE_HEIGHT / 2);

        label.setIcon(new ImageIcon(image));
        add(new JScrollPane(label), BorderLayout.CENTER);
        JPanel panel = new JPanel();

        JButton copyButton = new JButton("Copy");
        JButton pasteButton = new JButton("Paste");
        panel.add(copyButton);
        panel.add(pasteButton);
        add(panel, BorderLayout.SOUTH);
    }
}
```

```
panel.add(copyButton);
copyButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        copy();
    }
});

JButton pasteButton = new JButton("Paste");
panel.add(pasteButton);
pasteButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        paste();
    }
});

add(panel, BorderLayout.SOUTH);
pack();
}

/**
 * Kopiuje bieżący obraz do schowka systemowego.
 */
private void copy()
{
    Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
    ImageTransferable selection = new ImageTransferable(image);
    clipboard.setContents(selection, null);
}

/**
 * Wkleja obraz ze schowka systemowego.
 */
private void paste()
{
    Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
    DataFlavor flavor = DataFlavor.imageFlavor;
    if (clipboard.isDataFlavorAvailable(flavor))
    {
        try
        {
            image = (Image) clipboard.getData(flavor);
            label.setIcon(new ImageIcon(image));
        }
        catch (UnsupportedFlavorException exception)
        {
            JOptionPane.showMessageDialog(this, exception);
        }
        catch (IOException exception)
        {
            JOptionPane.showMessageDialog(this, exception);
        }
    }
}
```

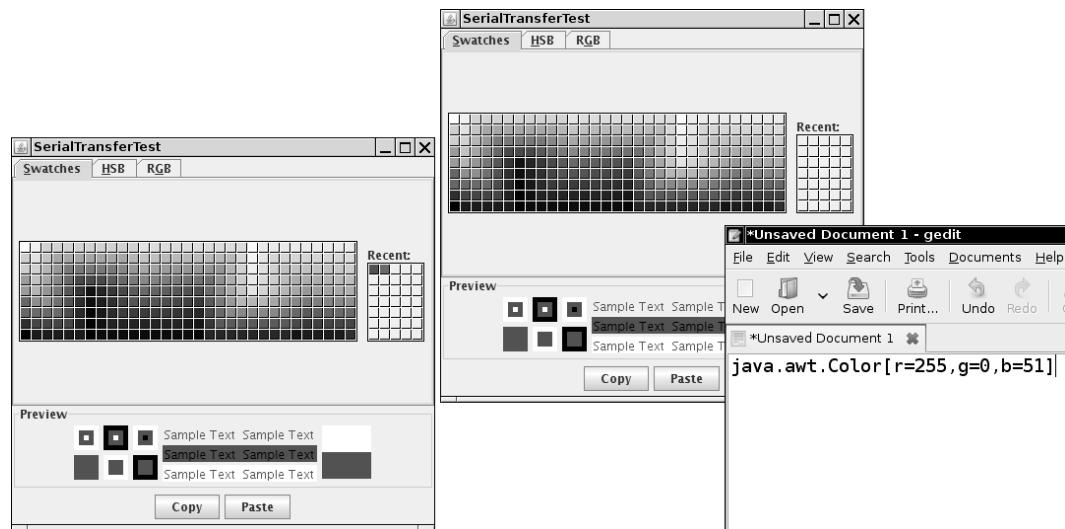
Program przekazywania obrazów powstał jako modyfikacja programu przekazywania tekstów. Formatem danych jest teraz `DataFlavor.imageFlavor`, a klasa `ImageSelection` umożliwia umieszczanie obrazów w schowku systemowym.

### 7.13.5. Wykorzystanie schowka systemowego do przekazywania obiektów Java

Załóżmy, że chcemy kopować obiekty pomiędzy aplikacjami Java. Możemy to osiągnąć, umieszczając serializowane obiekty Java w schowku systemowym.

Program, którego kod źródłowy zawiera listing 7.19, demonstruje taką możliwość. Program zawiera komponent wyboru kolorów. Wybranie przycisku *Copy* powoduje umieszczenie w schowku systemowym informacji o wybranym kolorze w postaci serializowanego obiektu klasy `Color`. Po wybraniu przycisku *Paste* program sprawdza, czy schowek systemowy zawiera serializowany obiekt klasy `Color`. Jeśli tak, to pobiera go i wykorzystuje jako bieżący wybrany kolor komponentu wyboru kolorów.

Serializowane obiekty możemy przekazywać między aplikacjami Java (rysunek 7.43). W tym celu najlepiej uruchomić dwie kopie przykładowego programu `SerialTransferTest`. W jednej z nich wybierzmy przycisk *Copy*, a następnie w drugiej — przycisk *Paste*. Obiekt klasy `Color` zostanie przekazany w ten sposób z jednej maszyny wirtualnej do drugiej.



Rysunek 7.43. Kopiowanie danych między dwiema instancjami aplikacji Java

**Listing 7.19.** *serialTransfer/SerialTransferFrame.java*

```
package serialTransfer;

import java.awt.*;
import java.awt.datatransfer.*;
```

```
import java.awt.event.*;
import java.io.*;
import javax.swing.*;

/**
 * Ramka zawierająca komponent wyboru kolorów
 * oraz przyciski operacji kopiowania i wklejania.
 */
class SerialTransferFrame extends JFrame
{
    private JColorChooser chooser;

    public SerialTransferFrame()
    {
        chooser = new JColorChooser();
        add(chooser, BorderLayout.CENTER);
        JPanel panel = new JPanel();

        JButton copyButton = new JButton("Copy");
        panel.add(copyButton);
        copyButton.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                copy();
            }
        });
        JButton pasteButton = new JButton("Paste");
        panel.add(pasteButton);
        pasteButton.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                paste();
            }
        });

        add(panel, BorderLayout.SOUTH);
        pack();
    }

    /**
     * Kopiuje wybrany kolor do schowka systemowego.
     */
    private void copy()
    {
        Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
        Color color = chooser.getColor();
        Serializable selection = new Serializable(color);
        clipboard.setContents(selection, null);
    }

    /**
     * Wkleja kolor ze schowka systemowego do komponentu wyboru koloru.
     */
    private void paste()
    {
```

```

Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard();
try
{
    DataFlavor flavor = new DataFlavor(
        "application/x-java-serialized-object;class=java.awt.Color");
    if (clipboard.isDataFlavorAvailable(flavor))
    {
        Color color = (Color) clipboard.getData(flavor);
        chooser.setColor(color);
    }
}
catch (ClassNotFoundException e)
{
    JOptionPane.showMessageDialog(this, e);
}
catch (UnsupportedFlavorException e)
{
    JOptionPane.showMessageDialog(this, e);
}
catch (IOException e)
{
    JOptionPane.showMessageDialog(this, e);
}
}

/**
 * Klasa obudowująca serializowane obiekty
 * przekazywane za pomocą schowka systemowego.
 */
class SerialTransferable implements Transferable
{
    private Serializable obj;

    /**
     * Tworzy obiekt klasy SerialSelection.
     * @param o dowolny serializowany obiekt
     */
    SerialTransferable(Serializable o)
    {
        obj = o;
    }

    public DataFlavor[] getTransferDataFlavors()
    {
        DataFlavor[] flavors = new DataFlavor[2];
        Class<?> type = obj.getClass();
        String mimeType = "application/x-java-serialized-object;class=" + type.getName();
        try
        {
            flavors[0] = new DataFlavor(mimeType);
            flavors[1] = DataFlavor.stringFlavor;
            return flavors;
        }
        catch (ClassNotFoundException e)
        {
            return new DataFlavor[0];
        }
    }
}

```

```
}

public boolean isDataFlavorSupported(DataFlavor flavor)
{
    return DataFlavor.stringFlavor.equals(flavor)
        || "application".equals(flavor.getPrimaryType())
        && "x-java-serialized-object".equals(flavor.getSubType())
        && flavor.getRepresentationClass().isAssignableFrom(obj.getClass());
}

public Object getTransferData(DataFlavor flavor) throws UnsupportedFlavorException
{
    if (!isDataFlavorSupported(flavor)) throw new UnsupportedFlavorException(flavor);

    if (DataFlavor.stringFlavor.equals(flavor)) return obj.toString();

    return obj;
}
}
```

---

Aby możliwe było przemieszczanie obiektów w ten sposób, Java umieszcza w schowku systemowym binarną reprezentację serializowanego obiektu. Inna aplikacja Java, niekoniecznie tego samego typu co aplikacja, która umieściła obiekt w schowku, może pobrać dane ze schowka i odtworzyć obiekt.

Oczywiście aplikacja, która nie została napisana w języku Java, nie będzie wiedzieć, jak zinterpretować dane schowka. Z tego powodu przykładowy program umożliwia pobranie danych ze schowka także w formacie tekstowym. Tekst ten uzyskiwany jest jako wynik zastosowania metody `toString` do obiektu przekazywanego za pomocą schowka. Aby to sprawdzić, wystarczy po umieszczeniu koloru w schowku uruchomić na przykład program edytora tekstu i wkleić do niego informację ze schowka. Łańcuch znaków w postaci

`java.awt.Color[r=255,g=51,b=51]`

zostanie wstawiony w dokumencie edytora tekstów.

Przekazywanie serializowanych obiektów za pośrednictwem schowka nie wymaga zbyt wiele programowania. Korzystamy z typu MIME:

`application/x-java-serialized-object;class=nazwaKlasy`

Podobnie jak w poprzednich przykładach tworzymy własną klasę obudowującą — szczegóły znajdziesz w kodzie programu.

## 7.13.6. Zastosowanie lokalnego schowka do przekazywania referencji obiektów

Czasami pojawia się potrzeba skopiowania i wklejenia danych, których typ nie jest obsługiwany przez schowek systemowy oraz nie jest serializowalny. Aby przekazać za pośrednictwem schowka dowolną referencję obiektu Java w obrębie tej samej maszyny wirtualnej, używamy następującego typu MIME:

`application/x-java-jvm-local-objectref;class=nazwaKlasy`

Dla tego typu musimy sami zdefiniować obiekt obudowujący Transferable. Proces ten jest analogiczny jak w przypadku obiektu obudowującego SerialTransferable z poprzedniego przykładu.

Ponieważ referencja obiektu ma sens jedynie w obrębie danej maszyny wirtualnej, to do jej przekazywania nie możemy użyć schowka systemowego. Zamiast nim posłużymy się schowkiem lokalnym:

```
Clipboard clipboard = new Clipboard("local");
```

Parametrem konstruktora schowka jest jego nazwa.

Korzystanie z lokalnego schowka ma jedną istotną wadę. Musimy synchronizować schowek lokalny z systemowym, aby użytkownicy ich nie pomyliły. Obecnie synchronizacja taka nie jest automatycznie wykonywana na platformie Java.

#### java.awt.datatransfer.Clipboard 1.1

- `Clipboard(String name)`  
tworzy lokalny schowek o podanej nazwie.

## 7.14. Mechanizm „przeciagnij i upuść”

Schowek odgrywa rolę pośrednika, gdy korzystamy z mechanizmu „skopiuj i wklej” do przekazywania danych między programami. Mechanizm „przeciagnij i upuść” pozwala pominać programom wszelkie pośrednictwo i komunikować się bezpośrednio. Platforma Java 2 oferuje podstawową obsługę tego mechanizmu. Obiekty możemy przeciągać między aplikacjami Java oraz macierzystymi aplikacjami danej platformy. W podrozdziale tym pokażemy, w jaki sposób zaimplementować aplikację w języku Java, która umożliwia upuszczanie obiektów oraz aplikację, która jest źródłem przeciąganych obiektów.

Zanim wnikniemy głębiej w implementację mechanizmu „przeciagnij i upuść” w języku Java, przyjrzyjmy się najpierw sposobowi jego działania. Zilustrujemy go na przykładzie programów Windows Explorer oraz WordPad — dla innych platform należy eksperymentować z dostępnymi aplikacjami korzystającymi z mechanizmu „przeciagnij i upuść”.

*Operację przeciągnięcia* inicjujemy odpowiednim *gestem* wewnętrz okna źródła — zwykle wybierając najpierw jeden lub więcej elementów, a następnie przeciągając je z wyjściowej lokalizacji. Jeśli zwolnimy klawisz myszy nad celem, to zapyta on źródło o informacje dotyczące upuszczanych elementów i zainicjuje pewne operacje. Na przykład jeśli przeciągniemy ikonę pliku z menedżera plików i upuścimy ją na ikonę reprezentującą katalog, to plik zostanie przeniesiony do tego katalogu. Jeśli natomiast upuścimy ją w oknie edytora tekstu, to edytor otworzy ten plik. (Oczywiście wymaga to użycia menedżera plików i edytora tekstu, które obsługują mechanizm „przeciagnij i upuść”, na przykład pary Explorer/WordPad w systemie Windows lub Nautilus/gedit w środowisku Gnome).

Jeśli podczas przeciągania przytrzymamy klawisz *Ctrl*, to typ operacji związanej z upuszczeniem zmieni się z *przesunięcia* na *kopiowanie* i kopia pliku zostanie umieszczona w danym katalogu. Jeśli przytrzymamy *oba* klawisze *Shift* i *Ctrl*, to w katalogu umieszczone zostanie *łącza* do pliku (inne platformy mogą wykorzystywać w tym celu inne kombinacje klawiszy).

Istnieją więc trzy rodzaje operacji dostępne za pomocą różnych gestów:

- przesunięcie,
- kopiowanie,
- utworzenie łącza.

Intencją operacji utworzenia łącza jest utworzenie referencji upuszczanego elementu. Operacja taka wymaga zwykle odpowiedniej obsługi ze strony systemu operacyjnego (na przykład tworzenia łączy dla plików lub łączy obiektów w dokumentach) i dlatego nie ma sensu tworzenie przenośnych aplikacji korzystających z tej operacji. Omawiając mechanizm „przeciągnij i upuść”, skoncentrujemy się zatem na operacjach przesunięcia i kopiowania.

Operacja przeciągnięcia zwykle posiada pewien rodzaj graficznej ilustracji. W najprostszym przypadku polega on na zmianie kształtu kurSORA. Kursor myszy zmienia swój kształt, gdy przemieszczany jest nad celami, w zależności od tego, czy *upuszczenie obiektu* jest możliwe, czy też nie. Jeśli upuszczenie jest możliwe, to kształt kursora ilustruje typ wykonywanej operacji. Tabela 7.6 pokazuje możliwe kształty, które przyjmuje kursor nad celami.

**Tabela 7.6. Kształty kurSORA podczas operacji upuszczania**

Akcja	Ikona Windows	Ikona Gnome
Przesunięcie		
Kopiowanie		
Utworzenie łącza		
Upuszczenie zabronione		

Oprócz ikon plików możemy także przeciągać inne obiekty. W programie WordPad możemy na przykład wybrać fragment tekstu i przeciągnąć go do pewnego celu, aby sprawdzić, jak on się zachowa.



Powyższy eksperyment pokazuje pewną wadę mechanizmu „przeciągnij i upuść”. Użytkownikowi trudno przewidzieć, jakie elementy interfejsu może przeciągać, gdzie je upuszczać i co to spowoduje. Ponieważ operacja przesunięcia powoduje usunięcie oryginału, to wielu użytkowników wyraża uzasadnioną obawę przed eksperymentowaniem z mechanizmem „przeciągnij i upuść”.

## 7.14.1. Przekazywanie danych pomiędzy komponentami Swing

Począwszy od wersji Java SE 1.4, niektóre komponenty biblioteki Swing obsługują mechanizm „przeciagnij i upuść” (patrz tabela 7.7). Wybrany tekst możemy przeciągać z szeregu komponentów i upuszczać na komponenty tekstowe. Ze względu na konieczność zachowania zgodności z wcześniejszymi wersjami przeciąganie należy uaktywnić, wywołując metodę `setDragEnabled`. Natomiast obsługa upuszczania jest zawsze włączona.

**Tabela 7.7.** Komponenty Swing umożliwiające przekazywanie danych

Komponent	Jako źródło	Jako cel
JFileChooser	Przekazuje listę plików.	Niedostępny.
JColorChooser	Przekazuje lokalną referencję do obiektu klasy <code>Color</code> .	Akceptuje dowolny obiekt klasy <code>Color</code> .
JTextField JformattedTextField	Przekazuje wybrany tekst.	Akceptuje tekst.
JPasswordField	Niedostępny (ze względów bezpieczeństwa).	Akceptuje tekst.
JTextArea JTextPane JEditorPane	Przekazuje wybrany tekst.	Akceptuje tekst i listy plików.
JList JLabel Jtree	Przekazuje tekstowy opis wyboru (tylko kopiowanie).	Niedostępny.



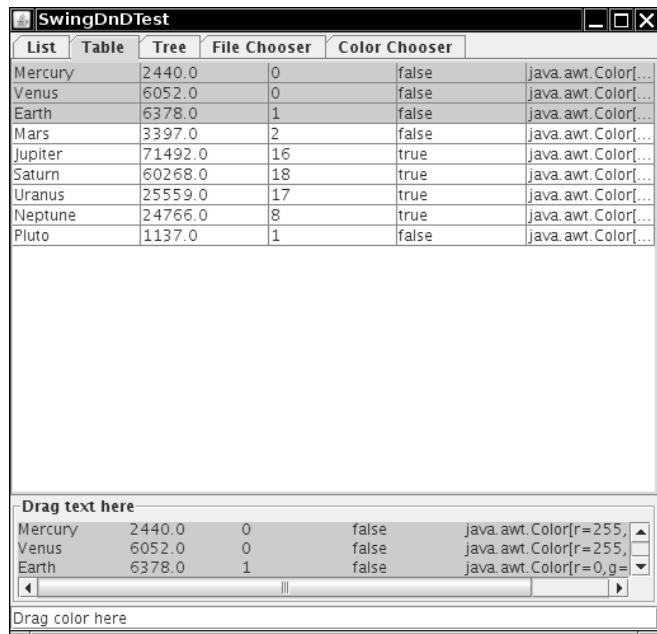
Pakiet `java.awt.dnd` dostarcza niskopoziomowy interfejs mechanizmu „przeciagnij i upuść”, który stanowi bazę dla obsługi tego mechanizmu przez komponenty Swing. Interfejsu tego nie będziemy tutaj omawiać.

Program przedstawiony na listingu 7.20 demonstruje zachowanie komponentów Swing. Zauważmy, że:

- Możemy wybierać wiele elementów list, tabel i drzew (patrz listing 7.21) i przeciągać je.
- Przeciąganie elementów tabeli jest nieco kłopotliwe. Najpierw musimy wybrać dany element za pomocą myszy, po czym należy zwolnić jej klawisz. Następnie jeszcze raz kliknąć i dopiero wtedy możemy przeciągnąć element.
- Upuszczając elementy w obszarze tekstowym, możemy zaobserwować sposób formatowania przeciąganej informacji. Komórki tabeli są oddzielone znakami tabulacji, a każdy wiersz tabeli zostaje umieszczony w osobnym wierszu tekstu (patrz rysunek 7.44).
- Elementy list, tabel, drzew oraz komponentów wyboru plików lub kolorów możemy jedynie kopować, a nie przesuwać. Usuwanie elementów list, tabel i drzew nie jest możliwe dla wszystkich modeli danych. W kolejnym podrozdziale pokażemy, w jaki sposób zaimplementować tę możliwość, gdy model danych umożliwia edycję.

**Rysunek 7.44.**

Program  
 SwingDnDTest  
 w działaniu

**Listing 7.20.** dnd/SwingDnDTest.java

```
package dnd;

import java.awt.*;
import javax.swing.*;

/**
 * Program demonstrujący podstawową obsługę mechanizmu "przeciągnij i upuść"
 * przez komponenty Swing.
 * @version 1.10 2007-09-20
 * @author Cay Horstmann
 */
public class SwingDnDTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new SwingDnDFrame();
                frame.setTitle("SwingDnDTest");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}
```

**Listing 7.21.** dnd/SampleComponents.java

```

package dnd;

import java.awt.*;
import javax.swing.*;
import javax.swing.tree.*;

public class SampleComponents
{
    public static JTree tree()
    {
        DefaultMutableTreeNode root = new DefaultMutableTreeNode("World");
        DefaultMutableTreeNode country = new DefaultMutableTreeNode("USA");
        root.add(country);
        DefaultMutableTreeNode state = new DefaultMutableTreeNode("California");
        country.add(state);
        DefaultMutableTreeNode city = new DefaultMutableTreeNode("San Jose");
        state.add(city);
        city = new DefaultMutableTreeNode("Cupertino");
        state.add(city);
        state = new DefaultMutableTreeNode("Michigan");
        country.add(state);
        city = new DefaultMutableTreeNode("Ann Arbor");
        state.add(city);
        country = new DefaultMutableTreeNode("Germany");
        root.add(country);
        state = new DefaultMutableTreeNode("Schleswig-Holstein");
        country.add(state);
        city = new DefaultMutableTreeNode("Kiel");
        state.add(city);
        return new JTree(root);
    }

    public static JList<String> list()
    {
        String[] words = { "quick", "brown", "hungry", "wild", "silent", "huge", "private",
                           "abstract", "static", "final" };

        DefaultListModel<String> model = new DefaultListModel<>();
        for (String word : words)
            model.addElement(word);
        return new JList<>(model);
    }

    public static JTable table()
    {
        Object[][] cells = { { "Mercury", 2440.0, 0, false, Color.YELLOW },
                             { "Venus", 6052.0, 0, false, Color.YELLOW },
                             { "Earth", 6378.0, 1, false, Color.BLUE }, { "Mars", 3397.0, 2, false,
                               Color.RED },
                             { "Jupiter", 71492.0, 16, true, Color.ORANGE },
                             { "Saturn", 60268.0, 18, true, Color.ORANGE },
                             { "Uranus", 25559.0, 17, true, Color.BLUE },
                             { "Neptune", 24766.0, 8, true, Color.BLUE },
                             { "Pluto", 1137.0, 1, false, Color.BLACK } };
    }
}

```

```

        String[] columnNames = { "Planet", "Radius", "Moons", "Gaseous", "Color" };
        return new JTable(cells, columnNames);
    }
}

```

- Nie możemy przeciągać elementów pomiędzy listami, tabelami, drzewami i komponentami wyboru plików.
- Jeśli uruchomimy dwie kopie programu, możemy przeciągnąć kolor pomiędzy komponentami wyboru kolorów.
- Nie możemy przeciągać tekstu z obszaru tekstowego, ponieważ dla tego komponentu nie wywołaliśmy metody `setDragEnabled`.

Pakiet Swing dostarcza potencjalnie użytecznego mechanizmu umożliwiającego przekształcenie komponentu w źródło przeciągania lub cel upuszczania. Wystarczy zainstalować *obiekt obsługi transferu* dla danej właściwości. Nasz przykładowy program używa w tym celu poniższego wywołania:

```
textField.setTransferHandler(new TransferHandler("background"));
```

Dzięki niemu przeciagnięcie koloru do pola tekstowego powoduje zmianę tła pola tekstowego.

Po upuszczeniu obiekt obsługi transferu sprawdza, czy jeden z formatów danych jest reprezentowany przez klasę `Color`. Jeśli tak, wywołuje metodę `setBackground`.

Instalując obiekt obsługi transferu dla pola tekstowego, powodujemy wyłączenie standardowego obiektu obsługi transferu. Oznacza to, że nie możemy już ani przeciągać tekstu z tego pola, ani upuszczać tekstu na to pole, a nawet wycinać i wklejać tekstu. Natomiast możemy wybrać tekst i przeciągnąć go do komponentu wyboru koloru, który pokaże wtedy kolor tła pola tekstowego.

#### javax.swing.JComponent 1.2

- `void setTransferHandler(TransferHandler handler) 1.4`  
określa procedurę obsługi komponentu związaną z przekazywaniem danych (wycinaniem, kopiowaniem, wklejaniem, przeciąganiem i upuszczeniem).

#### javax.swing.TransferHandler 1.4

- `TransferHandler(String propertyName)`  
tworzy procedurę obsługi, która odczytuje lub nadaje wartość właściwości JavaBeans o podanej nazwie, gdy wykonywana jest operacja przekazywania danych.

#### javax.swing.JFileChooser 1.2

`javax.swing.JColorChooser 1.2`

`javax.swing.JTextField 1.2`

`javax.swing.JList 1.2`

`javax.swing.JTable 1.2`

`javax.swing.JTree 1.2`

■ void setDragEnabled(boolean b) **1.4**

aktywuje lub dezaktywuje przeciąganie danych z komponentu.

## 7.14.2. Źródła przeciaganych danych

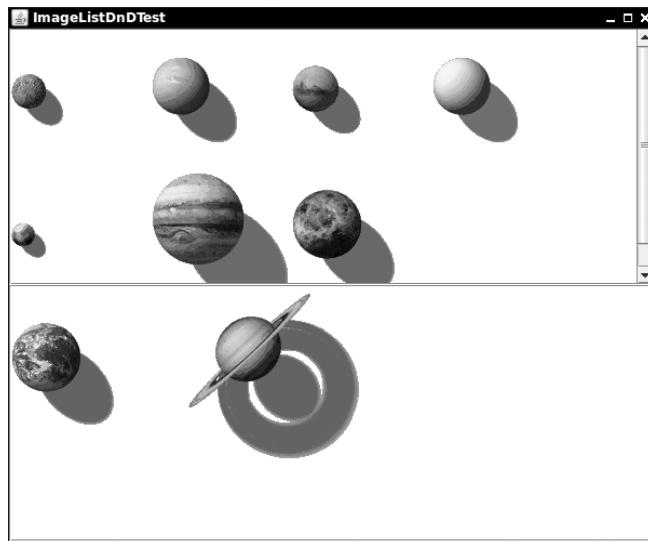
W poprzednim podrozdziale pokazaliśmy zalety podstawowej obsługi mechanizmu „przeciągnij i upuść” przez komponenty Swing. Teraz zajmiemy się konfigurowaniem komponentów jako źródeł przeciaganych danych. W następnym podrozdziale omówimy cele, na które można upuszczać przeciagane dane, i przedstawimy przykładowy komponent, który będzie zarówno źródłem jak i celem mechanizmu „przeciągnij i upuść” dla danych w postaci obrazów.

Aby zindywidualizować działanie mechanizmu „przeciągnij i upuść” dla komponentu Swing, musimy stworzyć klasę pochodną klasy TransferHandler. Najpierw zastępujemy metodę getSourceActions, która informuje o akcjach (COPY, MOVE, LINK) obsługiwanych przez nasz komponent. Następnie zastępujemy metodę createTransferable, która tworzy obiekt Transferable, tak samo jak w przypadku implementowanej wcześniej operacji kopowania do schowka.

W naszym przykładowym programie umożliwimy przeciaganie obrazów z listy JList wypełnionej ich ikonami (patrz rysunek 7.45). Poniżej przedstawiamy implementację metody createTransferable. Wybrany obraz zostaje po prostu obudowany obiektem ImageTransferable.

```
protected Transferable createTransferable(JComponent source)
{
    JList list = (JList) source;
    int index = list.getSelectedIndex();
    if (index < 0) return null;
    ImageIcon icon = (ImageIcon) list.getModel().getElementAt(index);
    return new ImageTransferable(icon.getImage());
}
```

**Rysunek 7.45.**  
Aplikacja *ImageList*  
w działaniu



W naszym przykładzie sprzyja nam fakt, że komponent `JList` obsługuje operację przeciągania. Wystarczy uaktywnić ją za pomocą metody `setDragEnabled`. Jeśli chcemy dodać operację przeciągania dla komponentu, który standardowo jej nie obsługuje, musimy sami zainicjować transfer. Poniżej przedstawiamy przykład inicjacji dla komponentu `JLabel`:

```
label.addMouseListener(new MouseAdapter()
{
    public void mousePressed(MouseEvent evt)
    {
        int mode;
        if ((evt.getModifiers() & (InputEvent.CTRL_MASK | InputEvent.SHIFT_MASK))
            != 0)
            mode = TransferHandler.COPY;
        else mode = TransferHandler.MOVE;
        JComponent comp = (JComponent) evt.getSource();
        TransferHandler th = comp.getTransferHandler();
        th.exportAsDrag(comp, evt, mode);
    }
});
```

W tym przypadku rozpoczynamy transfer, gdy użytkownik kliknie etykietę. Bardziej zaawansowana implementacja wykrywałaby raczej przeciągnięcie kurSORA myszy.

Gdy użytkownik zakończy operację upuszczania, wywołana zostaje metoda `exportDone` obiektu obsługującego transfer źródła. Metoda ta powinna usunąć przeciągnięty obiekt w przypadku, gdy użytkownik dokonał jego przesunięcia. Oto jej implementacja dla listy obrązków:

```
protected void exportDone(JComponent source, Transferable data, int action)
{
    if (action == MOVE)
    {
        JList list = (JList) source;
        int index = list.getSelectedIndex();
        if (index < 0) return;
        DefaultListModel model = (DefaultListModel) list.getModel();
        model.remove(index);
    }
}
```

Podsumowując, aby przekształcić komponent w źródło przyciąganych danych, dodajemy do niego obiekt obsługi transferu, który specyfikuje:

- Akcje obsługiwane przez komponent.
- Dane przekazywane przez komponent.
- Sposób usunięcia oryginalnych danych po wykonaniu operacji przesunięcia.

Dodatkowo, jeśli nasz komponent nie został wymieniony w tabeli 7.7, musimy sami wykrywać rozpoczęcie operacji przeciągania i inicjować transfer.

#### javax.swing.TransferHandler 1.4

- `int getSourceActions(JComponent c)`

metodę tę zastępujemy, aby poinformować o akcjach dopuszczalnych dla danego komponentu, gdy jest on źródłem przyciąganych danych (suma logiczna na bitach stałych `COPY`, `MOVE` i `LINK`).

- protected Transferable createTransferable(JComponent source)  
metodę tę zastępujemy, aby stworzyć obiekt dla przeciaganych danych.
- void exportAsDrag(JComponent comp, InputEvent e, int action)  
rozpoczyna akcję przeciagnięcia dla danego komponentu. Parametr action przyjmuje jedną z wartości COPY, MOVE lub LINK.
- protected void exportDone(JComponent source, Transferable data, int action)  
metodę tę zastępujemy, aby skorygować dane źródła po pomyślnym transferze danych.

### 7.14.3. Cele upuszczanych danych

W tym podrozdziale przedstawimy sposób implementacji celu upuszczanych danych. Naszym przykładem będzie ponownie komponent `JList` zawierający ikony obrazków. Wzbogacimy go o obsługę operacji upuszczania, co pozwoli użytkownikom upuszczać obrazki na listę.

Aby przekształcić komponent w cel upuszczanych danych, musimy skonfigurować dla niego obiekt klasy `TransferHandler` i zaimplementować metody `canImport` i `importData`.



Obiekt obsługujący transferu możemy dodać do komponentu `JFrame`. Możliwość ta jest najczęściej używana do obsługi upuszczania plików w oknie aplikacji. Dozwolone miejsca upuszczania zawierają dekoracje ramki oraz pasek menu, ale nie komponenty umieszczone w ramce (które mają własne obiekty obsługi transferu).

Metoda `canImport` jest ciągle wywoływana, gdy użytkownik przemieszcza kurSOR myszy nad komponentem będącym celem upuszczanych danych. Jeśli upuszczenie jest dozwolone, zwraca wartość `true`. Pozwala to zmienić ikonę kurSora myszy w taki sposób, aby sygnalizowała użytkownikowi możliwość upuszczania.

Metoda `canImport` ma parametr typu `TransferHandler.TransferSupport`. Za jego pośrednictwem otrzymujemy informacje o akcji upuszczania wybranej przez użytkownika, miejscu upuszczania i przekazywanych danych. (W wersjach wcześniejszych od Java SE 6 wywoływana była w tym celu inna metoda `canImport`, która dostarczała jedynie listę formatów danych).

Implementując metodę `canImport`, możemy również zmienić akcję upuszczania wybraną przez użytkownika. Na przykład jeśli użytkownik wybrał akcję przesunięcia, ale oryginał z pewnych względów nie powinien zostać usunięty, możemy wymusić na obiekcie obsługi transferu wykonanie akcji kopiowania.

Oto typowy przykład. Komponent w postaci listy obrazków będzie akceptować upuszczanie list plików oraz obrazków. W przypadku, gdy upuszczona zostanie lista plików, akcja `MOVE` wybrana przez użytkownika zostanie zastąpiona akcją `COPY`, aby zapobiec usunięciu plików obrazków.

```
public boolean canImport(TransferSupport support)
{
    if (support.isDataFlavorSupported(DataFlavor.javaFileListFlavor))
```

```

    {
        if (support.getUserDropAction() == MOVE) support.setDropAction(COPY);
        return true;
    }
    else return support.isDataFlavorSupported(DataFlavor.imageFlavor);
}

```

Bardziej zaawansowana implementacja metody canImport sprawdzałyby, czy pliki rzeczywiście zawierają obrazki.

Gdy użytkownik przesuwa kursor nad celem upuszczenia, komponenty JList, JTable, JTree i JTextField dostarczają wizualnej informacji o pozycji, na którą zostaną wstawione upuszczone dane. Domyślnie odbywa się to poprzez wybór pozycji komponentu (JList, JTable, JTree) lub odpowiednie umiejscowienie kurSORA (JTextComponent). Rozwiązanie takie nie jest szczególnie udane i pozostawiono je jedynie w celu zachowania zgodności z poprzednimi wersjami. Doskonalszej wizualizacji miejsca upuszczenia możemy dostarczyć, wywołując metodę setDropMode.

Możemy wybrać, czy upuszczane dane zastąpią istniejącą pozycję komponentu, czy zostaną umieszczone pomiędzy istniejącymi pozycjami. W naszym przykładowym programie wywołujemy

`setDropMode(DropMode.ON_OR_INSERT);`

aby umożliwić użytkownikowi zastąpienie istniejącej pozycji poprzez upuszczenie na nią nowych danych lub wstawienie danych pomiędzy istniejącymi pozycjami listy (patrz rysunek 7.46). Tryby upuszczenia obsługiwane przez komponenty Swing zostały przedstawione w tabeli 7.8.

Rysunek 7.46.

Wizualizacja upuszczenia na istniejący element lub pomiędzy elementy

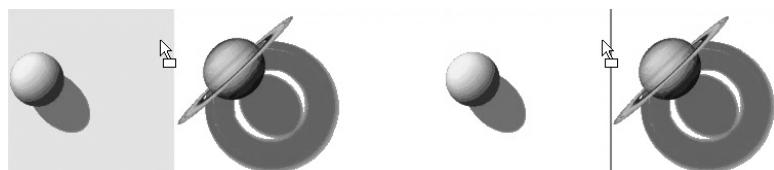


Tabela 7.8. Tryby upuszczenia

Komponent	Obsługiwane tryby upuszczenia
JList, JTree	ON, INSERT, ON_OR_INSERT, USE_SELECTION
Jtable	ON, INSERT, ON_OR_INSERT, INSERT_ROWS, INSERT_COLS, ON_OR_INSERT_ROWS, ON_OR_INSERT_COLS, USE_SELECTION
JtextComponent	INSERT, USE_SELECTION (w rzeczywistości przesuwa kursor, a nie wybór)

Gdy użytkownik upuści wreszcie dane, wywołana zostaje metoda importData. Musimy wtedy pobrać dane ze źródła przeciągnięcia. Wywołując metodę getTransferable dla parametru TransferSupport, uzyskujemy referencję do obiektu klasy Transferable. Jest to ten sam interfejs, który znajduje zastosowanie w przypadku mechanizmu „wytnij i wklej”.

Jednym z typów danych często używanych przez mechanizm „przeciągnij i upuść” jest DataFlavor.javaFileListFlavor. Reprezentuje on zbiór plików upuszczanych na komponent celu.

Przekazywane dane są w tym przypadku typu `List<File>`. A oto kod umożliwiający pobranie plików:

```
DataFlavor[] flavors = transferable.getTransferDataFlavors();
if (Arrays.asList(flavors).contains(DataFlavor.javaFileListFlavor))
{
    List<File> fileList = (List<File>)
        transferable.getTransferData(DataFlavor.javaFileListFlavor);
    for (File f : fileList)
    {
        operacje na f;
    }
}
```

Jeśli dane zostały upuszczone na jeden z komponentów wymienionych w tabeli 7.8, musimy poznać dokładne miejsce upuszczenia. W tym celu wywołujemy metodę `getDropLocation` dla parametru `TransferSupport`. Metoda ta zwraca obiekt klasy pochodnej klasy `TransferHandler.DropLocation`. Klasy `JTable`, `JTree` i `JTextComponent` definiują własne klasy pochodne klasy `TransferHandler.DropLocation` określające miejsce upuszczenia w określonym modelu danych. Na przykład miejsce upuszczenia na liście wystarczy określić za pomocą wartości indeksu, ale w przypadku drzewa wymaga podania ścieżki w drzewie. Poniżej przedstawiamy sposób określenia miejsca upuszczenia zastosowany dla naszej listy obrazków:

```
int index;
if (support.isDrop())
{
    JList.DropLocation location = (JList.DropLocation) support.getDropLocation();
    index = location.getIndex();
}
else index = model.size();
```

Klasa pochodna `JList.DropLocation` udostępnia metodę `getIndex` zwracającą indeks upuszczenia. (Klasa `JTree.DropLocation` dysponuje zamiast niej metodą `getPath`).

Metoda `importData` jest również wywoływana na skutek wklejenia danych do komponentu za pomocą kombinacji klawiszy *Ctrl+V*. W tym przypadku metoda `getDropLocation` wyrzuci wyjątek `IllegalStateException`. Dlatego jeśli metoda `isDrop` zwraca wartość `false`, umieszczymy wklejone dane na końcu listy.

W przypadku wstawiania danych na listę, do tabeli lub drzewa musimy sprawdzić także, czy dane mają być umieszczone pomiędzy istniejącymi elementami, czy zastąpić element znajdujący się w miejscu upuszczenia. W przypadku listy wywołujemy w tym celu metodę `isInsert` klasy `JList.DropLocation`. W przypadku innych komponentów sprawdź opis odpowiednich klas umieszczony na końcu tego podrozdziału.

Podsumowując, aby przekształcić komponent w cel upuszczenia, dodajemy do niego obiekt obsługi transferu specyfikujący:

- Sytuacje, gdy upuszczany element może zostać zaakceptowany.
- Sposób importu upuszczonych danych.

Dodatkowo, jeśli dodajemy obsługę upuszczania do komponentów `JList`, `JTable`, `JTree` lub `JTextComponent`, należy określić również tryb upuszczenia.

Kompletny kod przykładowego programu został przedstawiony na listingu 7.22. Zwróćmy uwagę, że klasa `ImageList` jest zarówno źródłem przeciagania, jak i celem upuszczania. Uruchom program i spróbuj przeciągnąć obrazki pomiędzy dwiema listami. Możesz również przeciągnąć listę plików obrazków z komponentu wyboru plików innego programu na listę obrazków w naszym programie.

**Listing 7.22.** `dndImage/ImageListDnDFrame.java`

---

```
package dndImage;

import java.awt.*;
import java.awt.datatransfer.*;
import java.io.*;
import java.nio.file.*;
import java.util.*;
import java.util.List;
import javax.imageio.*;
import javax.swing.*;

public class ImageListDnDFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 600;
    private static final int DEFAULT_HEIGHT = 500;

    private ImageList list1;
    private ImageList list2;

    public ImageListDnDFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        list1 = new ImageList(Paths.get(getClass().getPackage().getName(), "images1"));
        list2 = new ImageList(Paths.get(getClass().getPackage().getName(), "images2"));

        setLayout(new GridLayout(2, 1));
        add(new JScrollPane(list1));
        add(new JScrollPane(list2));
    }

    class ImageList extends JList<ImageIcon>
    {
        public ImageList(Path dir)
        {
            DefaultListModel<ImageIcon> model = new DefaultListModel<>();
            try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir))
            {
                for (Path entry : entries)
                    model.addElement(new ImageIcon(entry.toString()));
            }
            catch (IOException ex)
            {
                ex.printStackTrace();
            }
        }

        setModel(model);
        setVisibleRowCount(0);
    }
}
```

```

        setLayoutOrientation(JList.HORIZONTAL_WRAP);
        setDragEnabled(true);
        setDropMode(DropMode.ON_OR_INSERT);
        setTransferHandler(new ImageListTransferHandler());
    }
}

class ImageListTransferHandler extends TransferHandler
{
    // Obsługa przeciągania

    public int getSourceActions(JComponent source)
    {
        return COPY_OR_MOVE;
    }

    protected Transferable createTransferable(JComponent source)
    {
        ImageList list = (ImageList) source;
        int index = list.getSelectedIndex();
        if (index < 0) return null;
        ImageIcon icon = list.getModel().getElementAt(index);
        return new ImageTransferable(icon.getImage());
    }

    protected void exportDone(JComponent source, Transferable data, int action)
    {
        if (action == MOVE)
        {
            ImageList list = (ImageList) source;
            int index = list.getSelectedIndex();
            if (index < 0) return;
            DefaultListModel<?> model = (DefaultListModel<?>) list.getModel();
            model.remove(index);
        }
    }
}

// Obsługa upuszczania

public boolean canImport(TransferSupport support)
{
    if (support.isDataFlavorSupported(DataFlavor.javaFileListFlavor))
    {
        if (support.getUserDropAction() == MOVE) support.setDropAction(COPY);
        return true;
    }
    else return support.isDataFlavorSupported(DataFlavor.imageFlavor);
}

public boolean importData(TransferSupport support)
{
    ImageList list = (ImageList) support.getComponent();
    DefaultListModel<ImageIcon> model = (DefaultListModel<ImageIcon>) list.getModel();

    Transferable transferable = support.getTransferable();
    List<DataFlavor> flavors = Arrays.asList(transferable.getTransferDataFlavors());

    List<Image> images = new ArrayList<>();
}

```

```
try
{
    if (flavors.contains(DataFlavor.javaFileListFlavor))
    {
        @SuppressWarnings("unchecked") List<File> fileList
            = (List<File>) transferable.getTransferData(DataFlavor.javaFileListFlavor);
        for (File f : fileList)
        {
            try
            {
                images.add(ImageIO.read(f));
            }
            catch (IOException ex)
            {
                // obrazek nie został odczytany — pomija
            }
        }
    }
    else if (flavors.contains(DataFlavor.imageFlavor))
    {
        images.add((Image) transferable.getTransferData(DataFlavor.imageFlavor));
    }

    int index;
    if (support.isDrop())
    {
        JList.DropLocation location = (JList.DropLocation) support.getDropLocation();
        index = location.getIndex();
        if (!location.isInsert()) model.remove(index); // miejsce zastąpienia
    }
    else index = model.size();
    for (Image image : images)
    {
        model.add(index, new ImageIcon(image));
        index++;
    }
    return true;
}
catch (IOException ex)
{
    return false;
}
catch (UnsupportedFlavorException ex)
{
    return false;
}
}
```

---

**API** javax.swing.TransferHandler 1.4

■ **boolean canImport(TransferSupport support)** 6

metodę tę zastępujemy, aby poinformować, czy komponent będący celem upuszczania może zaakceptować dane opisane przez parametr TransferSupport.

■ **boolean importData(TransferSupport support) 6**

metodę tę zastępujemy, aby przeprowadzić operację upuszczenia lub wklejenia danych opisaną przez parametr TransferSupport. Metoda zwraca wartość true, jeśli import danych zakończył się pomyślnie.

**API javax.swing.JFrame 1.2**

■ **void setTransferHandler(TransferHandler handler) 6**

określa obiekt obsługi transferu tylko dla operacji upuszczenia i wklejenia.

**API javax.swing.JList 1.2**

**javax.swing.JTable 1.2**

**javax.swing.JTree 1.2**

**javax.swing.JTextField 1.2**

■ **void setDropMode(DropMode mode) 6**

określa tryb upuszczenia dla danego komponentu za pomocą jednej z wartości podanych w tabeli 7.8.

**API javax.swing.TransferHandler.TransferSupport 6**

■ **Component getComponent()**

zwraca komponent będący celem tego transferu.

■ **DataFlavor[] getDataFlavors()**

zwraca formaty transferowanych danych.

■ **boolean isDrop()**

zwraca wartość true, jeśli transfer jest wynikiem operacji upuszczenia, false — jeśli operacji wklejenia.

■ **int getUserDropAction()**

zwraca akcję upuszczenia wybraną przez użytkownika (MOVE, COPY lub LINK).

■ **getSourceDropActions()**

zwraca akcje upuszczenia dozwolone dla źródła przeciągania.

■ **getDropAction()**

■ **setDropAction()**

zwraca lub konfiguruje akcję upuszczenia dla danego transferu. Początkowo jest to akcja wybrana podczas upuszczenia przez użytkownika, ale może zostać zmieniona przez obiekt obsługi transferu.

■ **DropLocation getDropLocation()**

zwraca miejsce upuszczenia lub wyrzuca wyjątek IllegalStateException, jeśli dany transfer nie jest skutkiem upuszczenia.

### API javax.swing.TransferHandler.DropLocation 6

- `Point getDropPoint()`

zwraca miejsce upuszczenia dla komponentu będącego celem.

### API javax.swing.JList.DropLocation 6

- `boolean isInsert()`

zwraca wartość true, jeśli dane mają zostać wstawione przed danym elementem listy, false — jeśli zastąpią istniejące dane.

- `int getIndex()`

zwraca indeks modelu dla wstawienia nowych danych lub zastąpienia starych.

### API javax.swing.JTable.DropLocation 6

- `boolean isInsertRow()`

- `boolean isInsertColumn()`

zwraca wartość true, jeśli dane mają zostać wstawione przed danym wierszem lub kolumną.

- `int getRow()`

- `int getColumn()`

zwraca indeks wiersza lub kolumny w modelu danych właściwy dla wstawienia nowych danych lub zastąpienia starych. Gdy upuszczenie miało miejsce nad pustym obszarem komponentu, zwraca wartość -1.

### API javax.swing.JTree.DropLocation 6

- `TreePath getPath()`

- `int getChildIndex()`

zwraca ścieżkę drzewa lub węzeł podrzędny, które wraz z trybem upuszczenia komponentu będącego celem określają miejsce upuszczenia (patrz tabela 7.9).

**Tabela 7.9.** Obsługa miejsca upuszczenia dla komponentu JTree

Tryb upuszczenia	Akcja
INSERT	Wstawienie jako węzeł podrzędny ścieżki, przed indeksem węzła podzielnego.
ON lub USE_SELECTION	Zastąpienie danych ścieżki (indeks węzła podzielnego nie jest używany).
INSERT_OR_ON	Jeśli indeks węzła podzielnego ma wartość -1, akcja jak w trybie ON, w przeciwnym razie jak w trybie INSERT.

**API** javax.swing.JTextComponent.DropLocation 6

## ■ int getIndex()

zwraca indeks pozycji, na której mają być wstawione dane.

## 7.15. Integracja z macierzystą platformą

Rozdział zakończymy omówieniem kilku nowości wprowadzonych w wersji Java SE 6, które sprawiają, że aplikacje tworzone w języku Java coraz bardziej przypominają w użytkowaniu aplikacje poszczególnych systemów. Jedną z nich jest możliwość wyświetlania ekranu powitalnego aplikacji podczas ładowania maszyny wirtualnej. Natomiast klasa `java.awt.Desktop` pozwala uruchamiać macierzyste aplikacje systemu, na przykład domyślną przeglądarkę internetową czy klienta poczty elektronicznej. W wersji Java SE 6 uzyskujemy również dostęp do zasobnika systemowego, w którym, podobnie jak czyni to wiele aplikacji macierzystych, możemy umieszczać własne ikony.

### 7.15.1. Ekran powitalny

Jedną z częściej wymienianych wad aplikacji Java jest ich długi czas uruchamiania. Rzeczywiście maszyna wirtualna potrzebuje trochę czasu, aby załadować wszystkie potrzebne klasy, zwłaszcza w przypadku aplikacji Swing, które wymagają obszernego kodu bibliotek Swing i AWT. Może to być nieco irytujące dla użytkownika, który zniecierpliwiony może nawet próbować ponownie uruchomić aplikację, gdyż nie ma żadnej informacji o tym, czy pierwsze uruchomienie dało jakiś rezultat. Rozwiążaniem tej kłopotliwej sytuacji może okazać się wyświetlenie *ekranu powitalnego* aplikacji, czyli niewielkiego okna, które pojawia się bardzo szybko i informuje użytkownika, że aplikacja jest uruchamiana.

Dotychczas wyświetlenie takiego ekranu przez aplikację Java było co najmniej trudne, jeśli nie niemożliwe. Oczywiście kod aplikacji może wyświetlić takie okno, gdy tylko rozpocznie się wykonywanie metody `main`. Problem w tym, że metoda ta jest uruchamiana dopiero po załadowaniu wszystkich klas potrzebnych do jej działania, co zawsze wymaga pewnego czasu.

W wersji Java SE 6 rozwiązano problem ekranu powitalnego, umożliwiając wyświetlenie go przez maszynę wirtualną natychmiast po jej uruchomieniu. Istnieją dwa mechanizmy określenia obrazu wyświetlanego jako ekran powitalny. Możemy użyć opcji `-splash` w wierszu poleceń:

```
java -splash:myimage.png MyApp
```

Alternatywa polega na podaniu pliku zawierającego ekran powitalny w manifeście pliku JAR:

```
Main-Class: MyApp
SplashScreen-Image: myimage.gif
```

Ekran powitalny rzeczywiście zostaje wyświetlony błyskawicznie i znika automatycznie w momencie wyświetlenia pierwszego okna AWT. Ekran powitalny możemy zapisać w formacie GIF, JPEG lub PNG. Animacje (w formacie GIF) oraz przezroczystość (w formatach GIF i PNG) nie są obsługiwane.

Jeśli aplikacje, które tworzysz, są zawsze gotowe do działania w momencie uruchomienia metody `main`, to możesz pominąć lekturę pozostały części tego podrozdziału. Wiele aplikacji posiada jednak architekturę modularną, w której niewielki rdzeń ładuje po uruchomieniu zbiór wtyczek. Typowymi przykładami takich aplikacji na platformie Java są Eclipse i NetBeans. W przypadku takiej architektury warto informować użytkownika o postępie ładowania modułów, używając w tym celu ekranu powitalnego.

Istnieją dwa podejścia do tego zagadnienia. Możemy albo rysować bezpośrednio na ekranie powitalnym, albo zastąpić go oknem o identycznej zawartości i bez ramki, a następnie rysować wewnątrz okna. Nasz przykładowy program prezentuje zastosowanie obu technik.

Aby rysować bezpośrednio na ekranie powitalnym, musimy pobrać jego referencję, a następnie kontekst graficzny i rozmiary:

```
SplashScreen splash = SplashScreen.getSplashScreen();
Graphics2D g2 = splash.createGraphics();
Rectangle bounds = splash.getBounds();
```

Samo rysowanie grafiki odbywa się w tradycyjny sposób. Po zakończeniu rysowania wywołujemy metodę `update`, aby spowodować odświeżenie ekranu powitalnego. Nasz przykładowy program rysuje prosty pasek postępu pokazany w lewej części rysunku 7.47.

```
g.fillRect(x, y, width * percent / 100, height);
splash.update();
```

**Rysunek 7.47.**

Początkowy ekran powitalny zostaje zastąpiony oknem o takiej samej zawartości



Loading module 41



Ekran powitalny jest singletonem. Nie możemy utworzyć własnego ekranu powitalnego. Jeśli ekran powitalny nie zostanie określony w wierszu polecień lub w manifeście, metoda getSplashScreen zwróci wartość null.

Jednak rysowanie bezpośrednio na ekranie powitalnym ma swoje wady. Wyznaczanie pozycji pikseli jest żmudne i w efekcie nasz wskaźnik postępu znacznie odstaje w działaniu od podobnych rozwiązań stosowanych w macierzystych aplikacjach systemu. Aby uniknąć tego problemu, możemy zastąpić początkowy ekran powitalny oknem tego samego rozmiaru i o tej samej zawartości, jak tylko rozpoczęcie się wykonywanie metody main. Okno to może zawierać dowolne komponenty Swing.

Zastosowanie tej techniki ilustruje przykładowy program przedstawiony na listingu 7.23. W prawej części rysunku 7.47 widzimy okno pozbawione ramki i zawierające panel, który rysuje ekran powitalny oraz zawiera komponent JProgressBar. W ten sposób mamy pełen dostęp do interfejsu programowego Swing i możemy łatwo wyświetlać komunikaty bez zajmowania się szczegółami związanymi z wyznaczaniem położenia pikseli.

### **Listing 7.23.** *splashScreen/SplashScreenTest.java*

```
package SplashScreen;

import java.awt.*;
import java.util.List;
import javax.swing.*;

/**
 * Program demonstrujący sposób użycia ekranu powitalnego.
 * @version 1.00 2007-09-21
 * @author Cay Horstmann
 */
public class SplashScreenTest
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 300;

    private static SplashScreen splash;

    private static void drawOnSplash(int percent)
    {
        Rectangle bounds = splash.getBounds();
        Graphics2D g = splash.createGraphics();
        int height = 20;
        int x = 2;
        int y = bounds.height - height - 2;
        int width = bounds.width - 4;
        Color brightPurple = new Color(76, 36, 121);
        g.setColor(brightPurple);
        g.fillRect(x, y, width * percent / 100, height);
        splash.update();
    }

    /**
     * Metoda rysująca na ekranie powitalnym.
     */
    private static void init()
    {
        // KOD
    }
}
```

```

{
    splash = SplashScreen.getSplashScreen();
    if (splash == null)
    {
        System.err.println("Did you specify a splash image with -splash or in the
                           manifest?");
        System.exit(1);
    }

    try
    {
        for (int i = 0; i <= 100; i++)
        {
            drawOnSplash(i);
            Thread.sleep(100); //symuluje ładowanie aplikacji
        }
    }
    catch (InterruptedException e)
    {
    }
}

/**
 * Metoda wyświetlająca ramkę o tej samej zawartości
 * co ekran powitalny.
 */
private static void init2()
{
    final Image img = new ImageIcon(splash.getImageURL()).getImage();

    final JFrame splashFrame = new JFrame();
    splashFrame.setUndecorated(true);

    final JPanel splashPanel = new JPanel()
    {
        public void paintComponent(Graphics g)
        {
            g.drawImage(img, 0, 0, null);
        }
    };

    final JProgressBar progressBar = new JProgressBar();
    progressBar.setStringPainted(true);
    splashPanel.setLayout(new BorderLayout());
    splashPanel.add(progressBar, BorderLayout.SOUTH);

    splashFrame.add(splashPanel);
    splashFrame.setBounds(splash.getBounds());
    splashFrame.setVisible(true);

    new SwingWorker<Void, Integer>()
    {
        protected Void doInBackground() throws Exception
        {
            try
            {
                for (int i = 0; i <= 100; i++)
                {
                    drawOnSplash(i);
                    Thread.sleep(100);
                }
            }
        }
    }.execute();
}

```

```
        publish(i);
        Thread.sleep(100);
    }
}
catch (InterruptedException e)
{
}
return null;
}

protected void process(List<Integer> chunks)
{
    for (Integer chunk : chunks)
    {
        progressBar.setString("Loading module " + chunk);
        progressBar.setValue(chunk);
        splashPanel.repaint(); //ponieważ obrazek został załadowany
    }
}

protected void done()
{
    splashFrame.setVisible(false);

    JFrame frame = new JFrame();
    frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setTitle("SplashScreenTest");
    frame.setVisible(true);
}
}.execute();
}

public static void main(String args[])
{
    init1();

    EventQueue.invokeLater(new Runnable()
    {
        public void run()
        {
            init2();
        }
    });
}
}
```

Zwróćmy uwagę, że nie musimy usuwać początkowego ekranu powitalnego. Zostaje on automatycznie usunięty, gdy tylko pojawia się okno, które go zastępuje.



Niestety moment zastąpienia ekranu powitalnego przez okno jest dość łatwy do zauważenia.

**API** **java.awt.SplashScreen** 6

- static SplashScreen getSplashScreen()
 

zwraca referencję ekranu powitalnego lub wartość null, jeśli nie został skonfigurowany.
- URL getImageURL()
- void setImageURL(URL imageURL)
 

zwraca lub określa adres URL obrazka wyświetlanego jako ekran powitalny. Wywołanie drugiej wersji powoduje aktualizację ekranu powitalnego.
- Rectangle getBounds()
 

zwraca granice ekranu powitalnego.
- Graphics2D createGraphics()
 

zwraca kontekst graficzny umożliwiający rysowanie na ekranie powitalnym.
- void update()
 

aktualizuje wyświetlany ekran powitalny.
- void close()
 

zamyka ekran powitalny. Ekran powitalny zostaje automatycznie zamknięty w momencie wyświetlenia pierwszego okna AWT.

## 7.15.2. Uruchamianie macierzystych aplikacji pulpitu

Klasa `java.awt.Desktop` pozwala uruchomić domyślną przeglądarkę internetową i program obsługi poczty elektronicznej. Możemy również otwierać, edytować i drukować pliki za pomocą aplikacji, które zostały zarejestrowane dla określonych typów plików.

Posługiwanie się wspomnianą klasą jest bardzo proste. Najpierw wywołujemy metodę statyczną `isDesktopSupported`. Jeśli zwróci ona wartość `true`, to platforma na której działa maszyna wirtualna obsługuje uruchamianie aplikacji pulpitu. Następnie wywołujemy metodę statyczną `getDesktop`, aby uzyskać instancję klasy `Desktop`.

Nie wszystkie środowiska wykorzystujące koncepcję pulpitu obsługują wszystkie operacje. Na przykład pulpit Gnome w systemie Linux umożliwia otwieranie plików, ale nie pozwala na ich drukowanie. Aby sprawdzić czy operacja jest dostępna na danej platformie, wywołujemy metodę `isSupported` przekazując jej wartość zdefiniowaną przez wyliczenie `Desktop.Action`. Nasz program zawiera na przykład poniższy test:

```
if (desktop.isSupported(Desktop.Action.PRINT)) printButton.setEnabled(true);
```

Zanim otworzymy, poddamy edycji lub wydrukujemy plik, musimy najpierw sprawdzić czy dana akcja jest obsługiwana, a dopiero potem wywołać metodę `open`, `edit` lub `print`. Aby uruchomić przeglądarkę, należy przekazać obiekt klasy `URI`. (Więcej informacji na temat klasy `URI` znajdziesz w rozdziale 3.). W tym celu wystarczy wywołać konstruktor klasy `URI` i przekazać mu łańcuch URL rozpoczynający się przedrostkiem `http` lub `https`.

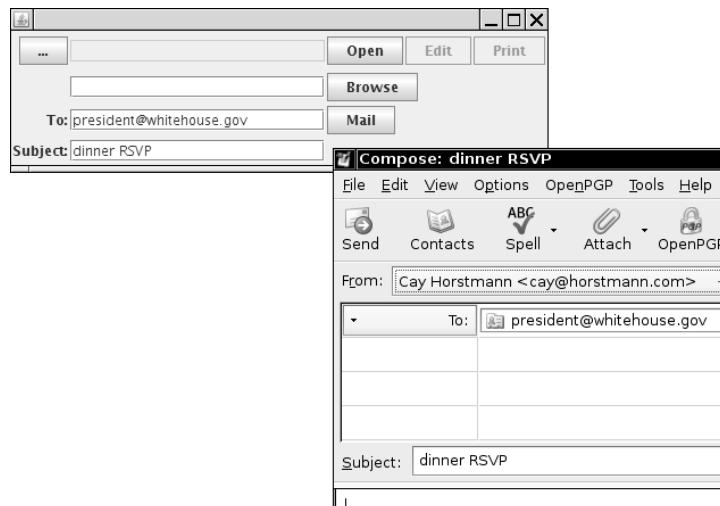
Aby uruchomić domyślny program obsługi poczty elektronicznej, musimy skonstruować obiekt URI w poniższym formacie:

```
mailto:odbiorca?kwerenda
```

Przez *odbiorcę* rozumiemy w tym przypadku adres poczty elektronicznej, na przykład *president@whitehouse.gov*, a *kwerenda* zawiera pary postaci *nazwa=wartość* rozdzielone znakiem & i zakodowane przy użyciu znaku %. (Kodowanie przy pomocy znaku % jest zasadniczo tożsame z kodowaniem łańcuchów URL omówionym w rozdziale 3., z tą różnicą, że znak spacji jest kodowany jako %20, a nie za pomocą znaku +). Przykładem kwerendy może być łańcuch *subject=dinner%20RSVP&bcc=putin%40kremvax.ru*. Format ten jest opisany szczegółowo w dokumencie RFC 2368 (<http://www.ietf.org/rfc/rfc2368.txt>). Niestety klasa URI nie rozpoznaje łańcuchów mailto i wobec tego musimy tworzyć je samodzielnie.

Program przykładowy przedstawiony na listingu 7.24 pozwala użytkownikowi otwierać, edytować i drukować wybrany plik, a także przeglądać podany URL lub uruchamiać klienta poczty elektronicznej (patrz rysunek 7.48).

**Rysunek 7.48.**  
Uruchamianie aplikacji pulpu



**Listing 7.24.** *desktopApp/DesktopAppFrame.java*

```
package desktopApp;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import javax.swing.*;

class DesktopAppFrame extends JFrame
{
    public DesktopAppFrame()
    {
        setLayout(new GridBagLayout());
        final JFileChooser chooser = new JFileChooser();
        JButton fileChooserButton = new JButton("...");
```

```
final JTextField fileField = new JTextField(20);
fileField.setEditable(false);
JButton openButton = new JButton("Open");
JButton editButton = new JButton("Edit");
JButton printButton = new JButton("Print");
final JTextField browseField = new JTextField();
JButton browseButton = new JButton("Browse");
final JTextField toField = new JTextField();
final JTextField subjectField = new JTextField();
JButton mailButton = new JButton("Mail");

openButton.setEnabled(false);
editButton.setEnabled(false);
printButton.setEnabled(false);
browseButton.setEnabled(false);
mailButton.setEnabled(false);

if (Desktop.isDesktopSupported())
{
    Desktop desktop = Desktop.getDesktop();
    if (desktop.isSupported(Desktop.Action.OPEN)) openButton.setEnabled(true);
    if (desktop.isSupported(Desktop.Action.EDIT)) editButton.setEnabled(true);
    if (desktop.isSupported(Desktop.Action.PRINT)) printButton.setEnabled(true);
    if (desktop.isSupported(Desktop.Action.BROWSE)) browseButton.setEnabled(true);
    if (desktop.isSupported(Desktop.Action.MAIL)) mailButton.setEnabled(true);
}

fileChooserButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        if (chooser.showOpenDialog(DesktopAppFrame.this) ==
            JFileChooser.APPROVE_OPTION)
            fileField.setText(chooser.getSelectedFile().getAbsolutePath());
    }
});

openButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        try
        {
            Desktop.getDesktop().open(chooser.getSelectedFile());
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
});

editButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        try
```

```
{  
    Desktop.getDesktop().edit(chooser.getSelectedFile());  
}  
catch (IOException ex)  
{  
    ex.printStackTrace();  
}  
}  
});  
  
printButton.addActionListener(new ActionListener()  
{  
    public void actionPerformed(ActionEvent e)  
{  
        try  
        {  
            Desktop.getDesktop().print(chooser.getSelectedFile());  
        }  
        catch (IOException ex)  
        {  
            ex.printStackTrace();  
        }  
    }  
});  
  
browseButton.addActionListener(new ActionListener()  
{  
    public void actionPerformed(ActionEvent e)  
{  
        try  
        {  
            Desktop.getDesktop().browse(new URI/browseField.getText());  
        }  
        catch (URISyntaxException ex)  
        {  
            ex.printStackTrace();  
        }  
        catch (IOException ex)  
        {  
            ex.printStackTrace();  
        }  
    }  
});  
  
mailButton.addActionListener(new ActionListener()  
{  
    public void actionPerformed(ActionEvent e)  
{  
        try  
        {  
            String subject = percentEncode(subjectField.getText());  
            URI uri = new URI("mailto:" + toField.getText() + "?subject=" + subject);  
  
            System.out.println(uri);  
            Desktop.getDesktop().mail(uri);  
        }  
        catch (URISyntaxException ex)
```

```

        {
            ex.printStackTrace();
        }
    catch (IOException ex)
    {
        ex.printStackTrace();
    }
}
});

JPanel buttonPanel = new JPanel();
((FlowLayout) buttonPanel.getLayout()).setHgap(2);
buttonPanel.add(openButton);
buttonPanel.add(editButton);
buttonPanel.add(printButton);

add(fileChooserButton, new GBC(0, 0).setAnchor(GBC.EAST).setInsets(2));
add(fileField, new GBC(1, 0).setFill(GBC.HORIZONTAL));
add(buttonPanel, new GBC(2, 0).setAnchor(GBC.WEST).setInsets(0));
add(browseField, new GBC(1, 1).setFill(GBC.HORIZONTAL));
add(browseButton, new GBC(2, 1).setAnchor(GBC.WEST).setInsets(2));
add(new JLabel("To:"), new GBC(0, 2).setAnchor(GBC.EAST).setInsets(5, 2, 5, 2));
add(toField, new GBC(1, 2).setFill(GBC.HORIZONTAL));
add(mailButton, new GBC(2, 2).setAnchor(GBC.WEST).setInsets(2));
add(new JLabel("Subject:"), new GBC(0, 3).setAnchor(GBC.EAST).setInsets(5, 2, 5, 2));
add(subjectField, new GBC(1, 3).setFill(GBC.HORIZONTAL));

pack();
}

private static String percentEncode(String s)
{
    try
    {
        return URLEncoder.encode(s, "UTF-8").replaceAll("[+]", "%20");
    }
    catch (UnsupportedEncodingException ex)
    {
        return null; // UTF-8 jest zawsze obsługiwany
    }
}
}

```

### java.awt.Desktop 6

- static boolean isDesktopSupported()

zwraca wartość true, jeśli platforma obsługuje uruchamianie aplikacji pulpitu.

- static Desktop getDesktop()

zwraca obiekt Desktop umożliwiający wykonywanie operacji pulpitu. Wyrzuca wyjątek UnsupportedOperationException jeśli platforma nie obsługuje uruchamianie aplikacji pulpitu.

- `boolean isSupported(Desktop.Action action)`  
zwraca wartość true jeśli dana akcja jest obsługiwana. Parametr `action` może przyjmować jedną z wartości OPEN, EDIT, PRINT, BROWSE lub MAIL.
- `void open(File file)`  
uruchamia aplikację zarejestrowaną do prezentacji zawartości plików danego typu.
- `void edit(File file)`  
uruchamia aplikację zarejestrowaną w celu edycji zawartości plików danego typu.
- `void print(File file)`  
drukuje dany plik.
- `void browse(URI uri)`  
uruchamia domyślną przeglądarkę dla podanego URI.
- `void mail()`
- `void mail(URI uri)`  
uruchamia domyślny program obsługi poczty elektronicznej. Druga z wersji metody może być używana do wypełniania poszczególnych części wiadomości.

### 7.15.3. Zasobnik systemowy

Wiele środowisk wykorzystujących koncepcję pulpu posiada również specjalny obszar zawierający ikony programów wykonywanych w tle, które okazjonalnie powiadamiają użytkownika o pewnych zdarzeniach. W systemie Windows obszar ten nosi nazwę *zasobnika systemowego*, a znajdujące się w nim ikony nazwane zostały *ikonami zasobnika*. Na platformie Java przyjęto tę samą terminologię. Typowym przykładem takiego programu jest monitor sprawdzający dostępność nowych wersji oprogramowania. Jeśli pojawi się aktualizacja oprogramowania, monitor zmienia wygląd swojej ikony bądź wyświetla komunikat w jej pobliżu.

Zasobnik systemowy bywa nadużywany przez twórców oprogramowania i użytkownicy zwykle nie są zachwyeni odkryciem jeszcze jednej ikony zasobnika. Nasz przykładowy program nie stanowi wyjątku od tej reguły.

Klasa `java.awt.SystemTray` umożliwia wykorzystanie zasobnika systemowego na różnych platformach. Podobnie jak w przypadku klasy `Desktop` omówionej w poprzednim podrozdziale, najpierw wywołujemy metodę statyczną `isSupported`, aby sprawdzić czy dana platforma obsługuje zasobnik systemowy. Jeśli tak, pobieramy singleton `SystemTray` wywołując w tym celu metodę statyczną `getSystemTray`.

Najważniejszą metodą klasy `SystemTray` jest oczywiście metoda `add` pozwalająca dodawać instancję klasy `TrayIcon`. Instancja taka posiada trzy właściwości:

- Obrazek ikony.
- Podpowiedź wyświetlana gdy kurSOR myszy przesuwa się nad ikoną.
- Podręczne menu wyświetlane, gdy użytkownik kliknie ikonę prawym klawiszem myszki.

Menu podręczne jest instancją klasy `PopupMenu` z biblioteki AWT, reprezentującej menu macierzystej platformy, a nie menu Swing. Do menu dodajemy instancje klasy `MenuItem` (również z biblioteki AWT), z których każda posiada obiekt nasłuchujący akcji podobnie jak jej odpowiednik z biblioteki Swing.

Ikona zasobnika może wyświetlać powiadomienia (patrz rysunek 7.49). W tym celu wywołujemy metodę `displayMessage` określając nagłówek, tekst powiadomienia i jego typ.

```
trayIcon.displayMessage("Your Fortune", fortunes.get(index),
    TrayIcon.MessageType.INFO);
```

**Rysunek 7.49.**

Powiadomienie  
wyświetlane przez  
ikonę zasobnika



Listing 7.25 przedstawia kod aplikacji, która umieszcza w zasobniku systemowym ikonę wyświetlającą sentencje, których źródłem jest plik (utworzony za pomocą nieśmiertelnego programu fortune z systemu Unix). Poszczególne sentencje są oddzielone wierszem zawierającym znak %. Program wyświetla kolejną sentencję co 10 sekund. Na szczęście posiada również menu umożliwiające zakończenie jego pracy. Gdyby tylko wszyscy twórcy ikon zasobnika byli równie przewidujący!

**Listing 7.25.** *systemTray/SystemTrayTest.java*

```
package systemTray;

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

/**
 * Program demonstrujący wykorzystanie zasobnika systemowego.
 * @version 1.01 2012-01-26
 * @author Cay Horstmann
 */
public class SystemTrayTest
{
    public static void main(String[] args)
```

```

        {
            SystemTrayApp app = new SystemTrayApp();
            app.init();
        }
    }

    class SystemTrayApp
    {
        public void init()
        {
            final TrayIcon trayIcon;

            if (!SystemTray.isSupported())
            {
                System.err.println("System tray is not supported.");
                return;
            }

            SystemTray tray = SystemTray.getSystemTray();
            Image image = new ImageIcon(getClass().getResource("cookie.png")).getImage();

            PopupMenu popup = new PopupMenu();
            MenuItem exitItem = new MenuItem("Exit");
            exitItem.addActionListener(new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    System.exit(0);
                }
            });
            popup.add(exitItem);

            trayIcon = new TrayIcon(image, "Your Fortune", popup);

            trayIcon.setImageAutoSize(true);
            trayIcon.addActionListener(new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    trayIcon.displayMessage("How do I turn this off?",
                        "Right-click on the fortune cookie and select Exit.",
                        TrayIcon.MessageType.INFO);
                }
            });

            try
            {
                tray.add(trayIcon);
            }
            catch (AWTException e)
            {
                System.err.println("TrayIcon could not be added.");
                return;
            }

            final List<String> fortunes = readFortunes();
            Timer timer = new Timer(10000, new ActionListener()

```

```
{  
    public void actionPerformed(ActionEvent e)  
    {  
        int index = (int) (fortunes.size() * Math.random());  
        trayIcon.displayMessage("Your Fortune", fortunes.get(index),  
                               TrayIcon.MessageType.INFO);  
    }  
};  
timer.start();  
}  
  
private List<String> readFortunes()  
{  
    List<String> fortunes = new ArrayList<>();  
    try (InputStream inStream = getClass().getResourceAsStream("fortunes"))  
    {  
        Scanner in = new Scanner(inStream);  
        StringBuilder fortune = new StringBuilder();  
        while (in.hasNextLine())  
        {  
            String line = in.nextLine();  
            if (line.equals("%"))  
            {  
                fortunes.add(fortune.toString());  
                fortune = new StringBuilder();  
            }  
            else  
            {  
                fortune.append(line);  
                fortune.append(' ');  
            }  
        }  
    }  
    catch (IOException ex)  
    {  
        ex.printStackTrace();  
    }  
    return fortunes;  
}  
}
```

---

### API java.awt.SystemTray 6

- static boolean isSupported()  
zwraca wartość true, jeśli dana platforma udostępnia zasobnik systemowy.
- static SystemTray getSystemTray()  
zwraca obiekt SystemTray umożliwiający dostęp do zasobnika systemowego.  
Wyrzuca wyjątek UnsupportedOperationException jeśli platforma nie udostępnia zasobnika systemowego.
- Dimension getTrayIconSize()  
zwraca rozmiary ikony w zasobniku systemowym.

- void add(TrayIcon trayIcon)
  - void remove(TrayIcon trayIcon)
- dodaje lub usuwa ikonę w zasobniku systemowym.

**API java.awt.TrayIcon** 

- TrayIcon(Image image)
  - TrayIcon(Image image, String tooltip)
  - TrayIcon(Image image, String tooltip, PopupMenu popupMenu)
- tworzy ikonę zasobnika o podanym obrazku, tekście podpowiedzi i menu podręcznym.
- Image getImage()
  - void setImage(Image image)
  - String getTooltip()
  - void setTooltip(String tooltip)
  - PopupMenu getPopupMenu()
  - void setPopupMenu(PopupMenu popupMenu)
- zwracają lub określają obrazek ikony, tekst podpowiedzi i menu podręczne.
- boolean isImageAutoSize()
  - void setImageAutoSize(boolean autosize)
- zwracają lub konfigurują właściwość `imageAutoSize`. Jeśli właściwość jest skonfigurowana, obrazek zostaje przeskalowany do obszaru ikony. Jeśli nie (domyślnie) zostaje przycięty (jeśli jest zbyt duży) lub wycentrowany (jeśli jest za mały).
- void displayMessage(String caption, String text, TrayIcon.MessageType messageType)
- wyświetla powiadomienie w pobliżu ikony zasobnika. Parametr `messageType` może przyjmować jedną z wartości INFO, WARNING, ERROR lub NONE.
- public void addActionListener(ActionListener listener)
  - public void removeActionListener(ActionListener listener)
- dodaje lub usuwa obiekt nasłuchujący akcji zależnej od konkretnej platformy. Typowym przypadkiem jest kliknięcie powiadomienia lub podwójne kliknięcie ikony w zasobniku systemowym.

W ten sposób dotarliśmy do końca tego długiego rozdziału poświęconego zaawansowanym możliwościami AWT. W następnym rozdziale przedstawimy specyfikację JavaBeans i jej wykorzystanie przez narzędzia budowy interfejsów użytkownika.



# 8

## JavaBeans

W tym rozdziale:

- Dlaczego ziarnka?
- Proces tworzenia ziarnek JavaBeans.
- Wykorzystanie ziarnek do tworzenia aplikacji.
- Wzorce nazw właściwości ziarnek i zdarzeń.
- Typy właściwości ziarnek.
- Klasa informacyjna ziarnka.
- Edytory właściwości.
- Indywidualizacja ziarnka.
- Trwałość ziarnek.

Oficjalna definicja ziarnka podana w specyfikacji JavaBeans brzmi następująco: „Ziarnko jest komponentem programowym wielokrotnego użytku opartym na specyfikacji JavaBeans, który wykorzystywany jest przez narzędzia wizualnego tworzenia aplikacji”.

Raz zaimplementowane ziarnko może być następnie wielokrotnie wykorzystywane przez innych programistów korzystających z narzędzi, takich jak NetBeans, co czyni proces tworzenia aplikacji i appletów bardziej efektywnym.

W tym rozdziale przedstawimy sposób implementacji ziarnek, które mogą być łatwo wykorzystywane przez innych programistów.



Zanim przejdziemy do szczegółów, chcielibyśmy wyjaśnić często powstające nieporozumienie: ziarnka JavaBeans omawiane w tym rozdziale nie mają wiele wspólnego z ziarnkami Enterprise JavaBeans (EJB). Ziarnka EJB są komponentami działającymi na serwerach wyposażonymi w mechanizmy obsługi transakcji, trwałości, replikacji i bezpieczeństwa. Na najbardziej podstawowym poziomie są one również komponentami, które możemy przetwarzać za pomocą wymienionych wcześniej narzędzi tworzenia aplikacji. Jednak technologia Enterprise JavaBeans jest bardziej złożona niż „zwykłe” ziarnka JavaBeans, które omówimy w tym rozdziale.

Nie oznacza to jednak, że zastosowanie ziarenek JavaBeans ogranicza się wyłącznie do aplikacji działających po stronie klienta. Technologie Web takie jak JavaServer Faces (JSF) i JavaServer Pages (JSP) wykorzystują intensywnie model komponentów JavaBeans.

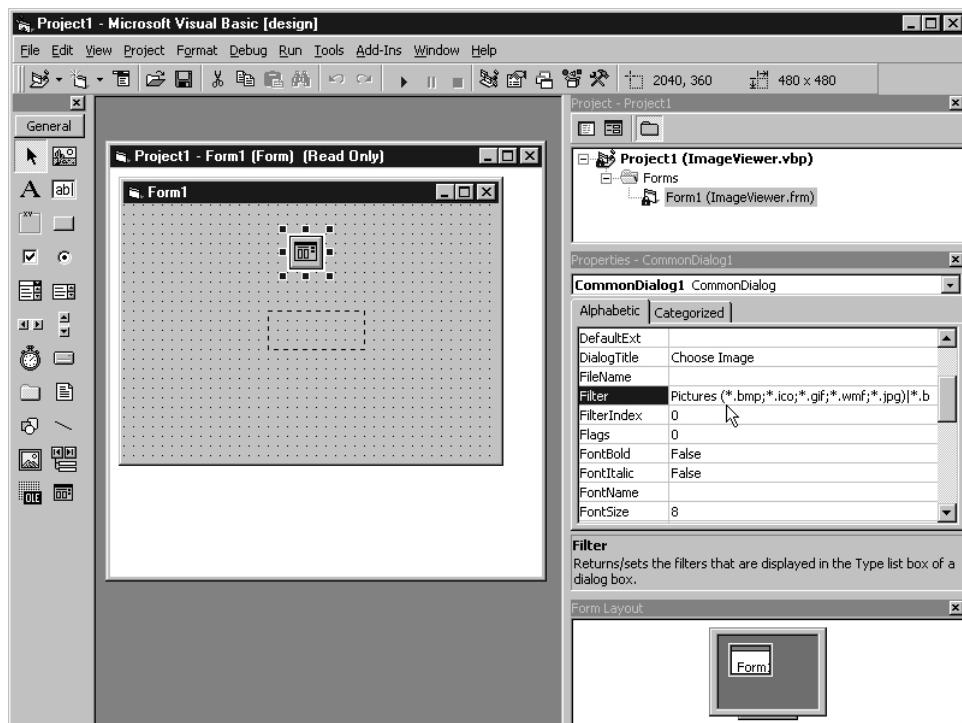
## 8.1. Dlaczego ziarnka?

Programiści tworzący aplikacje systemu Windows w języku Visual Basic natychmiast zorientują się, na czym polega zaleta wykorzystania ziarenek. Programiści tworzący aplikacje dla innych platform, gdzie aplikacje powstają w bardziej tradycyjny sposób oparty o samodzielne tworzenie ich komponentów, mogą z początku mieć trudności ze zrozumieniem idei ziarenek. Z naszego doświadczenia wynika, że największym zaskoczeniem jest zwłaszcza to, że Visual Basic jest jednym z najbardziej udanych przykładów wykorzystania technologii obiektów wielokrotnego użytku. Popularność języka Visual Basic staje się oczywista, dopiero gdy zapoznamy się ze sposobem tworzenia aplikacji w tym języku. Sposób ten przedstawiamy poniżej w skrócie czytelnikom, którzy nie korzystali nigdy z tego języka programowania.

- 1.** Interfejs użytkownika aplikacji tworzony jest przez przeciąganie jego komponentów (zwanych *kontrolkami*) do formatki okna aplikacji.
- 2.** Za pomocą *arkuszy właściwości* określane są właściwości komponentów, takie jak na przykład ich rozmiary, kolor czy sposób zachowania.
- 3.** Arkusze właściwości zawierają także listy zdarzeń związanych z komponentami. Dla niektórych pisze się krótkie fragmenty kodu obsługi.

W 2. rozdziale książki *Java. Podstawy* napisaliśmy program, który wyświetla obrazek w ramce. Jego implementacja zajmuje ponad stronę kodu. A oto sposób, w jaki program o takiej samej funkcjonalności można utworzyć, korzystając z języka Visual Basic.

- 1.** W oknie aplikacji umieszczamy dwie kontrolki: kontrolkę *Image* służącą wyświetlaniu grafiki oraz *Common Dialog* umożliwiającą wybranie pliku.
- 2.** Dla kontrolki *CommonDialog* określamy odpowiednio właściwość *Filter*, tak by możliwy był wybór tylko plików graficznych, które może wyświetlić kontrolka *Image*. Odbywa się to w oknie *Properties* pakietu Visual Basic (patrz rysunek 8.1).
- 3.** Należy jeszcze dodać cztery wiersze kodu w języku Visual Basic, który wykonany zostanie zaraz po uruchomieniu programu. Uruchomieniu programu odpowiada zdarzenie *Form\_Load*, dlatego też kod ten umieścimy w procedurze obsługi tego zdarzenia. Spowoduje on otwarcie okna dialogowego wyboru plików



**Rysunek 8.1.** Określanie właściwości kontrolek Visual Basic

o rozszerzeniu nazwy określonym przez właściwość *Filter*. Po wybraniu przez użytkownika pliku graficznego program wyświetli jego zawartość za pomocą kontrolki *Image*. Kod, który wykona opisane operacje, przedstawia się następująco:

```
Private Sub Form_Load()
    CommonDialog1.ShowOpen
    Image1.Picture = LoadPicture(CommonDialog1.FileName)
End Sub
```

I to wszystko. Kilka przeciągnięć kontrolek i zmian ich właściwości w połączeniu z powyższym, niewielkim fragmentem kodu daje taki sam efekt jak ponad strona kodu w języku Java. Z pewnością łatwiej też nauczyć się posługiwania kontrolkami i ich właściwościami niż pisania kodu programów.

Należy zaznaczyć, że Visual Basic nie jest najlepszym narzędziem do rozwiązywania każdego problemu. Zoptymalizowano go pod kątem szczególnego rodzaju zastosowań — tworzenia aplikacji systemu Windows intensywnie korzystających z możliwości interfejsu użytkownika. Aby technologia Java była konkurencyjna w tym obszarze zastosowań, wymyślono JavaBeans. Specyfikacja ta pozwala producentom tworzyć środowiska podobne do Visual Basica, które umożliwiają tworzenie interfejsów użytkownika, wymagając od projektanta jedynie minimum programowania.

## 8.2. Proces tworzenia ziarenek JavaBeans

Tworzenie ziarenek nie przedstawia technicznie większej trudności — należy jedynie opa-nować sposób wykorzystania kilku nowych klas i interfejsów. Najprostsze ziarnko nie jest niczym innym jak tylko klasą języka Java, która stosuje określone wzorce nazewnictwa dla swoich metod.



Niektórzy autorzy twierdzą, że ziarnko musi posiadać domyślny konstruktor. W rzeczywistości dokumentacja JavaBeans milczy na ten temat. W praktyce większość narzędzi tworzenia aplikacji za pomocą komponentów JavaBeans wymaga od nich domyślnego konstruktora umożliwiającego tworzenie ziarenek bez określania ich parametrów.

Listing 8.1 zamieszczony na końcu bieżącego podrozdziału pokazuje kod ziarka wyświetlania zawartości plików graficznych, które udostępnia narzędziom wizualnego tworzenia aplikacji na platformie Java taką samą funkcjonalność jak wspomniana już kontrolka Image pakietu Visual Basic. Kiedy przyjrzymy się kodowi programu, to zauważymy, że implementacja klasy ImageViewerBean nie różni się od sposobu implementacji innych klas. Wszystkie nazwy metod dostępu do składowych klasy ziarka rozpoczynają się jedynie od prefiksu get, a metody ich modyfikacji od prefiksu set. Jak pokażemy wkrótce, taka konwencja nazw wykorzystywana jest przez narzędzia wizualnego tworzenia aplikacji do wykrywania właściwości ziarenek. Na przykład składowa `fileName` klasy ImageViewerBean jest jej właściwością, ponieważ posiada odpowiednie metody `get` i `set`.

**Listing 8.1.** `imageViewer/ImageViewerBean.java`

```
package imageViewer;

import java.awt.*;
import java.io.*;
import java.nio.file.*;
import javax.imageio.*;
import javax.swing.*;

/**
 * Ziarnko wyświetlania plików graficznych.
 * @version 1.22 2012-06-10
 * @author Cay Horstmann
 */
public class ImageViewerBean extends JLabel
{
    private Path path = null;
    private static final int XREFSIZE = 200;
    private static final int YREFSIZE = 200;

    public ImageViewerBean()
    {
        setBorder(BorderFactory.createEtchedBorder());
    }

    /**
     * Nadaje wartość właściwości fileName.
     * @param fileName the image file name
     */
}
```

```

public void setFileName(String fileName)
{
    path = Paths.get(fileName);
    try (InputStream in = Files.newInputStream(path))
    {
        setIcon(new ImageIcon(ImageIO.read(in)));
    }
    catch (IOException e)
    {
        path = null;
        setIcon(null);
    }
}

/**
 * Zwraca wartość właściwości fileName.
 * @return the image file name
 */
public String getFileName()
{
    if (path == null) return "";
    else return path.toString();
}

public Dimension getPreferredSize()
{
    return new Dimension(XPREFSIZE, YPREFSIZE);
}
}

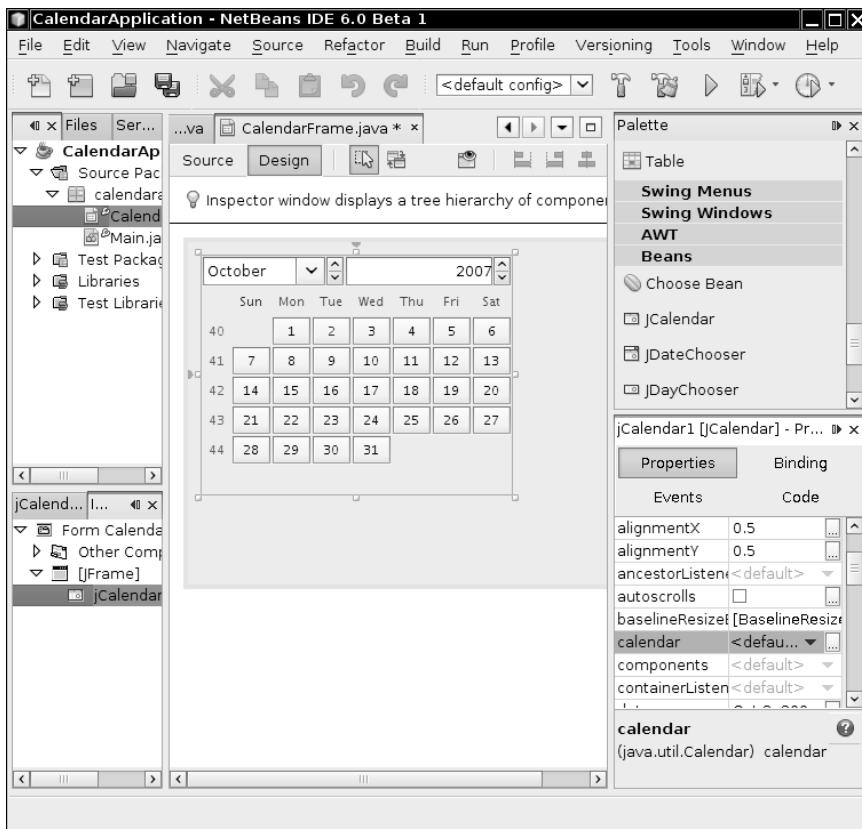
```

Zwróćmy przy tym uwagę, że właściwości nie są po prostu odpowiednikami zmiennych instancji. W naszym przykładzie właściwość `fileName` wyznaczana jest na podstawie zmiennej instancji o nazwie `file`. Właściwości znajdują się na wyższym poziomie koncepcji niż zmienne instancji — stanowią część interfejsu klasy, podczas gdy zmienne instancji należą do implementacji klasy.

Podczas lektury przykładów umieszczonych w tym rozdziale należy pamiętać, że w rzeczywistości ziarnka są bardziej skomplikowane i ich implementacja wymaga większego wysiłku. Dzieje się tak z dwóch powodów:

- 1.** Ziarnka mogą być wykorzystywane przez programistów, którzy nie są ekspertami platformy Java. Korzystać oni będą z funkcjonalności ziarnek głównie za pomocą narzędzi wizualnego tworzenia aplikacji, a nie kodu programów. Wymaga to zaimplementowania *wielu właściwości* ziarnek w celu umożliwienia użytkownikom dopasowania ich zachowania do określonych potrzeb.
- 2.** Ziarnko powinno być przydatne w *najróżniejszych kontekstach*. Wygląd i zachowanie ziarnka muszą mieć możliwość dopasowania do potrzeb konkretnego użytkownika. I tym razem oznacza to konieczność udostępnienia wielu właściwości.

Dobrym przykładem ziarnka o bogatej funkcjonalności jest `CalendarBean` opracowany przez Kaia Toedtera (patrz rysunek 8.2). Ziarnko to i jego kod źródłowy są dostępne na stronie <http://www.toedter.com/en/jcalendar/>. Umożliwia ono wygodne wprowadzanie dat przez



Rysunek 8.2. Ziarnko kalendarza

wskazywanie ich w kalendarzu. Implementacja takiego ziarnka jest dość skomplikowana i nie ma sensu wykonywać jej samodzielnie. Lepiej skorzystać z pracy innych, umieszczając gotowe ziarnko jako element interfejsu tworzonej aplikacji.

Na szczęście aby tworzyć ziarnka o rozbudowanych możliwościach, musimy opanować jedynie niewielką liczbę podstawowych koncepcji. Przykłady ziarenek przedstawione w tym rozdziale nie są trywialne, ale staraliśmy się, aby były wystarczająco proste i czytelnie ilustrowały poszczególne koncepcje.

## 8.3. Wykorzystanie ziarenek do tworzenia aplikacji

Zanim przejdziemy do omówienia sposobu tworzenia ziarenek, pokażemy najpierw, w jaki sposób należy ich używać lub testować je. Chociaż klasa ImageViewerBean stanowi zupełnie wystarczający przykład ziarka, to jednak użyta poza narzędziem wizualnego tworzenia aplikacji nie ujawnia swoich właściwości.

Każde z takich narzędzi korzysta w tym celu z własnego zestawu strategii ułatwiających tworzenie aplikacji. W dalszej części rozdziału posłużymy się pakietem NetBeans, dostępnym w witrynie <http://netbeans.org>.

W dalszej części bieżącego podrozdziału korzystać będziemy z dwu ziarenek, `ImageViewerBean` oraz `FilePickerBean`. Implementację pierwszego z nich pokazaliśmy już wcześniej. Kod ziarnka `FilePickerBean` jest nieco bardziej skomplikowany i przeanalizujemy go w dalszej części rozdziału. Teraz wystarczy jedynie informacja, że po wybraniu przycisku opisanego jako „...” otworzy się okno dialogowe umożliwiające wybór pliku.

### 8.3.1. Umieszczanie ziarenek w plikach JAR

Aby ziarnko mogło być użyte przez narzędzia wizualnego tworzenia aplikacji, wszystkie wykorzystywane przez nie pliki klas muszą zostać umieszczone we wspólnym pliku JAR. W przeciwieństwie do plików JAR zawierających aplenty, dla plików JAR ziarenek musimy utworzyć plik manifestu określający, które z klas zawartych w pliku JAR reprezentują ziarnka i powinny zostać umieszczone w przyborniku narzędzia. Poniżej przedstawiamy zawartość pliku manifestu `ImageViewer.mf` dla ziernka `ImageViewerBean`.

```
Manifest-Version: 1.0
```

```
Name: com/horstmann/corejava/ImageViewerBean.class
Java-Bean: True
```

Zwróćmy uwagę na pusty wiersz oddzielający wersję pliku manifestu od nazwy ziarka.



Niektóre programy narzędziowe mają problemy z ładowaniem ziarenek z pakietu domyślnego, dlatego ziarka należy zawsze umieszczać w pakiecie.

Jeśli ziarko składa się z wielu plików klas, to w pliku manifestu wymieniamy tylko klasy reprezentujące ziarka, które powinny pojawić się w przyborniku narzędzia. Możemy na przykład umieścić klasy `ImageViewerBean` oraz `FilePickerBean` w tym samym pliku, korzystając z poniższego pliku manifestu:

```
Manifest-Version: 1.0
```

```
Name: imageViewer/ImageViewerBean.class
Java-Bean: True
```

```
Name: imageViewer/FilePickerBean.class
Java-Bean: True
```



Niektóre z narzędzi tworzenia aplikacji są szczególnie wrażliwe na zawartość pliku manifestu. Dlatego należy upewnić się, że przed końcem każdego z jego wierszy nie występują zbędne odstępy, że istnieją puste wiersze między informacją o wersji manifestu i informacjami o kolejnych ziarkach oraz, że ostatni wiersz pliku zawiera znak nowego wiersza.

Aby utworzyć odpowiedni plik JAR, należy kolejno wykonać następujące kroki.

- 1.** Tworzymy właściwy plik manifestu.
- 2.** Zbieramy wszystkie potrzebne pliki klas w jednym katalogu.
- 3.** Uruchamiamy program jar w poniższy sposób:

```
jar cvfm PlikJar PlikManifestu PlikiKlas
```

na przykład:

```
jar cvfm ImageViewerBean.jar ImageViewerBean.mf imageViewer/*.class
```

W pliku JAR możemy umieścić także elementy inne niż klasy — na przykład pliki w formacie GIF zawierające ikony. Ikony ziarek omówimy w dalszej części rozdziału.



Należy upewnić się, że plik JAR zawiera wszystkie pliki niezbędne do działania ziarka. W szczególności należy zwrócić uwagę, czy umieściliśmy w nim pliki klasewnętrznych, takie jak na przykład *FilePickerBean\$1.class*.

Narzędzia wizualnego tworzenia aplikacji posiadają odpowiedni mechanizm dodawania nowych ziarek zwykle oparty o wykorzystanie plików JAR. Poniżej przedstawiamy sposób dodania nowych ziarek w środowisku NetBeans wersja 7.

Klasy *ImageViewerBean* oraz *FilePickerBean* należy skompilować i umieścić w pliku JAR. Następnie uruchamiamy pakiet NetBeans i wykonujemy następujące kroki.

- 1.** Z menu wybieramy pozycję *Tools/Palette/Swing/AWT Components*.
- 2.** W oknie dialogowym wybieramy przycisk *Add*.
- 3.** Otwarte zostaje okno dialogowe wyboru pliku, w którym przechodzimy do katalogu *ImageViewerBean* i wybieramy *ImageViewerBean.jar*.
- 4.** W kolejnym oknie dialogowym przedstawiona zostaje lista wszystkich ziarenek, które zostały odnalezione w wybranym pliku JAR. Wybieramy ziarnko *ImageViewerBean*.
- 5.** Na koniec zostaniemy poproszeni o wybór palety, w której umieszczone będzie ziarnko. Wybieramy paletę Beans (istnieją również inne palety dla komponentów Swing, AWT i tak dalej).
- 6.** Przyjrzyjmy się palecie Beans. Zawiera ona teraz ikonę reprezentującą nowe ziarnko. Jest to ikona domyślna — w dalszej części rozdziału pokażmy, w jaki sposób dodawać ikony ziarenek.

Te same kroki należy powtórzyć dla ziarka *FilePickerBean*. Uzyskamy w ten sposób możliwość wykorzystania obu ziarenek w tworzonych aplikacjach.

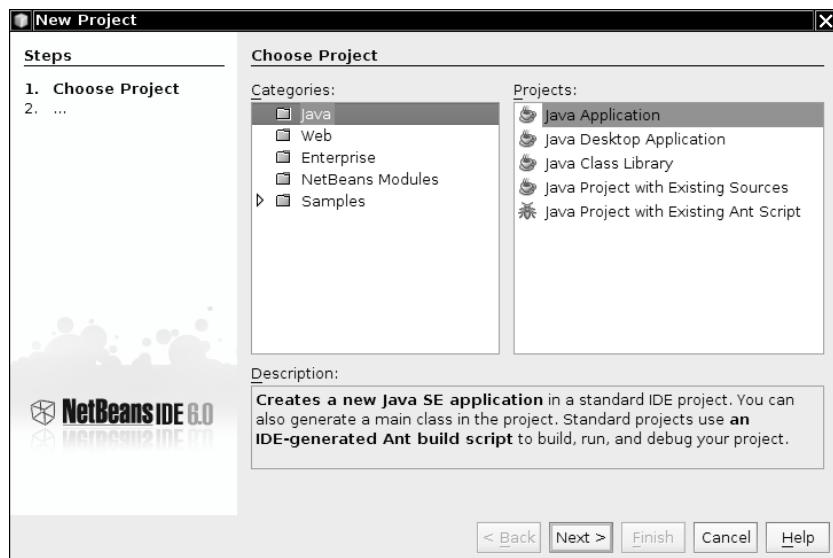
### 8.3.2. Korzystanie z ziarenek

Zaletą korzystania z pakietu takiego jak NetBeans jest możliwość tworzenia aplikacji z prefabrykowanych komponentów przy minimalnym udziale własnego programowania. W tym punkcie pokażemy, w jaki sposób stworzyć aplikację z komponentów *ImageViewerBean* i *FilePickerBean*.

W przypadku pakietu NetBeans wybierzmy pozycję menu *File/New Project*. Otworzy się okno dialogowe, w którym wybieramy Java, a następnie *Java Application* (patrz rysunek 8.3).

**Rysunek 8.3.**

Tworzenie nowego projektu



Wybieramy przycisk *Next*. Określamy nazwę aplikacji (na przykład *ImageViewer*) i wybieramy przycisk *Finish*. Po lewej stronie okna NetBeans pojawia się widok projektu, a w centralnej edytor kodu źródłowego.

Prawym przyciskiem myszy rozwijamy menu naszego projektu w widoku projektu i wybieramy z niego pozycje *New/JFrame Form...* (patrz rysunek 8.4).

Pojawi się okno dialogowe, w którym wprowadzimy nazwę klasy ramki (na przykład *ImageViewerFrame*), a następnie wybierzemy przycisk *Finish*. Pojawi się edytor formatki zawierający pustą ramkę. Aby umieścić ziarnko na formatce, musimy wybrać je z palety znajdującej się po prawej stronie edytora formatek. Następnie klikamy ramkę w edytorze.

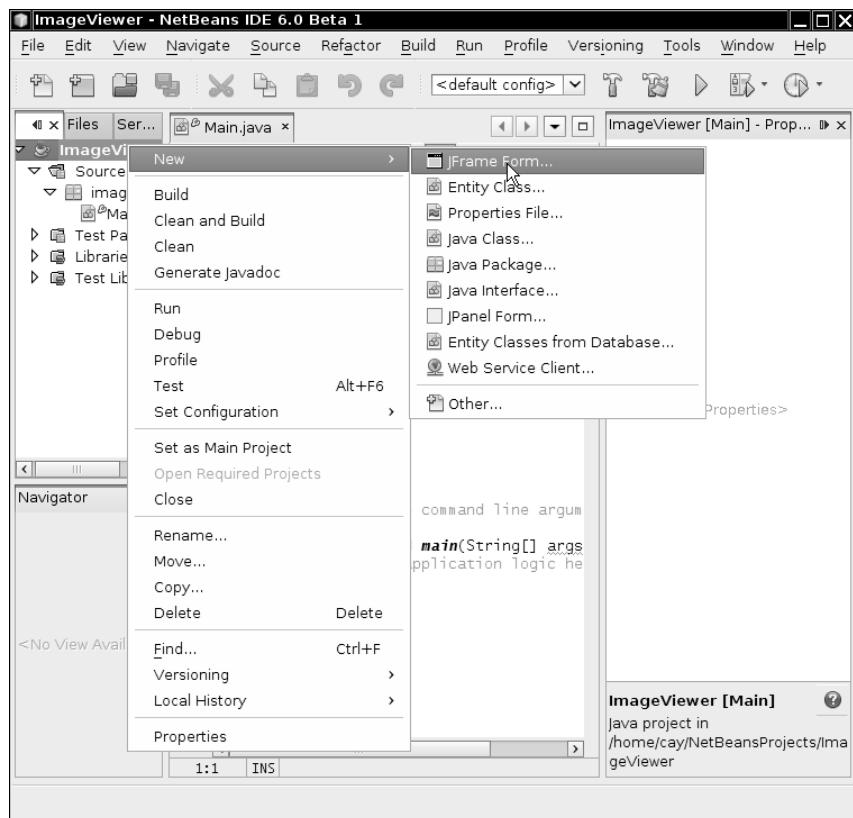
Rysunek 8.5 pokazuje efekt umieszczenia ziarnka *ImageViewerBean* na środkowej pozycji formatki.

Jeśli przyjrzymy się teraz bliżej wierszom programu w edytorze kodu, to zauważymy, że dodane w nim zostały instrukcje związane z umieszczeniem ziarnka w ramce (patrz rysunek 8.6). Fragment kodu w edytorze ujęty jest w linie komentarza zabraniające jego ręcznej modyfikacji. Jeśli mimo to wprowadzimy w nim zmiany, to zostaną one stracone w momencie gdy fragment ten zostanie automatycznie zaktualizowany na skutek zmian dokonanych w edytorze formatki.

Wybierzmy teraz myszą ziarnko umieszczone na formatce. Po prawej stronie edytora formatek znajduje się okno inspektora właściwości, które pokazuje listę właściwości ziarnka i ich bieżących wartości. Jest to jeden z najistotniejszych modułów narzędzi tworzenia aplikacji w oparciu o komponenty, ponieważ pozwala na określenie początkowych właściwości wykorzystywanych komponentów.

Rysunek 8.4.

Tworzenie formatki



Nie wszystkie narzędzia tworzenia aplikacji aktualizują kod programu na bieżąco. Mogą one generować kod programu dopiero po zakończeniu edycji lub serializować jedynie zmodyfikowane ziarnka czy tworzyć opis aplikacji w jeszcze inny sposób.

Na przykład eksperymentalne narzędzie Bean Builder z witryny <http://java.net/projects/bean-builder> pozwala projektować aplikacje wyposażone w graficzny interfejs użytkownika bez pisania jakiegokolwiek kodu.

Mechanizm JavaBeans nie wymusza określonej strategii implementacji narzędzi tworzenia aplikacji. Dostarcza jedynie tym narzędziom informacji o ziarnkach, które mogą zostać wykorzystane w taki lub inny sposób.

Inspektora właściwości możemy wykorzystać na przykład do zmiany właściwości text etykiety opisującej nasze ziarnko. Wystarczy po prostu wpisać tekst w polu tekstowym wartości — na przykład "Hello", a formatka zostanie natychmiast zaktualizowana (patrz rysunek 8.7).



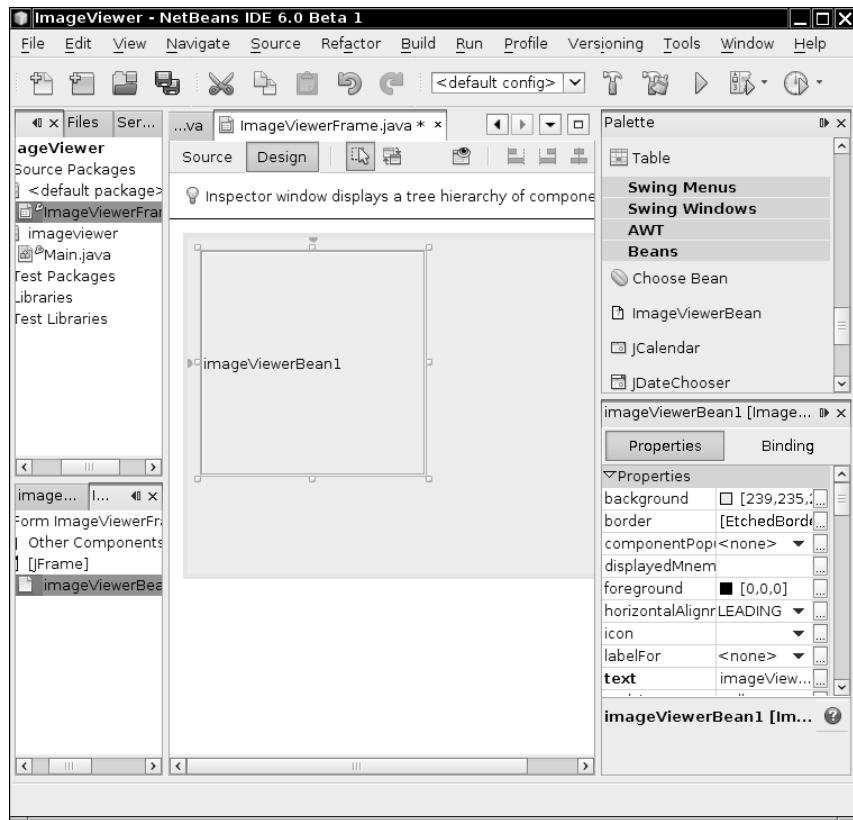
Gdy zmieniamy wartość właściwości, NetBeans aktualizuje kod źródłowy tworzonej aplikacji. Jeśli na przykład właściwości text nadamy wartość w postaci łańcucha Hello, to wywołanie

```
imageViewerBean.setText("Hello");
```

zostanie umieszczone w kodzie metody initComponents. Jak już wspomnieliśmy, inne narzędzia mogą korzystać z innych strategii przechowywania wartości właściwości.

**Rysunek 8.5.**

Ziarnko  
umieszczone  
na formacie



Oczywiście właściwości nie muszą byćłańcuchami znaków, mogą posiadać wartości dowolnego typu języka Java. Aby umożliwić użytkownikowi określanie wartości właściwości dowolnego typu, pakiety narzędziowe korzystają ze specjalizowanych edytorów właściwości. (Są to części pakietu narzędziowego bądź dostarczane są przez twórcę ziarnka. Sposób tworzenia własnych edytorów właściwości pokażemy w dalszej części tego rozdziału).

Działanie takiego edytora możemy zobaczyć na przykładzie właściwości `foreground`. Jej typem jest `Color`. Edytor koloru posiada pole tekstowe zawierającełańcuch `[0, 0, 0]` oraz przycisk oznaczony „...”, którego wybranie powoduje pojawięcie się okna wyboru kolorów. Jeśli wybierzemy w nim nowy kolor, to spowoduje to natychmiastową zmianę wartości właściwości — tekst etykiety zmieni kolor.

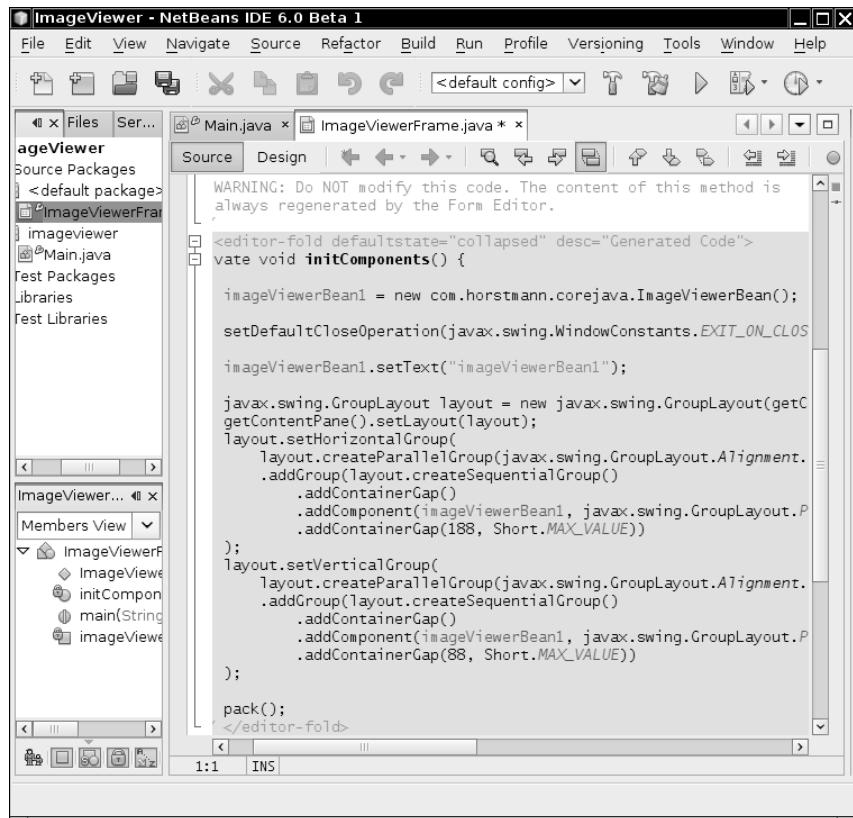
Za pomocą inspektora właściwości możemy w ten sam sposób zmienić wartość zaimplementowanej przez nas właściwości `fileName` ziarnka. Jeśli określmy nazwę pliku graficznego, to jego zawartość zostanie automatycznie wyświetlona przez ziarnko w oknie formatki.



Jeśli przyjrzymy się zawartości okna inspektora właściwości NetBeans, to zauważymy, że zawiera ono wiele właściwości, których przeznaczenie nie jest oczywiste — na przykład `focusCycleRoot` czy `iconTextGap`. Nasze ziarnko odziedziczyło je z klasy bazowej `JLabel`. W dalszej części rozdziału pokażemy, jak usunąć je z okna inspektora.

## Rysunek 8.6.

Kod źródłowy  
związanego  
ze wstawieniem  
ziarnka



Aby zakończyć projektowanie naszej aplikacji, umieścmy jeszcze ziarnko FilePickerBean w dolnej części ramki. Będziemy chcieli, by zmiana wartości jego właściwości fileName powodowała wyświetlenie przez aplikację zawartości nowego pliku graficznego. Jest to możliwe dzięki wykorzystaniu zdarzenia PropertyChange, które omówimy w dalszej części rozdziału.

Aby aplikacja zareagowała na to zdarzenie, musimy dla ziarnka FilePickerBean wybrać zakładkę *Events* w oknie jego inspektora właściwości. Następnie wybierzmy przycisk „...” towarzyszący pozycji *propertyChange*. Pojawi się okno dialogowe, które pokazuje, że ze zdarzeniem tym nie jest związana żadna procedura obsługi. Wybierzmy więc przycisk *Add* i wprowadźmy nazwę metody *loadImage* (patrz rysunek 8.8).

Przyjrzyjmy się teraz oknu edytora kodu. Pojawił się w nim kod obsługi zdarzenia oraz nowa metoda:

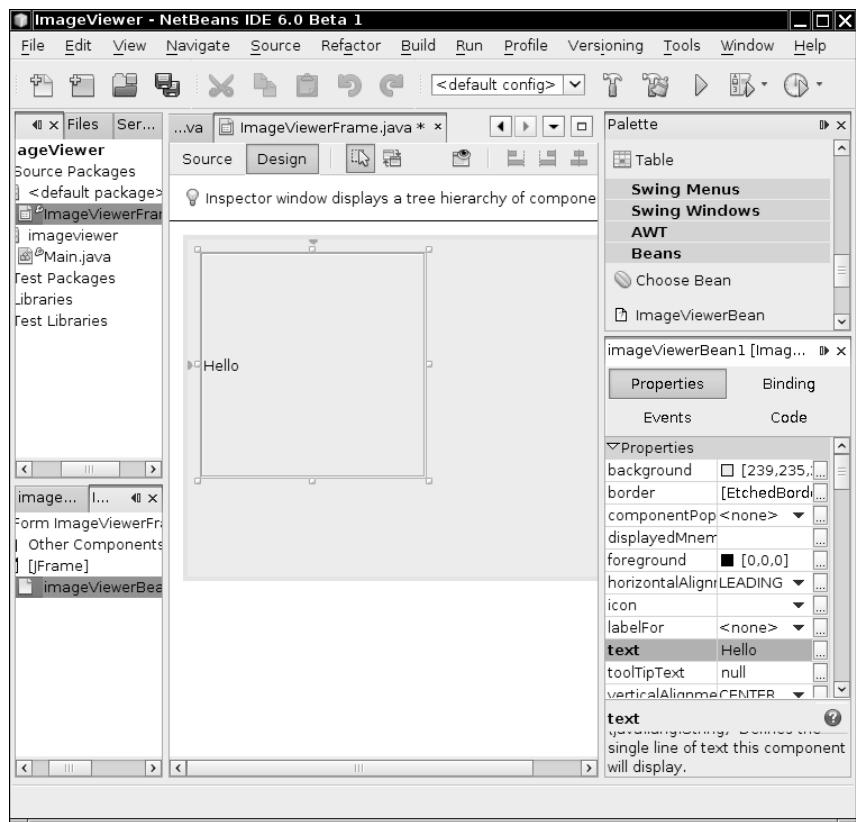
```
private void loadImage(java.beans.PropertyChange evt)
{
    // TODO Add your handling code here
}
```

Umieścimy w jej ciele poniższy wiersz kodu:

```
imageViewerBean1.setFileName(filePickerBean1.getFileName());
```

**Rysunek 8.7.**

Inspektor właściwości



Po skompilowaniu i uruchomieniu programu wybierzmy przycisk oznaczony etyktą „...” i następnie plik graficzny. Zostanie on wyświetlony przez aplikację (patrz rysunek 8.9).

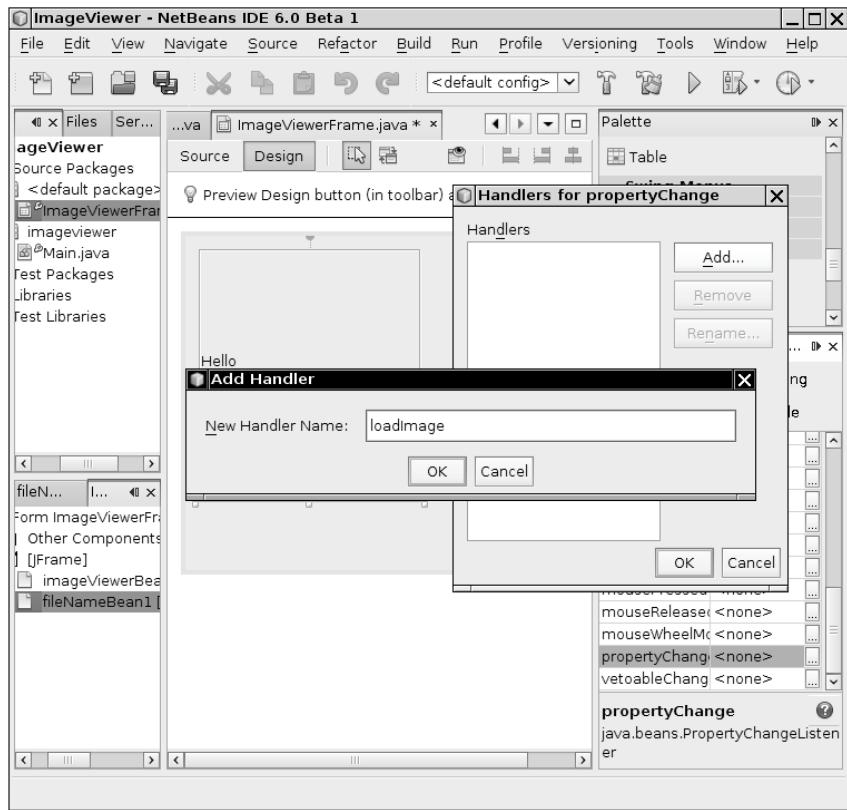
Pokazaliśmy więc, że aplikację w języku Java możemy utworzyć, składając ją z ziarenek, określając ich właściwości oraz pisząc stosunkowo małą ilość kodu obsługi zdarzeń.

## 8.4. Wzorce nazw właściwości ziarenek i zdarzeń

W niniejszym podrozdziale omówimy podstawowe zasady projektowania własnych ziarenek. Na wstępnie warto przede wszystkim podkreślić, że *nie* istnieje żadna „magiczna” klasa bazowa ziarka, która rozszerzać muszą tworzone przez nas ziarka. Ziarka posiadające wizualną reprezentację stanowią (bezpośrednio lub pośrednio) klasę pochodną klasy Component. Natomiast pozostałe ziarka nie muszą korzystać z żadnej określonej klasy bazowej. Zapamiętajmy więc, że ziarko jest *dowolną* klasą, którą potrafią wykorzystać narzędzia wizualnego tworzenia aplikacji. Narzędzia te, chcąc sprawdzić, czy klasa jest ziarkiem, nie poszukują żadnej klasy bazowej w hierarchii dziedziczenia danej klasy, ale analizują jedynie nazwy jej metod. Aby analiza ta była możliwa, nazwy metod klas ziarenek muszą być tworzone według pewnych wzorców.

Rysunek 8.8.

Dodawanie zdarzenia do ziarnka



Rysunek 8.9.

Działanie ziarnka  
*ImageViewerBean*



Istnieje klasa `java.beans.Beans`, której wszystkie metody są statyczne. Dlatego też dziedziczenie z niej jest raczej bezcelowe. Czasami można jednak spotkać przykłady rozszerzania tej klasy. Ponieważ jednak klasa ziarnka nie może jednocześnie rozszerzać klasy Beans i klasy Component, to oczywiście rozwiązanie takie nie jest możliwe dla ziarenek posiadających reprezentację wizualną. Klasa Beans zawiera metody przeznaczone do wykorzystania przez narzędzia tworzenia aplikacji — na przykład umożliwiające sprawdzenie, czy narzędzie działa w trybie projektowania, czy w trybie wykonania ziarnka.

Inne języki programowania umożliwiające wizualne projektowanie aplikacji, na przykład Visual Basic czy C#, posiadają specjalne słowa kluczowe, takie jak *Property* czy *Event*, umożliwiające zdefiniowanie właściwości i zdarzeń związanych z danym komponentem. Projektanci języka Java zdecydowali się jednak nie wprowadzać osobnych konstrukcji języka, które umożliwiłyby programowanie wizualne. Potrzebne więc było inne rozwiązanie, które pozwoliłoby na analizę ziarnka w celu uzyskania informacji o jego właściwościach i zdarzeniach z nim związanych. Istnieją dwa alternatywne sposoby dostarczenia tej informacji. Twórca ziarnka może zastosować standardowe wzorce nazewnictwa właściwości i zdarzeń, które narzędziem tworzenia aplikacji odnajdzie i przeanalizuje, korzystając z mechanizmu refleksji. Druga z metod polega na dołączeniu *klasy informacyjnej ziarnka*, która dostarcza narzędziom informacji o właściwościach i zdarzeniach związanych z ziarnkiem. Na początku korzystać będziemy ze wzorców nazw, ponieważ metoda ta jest prostsza w użyciu. Później przedstawimy także sposób tworzenia klasy informacyjnej ziarnka.



Choć dokumentacja platformy Java używa w odniesieniu do wzorców nazw terminu „wzorce projektowe”, są to tylko konwencje tworzenia nazw i nie mają nic wspólnego z wzorcami projektowymi stosowanymi w projektowaniu obiektowym.

Wzorzec nazw właściwości przedstawia się następująco: dowolna para metod

```
public Type getPropertyName()
public void setPropertyName(Type newValue)
```

odpowiada właściwości o nazwie *PropertyName* i typie *Type*, której wartość może być odczytywana i zapisywana.

W przypadku naszego ziarnka *ImageViewerBean* istnieje tylko jedna taka właściwość (określająca nazwę wyświetlanego pliku graficznego) o następujących metodach:

```
public String getFileName()
public void setFileName(String newValue)
```

Jeśli zdefiniujemy jedynie metodę *get*, to możliwy będzie tylko odczyt wartości właściwości. Podobnie, gdy dostarczymy jedynie metodę *set*, to możliwy będzie tylko zapis wartości właściwości.



Metody *set* i *get* nie służą jedynie do nadawania i pobierania wartości prywatnej składowej klasy. Podobnie jak każda inna metoda języka Java, wykonywać mogą dowolne operacje. Na przykład metoda *setFileName* klasy *ImageViewerBean* nie tylko nadaje wartość zmiennej *fileName*, ale także otwiera plik i ładuje jego zawartość.



W językach Visual Basic i C# właściwości także posiadają metody *get* i *set*. Jednak w obu językach należy jawnie zdefiniować właściwość zamiast oczekiwania, że narzędzie wizualnego projektowania aplikacji samo odgadnie intencje programisty na podstawie analizy nazw metod. Rozwiążanie zastosowane w tych językach posiada jeszcze inną zaletę. Umieszczenie nazwy właściwości po lewej stronie operatora przypisania wywołuje metodę *set* dla danej właściwości. Natomiast umieszczenie nazwy właściwości w dowolnym wyrażeniu wywołuje metodę *get*. W języku Visual Basic możemy napisać na przykład

```
imageBean.fileName = "corejava.gif"
```

zamiast

```
imageBean.setFileName("corejava.gif");
```

Składnię taką rozważano także dla języka Java, ale ostatecznie uznano, że ukrywanie wywołań metod za pomocą składni nie jest dobrym pomysłem.

Od reguły tworzenia nazw metod za pomocą prefiksów get/set istnieje jeden wyjątek. Jeśli właściwość posiada wartości typu boolean, to korzystamy z prefiksów is/set, jak pokazano poniżej:

```
public boolean isPropertyName()
public void setPropertyName(boolean b)
```

Na przykład dla ziarnka animacji może istnieć właściwość running o następujących metodach:

```
public boolean isRunning()
public void setRunning(boolean b)
```

Metoda setRunning wykorzystywana będzie do uruchomienia bądź zatrzymania animacji, natomiast metoda isRunning umożliwiać będzie sprawdzenie jej bieżącego stanu.



W przypadku metody dostępu do właściwości typu boolean dozwolone jest użycie prefiksu get (np. getRunning), ale zalecane jest stosowanie prefiksu is.

Należy zwrócić uwagę na sposób wykorzystania wielkich liter w nazwach metod. Nazwą właściwości w naszym przykładzie jest fileName rozpoczynająca się od małej litery f, chociaż nazwy jej metod zawierają wielką literę F (getFileName, setFileName). Podczas analizy ziarnka pierwsza litera następująca w nazwie metody po prefiksie get lub set zawsze zamieniana jest na małą literę. Uzyskane w ten sposób nazwy właściwości są zgodne z konwencjami przyjętymi dla nazewnictwa zmiennych i metod.

Jeśli natomiast pierwsze *dwie* litery nazwy są duże (tak jak na przykład w nazwie getURL), to wtedy pierwsza litera nazwy właściwości *nie* jest zamieniana na małą. Nazwa właściwości uRL wyglądałaby nienaturalnie.



Zastanówmy się nad przypadkiem, gdy nasza klasa ziarnka posiada metody o nazwach get i set, które nie są związane z właściwością ziarnka i nie powinny być wykorzystywane przez inspektora właściwości. W przypadku gdy daną klasę tworzymy sami od podstaw, możemy oczywiście uniknąć takiej sytuacji, zmieniając nazwy metod. Jeśli jednak klasa dziedziczy po innej klasie, na przykład gdy ziarnko rozszerza klasę JPanel lub JLabel, to w oknie inspektora pojawi się wiele zbędnych właściwości. Przykład ten pokazuje, że wzorce nazw są mało precyzyjną metodą projektowania właściwości ziarenek. W dalszej części tego rozdziału pokażemy, w jaki sposób zastąpić proces automatycznego wykrywania właściwości na podstawie nazw za pomocą informacji ziarka precyzyjnie określających, które metody odpowiadają właściwościom.

W przypadku zdarzeń, wzorzec nazw jest jeszcze prostszy. Przyjmuje się, że ziarnko generuje zdarzenia, jeśli dostarczymy metod dodawania i usuwania obiektów nasłuchujących. Wszystkie nazwy klas zdarzeń muszą kończyć się przyrostkiem Event, a same klasy muszą rozszerzać klasę EventObject.

Załóżmy, że ziarnko generuje zdarzenia typu *EventNameEvent*. Wtedy interfejs obiektu nasłuchującego musi posiadać nazwę *EventNameListener*, a nazwy metod umożliwiających dodanie i usunięcie obiektu nasłuchującego określone są, jak poniżej.

```
public void addEventNameListener(EventNameListener e)
public void removeEventNameListener(EventNameListener e)
public EventNameListener getEventNameListeners()
```

Kod naszego ziarnka *ImageViewerBean* nie posiada żadnych zdarzeń. Jednak większość komponentów Swing generuje zdarzenia. Na przykład klasa *AbstractButton* generuje obiekty *ActionEvent* i posiada następujące metody zarządzania obiektami nasłuchującymi:

```
public void addActionListener(ActionListener e)
public void removeActionListener(ActionListener e)
ActionListener[] getActionListeners()
```



Jeśli stworzona przez nas klasa zdarzeń nie jest pochodną klasy *EventObject*, to może się zdarzyć, że taki kod zostanie skompilowany poprawnie, ponieważ żadna z metod klasy *EventObject* nie jest w rzeczywistości wykorzystywana. Jednak ziarnko nie będzie działać poprawnie, ponieważ mechanizm introspekcji nie rozpozna zdarzeń.

## 8.5. Typy właściwości ziarenek

Złożone ziarnko posiada zwykle wiele różnych właściwości i zdarzeń. Właściwości ziarka mogą być równie proste jak właściwość *fileName* pokazanego wcześniej ziarka *ImageViewerBean* lub bardziej złożone — na przykład klasy *Color* lub tablica punktów wykresu — oba te przypadki pokażemy w dalszej części rozdziału. Specyfikacja JavaBeans wyróżnia cztery typy właściwości, które zilustrujemy przykładami.

### 8.5.1. Właściwości proste

Właściwość prosta posiada jedną wartość, taką jak łańcuch znaków bądź liczba. Właściwość *fileName* ziarka *ImageViewerBean* jest właśnie przykładem właściwości prostej. Korzystanie z takich właściwości wymaga jedynie zaimplementowania metod *get* i *set*. Jeśli przyjrzymy się kodowi z listingu 8.1, to zobaczymy, że zaimplementowanie właściwości *fileName* sprowadziło się do kodu poniższych metod:

```
public void setFileName(String f)
{
    fileName = f;
    image = . . .
    repaint();
}

public String getFileName()
{
    if (file == null) return "";
    else return file.getPath();
}
```

## 8.5.2. Właściwości indeksowane

Właściwość indeksowana specyfikuje tablicę. W przypadku właściwości indeksowanej musimy dostarczyć dwie pary metod get i set: jedną dla tablicy i drugą dla jej elementów. Ich nazwy tworzymy według poniższego wzorca:

```
Type[] getPropertyNames()
void setPropertyNames(Type[] x)
Type getProperty(int i)
void setProperty(int i, Type x)
```

Na przykład ziarnko FilePickerBean używa właściwości indeksowanej dla rozszerzeń nazw plików. Dostarcza ono cztery poniższe metody:

```
private String[] extensions;

public String[] getExtensions() { return extensions; }
public void setExtensions(String[] newValue) { extensions = newValue; }
public String getExtensions(int i)
{
    if (0 <= i && i < extensions.length) return extensions[i];
    else return "";
}
public void setExtensions(int i, String newValue)
{
    if (0 <= i && i < extensions.length) extensions[i] = value;
}
```

Metoda `setPropertyNames(int, Type)` nie może być wykorzystana do zwiększenia rozmiaru tablicy. W tym celu musimy utworzyć nową tablicę i przekazać ją metodzie `setPropertyName(Type[])`.

## 8.5.3. Właściwości powiązane

Właściwości powiązane informują obiekty nasłuchujące o zmianie swojej wartości. Przykładem właściwości powiązanej jest właściwość `fileName` ziarnka `FilePickerBean`. Zmiana nazwy pliku powoduje automatycznie zawiadomienie ziarnka `ImageViewerBean` i załadowanie nowego pliku.

Tworząc właściwość powiązaną, musimy zaimplementować dwa mechanizmy.

1. Gdy wartość takiej właściwości zmienia się, to musi ona wysłać zdarzenie `PropertyChange` do wszystkich zarejestrowanych obiektów nasłuchujących. Zmiana wartości właściwości może zajść na skutek wywołania metody `set` w programie lub wykonania akcji przez użytkownika polegającej na przykład na edycji tekstu bądź wyborze pliku.
2. Aby obiekty nasłuchujące mogły zostać zarejestrowane, ziarnko musi implementować poniższe metody:

```
void addPropertyChangeListener(PropertyChangeListener listener)
void removePropertyChangeListener(PropertyChangeListener listener)
```

Zaleca się również (ale nie jest to wymagane), aby ziarnko dostarczało metodę

```
PropertyChangeListener[] getPropertyChangeListeners()
```

Pakiet `java.beans` zawiera klasę o nazwie `PropertyChangeSupport`, która umożliwia zarządzanie obiektami nasłuchującymi. Aby z niej skorzystać, ziarnko powinno zawierać zmienną składową typu tej klasy zainicjowaną w sposób pokazany poniżej:

```
private PropertyChangeSupport changeSupport
    = new PropertyChangeSupport(this);
```

Zadanie dodawania i usuwania obiektów nasłuchujących zmiany właściwości delegujemy do obiektu `changeSupport`.

```
public void addPropertyChangeListener(PropertyChangeListener listener)
{
    changeSupport.addPropertyChangeListener(listener);
}

public void removePropertyChangeListener(PropertyChangeListener listener)
{
    changeSupport.removePropertyChangeListener(listener);
}

public PropertyChangeListener[] getPropertyChangeListeners()
{
    return changeSupport.getPropertyChangeListeners();
}
```

Kiedy zajdzie zmiana wartości właściwości, wywołamy metodę `firePropertyChange` klasy `PropertyChangeSupport`, aby powiadomić o zdarzeniu wszystkie zarejestrowane obiekty nasłuchujące. Metoda ta posiada trzy parametry: nazwę właściwości oraz jej poprzednią i nową wartość. Oto schemat implementacji typowej metody `set` dla właściwości powiązanej:

```
public void setValue(Type newValue)
{
    Type oldValue = getValue();
    value = newValue;
    changeSupport.firePropertyChange("propertyName", oldValue, newValue);
}
```

Aby zatem powiadomić obiekty nasłuchujące o zmianie wartości właściwości indeksowanej, używamy poniższego wywołania:

```
changeSupport.firePropertyChange("propertyName",
    oldValue, newValue);
```



Jeśli ziarnko stanowi rozszerzenie jakiejkolwiek klasy biblioteki Swing, która z kolei dziedziczy z klasy `Component`, to wtedy nie musimy implementować metod `addPropertyChangeListener`, `removePropertyChangeListener` ani `getPropertyChangeListeners`. Metody te posiadają już implementację w klasie bazowej `Component`. Aby zawiadomić obiekty nasłuchujące o zmianie wartości właściwości, wywołujemy wtedy metodę `firePropertyChange` klasy `JComponent`. Niestety powiadomianie o zmianie wartości właściwości indeksowanej nie jest obsługiwane.

Jeśli inne ziarnka chcą być powiadamiane o zmianie właściwości, to muszą implementować interfejs PropertyChangeListener. Zawiera on tylko poniższą metodę:

```
void propertyChange(PropertyChangeEvent event)
```

Obiekt PropertyChangeEvent zawiera starą i nową wartość właściwości, które możemy uzyskać za pomocą metod getPropertyName, getOldValue i getNewValue.

Jeśli typ właściwości nie jest klasą, to uzyskujemy w ten sposób obiekty klasy obudowującej.

## 8.5.4. Właściwości ograniczone

*Właściwości ograniczone* charakteryzują się tym, że dowolny z obiektów nasłuchujących może zgłosić wejście do proponowanej zmiany wartości właściwości, wymuszając tym samym po-zostawienie dotychczasowej wartości. Biblioteka języka Java zawiera tylko kilka przykładów zastosowania właściwości ograniczonych. Jednym z nich jest właściwość closed klasy JInternalFrame. Gdy wywołana zostanie metoda setClosed(true), to zawiadomione zostają wszystkie obiekty VetoableChangeListener. Jeśli którykolwiek z nich wygeneruje wyjątek PropertyVetoException, to wartość właściwości closed *nie* zostanie zmieniona, a metoda setClosed zgłosi ten sam wyjątek. Obiekt VetoableChangeListener może na przykład zgłosić wejście do zamknięcia ramki, jeśli jej zawartość nie została zapisana.

Aby korzystać z właściwości ograniczonych, ziarnko dysponować musi następującymi metodami rejestracji obiektów VetoableChangeListener:

```
public void addVetoableChangeListener(VetoableChangeListener listener);  
public void removeVetoableChangeListener(VetoableChangeListener listener);
```

Powinno również posiadać metodę zwracającą wszystkie obiekty nasłuchujące:

```
VetoableChangeListener[] getVetoableChangeListeners()
```

Podobnie jak w przypadku zarządzania obiektami nasłuchującymi zmiany właściwości, tak i w przypadku obiektów nasłuchujących zmiany, która może być odrzucona, dostępna jest odpowiednia klasa VetoableChangeSupport. Ziarnko korzystające z właściwości ograniczonych powinno zawierać obiekt tej klasy.

```
private VetoableChangeSupport vetoSupport  
= new VetoableChangeSupport(this);
```

Rejestrowanie i usuwanie obiektów nasłuchujących będzie oddelegowane do tego obiektu.

```
public void addVetoableChangeListener(VetoableChangeListener listener)  
{  
    vetoSupport.addVetoableChangeListener(listener);  
}  
  
public void removeVetoableChangeListener(VetoableChangeListener listener)  
{  
    vetoSupport.removeVetoableChangeListener(listener);  
}
```

Aby zaktualizować wartość właściwości ograniczonej, ziarnko stosuje metodę, na którą składają się trzy fazy.

- 1.** W pierwszej powiadamia wszystkie obiekty nasłuchujące właściwości ograniczonej o zamiarze zmiany jej wartości. (Korzysta w tym celu z metody fireVetoableChange klasy VetoableChangeSupport).
- 2.** Jeśli żaden z powiadomionych obiektów nasłuchujących nie wyrzucił w odpowiedzi wyjątku PropertyVetoException, to w drugiej fazie zmienia wartość.
- 3.** Powiadamia wszystkie obiekty nasłuchujące właściwości powiązanej, aby *potwierdzić* dokonanie zmiany.

Na przykład:

```
public void setValue(Type newValue) throws PropertyVetoException
{
    Type oldValue = getValue();
    vetoSupport.fireVetoableChange("value", oldValue, newValue);
    // jeśli zmiana nie została odrzucona
    value = newValue;
    changeSupport.firePropertyChange("value", oldValue, newValue);
}
```

Ważne jest, aby nie zmieniać wartości właściwości, jeśli nie wyraziły na nią zgody wszystkie obiekty nasłuchujące właściwości ograniczonej. Także obiekt nasłuchujący właściwości ograniczonej nie powinien zakładać, że jeśli zaakceptuje zmianę, to ona nastąpi. Jedyny niezwodny sposób powiadomienia o tym, czy zmiana rzeczywiście zaszła, możliwy jest za pośrednictwem obiektu nasłuchującego właściwości.



Jeśli ziarnko rozszerza klasę JComponent, to nie potrzebujemy osobnego obiektu klasy VetoableChangeSupport. Wystarczy wywołać metodę fireVetoableChange klasy bazowej JComponent. Zauważmy jednak, że dla JComponent nie możemy zainstalować obiektu nasłuchującego zmiany wartości wybranej właściwości, która może zostać zawetowana. Musimy nasłuchiwać zmian wszystkich takich właściwości.

Listing 8.2 zawiera kompletny kod źródłowy ziarnka FilePickerBean. Ziarnko FilePickerBean posiada indeksowaną właściwość extensions oraz właściwość ograniczoną filename. Ponieważ klasa FilePickerBean rozszerza klasę JPanel, nie musimy używać klasy PropertyChangeSupport. W zamian korzystamy z zarządzania obiektami nasłuchującymi zaimplementowanego w klasie JPanel.

**Listing 8.2.** filePicker/FilePickerBean.java

```
package filePicker;

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

```
/**  
 * Ziarnko wyboru pliku.  
 * @version 1.3 | 2012-06-10  
 * @author Cay Horstmann  
 */  
public class FilePickerBean extends JPanel  
{  
    private static final int XPREFSIZE = 200;  
    private static final int YPREFSIZE = 20;  
  
    private JButton dialogButton;  
    private JTextField nameField;  
    private JFileChooser chooser;  
    private String[] extensions = { "gif", "png" };  
  
    public FilePickerBean()  
    {  
        dialogButton = new JButton("...");  
        nameField = new JTextField(30);  
  
        chooser = new JFileChooser();  
        setPreferredSize(new Dimension(XPREFSIZE, YPREFSIZE));  
  
        setLayout(new GridBagLayout());  
        GridBagConstraints gbc = new GridBagConstraints();  
        gbc.weightx = 100;  
        gbc.weighty = 100;  
        gbc.anchor = GridBagConstraints.WEST;  
        gbc.fill = GridBagConstraints.BOTH;  
        gbc.gridwidth = 1;  
        gbc.gridheight = 1;  
        add(nameField, gbc);  
  
        dialogButton.addActionListener(new ActionListener()  
        {  
            public void actionPerformed(ActionEvent event)  
            {  
                chooser.setFileFilter(new FileNameExtensionFilter(Arrays.toString(extensions),  
                    extensions));  
                int r = chooser.showOpenDialog(null);  
                if (r == JFileChooser.APPROVE_OPTION)  
                {  
                    File f = chooser.getSelectedFile();  
                    String name = f.getAbsolutePath();  
                    setFileName(name);  
                }  
            }  
        });  
        nameField.setEditable(false);  
  
        gbc.weightx = 0;  
        gbc.anchor = GridBagConstraints.EAST;  
        gbc.fill = GridBagConstraints.NONE;  
        gbc.gridx = 1;  
        add(dialogButton, gbc);  
    }  
}
```

```

    /**
     * Metoda set właściwości fileName.
     * @param newValue nowa nazwa pliku
     */
    public void setFileName(String newValue)
    {
        String oldValue = nameField.getText();
        nameField.setText(newValue);
        firePropertyChange("fileName", oldValue, newValue);
    }

    /**
     * Metoda get właściwości fileName.
     * @return nazwa wybranego pliku
     */
    public String getFileName()
    {
        return nameField.getText();
    }

    /**
     * Metoda get właściwości extensions.
     * @return domyślne rozszerzenia nazwy pliku
     */
    public String[] getExtensions()
    {
        return extensions;
    }

    /**
     * Metoda set właściwości extensions.
     * @param newValue nowe domyślne rozszerzenia nazwy pliku
     */
    public void setExtensions(String[] newValue)
    {
        extensions = newValue;
    }

    /**
     * Zwraca jedną z wartości właściwości extensions.
     * @param i indeks wartości właściwości
     * @return wartość o podanym indeksie
     */
    public String getExtensions(int i)
    {
        if (0 <= i && i < extensions.length) return extensions[i];
        else return "";
    }

    /**
     * Nadaje wartość właściwości extensions.
     * @param i indeks wartości właściwości
     * @param newValue nowa wartość o podanym indeksie
     */
    public void setExtensions(int i, String newValue)
    {
        if (0 <= i && i < extensions.length) extensions[i] = newValue;
    }
}

```

**API `java.beans.PropertyChangeListener 1.1`**

- `void propertyChange(PropertyChangeEvent event)`

metoda wywoływana na skutek zdarzenia zmiany wartości właściwości.

**API `java.beans.PropertyChangeSupport 1.1`**

- `PropertyChangeSupport(Object sourceBean)`

tworzy obiekt zarządzający obiektami nasłuchującymi zmiany właściwości powiązanej danego ziarnka.

- `void addPropertyChangeListener(PropertyChangeListener listener)`

- `void addPropertyChangeListener(String propertyName, PropertyChangeListener listener) 1.2`

rejestruje obiekt nasłuchujący zainteresowany zdarzeniem zmiany wartości wszystkich właściwości powiązanych lub tylko właściwości powiązanej o podanej nazwie.

- `void removePropertyChangeListener(PropertyChangeListener listener)`

- `void removePropertyChangeListener(PropertyChangeListener listener) 1.2`

usuwa zarejestrowany wcześniej obiekt nasłuchujący zmiany wartości właściwości powiązanej.

- `void firePropertyChange(String propertyName, Object oldValue, Object newValue)`

- `void firePropertyChange(String propertyName, int oldValue, int newValue) 1.2`

- `void firePropertyChange(String propertyName, boolean oldValue, boolean newValue) 1.2`

wysyła `PropertyChangeEvent` zarejestrowanym obiektom nasłuchującym.

- `void fireIndexedPropertyChange(String propertyName, int index, Object oldValue, Object newValue) 5.0`

- `void fireIndexedPropertyChange(String propertyName, int index, int oldValue, int newValue) 5.0`

- `void fireIndexedPropertyChange(String propertyName, int index, boolean oldValue, boolean newValue) 5.0`

wysyła `IndexedPropertyChangeEvent` zarejestrowanym obiektom nasłuchującym.

- `PropertyChangeListener[] getPropertyChangeListeners() 1.4`

- `PropertyChangeListener[] getPropertyChangeListeners(String propertyName) 1.4`

zwracają obiekty nasłuchujące zmiany wartości wszystkich właściwości powiązanych lub tylko właściwości powiązanej o podanej nazwie.

**API** `java.beans.PropertyChangeEvent 1.1`

- `PropertyChangeEvent(Object source, String propertyName, Object oldValue, Object newValue)`  
tworzy obiekt klasy `PropertyChangeEvent` informujący o zmianie wartości właściwości z `oldValue` na `newValue`.
- `Object getNewValue()`  
zwraca nową wartość właściwości.
- `Object getOldValue()`  
zwraca poprzednią wartość właściwości.
- `String getPropertyName()`  
zwraca nazwę właściwości.

**API** `java.beans.IndexedPropertyChangeEvent 5.0`

- `IndexedPropertyChangeEvent(Object sourceBean, String propertyName, int index, Object oldValue, Object newValue)`  
tworzy nowy obiekt `IndexedPropertyChangeEvent` informujący o zmianie wartości właściwości z `oldValue` na `newValue` dla podanej wartości indeksu.
- `int getIndex()`  
zwraca indeks pozycji, na której zaszła zmiana.

**API** `java.beans.VetoableChangeListener 1.1`

- `void vetoableChange(PropertyChangeEvent event)`  
metoda wywoływana, gdy ma nastąpić zmiana wartości właściwości. Powinna wyrzucić wyjątek `PropertyVetoException`, jeśli nie akceptuje proponowanej zmiany.

**API** `java.beans.VetoableChangeSupport 1.1`

- `VetoableChangeSupport(Object sourceBean)`  
tworzy obiekt zarządzający obiektami nasłuchującymi zmian wartości właściwości ograniczonej danego ziarnka.
- `void addVetoableChangeListener(VetoableChangeListener listener)`
- `void addVetoableChangeListener(String propertyName, VetoableChangeListener listener) 1.2`  
rejestruje obiekt nasłuchujący zmian wartości wszystkich właściwości ograniczonych lub tylko właściwości ograniczonej o podanej nazwie.
- `void removeVetoableChangeListener(VetoableChangeListener listener)`
- `void removeVetoableChangeListener(String propertyName, VetoableChangeListener listener) 1.2`  
usuwa zarejestrowany obiekt nasłuchujący właściwości ograniczonej.

- void fireVetoableChange(String propertyName, Object oldValue, Object newValue)
- void fireVetoableChange(String propertyName, int oldValue, int newValue) **1.2**
- void fireVetoableChange(String propertyName, boolean oldValue, boolean newValue) **1.2**  
wysyła zdarzenie PropertyChangeEvent do zarejestrowanych obiektów nasłuchujących.
- VetoableChangeListener[] getVetoableChangeListeners() **1.4**
- VetoableChangeListener[] getVetoableChangeListeners(String propertyName) **1.4**  
zwraca obiekty nasłuchujące zmian wartości wszystkich właściwości ograniczonych lub tylko właściwości ograniczonej o podanej nazwie.

**API javax.awt.Component 1.0**

- void addPropertyChangeListener(PropertyChangeListener listener) **1.2**
- void addPropertyChangeListener(String propertyName, PropertyChangeListener listener) **1.2**  
rejestruje obiekt nasłuchujący zmian wszystkich właściwości powiązanych lub tylko właściwości powiązanej o podanej nazwie.
- void removePropertyChangeListener(PropertyChangeListener listener) **1.2**
- void removePropertyChangeListener(String propertyName, PropertyChangeListener listener) **1.2**  
usuwa zarejestrowany wcześniej obiekt nasłuchujący zmiany wartości właściwości powiązanej.
- void firePropertyChange(String propertyName, Object oldValue, Object newValue) **1.2**  
wysyła zdarzenie PropertyChangeEvent zarejestrowanym obiektem nasłuchującym.

**API javax.swing.JComponent 1.2**

- void addVetoableChangeListener(VetoableChangeListener listener)  
rejestruje obiekt nasłuchujący VetoableChangeListener.
- void removeVetoableChangeListener(VetoableChangeListener listener)  
usuwa zarejestrowany wcześniej obiekt VetoableChangeListener.
- void fireVetoableChange(String propertyName, Object oldValue, Object newValue) wysyła VetoableChangeEvent  
zarejestrowanym obiektem nasłuchującym.

**API java.beans.PropertyVetoException 1.1**

- PropertyVetoException(String reason, PropertyChangeEvent event)  
tworzy obiekt klasy PropertyVetoException.

- `PropertyChangeEvent getPropertyChangeEvent()`  
zwraca obiekt `PropertyChangeEvent` wykorzystywany do tworzenia wyjątku.

## 8.6. Klasa informacyjna ziarnka

Zastosowanie standardowych konwencji nazw dla składowych ziarnka umożliwia narzędziom tworzenia aplikacji skorzystanie z mechanizmu refleksji w celu ustalenia właściwości, zdarzeń i metod ziarnka. Rozwiążanie takie z pewnością ułatwia na początku programowanie ziarnek, ale z czasem staje się także ograniczeniem. Gdy powstają coraz bardziej skomplikowane ziarnka, to konwencje nazw oraz wykorzystanie refleksji okazują się niewystarczające. Co więcej, kod ziarnka może zawierać także metody `get/set`, które *nie* opisują właściwości ziarnka.

Jeśli potrzebny jest bardziej uniwersalny mechanizm opisu ziarnka, należy zdefiniować obiekt implementujący interfejs `BeanInfo`. Po zaimplementowaniu tego interfejsu narzędzia wizualnego tworzenia aplikacji będą używać jego metod w celu ustalenia właściwości ziarnka.

Nazwa klasy informacyjnej ziarnka musi zostać utworzona przez dodanie do nazwy klasy ziarnka przyrostka `BeanInfo`. Na przykład klasa informacyjna ziarnka `ImageViewerBean` *musi* posiadać nazwę `ImageViewerBeanBeanInfo`. Klasa ta powinna także należeć do tego samego pakietu co klasa ziarnka.

Zwykle nie będziemy tworzyć klasy implementującej interfejs `BeanInfo` od podstaw, lecz wykorzystamy klasę `SimpleBeanInfo`, która posiada domyślne implementacje wszystkich metod interfejsu.

Najczęstszym powodem tworzenia klasy implementującej interfejs `BeanInfo` jest udostępnienie właściwości ziarnka. Dla każdej właściwości tworzymy obiekt `PropertyDescriptor`, dostarczając jego konstruktorowi nazwę właściwości i klasę ziarnka, które zawiera tę właściwość.

```
PropertyDescriptor descriptor = new PropertyDescriptor("fileName",
    ➔ImagePickerBean.class);
```

Następnie implementujemy metodę `getPropertyDescriptors` interfejsu `BeanInfo` tak, aby zwróciła tablicę wszystkich deskryptorów właściwości.

Załóżmy na przykład, że ziarnko `ImageViewerBean` powinno ukryć wszystkie właściwości odziedziczone po klasie bazowej `JLabel` i udostępnić jedynie właściwość `fileName`. W tym celu należy zdefiniować następującą klasę informacyjną:

```
// klasa informacyjna ziarnka ImageViewerBean
public class ImageViewerBeanBeanInfo extends SimpleBeanInfo
{
    private PropertyDescriptor[] propertyDescriptors;

    public ImageViewerBeanBeanInfo()
    {
        try
```

```

    {
        propertyDescriptors = new PropertyDescriptor[]
        {
            new PropertyDescriptor("fileName", FilePickerBean.class);
        }
    }
    catch (IntrospectionException e)
    {
        e.printStackTrace();
    }
}

public PropertyDescriptor[] getPropertyDescriptors()
{
    return propertyDescriptors;
}
}

```

Pozostałe metody interfejsu BeanInfo również zwracają tablice deskryptorów EventSetDescriptor i MethodDescriptor, ale są rzadziej używane. Jeśli jedna z tych metod zwróci wartość null (tak zaimplementowane są metody klasy SimpleBeanInfo), to wtedy zastosowanie znajduje standardowa konwencja nazw. Natomiast, gdy przesłoniemy metodę klasy SimpleBeanInfo własną implementacją zwracającą tablicę różną od null, to tablica ta musi zawierać *wszystkie* właściwości, zdarzenia lub metody.



Czasami zachodzi potrzeba stworzenia ogólnego kodu potrafiącego wykryć właściwości lub zdarzenia dowolnego ziarnka. Metoda statyczna getBeanInfo należąca do klasy Introspector tworzy klasę informacyjną ziarnka.

Inną przydatną metodą interfejsu BeanInfo jest getIcon, która pozwala nadać ziarnku własną ikonę. Ikona ta wykorzystywana jest zwykle przez narzędzie tworzenia aplikacji dla reprezentacji ziarnka w jednej z palet ziarenek. W rzeczywistości możemy nawet wyspecyfikować cztery różne mapy bitowe ikony. Interfejs BeanInfo definiuje cztery stałe dla standaryzowanych rozmiarów map bitowych ikon.

```

ICON_COLOR_16X16
ICON_COLOR_32X32
ICON_MONO_16X16
ICON_MONO_32X32

```

Poniżej przedstawiamy przykład wykorzystania metody loadImage klasy SimpleBeanInfo w celu dodania ikony ziarnka:

```

public class ImageViewerBeanBeanInfo extends SimpleBeanInfo
{
    private Image iconColor16;
    private Image iconColor32;
    private Image iconMono16;
    private Image iconMono32;

    public ImageViewerBeanBeanInfo()
    {
        iconColor16 = loadImage("ImageViewerBean_COLOR_16x16.gif");
        iconColor32 = loadImage("ImageViewerBean_COLOR_32x32.gif");
    }
}

```

```

        iconMono16 = loadImage("ImageViewerBean_MONO_16x16.gif");
        iconMono32 = loadImage("ImageViewerBean_MONO_32x32.gif");
    }

    public Image getIcon(int iconType)
    {
        if (iconType == BeanInfo.ICON_COLOR_16x16) return iconColor16;
        else if (iconType == BeanInfo.ICON_COLOR_32x32) return iconColor32;
        else if (iconType == BeanInfo.ICON_MONO_16x16) return iconMono16;
        else if (iconType == BeanInfo.ICON_MONO_32x32) return iconMono32;
        else return null;
    }
}

```

**API** **java.beans.Introspector 1.1**

- static BeanInfo getBeanInfo(Class<?> beanClass)  
zwraca obiekt klasy informacyjnej ziarnka.

**API** **java.beans.BeanInfo 1.1**

- PropertyDescriptor[] getPropertyDescriptors()

Zwraca tablicę deskryptorów odpowiedniego rodzaju. Jeśli zwróci wartość null, jest to dla narzędzi tworzenia aplikacji sygnałem, że należy wykorzystać konwencje nazw i mechanizm refleksji.

- Image getIcon(int iconType)

zwraca obiekt graficzny, który może zostać wykorzystany do reprezentacji ziarnka w paletach, przybornikach itp. Dla standardowych typów ikon zdefiniowane są cztery stałe przedstawione już wcześniej.

**API** **java.beans.SimpleBeanInfo 1.1**

- Image loadImage(String resourceName)

zwraca obiekt graficzny związany z podanym zasobem. Nazwa zasobu określa ścieżkę względem katalogu zawierającego klasę informacyjną ziarnka.

**API** **java.beans.FeatureDescriptor 1.1**

- String getName()

- void setName(String name)

zwraca lub nadaje nazwę cechy (używaną w kodzie programu).

- String getDisplayName()

- void setDisplayName(String displayName)

zwraca lub nadaje lokalizowaną nazwę cechy. Domyślnie jest to ta sama nazwa, którą zwraca metoda getName. W obecnej wersji nie jest dostępna obsługa nazw w wielu językach.

- `String getShortDescription()`
- `void setShortDescription(String text)`

zwraca lub konfiguruje łańcuch znaków, który może zostać wykorzystany przez narzędzia tworzenia aplikacji do przedstawienia krótkiego opisu danej cechy. Domyślnie zwracany jest ten sam łańcuch, który zwraca metoda `getDisplayName`.

- `boolean isExpert()`
  - `void setExpert(boolean b)`
- pozwala sprawdzić lub ustawić znacznik, który określa, czy daną cechę należy ukryć przed zwykłym użytkownikiem (nie każdy pakiet tworzenia aplikacji umożliwia jego wykorzystanie).
- `boolean isHidden()`
  - `void setHidden(boolean b)`

sprawdza lub ustawia znacznik określający, czy cecha dostępna jest jedynie dla narzędzia tworzenia aplikacji.

#### `java.beans.PropertyDescriptor` 1.1

- `PropertyDescriptor(String propertyName, Class<?> beanClass)`
- `PropertyDescriptor(String propertyName, Class<?> beanClass, String getMethod, String setMethod)`

Tworzą deskryptor klasy `PropertyDescriptor`. Wyrzucają wyjątek `IntrospectionException`, jeśli wystąpi błąd podczas introspekcji. Pierwszy z konstruktorów zakłada wykorzystanie standardowej konwencji tworzenia nazw metod `get` i `set`.

- `Class<?> getPropertyType()`  
zwraca obiekt `Class` reprezentujący typ właściwości.
- `Method getReadMethod()`
- `Method getWriteMethod()`

Zwracają metodę `get` lub `set`.

#### `java.beans.IndexedPropertyDescriptor` 1.1

- `IndexedPropertyDescriptor(String propertyName, Class<?> beanClass)`
- `PropertyDescriptor(String propertyName, Class<?> beanClass, String getMethod, String setMethod, String indexedGetMethod, String indexedSetMethod)`

Tworzą deskryptor klasy `IndexedPropertyDescriptor`. Wyrzucają wyjątek `IntrospectionException`, jeśli wystąpi błąd podczas introspekcji. Pierwszy z konstruktorów zakłada wykorzystanie standardowej konwencji tworzenia nazw metod `get` i `set`.

- Method getIndexReadMethod()
- Method getIndexWriteMethod()

Zwracają indeksowaną metodę get lub set.

## 8.7. Edytory właściwości

Jeśli wyposażymy ziarnko we właściwość, której wartościami są na przykład liczby całkowite lub łańcuchy znaków, to zostanie ona automatycznie umieszczona w oknie inspektora właściwości. Jednak w przypadku gdy właściwość posiada wartości, które nie mogą być edytowane za pomocą pola tekstowego — na przykład daty lub kolory — musimy sami dostarczyć komponent, który umożliwi użytkownikowi wybór odpowiedniej wartości. Komponenty takie nazywamy *edytorami właściwości*. Edytor właściwości, której wartościami są daty, może mieć na przykład postać kalendarza umożliwiającego wybór daty. Edytor właściwości, której wartości opisują kolory, może pozwalać natomiast na określenie składowej czerwonej, niebieskiej i zielonej.

Pakiet NetBeans zawiera edytor właściwości, których wartości reprezentują kolory. Pakiet ten posiada także edytory dla podstawowych typów języka Java, na przykład String (edytor w postaci pola tekstowego) czy boolean (pole wyboru).

Utworzenie nowego edytora właściwości jest dość skomplikowane. Pierwszym krokiem jest stworzenie klasy informacyjnej dla danego ziarnka. Następnie dostarczamy implementacji metody getPropertyDescriptors. Zwraca ona tablicę obiektów klasy PropertyDescriptor. Musimy utworzyć po jednym takim obiekcie dla każdej właściwości, *także dla tych, w przypadku których wystarczy domyślny edytor*.

Obiekt klasy PropertyDescriptor tworzymy, przekazując jego konstruktorowi nazwę właściwości i klasę jej ziarnka.

```
PropertyDescriptor descriptor
    = new PropertyDescriptor("titlePosition", ChartBean.class);
```

Aby określić edytor danej właściwości, korzystamy z metody setPropertyEditorClass klasy PropertyDescriptor.

```
descriptor.setPropertyEditorClass(TitlePositionEditor.class);
```

Następnie utworzone deskryptory właściwości umieszczamy w tablicy. Ziarnko wykresu, które omówimy w tym podrozdziale, posiadać będzie pięć właściwości:

- graphColor typu Color,
- title typu String,
- titlePosition typu int,
- values typu double[],
- inverse typu boolean.

Listing 8.3 zawiera kod klasy ChartBeanBeanInfo określający edytory właściwości ziarnka. Działa on następująco.

1. Metoda getPropertyDescriptors zwraca deskryptor dla każdej właściwości. Właściwości title i graphColor otrzymują domyślny edytor dostarczany przez narzędzie tworzenia aplikacji.
2. Właściwości titlePosition, values, i inverse otrzymują dedykowane edytory klas TitlePositionEditor, DoubleArrayEditor i InverseEditor.

Rysunek 8.10 pokazuje ziarnko wykresu. W górnej części okna znajduje się tytuł wykresu, który może być wycentrowany lub wyrównany do jednego z marginesów. Właściwość values określa wartości wykresu. Jeśli właściwość inverse posiada wartość true, to tło wykresu wypełniane jest kolorem, a słupki są białe. Kod źródłowy ziarnka znajdziesz w przykładach dołączonych do tej książki. Jest on modyfikacją kodu apletu przedstawionego w 10. rozdziale książki Java. Podstawy.

**Listing 8.3.** chart/ChartBeanBeanInfo.java

```
package chart;

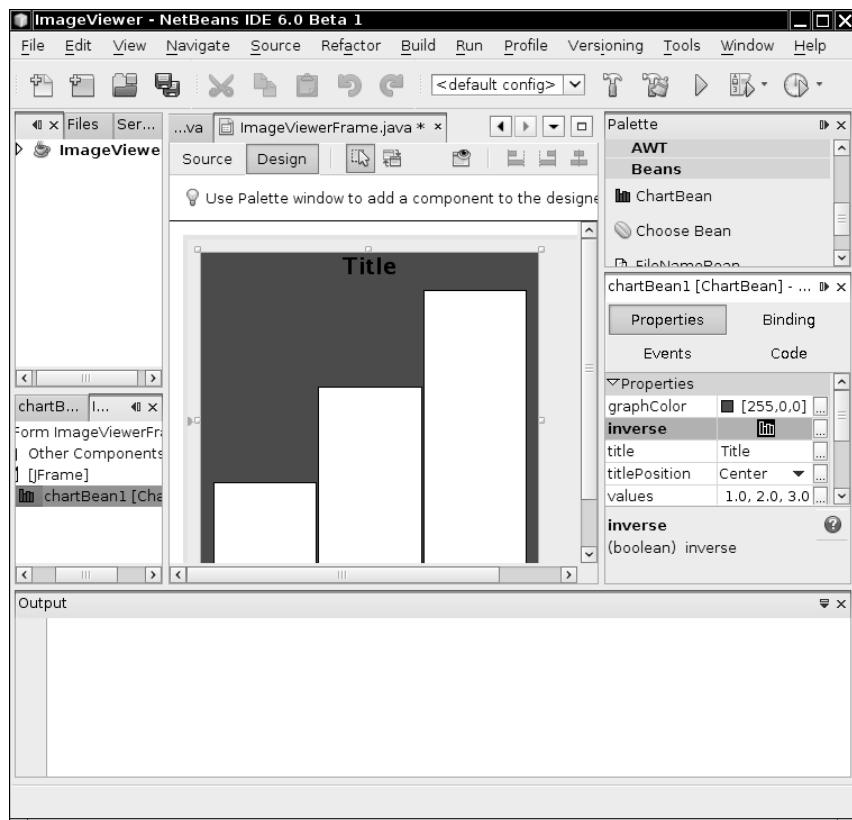
import java.awt.*;
import java.beans.*;

/*
 * Klasa informacyjna ziarnka wykresu. Specyfikuje edytory właściwości.
 * @version 1.20 2007-10-05
 * @author Cay Horstmann
 */
public class ChartBeanBeanInfo extends SimpleBeanInfo
{
    private PropertyDescriptor[] propertyDescriptors;
    private Image iconColor16;
    private Image iconColor32;
    private Image iconMono16;
    private Image iconMono32;

    public ChartBeanBeanInfo()
    {
        iconColor16 = loadImage("ChartBean_COLOR_16x16.gif");
        iconColor32 = loadImage("ChartBean_COLOR_32x32.gif");
        iconMono16 = loadImage("ChartBean_MONO_16x16.gif");
        iconMono32 = loadImage("ChartBean_MONO_32x32.gif");

        try
        {
            PropertyDescriptor titlePositionDescriptor = new PropertyDescriptor("titlePosition",
                ChartBean.class);
            titlePositionDescriptor.setPropertyEditorClass(TitlePositionEditor.class);
            PropertyDescriptor inverseDescriptor = new PropertyDescriptor("inverse",
                ChartBean.class);
            inverseDescriptor.setPropertyEditorClass(InverseEditor.class);
            PropertyDescriptor valuesDescriptor = new PropertyDescriptor("values",
                ChartBean.class);
            valuesDescriptor.setPropertyEditorClass(DoubleArrayEditor.class);
            propertyDescriptors = new PropertyDescriptor[] {
```

**Rysunek 8.10.**  
Ziarnko wykresu



```

valuesDescriptor.setPropertyEditorClass(DoubleArrayEditor.class);
propertyDescriptors = new PropertyDescriptor[] {
    new PropertyDescriptor("title", ChartBean.class), titlePositionDescriptor,
    valuesDescriptor, new PropertyDescriptor("graphColor", ChartBean.class),
    inverseDescriptor };
}
catch (IntrospectionException e)
{
    e.printStackTrace();
}
}

public PropertyDescriptor[] getPropertyDescriptors()
{
    return propertyDescriptors;
}

public Image getIcon(int iconType)
{
    if (iconType == BeanInfo.ICON_COLOR_16x16) return iconColor16;
    else if (iconType == BeanInfo.ICON_COLOR_32x32) return iconColor32;
    else if (iconType == BeanInfo.ICON_MONO_16x16) return iconMono16;
    else if (iconType == BeanInfo.ICON_MONO_32x32) return iconMono32;
    else return null;
}
}

```

**API** `java.beans.PropertyDescriptor 1.1`

- `PropertyDescriptor(String name, Class beanClass)`  
tworzy obiekt klasy `PropertyDescriptor`.  
*Parametry:*      `name`               nazwa właściwości,  
                        `beanClass`        klasa ziarnka, do którego należy właściwość.
- `void setPropertyEditorClass(Class editorClass)`  
określa klasę edytora dla danej właściwości.

**API** `java.beans.BeanInfo 1.1`

- `PropertyDescriptor[] getPropertyDescriptors()`  
zwraca deskryptor dla każdej z właściwości wyświetlanego dla ziarnka w oknie inspektora właściwości.

## 8.7.1. Implementacja edytora właściwości

Zanim przejdziemy do omówienia sposobu tworzenia edytorów, należy zaznaczyć, że edytory właściwości wykonywane są przez narzędzie tworzenia aplikacji, a nie samo ziarnko. Narzędzia takie, aby wyświetlić bieżącą wartość właściwości, korzystają z następującej procedury.

1. Tworzą instancję edytorów właściwości dla każdej właściwości ziarnka.
2. Żądają od ziarnka podania bieżącej wartości właściwości.
3. Zlecają edytoriowi właściwości wyświetlenie tej wartości.

Edytor właściwości musi posiadać domyślny konstruktor i implementować interfejs `PropertyEditor`. Zwykle rozszerza on klasę `PropertyEditorSupport`, która dostarcza domyślnych wersji metod tego interfejsu.

Dla każdego edytora właściwości wybieramy jeden z trzech sposobów wyświetlania i edycji wartości właściwości:

- Jako łańcuch znaków (definiujemy metody `getAsString` i `setAsString`).
- Jako pole wyboru (definiujemy metody `getAsText`, `setAsText` i `getTags`).
- Graficznie, poprzez jej narysowanie (definiujemy metody `isPaintable`, `paintValue`, `supportsCustomEditor` i `getCustomEditor`).

Sposoby te omówimy w kolejnych podrozdziałach.

### 8.7.1.1. Edytory właściwości oparte na łańcuchach

Proste edytory właściwości operują na łańcuchach znaków. Implementując je, zastępujemy metody `setAsString` oraz `getAsString` własnymi ich wersjami. Na przykład, nasze przykładowe

ziarnko wykresu posiada właściwość umożliwiającą określenie położenia tytułu wykresu. Możliwe są trzy różne położenia tytułu zdefiniowane za pomocą typu wyliczeniowego:

```
public enum Position { LEFT, CENTER, RIGHT };
```

Oczywiście w polu tekstowym właściwości nie będziemy prezentowaćłańcuchów LEFT, CENTER i RIGHT. Zdefiniujemy raczej edytor właściwości, którego metoda `getAsString` zwrócić będzie odpowiedni łańcuch znaków:

```
class TitlePositionEditor extends PropertyEditorSupport
{
    private String[] tags = { "Left", "Center", "Right" };
    ...
    public String getAsString()
    {
        int index = ((ChartBean.Position) getValue()).ordinal();
        return tags[index];
    }
}
```

Idealnie byłoby, gdyby podczas wyświetlania tych łańcuchów wykorzystany został również lokalizator, ale zadanie to pozostawiamy Czytelnikowi jako ćwiczenie.

Musimy oczywiście dostarczyć jeszcze metodę, która przekształci łańcuch tekstowy z powrotem na wartość właściwości:

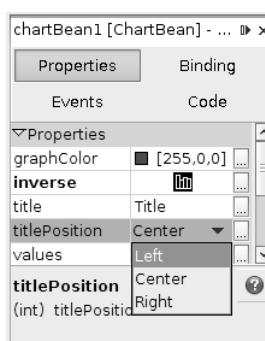
```
public void setAsString(String s)
{
    int index = Arrays.asList(tags).indexOf(s);
    if (index >= 0) setValue(ChartBean.Position.values()[index]);
}
```

Jeśli zdefiniujemy te dwie metody, inspektor właściwości utworzy pole tekstowe. Zostanie ono zainicjowane poprzez wywołanie metody `getAsString`, a metoda `setAsString` ostatecznie użyta po zakończeniu edycji. W naszym przypadku rozwiązanie takie będzie posiadać istotną wadę, gdyż użytkownik nie będzie wiedzieć, jakie są dopuszczalne łańcuchy znaków reprezentujące wartości właściwości. Dlatego też lepszym rozwiązaniem będzie prezentacja wszystkich wartości za pomocą listy wyboru (patrz rysunek 8.11). Klasa `PropertyEditorSupport` dostarcza prostej metody umożliwiającej wyświetlanie list wyboru przez edytor właściwości. Wystarczy zdefiniować metodę `getTags`, która zwróci tablicę łańcuchów znaków.

```
public String[] getTags() { return tags; }
```

### Rysunek 8.11.

Edytor  
`TitlePositionEditor`  
w działaniu



Domyślna wersja metody `getTags` zwraca wartość `null`, co oznacza, że do edycji wartości właściwości wystarcza pole tekstowe.

Dostarczając metody `getTags` nadal musimy dostarczyć implementacji metod `getAsText` i `setAsText`. Metoda `getTags` specyfikuje jedynie wartości wyświetlane na liście. Metody `getAsText`/ `setAsText` dokonują przekładu łańcuchów znaków na typ właściwości (który może być typem numerycznym lub dowolnym innym typem).

Edytory właściwości powinny również implementować metodę `getJavaInitializationString`. Za pomocą tej metody możemy przekazać narzędziom tworzenia aplikacji kod w języku Java, który nadaje właściwości bieżącą wartość. Narzędzia te używają go podczas automatycznego generowania kodu. Oto implementacja tej metody dla edytora `TitlePositionEditor`:

```
public String getJavaInitializationString()
{
    return ChartBean.Position.class.getName().replace('$', '.') + "." + getValue();
}
```

Metoda ta zwraca łańcuch postaci "chart.ChartBean.Position.LEFT". Wypróbujmy jej działanie w NetBeans: jeśli będziemy edytować pole `titlePosition`, NetBeans wstawi kod o postaci:

```
chartBean1.setTitlePosition(chart.ChartBean.Position.LEFT);
```

W naszej sytuacji kod ten jest nieco nieporządzony, ponieważ `ChartBean.Position.class.get  
Name()` zwraca łańcuch "chart.ChartBean\$Position". Zastępujemy zatem znak \$ kropką i dodajemy do łańcucha wynik zastosowania metody `toString` do wartości typu wyliczeniowego.



Jeśli właściwość posiada niestandardowy edytor, który nie implementuje metody `getJavaInitializationString`, NetBeans nie może wygenerować kodu i tworzy metodę `set` o parametrze ???.

Listing 8.4 zawiera kompletny kod źródłowy omówionego edytora.

#### **Listing 8.4.** chart/TitlePositionEditor.java

```
package chart;

import java.beans.*;
import java.util.*;

/**
 * Niestandardowy edytor właściwości titlePosition ziarnka
 * ChartBean. Umożliwia użytkownikowi wybór wartości
 * Left, Center i Right
 * @version 1.20 2007-12-14
 * @author Cay Horstmann
 */
public class TitlePositionEditor extends PropertyEditorSupport
{
    private String[] tags = { "Left", "Center", "Right" };

    public String[] getTags()
```

```

    {
        return tags;
    }

    public String getJavaInitializationString()
    {
        return ChartBean.Position.class.getName().replace('$', '.') + "." + getValue();
    }

    public String getAsText()
    {
        int index = ((ChartBean.Position) getValue()).ordinal();
        return tags[index];
    }

    public void setAsText(String s)
    {
        int index = Arrays.asList(tags).indexOf(s);
        if (index >= 0) setValue(ChartBean.Position.values()[index]);
    }
}

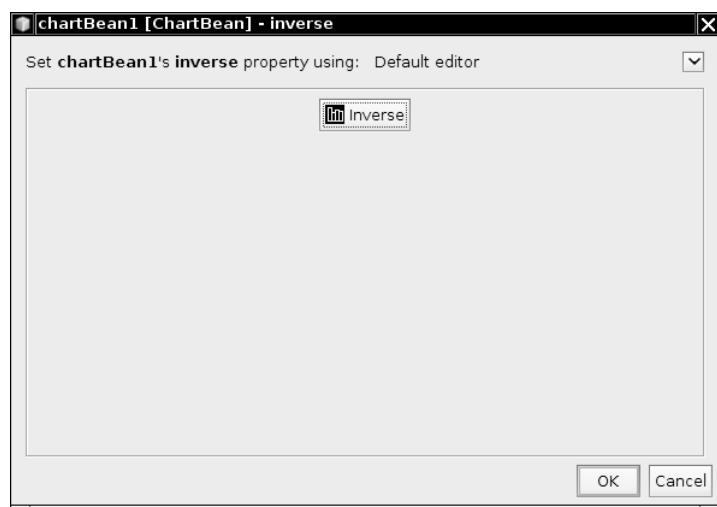
```

### 8.7.1.2. Złożone edytory właściwości

W przypadku złożonych typów właściwości ich wartości nie mogą być reprezentowane za pomocą pól tekstowych. Okno inspektora właściwości zawiera niewielki obszar (w nim zwykle umieszczone są pole tekstowe lub lista wyboru), w którym edytor właściwości tworzy graficzną reprezentację bieżącej wartości. Gdy użytkownik wybierze ten obszar myszą, to otwiera się okno dialogowe (patrz rysunek 8.12). Zawiera ono komponent umożliwiający edycję wartości dostarczony przez edytor właściwości oraz różne przyciski umieszczone przez narzędzie tworzenia aplikacji. W naszym przykładzie edytor jest bardzo oszczędny i zawiera tylko pojedynczy przycisk. Przykłady dołączone do książki zawierają kod bardziej złożonego edytora wartości wykresu.

**Rysunek 8.12.**

Okno edytora właściwości



Aby utworzyć graficzny edytor właściwości, należy najpierw poinformować inspektora właściwości, że sami będziemy rysować reprezentację wartości zamiast używać łańcucha znaków. Implementujemy zatem metodę `getAsText` interfejsu `PropertyEditor`, tak by zwracała wartość `null` oraz metodę `isPaintable`, tak by zwracała wartość `true`.

Następnie implementujemy metodę `paintValue`. Jako wartości parametrów otrzymuje ona kontekst graficzny i prostokąt, w którym może tworzyć rysunek. Prostokąt ten jest zwykle małych rozmiarów, dlatego też reprezentacja graficzna wartości nie powinna być zbyt skomplikowana. W naszym przypadku narysujemy jedną z dwóch ikon (patrz rysunek 8.11).

```
public void paintValue(Graphics g, Rectangle box)
{
    ImageIcon icon = (Boolean) getValue() ? inverseIcon : normalIcon;
    int x = bounds.x + (bounds.width - icon.getIconWidth()) / 2;
    int y = bounds.y + (bounds.height - icon.getIconHeight()) / 2;
    g.drawImage(icon.getImage(), x, y, null);
}
```

Oczywiście utworzonej w ten sposób graficznej reprezentacji wartości nie możemy bezpośrednio edytować. Jeśli wybierzemy ją myszą, to otwarte zostanie okno dialogowe edytora.

Informację o wykorzystaniu własnego edytora przekazujemy, implementując metodę `supportsCustomEditor` interfejsu `PropertyEditor`, tak by zwracała ona wartość `true`.

Następnie a pomocą metody `getCustomEditor` interfejsu `PropertyEditor` tworzymy i zwracamy obiekt reprezentujący nowy edytor.

Listing 8.5 zawiera kompletny kod klasy `InverseEditor` wyświetlającej bieżącą wartość w oknie inspektora właściwości, a listing 8.6 przedstawia implementację panelu edytora.

---

**Listing 8.5.** chart/InverseEditor.java

```
package chart;

import java.awt.*;
import java.beans.*;
import javax.swing.*;

/**
 * Edytor właściwości inverse ziarnka ChartBean.
 * Właściwość inverse umożliwia wybór kolorowych słupków
 * wykresu lub kolorowego tła.
 * @version 1.30 2007-10-03
 * @author Cay Horstmann
 */
public class InverseEditor extends PropertyEditorSupport
{
    private ImageIcon normalIcon = new
        ImageIcon(getClass().getResource("ChartBean_MONO_16x16.gif"));

    private ImageIcon inverseIcon = new ImageIcon(getClass().getResource(
        "ChartBean_INVERSE_16x16.gif"));

    public Component getCustomEditor()
    {
        return new InverseEditorPanel(this);
    }
}
```

```

    }

    public boolean supportsCustomEditor()
    {
        return true;
    }

    public boolean isPaintable()
    {
        return true;
    }

    public String getAsText()
    {
        return null;
    }

    public String getJavaInitializationString()
    {
        return "" + getValue();
    }

    public void paintValue(Graphics g, Rectangle bounds)
    {
        ImageIcon icon = (Boolean) getValue() ? inverseIcon : normalIcon;
        int x = bounds.x + (bounds.width - icon.getIconWidth()) / 2;
        int y = bounds.y + (bounds.height - icon.getIconHeight()) / 2;
        g.drawImage(icon.getImage(), x, y, null);
    }
}

```

**Listing 8.6.** chart/InverseEditorPanel.java

```

package chart;

import java.awt.event.*;
import java.beans.*;
import javax.swing.*;

/**
 * Panel konfiguracji właściwości inverse. Zawiera przyciski
 * wyboru wartości właściwości.
 * @version 1.30 2007-10-03
 * @author Cay Horstmann
 */
public class InverseEditorPanel extends JPanel
{
    private JButton button;
    private PropertyEditorSupport editor;
    private ImageIcon normalIcon = new
    ImageIcon(getClass().getResource("ChartBean_MONO_16x16.gif"));

    public InverseEditorPanel(PropertyEditorSupport ed)
    {
        editor = ed;
        button = new JButton();
    }
}

```

```

updateButton();
button.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        editor.setValue(!(Boolean) editor.getValue());
        updateButton();
    }
});
add(button);
}

private void updateButton()
{
    if ((Boolean) editor.getValue())
    {
        button.setIcon(inverseIcon);
        button.setText("Inverse");
    }
    else
    {
        button.setIcon(normalIcon);
        button.setText("Normal");
    }
}

private ImageIcon inverseIcon = new ImageIcon(getClass().getResource(
    "ChartBean_INVERSE_16x16.gif"));
}

```

### *java.beans.PropertyEditor 1.1*

■ `Object getValue()`

zwraca bieżącą wartość właściwości. Typy proste są obudowywane obiektami.

■ `void setValue(Object newValue)`

nadaje właściwości nową wartość. Typy proste muszą być obudowane odpowiednim obiektem.

*Parametry:* `newValue` nowa wartość właściwości; powinna być nowo utworzonym obiektem odpowiedniej klasy.

■ `String getAsText()`

metodę tę należy zastąpić, tak by zwracała reprezentację bieżącej wartości właściwości w postaci łańcucha znaków. Domyślona implementacja zwraca wartość `null`, co oznacza, że wartość właściwości nie może być reprezentowana przez łańcuch znaków.

■ `void setAsText(String text)`

metodę tę należy zastąpić, tak by nadawała właściwości odpowiednią wartość na podstawie otrzymanego łańcucha znaków. Metoda może wyrzucać wyjątek `IllegalArgumentException`, w przypadku gdy łańcuch nie reprezentuje dozwolonej

wartości lub wartość właściwości nie może być reprezentowana przez łańcuch znaków.

■ `String[] getTags()`

metodę tę należy zastąpić, tak by zwracała tablicę reprezentacji wszystkich dozwolonych wartości właściwości, które zostaną wyświetcone na liście wyboru. Domyślna implementacja zwraca wartość `null`, co oznacza, że nie istnieje skończony zbiór wartości właściwości.

■ `boolean isPaintable()`

metodę tę należy zaimplementować, tak by zwracała wartość `true`, jeśli klasa używa metody `paintValue` do wyświetlania wartości właściwości.

■ `void paintValue(Graphics g, Rectangle box)`

metodę tę należy zastąpić, tak by tworzyła reprezentację graficzną wartości, korzystając z dostarczonego kontekstu graficznego i w określonym miejscu komponentu inspektora właściwości.

■ `boolean supportsCustomEditor()`

metodę tę należy zastąpić, tak by zwracała wartość `true`, jeśli korzystamy z własnego edytora właściwości.

■ `Component getCustomEditor()`

metodę tę należy zastąpić, tak by zwracała komponent zawierający interfejs użytkownika edytora właściwości.

■ `String getJavaInitializationString()`

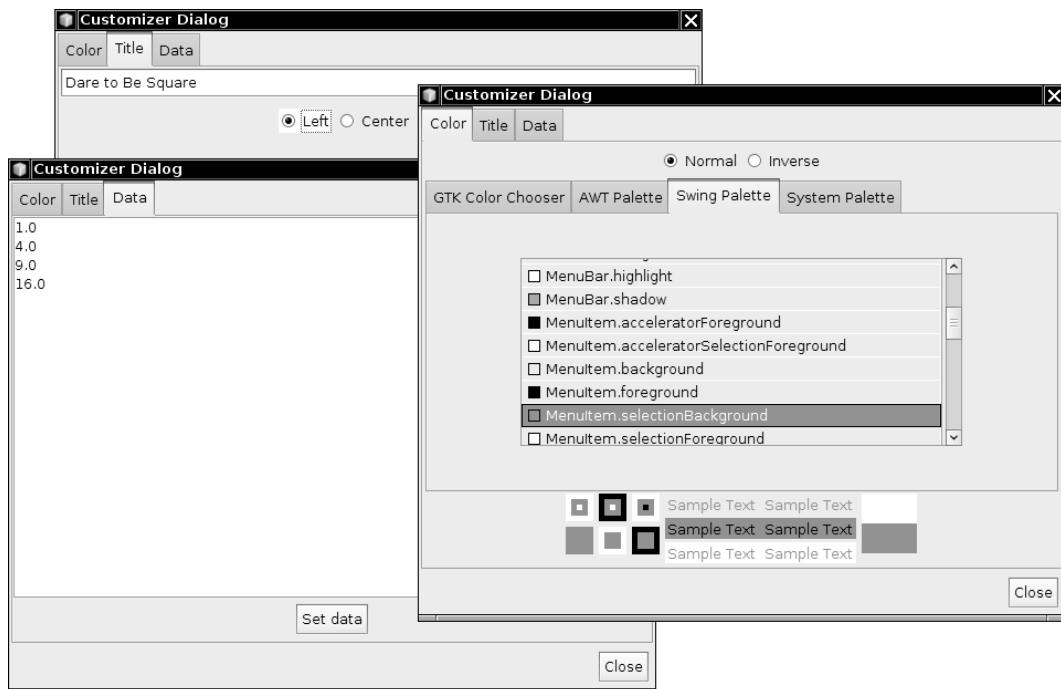
metodę tę należy zastąpić, tak by zwracała łańcuch znaków reprezentujący kod w języku Java, który zostanie użyty do zainicjowania wartości właściwości, na przykład "`0`", "`new Color(64, 64, 64)`".

## 8.8. Indywidualizacja ziarnka

Edytor właściwości, nawet bardzo skomplikowany, wykorzystywany jest jedynie do edycji pojedynczej właściwości. Jeśli niektóre właściwości ziarnka są ze sobą powiązane, to przydatna może okazać się możliwość edycji wielu właściwości jednocześnie. W takim przypadku mówimy o *indywidualizacji* ziarnka.

Niektóre ziarnka mogą również posiadać istotne cechy, które nie są eksponowane w postaci właściwości i tym samym nie mogą być edytowane za pomocą inspektora właściwości. W takim przypadku również zachodzi potrzeba indywidualizacji ziarnka.

W podrozdziale tym przedstawimy przykład indywidualizacji ziarnka wykresu. Umożliwimy wyspecyfikowanie wielu jego właściwości w jednym oknie dialogowym przedstawionym na rysunku 8.13.



Rysunek 8.13. Indywidualizacja ziarnka wykresu

Aby umożliwić indywidualizację ziarnka, musimy dostarczyć odpowiednią klasę informacyjną oraz zaimplementować metodę `getBeanDescriptor`, co pokazujemy poniżej.

```
public ChartBean2BeanInfo extends SimpleBeanInfo
{
    private BeanDescriptor beanDescriptor
        = new BeanDescriptor(ChartBean2.class, ChartBean2Customizer.class);

    public BeanDescriptor getBeanDescriptor()
    {
        return beanDescriptor;
    }
}
```

Zwróćmy uwagę, że w przypadku klasy indywidualizacji nie musimy stosować się do żadnego wzorca tworzenia nazw. (Mimo to przyjęło się tworzyć nazwy klasy indywidualizacji, dodając do nazwy klasy ziarnka przyrostek `Customizer`).

Sposób implementacji klasy indywidualizacji przedstawimy w kolejnym podrozdziale.

#### `java.beans.BeanInfo 1.1`

- `BeanDescriptor getBeanDescriptor()`  
zwraca obiekt `BeanDescriptor` opisujący cechy ziarnka.

**API** `java.beans.BeanDescriptor 1.1`

- `BeanDescriptor(Class<?> beanClass, Class<?> customizerClass)`

tworzy obiekt klasy `BeanDescriptor` dla ziarnka wyposażonego w klasę indywidualizacji.

*Parametry:* `beanClass` obiekt `Class` ziarnka,  
`customizerClass` obiekt `Class` klasy indywidualizacji.

## 8.8.1. Implementacja klasy indywidualizacji

Każda klasa indywidualizacji musi posiadać domyślny konstruktor, rozszerzać klasę `Component` i implementować interfejs `Customizer`. Interfejs ten zawiera tylko trzy metody:

- metodę `setObject`, której parametr określa indywidualizowane ziarnko,
- metody `addPropertyChangeListener` i `removePropertyChangeListener`, które zarządzają kolekcją obiektów nasłuchujących powiadamianych o zmianie właściwości na skutek indywidualizacji.

Dobrym zwyczajem jest aktualizacja wyglądu ziarnka za każdym razem, gdy użytkownik zmieni jedną z jego właściwości (przez wysłanie zdarzenia `PropertyChangeEvent`), a nie tylko po zakończeniu procesu indywidualizacji.

W przeciwieństwie do edytorów właściwości okna indywidualizacji nie są otwierane automatycznie. W przypadku NetBeans musimy w tym celu otworzyć menu ziarnka prawym klawiszem myszy i wybrać z niego pozycję `Customize`. Wywołana zostanie wtedy metoda `setObject` obiektu klasy indywidualizacji, której parametrem jest indywidualizowane ziarnko. Jak łatwo zauważać, oznacza to, że instancja klasy indywidualizacji tworzona jest, zanim zostanie powiązana z instancją ziarnka. Dlatego też klasa indywidualizacji nie posiada żadnej informacji o stanie ziarnka, a także musimy dodatkowo dostarczyć jej konstruktor domyślny (czyli nieposiadający parametrów).

Ponieważ klasa indywidualizacji z reguły prezentuje użytkownikowi wiele opcji, to często najwygodniej wykorzystać do jej implementacji panel z zakładkami. Z możliwości tej skorzystamy także w naszym przykładzie i klasa indywidualizacji będzie rozszerzać klasę `JTabbedPane`.

Nasze okno indywidualizacji posiada trzy panele umożliwiające określenie:

- koloru wykresu i jego tła,
- tytułu wykresu i jego położenia,
- danych wykresu.

Utworzenie odpowiedniego interfejsu użytkownika okna indywidualizacji może być pracochłonne — w naszym przykładzie zajmuje ponad 100 wierszy kodu w konstruktorze. Ponieważ jest to typowy przykład wykorzystania komponentów biblioteki Swing, nie będziemy omawiać szczegółów tego fragmentu kodu.

Istnieje jednak pewna sztuczka, którą warto zapamiętać. W oknie indywidualizacji często musimy umożliwiać użytkownikowi edycję właściwości. Zamiast implementować ją od początku, możemy zlokalizować istniejący edytor właściwości i dodać go do interfejsu użytkownika! Na przykład indywidualizując nasze ziarnko wykresu, musimy udostępnić użytkownikowi możliwość określenia jego koloru. Ponieważ wiemy, że NetBeans posiada odpowiedni edytor właściwości, to możemy odnaleźć go w poniższy sposób:

```
PropertyEditor colorEditor
    = PropertyEditorManager.findEditor(Color.class);
Component colorEditorComponent = colorEditor.getCustomEditor();
```

Po umieszczeniu wszystkich komponentów interfejsu użytkownika inicjujemy ich wartość za pomocą metody `setObject`. Metoda ta jest wywoływana, gdy otwierane jest okno indywidualizacji. Jako parametr przekazywane jest jej ziarnko poddawane procesowi indywidualizacji. Referencję ziarka zachowujemy, ponieważ przyda nam się do powiadamiania go o zmianach właściwości. Następnie inicjujemy każdy z komponentów interfejsu użytkownika. Poniżej przedstawiamy fragment kodu metody `setObject` odpowiedzialny za te operacje.

```
public void setObject(Object obj)
{
    bean = (ChartBean2)obj;
    titleField.setText(bean.getTitle());
    colorEditor.setValue(bean.getGraphColor());
    ...
}
```

Musimy jeszcze obsłużyć zdarzenia związane z akcjami użytkownika w oknie indywidualizacji, tak by na bieżąco aktualizować ziarnko. Za każdym razem, gdy użytkownik zmieni wartość za pomocą jednego z komponentów, komponent ten wygeneruje odpowiednie zdarzenie. Obsługując je, musimy zaktualizować wartość właściwości ziarka, ale i wygenerować zdarzenie `PropertyChangeEvent`, aby także inne obiekty nasłuchujące (na przykład inspektor właściwości) mogły zaktualizować swoje dane. Prześledźmy ten sposób działania na przykładzie kilku elementów interfejsu okna indywidualizacji ziarka wykresu.

Gdy użytkownik wprowadzi nowy tekst tytułu wykresu, musimy zaktualizować właściwość reprezentującą tytuł wykresu. W tym celu do pola tekstowego, w którym użytkownik wprowadza tytuł, dołączymy obiekt nasłuchujący `DocumentListener`.

```
titleField.getDocument().addDocumentListener(new
    DocumentListener()
{
    public void changedUpdate(DocumentEvent evt)
    {
        setTitle(titleField.getText());
    }
    public void insertUpdate(DocumentEvent evt)
    {
        setTitle(titleField.getText());
    }
    public void removeUpdate(DocumentEvent evt)
    {
        setTitle(titleField.getText());
    }
});
```

Wszystkie trzy metody obiektu nasłuchującego wywołują metodę `setTitle` klasy indywidualizacji. Aktualizuje ona właściwość ziarnka i generuje zdarzenie zmiany właściwości. (Aktualizacja ta konieczna jest jedynie w przypadku właściwości, które nie są powiązane). Poniżej przedstawiamy implementację metody `setTitle`.

```
public void setTitle(String newValue)
{
    if (bean == null) return;
    String oldValue = bean.getTitle();
    bean.setTitle(newValue);
    firePropertyChange("title", oldValue, newValue);
}
```

Użytkownik może zmienić kolor wykresu, korzystając z edytora właściwości koloru w oknie indywidualizacji. Zmiany koloru śledzimy, używając obiektu nasłuchującego, który dołączamy do edytora właściwości. Edytor ten również generuje zdarzenie zmiany właściwości. Obsługując je, zmieniamy kolor wykresu.

```
colorEditor.addPropertyChangeListener(new
    PropertyChangeListener()
{
    public void propertyChange(PropertyChangeEvent event)
    {
        setGraphColor((Color)colorEditor.getValue());
    }
});
```

Listing 8.7 zawiera pełen kod źródłowy klasy indywidualizacji ziarnka wykresu.

#### **Listing 8.7. chart2/ChartBeanCustomizer.java**

```
package chart2;

import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

/**
 * Klasa indywidualizacji ziernka wykresu
 * umożliwiająca edycję wszystkich jego właściwości
 * w jednym oknie dialogowym z zakładkami
 * @version 1.12 2007-10-03
 * @author Cay Horstmann
 */
public class ChartBeanCustomizer extends JTabbedPane implements Customizer
{
    private ChartBean bean;
    private PropertyEditor colorEditor;
    private JTextArea data;
    private JRadioButton normal;
    private JRadioButton inverse;
    private JRadioButton[] position;
    private JTextField titleField;
```

```
public ChartBeanCustomizer()
{
    data = new JTextArea();
    JPanel dataPane = new JPanel();
    dataPane.setLayout(new BorderLayout());
    dataPane.add(new JScrollPane(data), BorderLayout.CENTER);
    JButton dataButton = new JButton("Set data");
    dataButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            setData(data.getText());
        }
    });
    JPanel panel = new JPanel();
    panel.add(dataButton);
    dataPane.add(panel, BorderLayout.SOUTH);

    JPanel colorPane = new JPanel();
    colorPane.setLayout(new BorderLayout());

    normal = new JRadioButton("Normal", true);
    inverse = new JRadioButton("Inverse", false);
    panel = new JPanel();
    panel.add(normal);
    panel.add(inverse);
    ButtonGroup group = new ButtonGroup();
    group.add(normal);
    group.add(inverse);
    normal.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            setInverse(false);
        }
    });
    inverse.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            setInverse(true);
        }
    });
}

colorEditor = PropertyEditorManager.findEditor(Color.class);
colorEditor.addPropertyChangeListener(new PropertyChangeListener()
{
    public void propertyChange(PropertyChangeEvent event)
    {
        setGraphColor((Color) colorEditor.getValue());
    }
});

colorPane.add(panel, BorderLayout.NORTH);
colorPane.add(colorEditor.getCustomEditor(), BorderLayout.CENTER);
```

```

JPanel titlePane = new JPanel();
titlePane.setLayout(new BorderLayout());

group = new ButtonGroup();
position = new JRadioButton[3];
position[0] = new JRadioButton("Left");
position[1] = new JRadioButton("Center");
position[2] = new JRadioButton("Right");

panel = new JPanel();
for (int i = 0; i < position.length; i++)
{
    final ChartBean.Position pos = ChartBean.Position.values()[i];
    panel.add(position[i]);
    group.add(position[i]);
    position[i].addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            setTitlePosition(pos);
        }
    });
}

titleField = new JTextField();
titleField.getDocument().addDocumentListener(new DocumentListener()
{
    public void changedUpdate(DocumentEvent evt)
    {
        setTitle(titleField.getText());
    }

    public void insertUpdate(DocumentEvent evt)
    {
        setTitle(titleField.getText());
    }

    public void removeUpdate(DocumentEvent evt)
    {
        setTitle(titleField.getText());
    }
});

titlePane.add(titleField, BorderLayout.NORTH);
JPanel panel2 = new JPanel();
panel2.add(panel);
titlePane.add(panel2, BorderLayout.CENTER);
addTab("Color", colorPane);
addTab("Title", titlePane);
addTab("Data", dataPane);
}

/**
 * Określa dane prezentowane na wykresie.
 * @param s łańcuch zawierający sekwencję danych oddzielonych spacją
 */
public void setData(String s)
{

```

```
 StringTokenizer tokenizer = new StringTokenizer(s);

    int i = 0;
    double[] values = new double[tokenizer.countTokens()];
    while (tokenizer.hasMoreTokens())
    {
        String token = tokenizer.nextToken();
        try
        {
            values[i] = Double.parseDouble(token);
            i++;
        }
        catch (NumberFormatException e)
        {
        }
    }
    setValues(values);
}

< /**
 * Określa tytuł wykresu.
 * @param newValue nowy tytuł
 */
public void setTitle(String newValue)
{
    if (bean == null) return;
    String oldValue = bean.getTitle();
    bean.setTitle(newValue);
    firePropertyChange("title", oldValue, newValue);
}

< /**
 * Określa położenie tytułu wykresu.
 * @param i położenie tytułu (ChartBean.LEFT, ChartBean.CENTER lub ChartBean.RIGHT)
 */
public void setTitlePosition(ChartBean.Position pos)
{
    if (bean == null) return;
    ChartBean.Position oldValue = bean.getTitlePosition();
    bean.setTitlePosition(pos);
    firePropertyChange("titlePosition", oldValue, pos);
}

< /**
 * Określa wartość właściwości inverse.
 * @param b wartość true, jeśli kolory wykresu i tła mają być zamienione
 */
public void setInverse(boolean b)
{
    if (bean == null) return;
    boolean oldValue = bean.isInverse();
    bean.setInverse(b);
    firePropertyChange("inverse", oldValue, b);
}

< /**
 * Określa wartości prezentowane na wykresie.
 * @param newValue tablica nowych wartości
 */

```

```

public void setValues(double[] newValue)
{
    if (bean == null) return;
    double[] oldValue = bean.getValues();
    bean.setValues(newValue);
    firePropertyChange("values", oldValue, newValue);
}

/**
 * Określa kolor wykresu.
 * @param newValue nowy kolor
 */
public void setGraphColor(Color newValue)
{
    if (bean == null) return;
    Color oldValue = bean.getGraphColor();
    bean.setGraphColor(newValue);
    firePropertyChange("graphColor", oldValue, newValue);
}

public void setObject(Object obj)
{
    bean = (ChartBean) obj;

    data.setText("");
    for (double value : bean.getValues())
        data.append(value + "\n");

    normal.setSelected(!bean.isInverse());
    inverse.setSelected(bean.isInverse());

    titleField.setText(bean.getTitle());

    for (int i = 0; i < position.length; i++)
        position[i].setSelected(i == bean.getTitlePosition().ordinal());

    colorEditor.setValue(bean.getGraphColor());
}
}

```

#### API java.beans.Customizer 1.1

- void setObject(Object bean)  
określa ziarnko poddawane indywidualizacji.

## 8.9. Trwałość ziarnek JavaBeans

Mechanizm trwałości ziarnek JavaBeans używa właściwości JavaBeans do zapisu ziarnek do strumienia i ich późniejszego odczytu (być może na innej maszynie wirtualnej). Pod tym względem mechanizm trwałości ziarnek JavaBeans przypomina serializację obiektów (patrz rozdział 1.). Istnieje jednak ważna różnica: mechanizm trwałości ziarnek JavaBeans dużo lepiej nadaje się do ich *długotrwałego* przechowywania.

Gdy obiekt poddawany jest serializacji, to jego pola zostają zapisane w strumieniu. Gdy zmienia się implementacja klasy, to mogą zmienić się również jej pola. W takim przypadku odczytanie plików zawierających serializowane obiekty wcześniejszych wersji klasy staje się kłopotliwe. Oczywiście możliwe jest wykrywanie różnic pomiędzy wersjami i przekład pomiędzy starą a nową reprezentacją danych. Jednak proces taki jest wyjątkowo żmudny i powinien być stosowany jedynie w wyjątkowych sytuacjach. Serializacja nie jest odpowiednim rozwiązaniem dla długotrwałego przechowywania obiektów. Dlatego też dokumentacja wszystkich komponentów biblioteki Swing zawiera następującą uwagę „Ostrzeżenie: serializowane obiekty tej klasy nie będą zgodne z kolejnymi wersjami biblioteki Swing. Wykorzystanie serializacji jest właściwe tylko dla tymczasowego przechowywania obiektów bądź przez mechanizm RMI”.

Rozwiązaniem wspomnianego problemu jest mechanizm trwałości. Początkowo został on zaprojektowany z myślą o narzędziach projektowania interfejsu użytkownika wykorzystujących technikę przeciagnij-i-upuść. Narzędzia takie zapisują wynik kolejnych działań użytkownika — czyli kolekcję ramek, paneli, przycisków i innych komponentów Swing — w pliku, używając do tego mechanizmu trwałości umożliwiającego ich długotrwałe przechowywanie. Ich odtworzenie wymaga jedynie otwarcia pliku. W ten sposób znaczającej redukcji ulega kod potrzebny od odtworzenia układu i powiązań pomiędzy komponentami Swing. Niestety, rozwiązania takie nie zyskały większej popularności.

Podstawowa koncepcja trwałości komponentów JavaBeans jest bardzo prosta. Założymy, że chcemy zapisać w pliku obiekt klasy JFrame, aby później móc go odtworzyć. Jeśli zapoznamy się z kodem źródłowym klasy JFrame i jej klas bazowych, to zorientujemy się, że zawierają one bardzo wiele pól. Gdyby obiekt klasy JFrame poddać serializacji, to wszystkie te pola musiałyby zostać zapisane. Zastanówmy się jednak, w jaki sposób tworzona jest ramka klasy JFrame:

```
JFrame frame = new JFrame();
frame.setTitle("My Application");
frame.setVisible(true);
```

Domyślny konstruktor inicjuje wszystkie pola instancji oraz konfiguruje kilka właściwości. Mechanizm trwałości komponentów JavaBeans zapisuje powyższe instrukcje w formacie XML:

```
<object class="javax.swing.JFrame">
    <void property="title">
        <string>My Application</string>
    </void>
    <void property="visible">
        <boolean>true</boolean>
    </void>
</object>
```

Gdy obiekt jest odtwarzany, to instrukcje te zostają *wykonane*: zostaje utworzony obiekt klasy JFrame, a właściwości title i visible otrzymują odpowiednie wartości. Nie jest wtedy istotne, czy wewnętrzna reprezentacja klasy JFrame uległa w międzyczasie zmianie. Proces odtwarzania obiektu polega bowiem na konfiguracji jego właściwości.

Zauważmy przy tym, że archiwizowane są tylko te właściwości, których wartości są różne od domyślnych. XMLEncoder tworzy domyślny obiekt klasy JFrame i porównuje jego właściwości z właściwościami archiwizowanej ramki. Instrukcje konfigurujące właściwości zostają

wygenerowane tylko dla tych właściwości, które różnią się od domyślnych. Rozwiążanie takie powoduje *eliminację redundantnych danych* i przyczynia się do powstawania mniejszych plików niż podczas serializacji. (Różnica ta jest szczególnie widoczna właśnie w przypadku serializacji komponentów Swing, ponieważ zawierają one wiele pól, których wartość prawie zawsze pozostaje domyślna).

Z rozwiązaniem takim związane są pewne drobne niuanse. Na przykład wywołanie:

```
frame.setSize(600, 400)
```

nie jest operacją zmiany wartości właściwości. Jednak XMLEncoder potrafi sobie poradzić z tym problemem, zapisując instrukcję jako:

```
<void property="bounds">
<object class="java.awt.Rectangle">
<int>0</int>
<int>0</int>
<int>600</int>
<int>400</int>
</object>
</void>
```

Aby zapisać obiekt w strumieniu, użyjemy obiektu XMLEncoder w następujący sposób:

```
XMLEncoder out = new XMLEncoder(new FileOutputStream(. . .));
out.writeObject(frame);
out.close();
```

Aby odtworzyć obiekt zapisany w ten sposób, użyjemy obiektu klasy XMLDecoder:

```
XMLDecoder in = new XMLDecoder(new FileInputStream(. . .));
JFrame newFrame = (JFrame) in.readObject();
in.close();
```

Program przedstawiony na listingu 8.8 ilustruje sposób załadowania i zapisania ramki (patrz rysunek 8.14). Po uruchomieniu programu należy najpierw wybrać przycisk *Save* i zapisać w ten sposób ramkę w pliku. Następnie trzeba przesunąć ramkę w inne miejsce i wybrać przycisk *Load*, co spowoduje pojawienie się nowej ramki w poprzednim miejscu. Przyjrzyjmy się zawartości pliku XML utworzonego przez program.

#### **Listing 8.8. persistentFrame/PersistentFrameTest.java**

```
package persistentFrame;

import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.io.*;
import javax.swing.*;

/**
 * Program demonstrujący użycie kodera i dekodera XML
 * do zapisu i odtwarzania ramki.
 * @version 1.01 2007-10-03
 * @author Cay Horstmann
 */
```

```
public class PersistentFrameTest
{
    private static JFileChooser chooser;
    private JFrame frame;

    public static void main(String[] args)
    {
        chooser = new JFileChooser();
        chooser.setCurrentDirectory(new File("."));
        PersistentFrameTest test = new PersistentFrameTest();
        test.init();
    }

    public void init()
    {
        frame = new JFrame();
        frame.setLayout(new FlowLayout());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setTitle("PersistentFrameTest");
        frame.setSize(400, 200);

        JButton loadButton = new JButton("Load");
        frame.add(loadButton);
        loadButton.addActionListener(EventHandler.create(ActionListener.class, this, "load"));

        JButton saveButton = new JButton("Save");
        frame.add(saveButton);
        saveButton.addActionListener(EventHandler.create(ActionListener.class, this, "save"));

        frame.setVisible(true);
    }

    public void load()
    {
        // okno wyboru pliku
        int r = chooser.showOpenDialog(null);

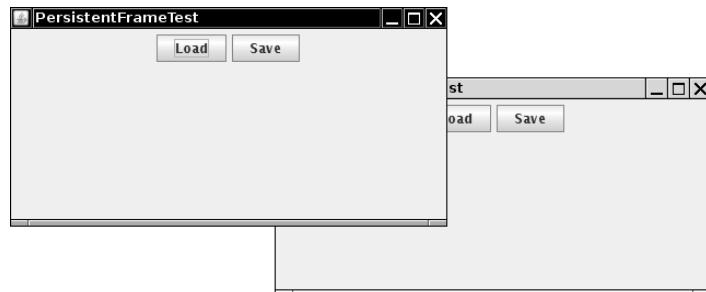
        // otwiera wybrany plik
        if(r == JFileChooser.APPROVE_OPTION)
        {
            try
            {
                File file = chooser.getSelectedFile();
                XMLDecoder decoder = new XMLDecoder(new FileInputStream(file));
                decoder.readObject();
                decoder.close();
            }
            catch (IOException e)
            {
                JOptionPane.showMessageDialog(null, e);
            }
        }
    }

    public void save()
    {
        if (chooser.showSaveDialog(null) == JFileChooser.APPROVE_OPTION)
```

```
        {
            try
            {
                File file = chooser.getSelectedFile();
                XMLEncoder encoder = new XMLEncoder(new FileOutputStream(file));
                encoder.writeObject(frame);
                encoder.close();
            }
            catch (IOException e)
            {
                JOptionPane.showMessageDialog(null, e);
            }
        }
    }
}
```

### Rysunek 8.14.

## *Program PersistentFrameTest w działaniu*



Zawartość pliku utworzonego przez program dowodzi, jak wiele pracy musi wykonać obiekt XMLEncoder podczas zapisywania ramki. XMLEncoder generuje instrukcje odpowiedzialne za wykonanie następujących akcji:

- Skonfigurowanie różnych właściwości: size, layout, defaultCloseOperation, title i tak dalej.
  - Dodanie przycisków do ramki.
  - Dodanie obiektów nasłuchujących przycisków.

W tym przypadku musimy skonstruować obiekty nasłuchujące klasy EventHandler. Obiekt XMLEncoder nie może bowiem zarchiwizować dowolnej klasy wewnętrznej, ale potrafi obsługiwać obiekty klasy EventHandler.

## **8.9.1. Zastosowanie mechanizmu trwałości JavaBeans dla dowolnych danych**

Mechanizm trwałości JavaBeans nie jest zarezerwowany wyłącznie dla komponentów Swing. Możemy go użyć do przechowania *dowolnej* kolekcji obiektów pod warunkiem, że przestrzegamy kilku prostych zasad. W kolejnych podrozdziałach pokażemy, w jaki sposób można użyć mechanizmu trwałości JavaBeans dla własnych danych.

Klasa XMLEncoder posiada wbudowaną obsługę następujących typów:

- null;
- wszystkich typów podstawowych i ich klas obudowujących;
- wyliczeń (począwszy od Java SE 6);
- String;
- tablic;
- kolekcji i map;
- typów refleksji (Class, Field, Method i Proxy);
- typów AWT (Color, Cursor, Dimension, Font, Insets, Point, Rectangle i ImageIcon);
- komponentów AWT i Swing, menedżerów układu komponentów i modeli;
- obiektów nasłuchujących zdarzeń.

### 8.9.1.1. Implementacja delegatu trwałości tworzącego obiekt

Zastosowanie mechanizmu trwałości JavaBeans jest trywialne, jeśli tylko potrafimy opisać stan każdego obiektu, konfigurując jego właściwości. Ale w praktyce większość programów zawiera klasy, których obiektów nie można opisać w ten sposób. Weźmy na przykład klasę Employee przedstawioną w rozdziale 4. książki *Java. Podstawy*. Klasa ta nie jest ziarnkiem, ponieważ nie posiada domyślnego konstruktora oraz metod setName, setSalary, setHireDay. Aby obejść ten problem, musimy dostarczyć obiektovi XMLEncoder *delegat trwałości*, który potrafi zapisać obiekt klasy Employee w języku XML:

Delegat trwałości dla klasy Employee przesyłania metodę instantiate, która generuje *wyrażenie tworzące obiekt*.

```
PersistenceDelegate delegate = new
    DefaultPersistenceDelegate()
{
    protected Expression instantiate(Object oldInstance, Encoder out)
    {
        Employee e = (Employee) oldInstance;
        GregorianCalendar c = new GregorianCalendar();
        c.setTime(e.getHireDay());
        return new Expression(oldInstance, Employee.class, "new",
            new Object[]
            {
                e.getName(),
                e.getSalary(),
                c.get(Calendar.YEAR),
                c.get(Calendar.MONTH),
                c.get(Calendar.DATE)
            });
    }
};
```

Powyższy fragment kodu oznacza: „Aby odtworzyć oldInstance, należy wywołać metodę new (czyli konstruktor) dla obiektu Employee.class i dostarczyć mu odpowiednich parametrów”.

Nazwa parametru `oldInstance` jest może nieco myląca, a oznacza po prostu zapisywana instancję.

Istnieją dwa sposoby instalowania delegatu trwałości. Możemy związać go z obiektem XML ↳Writer:

```
out.setPersistenceDelegate(Employee.class, delegate);
```

Lub skonfigurować atrybut `persistenceDelegate` *deskryptora ziarnka* w klasie BeanInfo:

```
BeanInfo info = Introspector.getBeanInfo(GregorianCalendar.class);
info.getBeanDescriptor().setValue("persistenceDelegate", delegate);
```

Po zainstalowaniu delegatu możemy już zapisywać obiekty klasy Employee. Na przykład instrukcje:

```
Object myData = new Employee("Harry Hacker", 50000, 1989, 10, 1);
out.writeObject(myData);
```

spowodują wygenerowanie następujących danych:

```
<object class="Employee">
<string>Harry Hacker</string>
<double>50000.0</double>
<int>1989</int>
<int>9</int>
<int>1</int>
</object>
```



Programista musi jedynie opisać proces *kodowania* obiektu. Nie są potrzebne żadne metody odkodowywania, ponieważ dekoder wykonuje po prostu instrukcje i wyrażenia, które znajdują się w pliku XML.

### 8.9.1.2. Tworzenie obiektu na podstawie właściwości

Jeśli wszystkie parametry konstruktora można uzyskać na podstawie właściwości obiektu `oldInstance`, to nie trzeba implementować metody `instantiate`. Wystarczy jedynie stworzyć obiekt `DefaultPersistenceDelegate` i dostarczyć nazwy właściwości.

Na przykład poniższe instrukcje instalują delegat dla klasy `Rectangle2D.Double`:

```
out.setPersistenceDelegate(Rectangle2D.Double.class,
    new DefaultPersistenceDelegate(new String[] { "x", "y", "width", "height" }));
```

W ten sposób informujemy obiekt kodujący, że: „aby zakodować obiekt klasy `Rectangle2D.Double`, należy pobrać jego właściwości `x`, `y`, `width` i `height` i wywołać konstruktor dla tych czterech wartości”. W efekcie uzyskamy następujące kodowanie obiektu:

```
<object class="java.awt.geom.Rectangle2D$Double">
<double>5.0</double>
<double>10.0</double>
<double>20.0</double>
<double>30.0</double>
</object>
```

Jeśli jesteś autorem klasy, możesz pokusić się o jeszcze lepsze rozwiązanie. Wystarczy oznaczyć konstruktor adnotacją @ConstructorProperties. Założymy na przykład, że klasa Employee ma konstruktor o trzech parametrach (name, salary i hireDay). Wtedy adnotacja konstruktora będzie mieć następującą postać:

```
@ConstructorProperties({"name", "salary", "hireDay"})
public Employee(String n, double s, Date d)
```

Informuje ona obiekt kodujący, że należy wywołać metody getName, getSalary i getHireDay i zapisać zwrócone przez nie wartości w wyrażeniu object.

Adnotację @ConstructorProperties wprowadzono w Java SE 6 i jak dotychczas znalazła ona zastosowanie tylko w przypadku klas interfejsu programowego JMX (Java Management Extensions).

### 8.9.1.3. Tworzenie obiektu przez metodę fabryki

Czasami musimy zapisać obiekty, które zostały uzyskane od metody fabryki, a nie konstruktora. Przykładem może być obiekt klasy InetAddress:

```
byte[] bytes = new byte[] { 127, 0, 0, 1 };
InetAddress address = InetAddress.getByAddress(bytes);
```

Metoda instantiate należąca do klasy PersistenceDelegate generuje wywołanie metody fabryki.

```
protected Expression instantiate(Object oldInstance, Encoder out)
{
    return new Expression(oldInstance, InetAddress.class, "getByAddress",
        new Object[] { ((InetAddress) oldInstance).getAddress() });
}
```

Przykładowy wynik działania może wyglądać następująco:

```
<object class="java.net.InetAddress" method="getByAddress">
<array class="byte" length="4">
<void index="0">
<byte>127</byte>
</void>
<void index="3">
<byte>1</byte>
</void>
</array>
</object>
```



Delegat ten należy zainstalować dla konkretnej klasy pochodnej, na przykład InetAddress, a nie dla klasy abstrakcyjnej InetAddress!

### 8.9.1.4. Operacje po utworzeniu obiektu

Odtworzenie stanu obiektów niektórych klas wymaga wywołania metod, które nie konfigurują właściwości obiektów. W tym celu musimy przesłonić metodę initialize klasy DefaultPer

→sistenceDelegate. Metoda initialize jest wywoływana po metodzie instantiate i pozwala wygenerować sekwencję instrukcji, które zostaną zapisane w archiwum.

Rozważmy na przykład klasę BitSet. Aby odtworzyć obiekt klasy BitSet, musimy ustawić wszystkie bity tak jak w oryginalnym obiekcie. Poniższa metoda initialize generuje odpowiednie instrukcje:

```
protected void initialize (Class type, Object oldInstance, Object newInstance,
→Encoder out)
{
    super.initialize(type, oldInstance, newInstance, out);
    BitSet bs = (BitSet) oldInstance;
    for (int i = bs.nextSetBit(0); i >= 0; i = bs.nextSetBit(i + 1))
        out.writeStatement(new Statement(bs, "set", new Object[] { i, i + 1, true } ));
}
```

Przykładowy wynik jej działania może wyglądać następująco:

```
<object class="java.util.BitSet">
    <void method="set">
        <int>1</int>
        <int>2</int>
        <boolean>true</boolean>
    </void>
    <void method="set">
        <int>4</int>
        <int>5</int>
        <boolean>true</boolean>
    </void>
</object>
```



Bardziej sensowne byłoby napisać instrukcję postaci new Statement(bs, "set", new Object[] { i }), ale wtedy XMLWriter utworzyłby niepotrzebną instrukcję, która konfigurowałaby właściwość o pustej nazwie.

### 8.9.1.5. Właściwości tymczasowe

Czasami klasa posiada właściwość wraz odpowiednimi metodami set i get. Właściwość ta zostaje wykryta przez XMLEncoder, ale nie chcemy, aby jej wartość została zapisana w archiwum. Aby zapobiec archiwizacji takiej właściwości, musimy oznaczyć ją jako transient w deskryptorze właściwości. Na przykład poniższy kod oznacza właściwość removeProperty klasy DamageReporter (która omówimy szczegółowo w następnym podrozdziale) jako transient.

```
BeanInfo info = Introspector.getBeanInfo(DamageReport.class);
for (PropertyDescriptor desc : info.getPropertyDescriptors())
    if (desc.getName().equals("removeMode"))
        desc.setValue("transient", Boolean.TRUE);
```

Program przedstawiony na listingu 8.9 ilustruje działanie różnych delegatów trwałości. Należy pamiętać, że stanowi on ilustrację najgorszego przypadku, ponieważ w prawdziwych aplikacjach wiele klas może być archiwizowanych bez użycia delegatów.

**Listing 8.9.** persistenceDelegate/PersistenceDelegateTest.java

```
package persistenceDelegate;

import java.awt.geom.*;
import java.beans.*;
import java.net.*;
import java.util.*;

/**
 * Program demonstrujący użycie różnych delegatów trwałości.
 * @version 1.01 2007-10-03
 * @author Cay Horstmann
 */
public class PersistenceDelegateTest
{
    public static class Point
    {
        private final int x, y;

        @ConstructorProperties( { "x", "y" } )
        public Point(int x, int y)
        {
            this.x = x;
            this.y = y;
        }

        public int getX()
        {
            return x;
        }

        public int getY()
        {
            return y;
        }
    }

    public static void main(String[] args) throws Exception
    {
        PersistenceDelegate delegate = new PersistenceDelegate()
        {
            protected Expression instantiate(Object oldInstance, Encoder out)
            {
                Employee e = (Employee) oldInstance;
                GregorianCalendar c = new GregorianCalendar();
                c.setTime(e.getHireDay());
                return new Expression(oldInstance, Employee.class, "new", new Object[] {
                    e.getName(), e.getSalary(), c.get(Calendar.YEAR),
                    c.get(Calendar.MONTH),
                    c.get(Calendar.DATE) });
            }
        };
        BeanInfo info = Introspector.getBeanInfo(Employee.class);
        info.getBeanDescriptor().setValue("persistenceDelegate", delegate);

        XMLEncoder out = new XMLEncoder(System.out);
    }
}
```

```

out.setExceptionListener(new ExceptionListener()
{
    public void exceptionThrown(Exception e)
    {
        e.printStackTrace();
    }
});

out.setPersistenceDelegate(Rectangle2D.Double.class, new DefaultPersistenceDelegate(
    new String[] { "x", "y", "width", "height" }));

out.setPersistenceDelegate(InetAddress.class, new DefaultPersistenceDelegate()
{
    protected Expression instantiate(Object oldInstance, Encoder out)
    {
        return new Expression(oldInstance, InetAddress.class, "getByAddress",
            new Object[] { ((InetAddress) oldInstance).getAddress() });
    }
});

out.setPersistenceDelegate(BitSet.class, new DefaultPersistenceDelegate()
{
    protected void initialize(Class<?> type, Object oldInstance, Object
        newInstance, Encoder out)
    {
        super.initialize(type, oldInstance, newInstance, out);
        BitSet bs = (BitSet) oldInstance;
        for (int i = bs.nextSetBit(0); i >= 0; i = bs.nextSetBit(i + 1))
            out.writeStatement(new Statement(bs, "set", new Object[] { i, i +
                1, true }));
    }
});

out.writeObject(new Employee("Harry Hacker", 50000, 1989, 10, 1));
out.writeObject(new Point(17, 29));
out.writeObject(new java.awt.geom.Rectangle2D.Double(5, 10, 20, 30));
out.writeObject(InetAddress.getLocalHost());
BitSet bs = new BitSet();
bs.set(1, 4);
bs.clear(2, 3);
out.writeObject(bs);
out.close();
}
}

```

## 8.9.2. Kompletny przykład zastosowania trwałości JavaBeans

Omówienie mechanizmu trwałości JavaBeans zakończymy kompletnym przykładem jego zastosowania (patrz rysunek 8.15). Program ten zapisuje raport o uszkodzeniach wynajmowanych samochodów. Agent wynajmujący samochód wprowadza numer wynajmu, wybiera typ samochodu, klikając myszą obszary uszkodzeń auta i zapisuje raport. Program może ładować istniejące raporty o uszkodzeniach. Kod programu przedstawiony został na listingu 8.10.

**Rysunek 8.15.**

Program  
*DamageReporter*  
w działaniu



**Listing 8.10.** damageReporter/DamageReporterFrame.java

```
package damageReporter;

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.beans.*;
import java.io.*;
import java.util.*;
import javax.swing.*;

public class DamageReporterFrame extends JFrame
{
    private JTextField rentalRecord;
    private JComboBox<DamageReport.CarType> carType;
    private JComponent carComponent;
    private JRadioButton addButton;
    private JRadioButton removeButton;
    private DamageReport report;
    private JFileChooser chooser;

    private static Map<DamageReport.CarType, Shape> shapes = new EnumMap<>(
        DamageReport.CarType.class);

    public DamageReporterFrame()
    {
        chooser = new JFileChooser();
        chooser.setCurrentDirectory(new File("."));

        report = new DamageReport();
        report.setCarType(DamageReport.CarType.SEDAN);

        // konfiguruje menu
        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);

        JMenu menu = new JMenu("File");
        menuBar.add(menu);
    }
}
```

```

menuBar.add(menu);

JMenuItem openItem = new JMenuItem("Open");
menu.add(openItem);
openItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent evt)
    {
        // okno wyboru pliku
        int r = chooser.showOpenDialog(null);

        // otwiera wybrany plik
        if (r == JFileChooser.APPROVE_OPTION)
        {
            try
            {
                File file = chooser.getSelectedFile();
                XMLDecoder decoder = new XMLDecoder(new FileInputStream(file));
                report = (DamageReport) decoder.readObject();
                decoder.close();
                rentalRecord.setText(report.getRentalRecord());
                carType.setSelectedItem(report.getCarType());
                repaint();
            }
            catch (IOException e)
            {
                JOptionPane.showMessageDialog(null, e);
            }
        }
    }
});

JMenuItem saveItem = new JMenuItem("Save");
menu.add(saveItem);
saveItem.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent evt)
    {
        report.setRentalRecord(rentalRecord.getText());
        chooser.setSelectedFile(new File(rentalRecord.getText() + ".xml"));

        // okno wyboru pliku
        int r = chooser.showSaveDialog(null);

        // zapisuje w wybranym pliku
        if (r == JFileChooser.APPROVE_OPTION)
        {
            try
            {
                File file = chooser.getSelectedFile();
                XMLEncoder encoder = new XMLEncoder(new FileOutputStream(file));
                report.configureEncoder(encoder);
                encoder.writeObject(report);
                encoder.close();
            }
            catch (IOException e)
            {
                JOptionPane.showMessageDialog(null, e);
            }
        }
    }
});

```

```
        }
    }
}
});

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

// lista rozwijana wyboru typu samochodu
rentalRecord = new JTextField();
carType = new JComboBox<>();
carType.addItem(DamageReport.CarType.SEDAN);
carType.addItem(DamageReport.CarType.WAGON);
carType.addItem(DamageReport.CarType.SUV);

carType.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        DamageReport.CarType item = carType.getItemAt(carType.getSelectedIndex());
        report.setCarType(item);
        repaint();
    }
});

// komponent prezentacji samochodu i uszkodzeń
carComponent = new JComponent()
{
    private static final int DEFAULT_WIDTH = 400;
    private static final int DEFAULT_HEIGHT = 200;

    public void paintComponent(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;
        g2.setColor(new Color(0.9f, 0.9f, 0.45f));
        g2.fillRect(0, 0, getWidth(), getHeight());
        g2.setColor(Color.BLACK);
        g2.draw(shapes.get(report.getCarType()));
        report.drawDamage(g2);
    }

    public Dimension getPreferredSize()
    {
        return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
    }
};
carComponent.addMouseListener(new MouseAdapter()
{
    public void mousePressed(MouseEvent event)
    {
        report.click(new Point2D.Double(event.getX(), event.getY()));
    }
});
```

```

        repaint();
    }
}):
// przyciski konfigurujące efekt klikania
addButton = new JRadioButton("Add");
removeButton = new JRadioButton("Remove");
ButtonGroup group = new ButtonGroup();
JPanel buttonPanel = new JPanel();
group.add(addButton);
buttonPanel.add(addButton);
group.add(removeButton);
buttonPanel.add(removeButton);
addButton.setSelected(!report.getRemoveMode());
removeButton.setSelected(report.getRemoveMode());
addButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        report.setRemoveMode(false);
    }
});
removeButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        report.setRemoveMode(true);
    }
});

// komponenty układu
JPanel gridPanel = new JPanel();
gridPanel.setLayout(new GridLayout(0, 2));
gridPanel.add(new JLabel("Rental Record"));
gridPanel.add(rentalRecord);
gridPanel.add(new JLabel("Type of Car"));
gridPanel.add(carType);
gridPanel.add(new JLabel("Operation"));
gridPanel.add(buttonPanel);

add(gridPanel, BorderLayout.NORTH);
add(carComponent, BorderLayout.CENTER);
pack();
}

static
{
    int width = 200;
    int x = 50;
    int y = 50;
    Rectangle2D.Double body = new Rectangle2D.Double(x, y + width / 6, width - 1,
    ↴width / 6);
    Ellipse2D.Double frontTire = new Ellipse2D.Double(x + width / 6, y + width /
    ↴3, width / 6, width / 6);
    Ellipse2D.Double rearTire = new Ellipse2D.Double(x + width * 2 / 3, y + width /
    ↴3, width / 6, width / 6);

    Point2D.Double p1 = new Point2D.Double(x + width / 6, y + width / 6);
}

```

```
Point2D.Double p2 = new Point2D.Double(x + width / 3, y);
Point2D.Double p3 = new Point2D.Double(x + width * 2 / 3, y);
Point2D.Double p4 = new Point2D.Double(x + width * 5 / 6, y + width / 6);

Line2D.Double frontWindshield = new Line2D.Double(p1, p2);
Line2D.Double roofTop = new Line2D.Double(p2, p3);
Line2D.Double rearWindshield = new Line2D.Double(p3, p4);

GeneralPath sedanPath = new GeneralPath();
sedanPath.append(frontTire, false);
sedanPath.append(rearTire, false);
sedanPath.append(body, false);
sedanPath.append(frontWindshield, false);
sedanPath.append(roofTop, false);
sedanPath.append(rearWindshield, false);
shapes.put(DamageReport.CarType.SEDAN, sedanPath);

Point2D.Double p5 = new Point2D.Double(x + width * 11 / 12, y);
Point2D.Double p6 = new Point2D.Double(x + width, y + width / 6);
roofTop = new Line2D.Double(p2, p5);
rearWindshield = new Line2D.Double(p5, p6);

GeneralPath wagonPath = new GeneralPath();
wagonPath.append(frontTire, false);
wagonPath.append(rearTire, false);
wagonPath.append(body, false);
wagonPath.append(frontWindshield, false);
wagonPath.append(roofTop, false);
wagonPath.append(rearWindshield, false);
shapes.put(DamageReport.CarType.WAGON, wagonPath);

Point2D.Double p7 = new Point2D.Double(x + width / 3, y - width / 6);
Point2D.Double p8 = new Point2D.Double(x + width * 11 / 12, y - width / 6);
frontWindshield = new Line2D.Double(p1, p7);
roofTop = new Line2D.Double(p7, p8);
rearWindshield = new Line2D.Double(p8, p6);

GeneralPath suvPath = new GeneralPath();
suvPath.append(frontTire, false);
suvPath.append(rearTire, false);
suvPath.append(body, false);
suvPath.append(frontWindshield, false);
suvPath.append(roofTop, false);
suvPath.append(rearWindshield, false);
shapes.put(DamageReport.CarType.SUV, suvPath);
}

}
```

Program używa mechanizmu trwałości JavaBeans do zapisu i ładowania obiektów DamageReport (patrz listing 8.11). Stanowi on ilustrację następujących aspektów technologii trwałości:

- Właściwości są automatycznie zapisywane i odtwarzane. Właściwości rentalRecord i carType nie wymagają dodatkowych działań ze strony programisty.
- Dodatkowe operacje po utworzeniu obiektu wymagane są do odtworzenia obszarów uszkodzeń. Delegat trwałości generuje instrukcje zawierające wywołania metody click.

- Klasa Point2D.Double potrzebuje delegatu DefaultPersistenceDelegate, który tworzy punkt na podstawie jego właściwości x i y.
- Właściwość removeMode (która określa, czy kliknięcie myszą dodało lub usunęło obszar uszkodzenia) jest tymczasowa i nie jest zapisywana w raportach uszkodzeń.

**Listing 8.11.** damageReporter/DamageReport.java

```
package damageReporter;

import java.awt.*;
import java.awt.geom.*;
import java.beans.*;
import java.util.*;

/**
 * Klasa reprezentująca raport o uszkodzeniach samochodu,
 * przechowywany i odtwarzany przy użyciu mechanizmu trwałości.
 * @version 1.22 2012-01-26
 * @author Cay Horstmann
 */
public class DamageReport
{
    private String rentalRecord;
    private CarType carType;
    private boolean removeMode;
    private java.util.List<Point2D> points = new ArrayList<>();

    private static final int MARK_SIZE = 5;

    public enum CarType
    {
        SEDAN, WAGON, SUV
    }

    // właściwość zapisywana automatycznie
    public void setRentalRecord(String newValue)
    {
        rentalRecord = newValue;
    }

    public String getRentalRecord()
    {
        return rentalRecord;
    }

    // właściwość zapisywana automatycznie
    public void setCarType(CarType newValue)
    {
        carType = newValue;
    }

    public CarType getCarType()
    {
        return carType;
    }

    // właściwość oznaczona jako transient
}
```

```
public void setRemoveMode(boolean newValue)
{
    removeMode = newValue;
}

public boolean getRemoveMode()
{
    return removeMode;
}

public void click(Point2D p)
{
    if (removeMode)
    {
        for (Point2D center : points)
        {
            Ellipse2D circle = new Ellipse2D.Double(center.getX() - MARK_SIZE, center.getY()
                - MARK_SIZE, 2 * MARK_SIZE, 2 * MARK_SIZE);
            if (circle.contains(p))
            {
                points.remove(center);
                return;
            }
        }
    }
    else points.add(p);
}

public void drawDamage(Graphics2D g2)
{
    g2.setPaint(Color.RED);
    for (Point2D center : points)
    {
        Ellipse2D circle = new Ellipse2D.Double(center.getX() - MARK_SIZE, center.getY()
            - MARK_SIZE, 2 * MARK_SIZE, 2 * MARK_SIZE);
        g2.draw(circle);
    }
}

public void configureEncoder(XMLEncoder encoder)
{
    // operacja konieczna w celu przechowywania obiektów Point2D.Double
    encoder.setPersistenceDelegate(Point2D.Double.class, new DefaultPersistenceDelegate(
        new String[] { "x", "y" }));

    // operacja konieczna, ponieważ lista punktów nie jest
    // (i nie powinna być) dostępna jako właściwość
    encoder.setPersistenceDelegate(DamageReport.class, new DefaultPersistenceDelegate()
    {
        protected void initialize(Class<?> type, Object oldInstance, Object newInstance,
                                  Encoder out)
        {
            super.initialize(type, oldInstance, newInstance, out);
            DamageReport r = (DamageReport) oldInstance;

            for (Point2D p : r.points)
                out.writeStatement(new Statement(oldInstance, "click", new Object[] {
                    p }));
        }
    });
}
```

```
        }
    });
}

// operacja konieczna ze względu na zapewnienie tymczasowości (transient) właściwości removeMode
static
{
    try
    {
        BeanInfo info = Introspector.getBeanInfo(DamageReport.class);
        for (PropertyDescriptor desc : info.getPropertyDescriptors())
            if (desc.getName().equals("removeMode")) desc.setValue("transient",
                Boolean.TRUE);
    }
    catch (IntrospectionException e)
    {
        e.printStackTrace();
    }
}
```

A oto przykładowy raport uszkodzeń:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.5.0" class="java.beans.XMLEncoder">
<object class="DamageReport">
    <object class="java.lang.Enum" method="valueOf">
        <class>DamageReport$CarType</class>
        <string>SEDAN</string>
    </object>
    <void property="rentalRecord">
        <string>12443-19</string>
    </void>
    <void method="click">
        <object class="java.awt.geom.Point2D$Double">
            <double>181.0</double>
            <double>84.0</double>
        </object>
    </void>
    <void method="click">
        <object class="java.awt.geom.Point2D$Double">
            <double>162.0</double>
            <double>66.0</double>
        </object>
    </void>
</object>
</java>
```



 Program przykładowy *nie* używa mechanizmu trwałości JavaBeans do zapisu interfejsu użytkownika. Zastosowania takie są interesujące przede wszystkim dla twórców narzędzi projektowania aplikacji. Natomiast w naszym przykładzie skoncentrowaliśmy się na zastosowaniu mechanizmu trwałości do przechowywania *danych aplikacji*.

Przykład ten kończy omówienie mechanizmu trwałości JavaBeans. Podsumowując, możemy stwierdzić, że mechanizm ten:

- jest przeznaczony do długotrwałego przechowywania obiektów,
- jest prosty i szybki,
- jest łatwy w implementacji,
- tworzy dane, które mogą być łatwo analizowane przez programistę,
- stanowi część standardu Java.

**API java.beans.XMLEncoder 1.4**

- `XMLEncoder(OutputStream out)`  
tworzy obiekt XMLEncoder, który wysyła tworzone dane do podanego strumienia.
- `void writeObject(Object obj)`  
archiwizuje podany obiekt.
- `void writeStatement(Statement stat)`  
zapisuje podaną instrukcję do archiwum. Metoda ta powinna być wywoływana tylko przez delegat trwałości.

**API java.beans.Encoder 1.4**

- `void setPersistenceDelegate(Class<?> type, PersistenceDelegate delegate)`
- `PersistenceDelegate getPersistenceDelegate(Class<?> type)`  
instaluje lub zwraca delegat archiwizacji obiektów podanego typu.
- `void setExceptionListener(ExceptionListener listener)`
- `ExceptionListener getExceptionListener()`  
instaluje lub zwraca obiekt nasłuchujący wyjątków powiadamiany o wyjątkach pojawiających się w procesie kodowania.

**API java.beans.ExceptionListener 1.4**

- `void exceptionThrown(Exception e)`  
metoda wywoływana, gdy zostanie zgłoszony wyjątek w procesie kodowania lub dekodowania.

**API java.beans.XMLDecoder 1.4**

- `XMLDecoder(InputStream in)`  
tworzy obiekt XMLDecoder wczytujący archiwum z podanego strumienia wejściowego.
- `Object readObject()`  
wczytuje następny obiekt z archiwum.
- `void setExceptionListener(ExceptionListener listener)`

- `ExceptionListener getExceptionListener()`  
instaluje lub zwraca obiekt nasłuchujący wyjątków powiadamiany o wyjątkach pojawiających się w procesie kodowania.

#### `java.beans.PersistenceDelegate 1.4`

- `protected abstract Expression instantiate(Object oldInstance, Encoder out)`  
zwraca wyrażenie tworzące obiekt będący odpowiednikiem `oldInstance`.
- `protected void initialize(Class<?> type, Object oldInstance, Object newInstance, Encoder out)`  
zapisuje instrukcje w `out`, która przekształca obiekt `newInstance` w odpowiednik obiektu `oldInstance`.

#### `java.beans.DefaultPersistenceDelegate 1.4`

- `DefaultPersistenceDelegate()`  
tworzy delegat trwałości dla klasy, której konstruktor nie posiada parametrów.
- `DefaultPersistenceDelegate(String[] propertyNames)`  
tworzy delegat trwałości dla klasy, której parametry konstruktora są wartościami podanych właściwości.
- `protected Expression instantiate(Object oldInstance, Encoder out)`  
zwraca wyrażenie wywołujące konstruktor bez parametrów lub dla wartości właściwości przekazanych konstruktorowi.
- `protected void initialize(Class<?> type, Object oldInstance, Object newInstance, Encoder out)`  
zapisuje instrukcje do `out`, które wywołują metody `set` obiektu `newInstance`, próbując przekształcić go w odpowiednik obiektu `oldInstance`.

#### `java.beans.Expression 1.4`

- `Expression(Object value, Object target, String methodName, Object[] parameters)`  
tworzy wyrażenie, które wywołuje podaną metodę dla obiektu `target`, korzystając z podanych parametrów. Wynikiem wyrażenia jest `value`. Aby wywołać w ten sposób konstruktor, `target` powinien być obiektem klasy `Class`, a parametr `methodName` powinien mieć wartość "new".

#### `java.beans.Statement 1.4`

- `Statement(Object target, String methodName, Object[] parameters)`  
tworzy instrukcję, która wywołuje podaną metodę obiektu `target`, używając podanych parametrów.

W ten sposób zakończyłeś lekturę trzech długich rozdziałów poświęconych programowaniu interfejsu użytkownika przy użyciu Swing, AWT i JavaBeans. W następnym rozdziale zajmujemy się zupełnie innym zagadnieniem: bezpieczeństwem. Bezpieczeństwo od początku było jedną z podstawowych właściwości platformy Java. Ponieważ świat, w którym żyjemy i programujemy, staje się coraz niebezpieczniejszy, znaczenie dogłębniego zrozumienia zasad bezpieczeństwa na platformie Java ciągle wzrasta.

# 9

## Bezpieczeństwo

W tym rozdziale:

- Ładowanie klas.
- Weryfikacja kodu maszyny wirtualnej.
- Menedżery bezpieczeństwa i pozwolenia.
- Uwierzytelnianie użytkownika
- Podpis cyfrowy.
- Podpisywanie kodu.
- Szyfrowanie.

Kiedy technologia Java pojawiła się po raz pierwszy, to skupiła na sobie uwagę nie jako doskonały język programowania, ale przede wszystkim dzięki możliwości bezpiecznego wykonywania appletów ładowanych przez Internet (więcej informacji na temat appletów odnajdziemy w 10. rozdziale książki *Java 2. Podstawy*). Oczywiście ładowanie i wykonywanie gotowych appletów ma sens jedynie wtedy, gdy odbiorca może być pewien, że nie wyrządzają one szkód na jego maszynie. Z tego powodu kwestie bezpieczeństwa były i są kluczowe zarówno dla projektantów, jak i użytkowników technologii Java. Inaczej więc niż w przypadku innych języków programowania i systemów, gdzie kwestiami bezpieczeństwa zajęto się dopiero podczas ich udoskonalania, często dopiero w reakcji na powstałe zagrożenia, mechanizmy bezpieczeństwa stanowią integralną część technologii Java od samego jej powstania.

Na platformie Java istnieją trzy zasadnicze mechanizmy zapewniające bezpieczeństwo. Są to:

- cechy samego języka programowania (na przykład sprawdzanie zakresu indeksów tablic, dopuszczanie tylko dozwolonych konwersji typów, brak arytmetyki wskaźników i wiele innych),
- mechanizm kontroli dostępu nadzorujący działania kodu (dostęp do plików, dostęp do sieci itp.),

- podpisywanie kodu umożliwiające wykorzystanie standardowych algorytmów kryptograficznych w celu uwierzytelnienia kodu w języku Java, dzięki czemu użytkownicy kodu mogą zawsze ustalić jego autora oraz upewnić się, że kod nie był modyfikowany od momentu jego podpisania.

Podczas ładowania plików klas na maszynę wirtualną sprawdzana jest ich integralność. Pokażemy, w jaki sposób ten mechanizm potrafi wykryć próby modyfikacji kodu klas.

Dla zapewnienia maksimum bezpieczeństwa tak w przypadku standardowego mechanizmu ładowania klas, jak i wykorzystania własnych procedur, niezbędna jest ich współpraca z klasą *menedżera bezpieczeństwa*, która nadzoruje operacje wykonywane przez kod. W rozdziale tym pokażemy w szczegółach sposób zarządzania bezpieczeństwem na platformie Java.

W dalszej części bieżącego rozdziału zajmiemy się algorytmami kryptograficznymi udostępnianymi przez pakiet `java.security`, które pozwalają podpisywać kod i sprawdzać autentyczność użytkowników.

Jak zwykle skoncentrujemy się na tych zagadnieniach, które są najważniejsze dla programistów tworzących aplikacje na platformie Java. Pełniejsze omówienie zagadnień bezpieczeństwa platformy Java znajdziemy w książce *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*, wyd. drugie, której autorami są Li Gong, Gary Ellison i Mary Dageforde (Prentice Hall 2003).

## 9.1. Ładowanie klas

Kompilator języka Java przekształca kod źródłowy programu w kod wykonywalny hipotetycznej maszyny zwanej *maszyną wirtualną*. Kod maszyny wirtualnej umieszczany jest w plikach klas posiadających rozszerzenie `.class`. Każdy plik klas zawiera kod definicji i implementacji jednej klasy lub interfejsu. Pliki klas są następnie interpretowane przez program, tłumaczący kod maszyny wirtualnej na kod maszyny macierzystej, na której wykonywany jest program.

Interpreter maszyny wirtualnej ładuje jedynie te pliki klas, które są niezbędne do wykonania programu. Założymy na przykład, że wykonanie programu rozpoczyna się od pliku klasy `MyProgram.class`. Maszyna wirtualna musi wtedy wykonać następujące działania.

1. Maszyna wirtualna dysponuje mechanizmem ładowania plików klas na przykład z dysku lub sieci Internet, który wykorzystuje do załadowania pliku klasy `MyProgram`.
2. Jeśli klasa `MyProgram` posiada zmienne instancji o typach innych klas lub klasy bazowe, to ładowane są także pliki tych klas. (Proces ładowania wszystkich klas, od których zależy dana klasa, nazywamy *rozwiązywaniem klasy*).
3. Maszyna wirtualna wykonuje metodę `main` klasy `MyProgram`. Ponieważ jest to metoda statyczna, nie jest tworzona instancja klasy `MyProgram`.

- 4.** Jeśli metoda `main` lub jakakolwiek wywoływaną przez nią metoda wymaga innego pliku klasy niż dotychczas załadowane, to jest on ładowany w trakcie wykonania programu.

Mechanizm ładowania klas nie korzysta z pojedynczej procedury ładowania klas. W przypadku każdego programu w języku Java wykorzystywane są przynajmniej trzy takie procedury:

- początkowa,
- rozszerzeń,
- systemowa (inaczej aplikacji).

Procedura początkowa ładuje klasy systemowe (typowo z pliku `rt.jar`). Stanowi ona integralną część maszyny wirtualnej zazwyczaj zaimplementowaną w języku C. Dla procedury początkowej nie jest dostępny reprezentujący ją obiekt `ClassLoader` i dlatego na przykład poniższe wywołanie

```
String.class.getClassLoader()
```

zwróci zawsze wartość `null`.

Procedura ładowania rozszerzeń jest używana dla standardowych rozszerzeń maszyny wirtualnej z katalogu `jre/lib/ext`. Jeśli w katalogu tym umieścimy własny plik typu JAR, to procedura rozszerzona odnajdzie zawarte w nim klasy bez konieczności podawania ścieżki dostępu do klas. (Niektórzy projektanci zalecają wykorzystanie tej możliwości w celu ograniczenia nadmiernie rozbudowanych ścieżek dostępu do klas, ale posiada ona także pewne wady omówione w ramce poniżej).

Systemowa procedura ładowania klas ładuje pliki klas aplikacji. Poszukuje ich w katalogach i plikach typu JAR i ZIP, korzystając ze ścieżki dostępu do klas określonej za pomocą zmiennej środowiskowej `CLASSPATH` lub opcji `-classpath` podanej w wierszu poleceń.

W implementacji technologii Java dostarczanej przez firmę Oracle rozszerzona procedura ładowania klas i systemowa procedura ładowania klas zaimplementowane zostały w języku Java. Obie są instancjami klasy `URLClassLoader`.



Jeśli w katalogu `jre/lib/ext` umieścimy plik JAR i jedna ze znajdujących się w nim klas musi załadować inną klasę, która nie jest klasą systemową lub klasą rozszerzeń, to operacja ta nie powiedzie się, ponieważ procedura ładowania klas rozszerzeń *nie używa ścieżki dostępu do klas*. Należy wziąć to pod uwagę, zanim zdecydujemy się wykorzystać katalog rozszerzeń jako cudowny środek na problemy związane z zarządzaniem plikami klas.



Klasy mogą być ładowane nie tylko z wymienionych wcześniej miejsc, ale i z katalogu `jre/lib/endorsed`. Jednak mechanizm ten może być używany wyłącznie w celu zastąpienia pewnych standardowych bibliotek Java (na przykład służących do obsługi XML i CORBA) nowszymi wersjami. Więcej informacji na ten temat znajdziesz na stronie <http://docs.oracle.com/javase/7/docs/technotes/guides/standards>.

## 9.1.1. Hierarchia klas ładowania

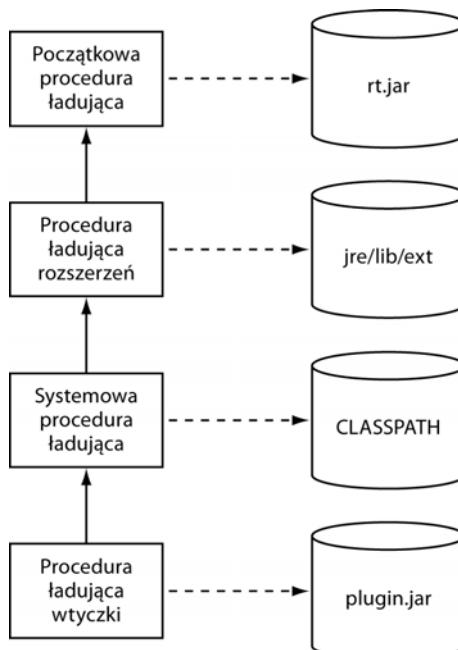
Klasy reprezentujące procedury ładowania są powiązane zależnością podlegania. Każda z procedur, z wyjątkiem procedury początkowej, posiada procedurę nadziedną. Każda procedura musi najpierw umożliwić załadowanie pliku klasy swojej procedurze nadziednej i dopiero próbuje załadować go sama, gdy nie uda się to procedurze nadziednej. Na przykład gdy procedura systemowa otrzymała polecenie załadowania klasy systemowej (na przykład `java.util.ArrayList`), to najpierw przekaże je procedurze rozszerzonej, która z kolei przekaże je procedurze początkowej. Ponieważ procedura początkowa odnajdzie plik klasy w pliku `rt.jar` i załaduje go, to żadna z pozostałych procedur ładowania klas nie będzie już kontynuować procesu poszukiwania tej klasy.

Niektóre programy posiadają modularną architekturę, w której pewne części kodu tworzą opcjonalne wtyczki. Jeśli wtyczki te są umieszczone w plikach JAR, to klasy wtyczek możemy załadować za pomocą instancji klasy `URLClassLoader`:

```
URL url = new URL("file:///path/to/plugin.jar");
URLClassLoader pluginLoader = new URLClassLoader(new URL[] { url });
Class<?> c1 = pluginLoader.loadClass("mypackage.MyClass");
```

Ponieważ w konstruktorze `URLClassLoader` nie została wyspecyfikowana żadna procedura nadziedzona, to nadziedzną procedurą ładowania dla `pluginLoader` będzie systemowa procedura ładowania. Rysunek 9.1 przedstawia tę hierarchię.

**Rysunek 9.1.**  
Hierarchia procedur ładowających klasy



W większości przypadków programista nie musi zajmować się w ogóle procedurami ładowania klas. Większość klas zostaje załadowana automatycznie, ponieważ jest używana przez inne klasy.

Czasami jednak musimy zainterweniować i określić procedurę ładowającą klasy. Rozważmy następujący scenariusz.

- Kod aplikacji zawiera pomocniczą metodę, która używa wywołania `Class.forName(classNameString)`.
- Metoda ta zostaje wywołana przez klasę wtyczki.
- Łącuch `classNameString` określa klasę umieszczoną w pliku JAR wtyczki.

Autor wtyczki spodziewał się, że klasa ta zostanie załadowana. Jednak klasa zawierająca metodę pomocniczą została załadowana przez systemową procedurę ładowającą, której używa również metoda `Class.forName`. Zatem klasy w pliku JAR wtyczki nie są dla niej widoczne. Zjawisko to nosi nazwę *inwersji procedury ładowającej*.

Rozwiążanie tego problemu jest możliwe, jeśli metoda pomocnicza zastosuje właściwą procedurę ładowającą klasę. Metoda ta może na przykład wymagać podania procedury ładowającej jako jednego ze swych parametrów. Alternatywnie, może wymagać skonfigurowania właściwej procedury ładowającej jako *procedury ładowającej kontekstu* bieżącego wątku. Strategia ta jest stosowana przez wiele szkieletów (na przykład szkielety JAXP i JNDI, które omówiliśmy w rozdziałach 2. i 4.).

Każdy wątek posiada referencję procedury ładowającej zwanej procedurą ładowającą kontekstu. Procedura ładowającą kontekstu w przypadku wątku głównego jest systemowa procedura ładowająca. Zatem dopóki sami tego nie zmienimy, procedurą ładowającą kontekstu dla wszystkich wątków będzie systemowa procedura ładowająca.

Możemy jednak skonfigurować dowolną procedurę ładowającą klasy jako procedurę ładowającą kontekstu wątku:

```
Thread t = Thread.currentThread();
t.setContextClassLoader(loader);
```

Metoda pomocnicza może następnie pobrać procedurę ładowającą kontekstu:

```
Thread t = Thread.currentThread();
ClassLoader loader = t.getContextClassLoader();
Class c1 = loader.loadClass(className);
```

Pozostaje jednak pytanie, kiedy skonfigurować procedurę ładowającą wtyczkę jako procedurę ładowającą kontekstu. Decyzję tę musi podjąć projektant aplikacji. Zwykle konfiguruje się procedurę ładowającą kontekstu, wywołując metodę klasy wtyczki, która została załadowana przez inną procedurę ładowającą klasy. Alternatywnie kod wywołujący metodę pomocniczą może sam skonfigurować procedurę ładowającą kontekstu.



Jeśli implementujemy metodę, która ładuje klasę, używając w tym celu nazwy klasy, to dobrze jest zaoferować kodowi wywołującemu możliwość wyboru pomiędzy jawnym przekazaniem procedury ładowającej i wykorzystaniem procedury ładowającej kontekstu. Nie należy po prostu używać tej samej procedury ładowającej, która załadowała klasę metody.

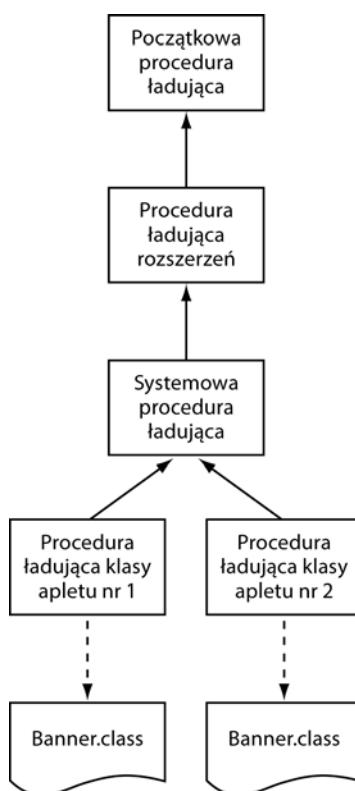
## 9.1.2. Zastosowanie procedur ładujących w roli przestrzeni nazw

Nazwy pakietów wykorzystujemy w języku Java, aby wyeliminować konflikty nazw klas. Na przykład w standardowej bibliotece języka Java istnieją dwie klasy o nazwie Date, których pełne nazwy związane z umieszczeniem ich w różnych pakietach to `java.util.Date` i `java.sql.Date`. Nazwa skrócona Date służy jedynie wygodzie programisty i wymaga podania w programie odpowiedniego polecenia `import`. Wykonywany program zawsze posługuje się pełnymi nazwami klas zawierającymi nazwy pakietów.

Dlatego zaskakujący może wydać się początkowo fakt, że jedna i ta sama maszyna wirtualna może wykonywać dwie klasy o takiej samej nazwie klasy *i pakietu*. Jest to możliwe, ponieważ klasa określona jest nie tylko przez jej pełną nazwę, ale i procedurę ładowającą. Okazuje się to przydatne podczas ładowania kodu z wielu różnych źródeł. Na przykład przeglądarka internetowa używa osobnej instancji klasy ładowającej apłyty dla każdej ze stron internetowych. Umożliwia to maszynie wirtualnej rozróżnić klas apletów na różnych stronach bez względu na sposób, w jaki je nazwano. Na rysunku 9.2 został przedstawiony przykład takiej sytuacji. Założymy, że strona witryny internetowej zawiera dwa apłyty dostarczane przez różnych reklamodawców, a każdy aparat zawiera klasę o nazwie Banner. Ponieważ każdy aparat zostaje załadowany przez osobną procedurę ładowającą klasy, to obie klasy Banner są poprawnie rozróżniane i nie występuje żaden konflikt nazw.

**Rysunek 9.2.**

Dwie procedury ładujące ładują różne klasy o tej samej nazwie





Rozwiążanie to posiada także inne zastosowania związane z wdrażaniem serwletów i komponentów Enterprise Java Beans. Więcej informacji na ten temat znajdziemy na stronie <http://zeroturnaround.com/labs/rjc301>.

### 9.1.3. Implementacja własnej procedury ładowającej

Własne procedury ładowające klasy tworzymy z myślą o wyspecjalizowanych zastosowaniach. W ten sposób możemy podjąć określone działania, zanim kod klasy zostanie przekazany maszynie wirtualnej. Możemy na przykład odmówić załadowania klasy, dla której użytkownik nie wykupił odpowiedniej licencji.

Tworząc własną procedurę ładowającą, musimy rozszerzyć klasę ClassLoader i zastąpić jej metodę

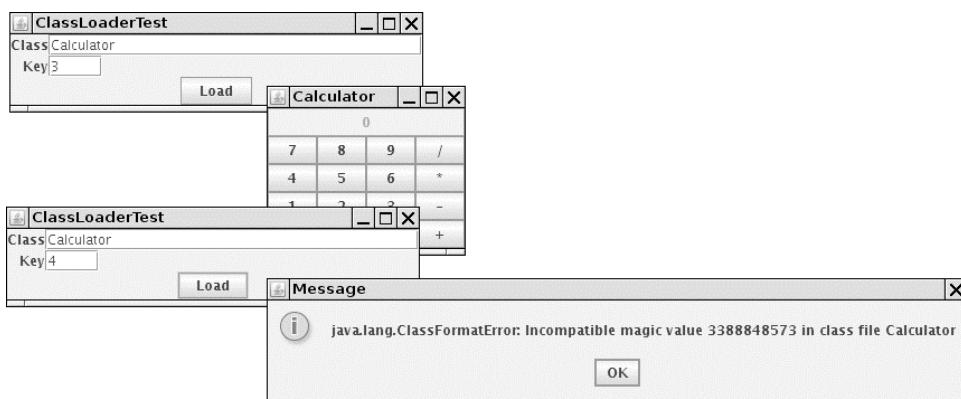
```
findClass(String className)
```

Metoda loadClass klasy bazowej ClassLoader zajmuje się delegacją żądań do procedury nadzędnej i wywołuje metodę findClass tylko, jeśli klasa nie jest jeszcze załadowana i procedura nadzędna nie zdołała jej odnaleźć i załadować.

Implementacja metody findClass musi:

1. załadować kod maszyny wirtualnej dla danej klasy, korzystając z lokalnego systemu plików lub innego źródła,
2. wywołać metodę defineClass klasy bazowej ClassLoader w celu przekazania kodu klasy maszynie wirtualnej.

Program, którego kod źródłowy zawiera listing 9.1, implementuje klasę procedury ładowającej zaszyfrowane pliki klas. Program wymaga podania nazwy pierwszej z ładowanych klas (czyli zawierającej metodę main) oraz klucza umożliwiającego jej odszyfrowanie. Następnie korzysta ze specjalizowanej procedury ładowającej i wywołuje metodę main załadowanej klasy. Procedura ładowająca odszyfrowuje podaną klasę oraz wszystkie używane przez nią klasy, które nie są systemowe. Na koniec program wywołuje metodę main załadowanej klasy (patrz rysunek 9.3).



**Rysunek 9.3.** Program ClassLoaderTest w działaniu

**Listing 9.1.** classLoader/ClassLoaderTest.java

```
package classLoader;

import java.io.*;
import java.lang.reflect.*;
import java.nio.file.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Program demonstrujący wykorzystanie własnej procedury ładowającej
 * zaszyfrowane pliki klas.
 * @version 1.23 2012-06-08
 * @author Cay Horstmann
 */
public class ClassLoaderTest
{
    public static void main(String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new ClassLoaderFrame();
                frame.setTitle("ClassLoaderTest");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
            }
        });
    }
}

/**
 * Ramka zawierająca dwa pola tekstowe
 * umożliwiające wprowadzenie nazwy ładowanej klasy
 * i klucza szyfrowania.
 */
class ClassLoaderFrame extends JFrame
{
    private JTextField keyField = new JTextField("3", 4);
    private JTextField nameField = new JTextField("Calculator", 30);
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;

    public ClassLoaderFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
        setLayout(new GridBagLayout());
        add(new JLabel("Class"), new GBC(0, 0).setAnchor(GBC.EAST));
        add(nameField, new GBC(1, 0).setWeight(100, 0).setAnchor(GBC.WEST));
        add(new JLabel("Key"), new GBC(0, 1).setAnchor(GBC.EAST));
        add(keyField, new GBC(1, 1).setWeight(100, 0).setAnchor(GBC.WEST));
        JButton loadButton = new JButton("Load");
        add(loadButton, new GBC(0, 2, 2, 1));
        loadButton.addActionListener(new ActionListener()
        {
```

```

        public void actionPerformed(ActionEvent event)
        {
            runClass(nameField.getText(), keyField.getText());
        }
    });
    pack();
}

< /**
 * Wywołuje metodę main danej klasy.
 * @param name nazwa klasy
 * @param key klucz szyfrowania pliku klasy
 */
public void runClass(String name, String key)
{
    try
    {
        ClassLoader loader = new CryptoClassLoader(Integer.parseInt(key));
        Class<?> c = loader.loadClass(name);
        Method m = c.getMethod("main", String[].class);
        m.invoke(null, (Object) new String[] {});
    }
    catch (Throwable e)
    {
        JOptionPane.showMessageDialog(this, e);
    }
}

}

< /**
 * Klasa procedury ładującej zaszyfrowane pliki klas.
 */
class CryptoClassLoader extends ClassLoader
{
    private int key;

    /**
     * Tworzy obiekt klasy CryptoClassLoader.
     * @param k klucz szyfrowania
     */
    public CryptoClassLoader(int k)
    {
        key = k;
    }

    protected Class<?> findClass(String name) throws ClassNotFoundException
    {
        try
        {
            byte[] classBytes = null;
            classBytes = loadClassBytes(name);
            Class<?> cl = defineClass(name, classBytes, 0, classBytes.length);
            if (cl == null) throw new ClassNotFoundException(name);
            return cl;
        }
        catch (IOException e)
        {
            throw new ClassNotFoundException(name);
        }
    }
}

```

```

        }

    /**
     * Ładuje i odszyfrowuje kod klasy.
     * @param name nazwa klasy
     * @return tablica zawierająca kod klasy
     */
    private byte[] loadClassBytes(String name) throws IOException
    {
        String cname = name.replace('.', '/') + ".caesar";
        byte[] bytes = Files.readAllBytes(Paths.get(cname));
        for (int i = 0; i < bytes.length; i++)
            bytes[i] = (byte) (bytes[i] - key);
        return bytes;
    }
}

```

W celu uproszczenia przykładu zignorowaliśmy dorobek ostatnich 2000 lat w dziedzinie kryptografii i do szyfrowania klas zastosowaliśmy prościutki szyfr Cezara.



Doskonała książka *The Codebreakers* autorstwa Davida Kahna (Macmillan, NY, 1967, s. 84) przypisuje autorstwo szyfru Cezara Swetoniuszowi. Szyfr ten polega na przesunięciu 24 liter alfabetu rzymskiego o trzy pozycje.

Gdy pisaliśmy tę książkę, wyrafinowane metody szyfrowania były przedmiotem ograniczeń eksportowych nałożonych przez rząd Stanów Zjednoczonych. Dlatego też zastosowaliśmy szyfr Cezara, który jest tak prosty, że prawdopodobnie jego eksport jest już legalny.

Nasza wersja szyfru Cezara używa jako klucza liczby z przedziału od 1 do 255. Aby odszyfrować pojedyńczy bajt, musimy dodać najpierw do niego klucz, a następnie wyznaczyć resztę z dzielenia przez 256. Klasa Caesar, której kod źródłowy pokazuje listing 9.2, przeprowadza proces szyfrowania.

#### **Listing 9.2.** classLoader/CaesarTest.java

```

package classLoader;

import java.io.*;

/**
 * Szyfruje plik za pomocą szyfru Cezara.
 * @version 1.01 2012-06-10
 * @author Cay Horstmann
 */
public class Caesar
{
    public static void main(String[] args) throws Exception
    {
        if (args.length != 3)
        {
            System.out.println("USAGE: java classLoader.Caesar in out key");
            return;
        }
    }
}

```

```
try(FileInputStream in = new FileInputStream(args[0]);
    FileOutputStream out = new FileOutputStream(args[1]));
{
    int key = Integer.parseInt(args[2]);
    int ch;
    while ((ch = in.read()) != -1)
    {
        byte c = (byte) (ch + key);
        out.write(c);
    }
}
```

Aby nie wprowadzać w błąd zwykłych procedur ladowania klas, zastosowaliśmy rozszerzenie *.caesar* w przypadku plików zawierających zaszyfrowane klasy.

Aby odszyfrować pliki klas, procedura ładowająca odejmuję wartość klucza od każdego bajtu. W kodzie przykładów dla tej książki zamieściliśmy cztery pliki klas zaszyfrowane za pomocą klucza o wartości 3. Aby uruchomić zaszyfrowany program,, należy użyć procedury ładowającej zdefiniowanej w programie ClassLoaderTest.

Szyfrowanie plików klas posiada wiele potencjalnych zastosowań (oczywiście pod warunkiem zastosowania lepszego szyfru niż szyfr Cezara). Jeśli użytkownik nie dysponuje kluczem, to zaszyfrowane pliki klas są bezużyteczne — nie można ich uruchomić ani odtworzyć ich zawartości.

W ten sposób możemy więc wykorzystać specjalizowaną procedurę ładującą w celu uwierzytelnienia użytkownika klasy lub sprawdzenia, że zapłacono za używany program. Oczywiście szyfrowanie jest tylko jednym z przykładów zastosowań klas ładujących. Inne rodzaje klas procedur ładujących mogą rozwiązywać odmienne kategorie problemów — na przykład przechowywanie plików klas w bazie danych.

API java.lang.Class 1.0

- `ClassLoader getClassLoader()`  
zwraca procedure ładowającą, której użyto do załadowania danej klasy

API java.lang.ClassLoader 1.0

- **ClassLoader getParent() 1.2**  
zwraca nadczną procedurę ładowającą lub wartość null, w przypadku gdy jest nią procedura początkowa.
  - **static ClassLoader getSystemClassLoader()1.2**  
zwraca klasę systemowej procedury ładowającej, czyli procedury użytej do załadowania pierwszej klasy danej aplikacji.

- `protected Class findClass(String name)` **1.2**  
klasa procedury ładującej powinna zastąpić tę metodę w celu odnalezienia kodu maszyny wirtualnej dla danej klasy i przekazania go maszynie wirtualnej za pomocą metody `defineClass`. Nazwa klasy może zawierać znak kropki jako separator w nazwie pakietu, ale nie przyrostek `.class`.
- `Class defineClass(String name, byte[] data, int offset, int length)`  
przekazuje nową klasę maszynie wirtualnej. Kod klasy umieszczony jest w podanym zakresie tablicy.

**API** `java.net.URLClassLoader` **1.2**

- `URLClassLoader(URL[] urls)`
- `URLClassLoader(URL[] urls, ClassLoader parent)`  
tworzy obiekt reprezentujący procedurę ładującą klasy z podanego adresu URL. Jeśli adres URL kończy się znakiem ukośnika `/`, zakłada się, że reprezentuje katalog. W przeciwnym razie reprezentuje plik JAR.

**API** `java.lang.Thread` **1.0**

- `ClassLoader getContextClassLoader()` **1.2**  
zwraca procedurę ładującą, którą twórca danego wątku wyznaczył jako najodpowiedniejszą do ładowania klas w danym wątku.
- `void setContextClassLoader(ClassLoader loader)` **1.2**  
określa procedurę ładującą wykorzystywaną w danym wątku. Jeśli nie zostanie ona określona za pomocą tej metody przed uruchomieniem wątku, to automatycznie będzie wykorzystywana kontekstowa procedura ładująca wątku nadrzędnego.

## 9.2. Weryfikacja kodu maszyny wirtualnej

Zanim kod nowej klasy zostanie przekazany przez procedurę ładującą maszynie wirtualnej, najpierw musi zostać sprawdzony przez *weryfikator*. Sprawdza on, czy załadowany kod nie zawiera instrukcji, których wykonanie może przynieść niepożądane skutki. W ten sposób sprawdzany jest kod wszystkich ładowanych klas z wyjątkiem klas systemowych.

Weryfikator sprawdza między innymi, czy:

- zmienne zostały zainicjowane przed ich użyciem,
- wywołania metod odpowiadają typom referencji obiektów,
- nie zostały naruszone zasady dostępu do składowych i metod o dostępie prywatnym,
- dostęp do zmiennych lokalnych odbywa się za pomocą stosu,
- nie nastąpiło przepelenie stosu.

Jeśli którakolwiek z wymienionych kontroli da wynik negatywny, to przyjmuje się, że kod klasy jest uszkodzony i nie jest on wykonywany.



Znając zasadę Gödla, możemy zastanawiać się, w jaki sposób weryfikator może stwierdzić, że kod klasy nie zawiera niezgodności typów, niezainicjowanych zmiennych oraz nie przepelnia stosu. Przypomnijmy, że zasada Gödla mówi, iż nie jest możliwe utworzenie algorytmu, którego wejście stanowiłoby plik źródłowy programu, a na wyjściu otrzymywaliśmy wartość logiczną stwierdzającą, czy program ten posiada określoną właściwość (na przykład nie przepelnia stosu). Czy więc pomiędzy zasadą Gödla a sposobem działania weryfikatora istnieje sprzeczność? Nie, ponieważ weryfikator *nie* jest algorymem decyzyjnym w sensie zasady Gödla. Jeśli weryfikator zaakceptuje kod klasy, to jest on rzeczywiście bezpieczny. Jednak weryfikator może odrzucić kod wielu innych klas, mimo że w rzeczywistości jego wykonanie także byłoby bezpieczne.

Dokładna weryfikacja kodu jest niezwykle istotna z punktu widzenia bezpieczeństwa. Nawet przypadkowe błędy polegające na przykład na braku zainicjowania zmiennych mogą się okazać groźne w skutkach. Musimy także uzyskać zabezzczenie przed kodem celowo tworzonym z myślą o dokonywaniu zniszczeń, który jest rozpowszechniany przez Internet. Na przykład dopuszczenie modyfikacji wartości na stosie lub prywatnych składowych obiektów systemowych może umożliwić przełamanie systemu zabezpieczeń przeglądarki internetowej.

Zastanawiać może jedynie fakt, po co osobny weryfikator kodu, skoro kompilator języka Java nie wygeneruje kodu klasy, która posiada niezainicjowane zmienne lub próbuje wykorzystać prywatne składowe innej klasy. Oczywiście kod klasy wygenerowany przez kompilator języka Java zawsze zostanie zaakceptowany przez weryfikator. Jednak kod maszyny wirtualnej jest dobrze udokumentowany i nawet programista z umiarkowanym doświadczeniem w tworzeniu programów w kodzie maszynowym może, korzystając z edytora kodu szesnastkowego, utworzyć plik klasy, który zawierać będzie dozwolone, lecz niebezpieczne instrukcje maszyny wirtualnej Java. Zadaniem weryfikatora jest właśnie niedopuszczenie do wykonania takiego kodu, a nie kontrola plików klas wygenerowanych przez kompilator języka Java.

Poniżej pokażemy sposób, w jaki może powstać celowo zmodyfikowany plik klasy. Wykorzystamy w tym celu program *VerifierTest.java* zamieszczony w listingu 9.3. Program ten wywołuje metodę i wyświetla jej wynik. Może być uruchomiony z linii poleceń bądź jako applet. Metoda *fun* wyznacza wynik działania  $1+2$ .

```
public static int fun()
{
    int m;
    int n;
    m = 1;
    n = 2;
    int r = m + n;
    return r;
}
```

**Listing 9.3.** *Verifier/VerifierTest.java*

```
package verifier;

import java.applet.*;
import java.awt.*;
```

```
/*
 * Program demonstrujący działanie weryfikatora kodu maszyny
 * wirtualnej. Jeśli zmodyfikujemy plik klasy, korzystając
 * z edytora kodu szesnastkowego, weryfikator wykryje to
 * podczas próby uruchomienia programu.
 * @version 1.00 1997-09-10
 * @author Cay Horstmann
 */
public class VerifierTest extends Applet
{
    public static void main(String[] args)
    {
        System.out.println("1 + 2 == " + fun());
    }

    /**
     * Metoda wyznaczająca wynik działania 1 + 2
     * @return 3, jeśli kod metody fun nie został zmodyfikowany
     */
    public static int fun()
    {
        int m;
        int n;
        m = 1;
        n = 2;
        // powyższą instrukcję należy zmienić w pliku klasy
        // na "m = 2", korzystając z edytora kodu szesnastkowego
        int r = m + n;
        return r;
    }

    public void paint(Graphics g)
    {
        g.drawString("1 + 2 == " + fun(), 20, 20);
    }
}
```

---

W ramach eksperymentu spróbujmy skompilować wersję programu zawierającą zmodyfikowaną postać metody fun:

```
public static int fun()
{
    int m;
    int n;
    m = 1;
    m = 2;
    int r = m + n;
    return r;
}
```

Zmienna n nie zostanie w tym przypadku zainicjowana i może posiadać dowolną, przypadkową wartość. Sytuacja ta zostanie oczywiście wykryta przez kompilator, który odmówi utworzenia kodu klasy. Aby go uzyskać, musimy spróbować innego sposobu. Najpierw uruchomimy program javap, aby przekonać się, w jaki sposób kompilator przetłumaczył pierwszą wersję metody fun.

```
javap -c VerifierTest
```

W rezultacie uzyskamy poniższy kod maszyny wirtualnej dla metody fun:

```
Method int fun()
0  iconst_1
1  istore_0
2  iconst_2
3  istore_1
4  iload_0
5  iload_1
6  iadd
7  istore_2
8  iload_2
9  ireturn
```

Z pomocą edytora kodu szesnastkowego zmienimy instrukcję 3. z `istore_1` na `istore_0`. W ten sposób lokalna zmienna 0 (czyli `m`) zostanie zainicjowana dwukrotnie, a lokalna zmienna 1 (czyli `n`) nie zostanie w ogóle zainicjowana. W tym celu musimy oczywiście poznać jeszcze kody szesnastkowe instrukcji maszyny wirtualnej Java. Możemy je odnaleźć na przykład w książce *The Java Virtual Machine Specification*, wyd. drugie, autorstwa Tima Lindholma i Franka Yellina (Prentice Hall, 1999).

```
0  iconst_1 04
1  istore_0 3B
2  iconst_2 05
3  istore_1 3C
4  iload_0 1A
5  iload_1 1B
6  iadd   60
7  istore_2 3D
8  iload_2 1C
9  ireturn AC
```

Do modyfikacji kodu możemy wykorzystać specjalizowany edytor szesnastkowy. Na rysunku 9.4 pokazano kod pliku klasy *VerifierTest.class* załadowany w edytorze szesnastkowym środowiska Gnome z zaznaczonym fragmentem kodów instrukcji metody fun.

Należy zmienić w nim wartość 3C na 3B i zapisać zmodyfikowany plik klasy. Następnie spróbujmy uruchomić program *VerifierTest*. Próba taka zakończy się następującą informacją o błędzie:

```
Exception in thread "main" java.lang.VerifyError: (class: VerifierTest, method:
fun signature: ()I) Accessing value from uninitialized register 1
```

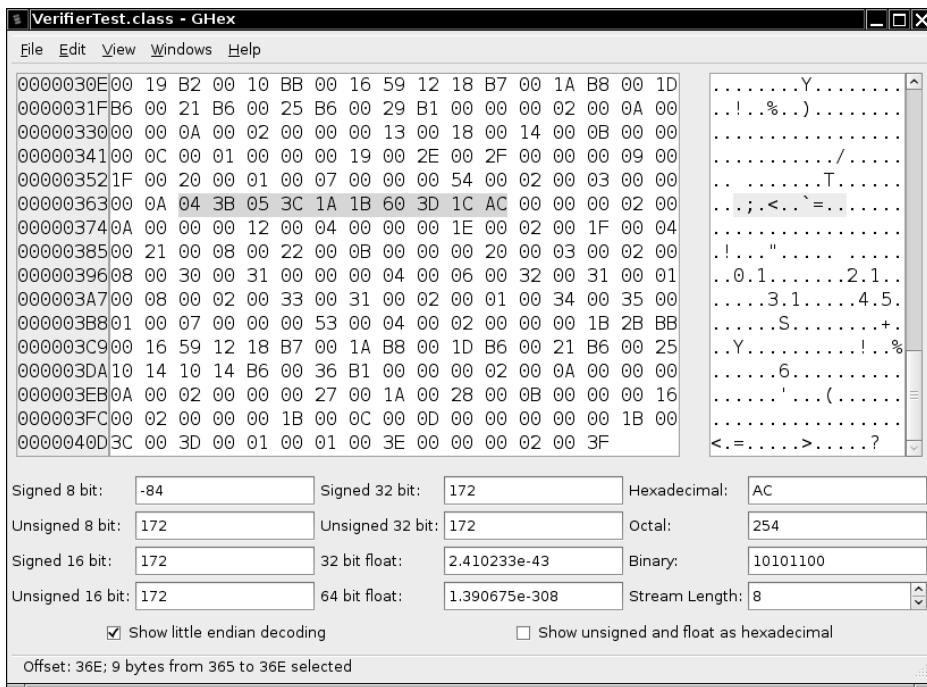
Weryfikator kodu maszyny wirtualnej wykrył więc naszą modyfikację.

A teraz spróbujmy raz jeszcze uruchomić zmodyfikowany program, korzystając z opcji `-noverify` (lub `-Xverify:none`).

```
java -noverify verifier.VerifierTest
```

Metoda fun zwróci przypadkową wartość, ponieważ do wartości 2 doda wartość niezainicjowanej zmiennej `n`. Rezultatem działania programu będzie więc na przykład:

```
1 + 2 = 15102330
```



Rysunek 9.4. Modyfikacja kodu maszyny wirtualnej za pomocą edytora kodu szesnastkowego

Aby pokazać, w jaki sposób przeglądarki internetowe obsługują weryfikację kodu, program VerifierTest napisaliśmy tak, by można go było uruchomić także jako applet. Należy go załadować do przeglądarki za pomocą adresu URL w postaci:

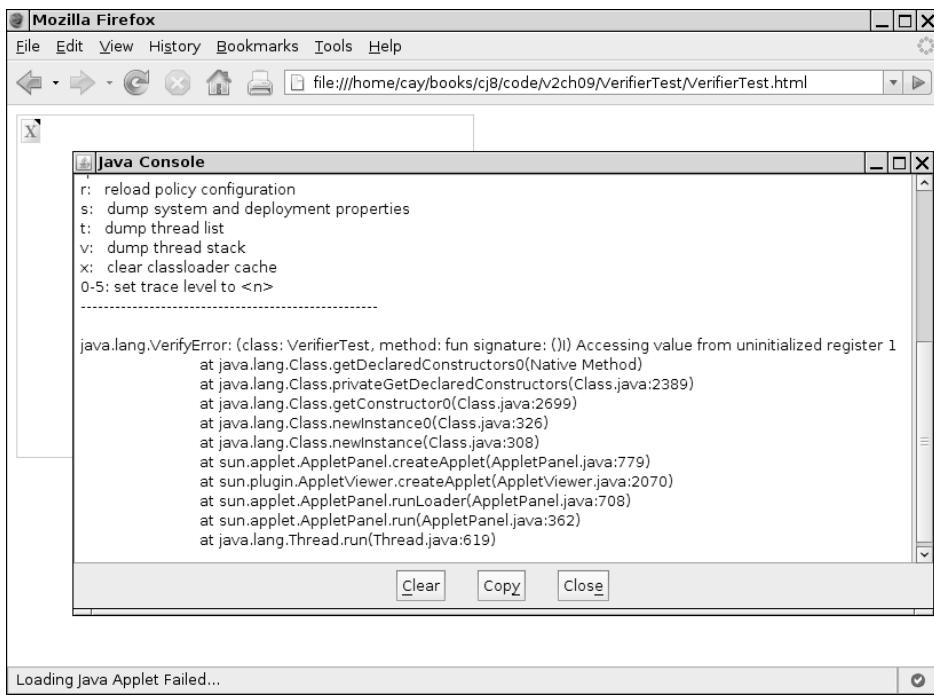
file:///C:/javtz9/r9/verifier/VerifierTest.html

W efekcie uzyskamy informację o błędzie w procesie weryfikacji kodu (patrz rysunek 9.5).

## 9.3. Menedżery bezpieczeństwa i pozwolenia

Po załadowaniu na maszynę wirtualną kodu klasy sprawdzonego przez weryfikator, do akcji wkracza trzeci z mechanizmów bezpieczeństwa platformy Java: *menedżer bezpieczeństwa*. Menedżer bezpieczeństwa jest klasą, która sprawdza, czy określone operacje są dozwolone. Do sprawdzanych operacji należą:

- utworzenie nowej procedury ładowającej,
- zatrzymanie maszyny wirtualnej,
- dostępu do składowej innej klasy z użyciem mechanizmu refleksji,
- dostęp do pliku,
- otwarcie połączenia poprzez gniazdko sieciowe,



**Rysunek 9.5.** Próba załadowania zmodyfikowanego pliku klasy powoduje błąd weryfikacji kodu metody

- uruchomienie zadania wydruku,
- dostęp do schowka systemowego,
- dostęp do kolejki zdarzeń AWT,
- otwarcie okna najwyższego poziomu.

Bibliotek języka Java umożliwia sprawdzanie również wielu innych uprawnień.

Domyślnie podczas uruchamiania aplikacji platformy Java *nie* jest zainstalowany żaden menedżer bezpieczeństwa i wobec tego wszystkie wymienione wyżej operacje są dozwolone. Natomiast program appletviewer służący do uruchamiania apliów natychmiast instaluje własnego menedżera bezpieczeństwa, który jest dość restrykcyjny.

Na przykład zabrania on aplatom kończenia pracy maszyny wirtualnej za pomocą metody exit. Jeśli aplet spróbuje ją wywołać, to spowoduje wyrzucenie odpowiedniego wyjątku związanego z naruszeniem bezpieczeństwa. Dzieje się tak, ponieważ metoda exit klasy Runtime wywołuje metodę checkExit menedżera bezpieczeństwa. Poniżej przedstawiamy kod metody exit.

```
public void exit(int status)
{
    SecurityManager security = System.getSecurityManager();
    if (security != null)
        security.checkExit(status);
    exitInternal(status);
}
```

Menedżer bezpieczeństwa sprawdza, czy żądanie zakończenia pracy maszyny wirtualnej pochodzi od przeglądarki czy od apletu. Jeśli zaakceptuje on żądanie, to wykonanie metody `checkExit` kończy się normalnie i sterowanie powraca do metody `exit`. W przeciwnym razie metoda `checkExit` wyrzuca wyjątek `SecurityException`.

Metoda `exit` może więc kontynuować swoje wykonanie, jeśli nie został wyrzucony wyjątek. Wywołuje ona następnie prywatną metodę macierzystą `exitInternal`, która kończy działania maszyny wirtualnej. Nie istnieje inny sposób zakończenia pracy maszyny wirtualnej. Ponieważ metoda `exitInternal` jest metodą prywatną, to nie może też zostać wywołana bezpośrednio przez inną klasę. Dlatego też każdy kod, który chce zakończyć pracę maszyny wirtualnej, musi wywołać metodę `exit` i tym samym zyskać akceptację metody `checkExit` menedżera bezpieczeństwa.

Zapewnienie integralności polityki bezpieczeństwa wymaga starannej implementacji. Szczególnie dostawcy usług systemowych standardowej biblioteki powinni zawsze odwoływać się do menedżera bezpieczeństwa, zanim wykonają jakąkolwiek operację.

Domyślny menedżer bezpieczeństwa platformy Java pozwala programistom i administratorom precyzyjnie określić prawa. Możliwości te omówimy w kolejnych podrozdziałach. Najpierw jednak zapoznamy się z modelem bezpieczeństwa na platformie Java. Następnie przedstawimy sposób udzielania pozwoleń za pomocą *plików polityki*. Pokażemy też, jak można definiować własne typy pozwoleń.



Na platformie Java można zaimplementować i zainstalować własnego menedżera bezpieczeństwa. Jeśli jednak nie jesteś ekspertem w dziedzinie bezpieczeństwa, to znacznie bezpieczniej będzie, gdy skorzystasz z możliwości, jakie daje konfiguracja standardowego menedżera bezpieczeństwa.

### 9.3.1. Bezpieczeństwo na platformie Java

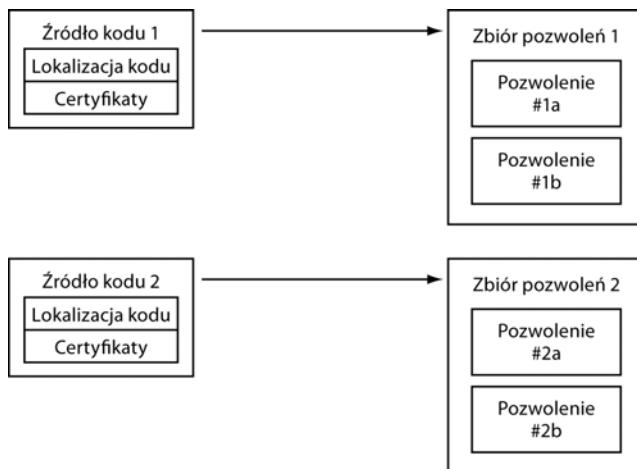
JDK 1.0 posiadał bardzo prosty model bezpieczeństwa: klasy dostępne lokalnie miały prawo wykonywania wszystkich operacji, natomiast uzyskane ze zdalnych źródeł umieszczane były w *piaskownicy* i menedżer bezpieczeństwa zezwalał im jedynie na wyświetlanie informacji na ekranie oraz interakcję z użytkownikiem. JDK 1.1 wprowadził do tego modelu drobną modyfikację: zdalny kod mógł uzyskać takie same prawa jak kod lokalny pod warunkiem, że został podpisany przez wiarygodną instancję. Obie wymienione wersje JDK dostarczały więc modelu pozwoleń typu „wszystko-albo-nic”. Programy uzyskiwały w ten sposób pełne prawa lub zmuszone były do działania w piaskownicy.

Java SE 1.2 wprowadziła znacznie bardziej uniwersalny mechanizm przydzielania praw. *Polityka bezpieczeństwa* umożliwia przyporządkowanie różnym źródłom kodu odmiennych zbiorów pozwoleń (patrz rysunek 9.6).

Źródło kodu posiada dwie właściwości: *lokalizację kodu* (określona za pomocą adresu URL HTTP kodu apletu lub adresu URL pliku JAR) oraz *certyfikaty*. Certyfikat, jeśli jest dostępny, stanowi gwarancję, że kod nie został naruszony. Certyfikaty kodu przedstawimy w dalszej części bieżącego rozdziału.

**Rysunek 9.6.**

Polityka bezpieczeństwa



Przez *pozwolenie* rozumiemy w tym przypadku dowolną właściwość, która kontrolowana jest przez menedżera bezpieczeństwa. Java dostarcza szeregu klas pozwoleń hermetyzujących szczegóły związane z różnymi rodzajami pozwoleń. Na przykład następująca instancja klasy `FilePermission` pozwala na odczyt i zapis dowolnego pliku w katalogu `/tmp`.

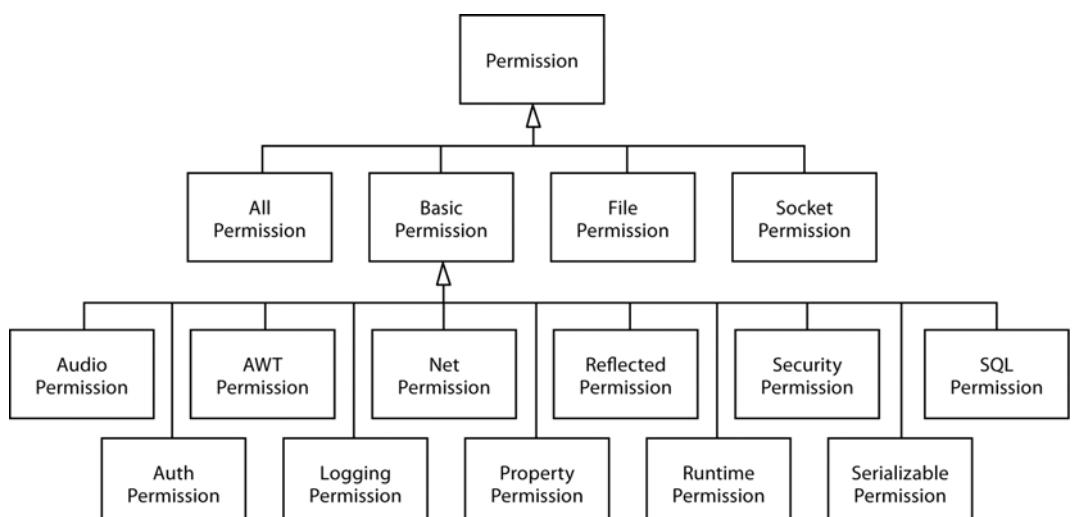
```
FilePermission p = new FilePermission("/tmp/*", "read,write");
```

Domyślna implementacja klasy `Policy` odczytuje pozwolenia z *pliku pozwoleń*. Powyższe pozwolenie możemy wyrazić w takim pliku w następujący sposób.

```
permission java.io.FilePermission "/tmp/*", "read,write";
```

Pliki pozwoleń omówimy w następnym podrozdziale.

Rysunek 9.7 prezentuje hierarchię klas pozwoleń wprowadzoną w Java SE 1.2. W kolejnych wersjach platformy Java wprowadzono dodatkowe klasy pozwoleń.

**Rysunek 9.7.** Fragment hierarchii klas pozwoleń

W poprzednim podrozdziale pokazaliśmy, że klasa SecurityManager posiada metody nadzoru, takie jak checkExit. Metody te dostępne są obecnie jedynie dla wygody programistów bądź ze względu na zachowanie zgodności ze starszymi wersjami platformy. W rzeczywistości korzystają one ze standardowych pozwoleń. Na przykład kod źródłowy metody checkExit wygląda następująco:

```
public void checkExit()
{
    checkPermission(new RuntimePermission("exitVM"));
}
```

Każda klasa platformy Java posiada *domenę ochronną*, która hermetyzuje źródło kodu oraz kolekcję pozwoleń dla danej klasy. Gdy menedżer bezpieczeństwa klasy SecurityManager musi sprawdzić pozwolenie wykonania danej operacji, to sprawdza najpierw klasy wszystkich metod, których wywołania znajdują się na stosie. Następnie pobiera domeny ochronne dla tych klas i sprawdza, czy ich kolekcje pozwoleń zezwalają na daną operację. Jeśli wszystkie domeny akceptują operację, to menedżer bezpieczeństwa zezwala na jej wykonanie. W przeciwnym wypadku wyrzucany jest wyjątek SecurityException.

Dlaczego wszystkie metody znajdujące się na stosie wywołań muszą akceptować daną operację? W celu wyjaśnienia posłużymy się przykładem. Założmy, że metoda init apletu zamierza otworzyć plik.

```
Reader in = new FileReader(name);
```

Konstruktor klasy FileReader wywołuje konstruktor klasy FileInputStream, który z kolei wywołuje metodę checkRead menedżera bezpieczeństwa. Jej implementacja polega na wywołaniu metody checkPermission dla obiektu FilePermission(name, "read"). Tabela 9.1 przedstawia powstały w ten sposób stos wywołań metod.

**Tabela 9.1.** Stos wywołań metod podczas kontroli pozwolenia

Klasa	Metoda	Źródło kodu	Pozwolenia
SecurityManager	checkPermission	null	AllPermission
SecurityManager	checkRead	null	AllPermission
FileInputStream	konstruktor	null	AllPermission
FileReader	konstruktor	null	AllPermission
applet	init	źródło kodu apletu	pozwolenia apletu
...			

Klasy FileInputStream i SecurityManager są *klasami systemowymi*, dla których CodeSource posiada wartość null, a pozwolenia określone są przez instancję klasy AllPermissions zezwalającej na wykonywanie wszystkich operacji. Pozwolenia tych klas nie są oczywiście wystarczające do ustalenia ostatecznego wyniku kontroli. Metoda checkPermission musi wziąć pod uwagę ograniczenia pozwoleń klasy apletu. Sprawdzając cały stos wywołań, mechanizm bezpieczeństwa upewnia się, że żadna klasa nie zleciła innej klasie wykonania operacji w swoim imieniu.



Powyższe, krótkie omówienie sposobu kontroli pozwoleń ilustruje podstawowe koncepty działania mechanizmów bezpieczeństwa. Pomija ono szereg szczegółów technicznych, które w przypadku zagadnień związanych z bezpieczeństwem są bardzo ważne. Dlatego też raz jeszcze zachęcamy do lektury książki autorstwa Li Gong. Szczegółowe omówienie modelu bezpieczeństwa platformy Java zawiera także książka *Securing Java: Getting Down to Business with Mobile Code*, wyd. drugie, napisana przez Gary'ego McGrawa i Eda W. Feltena (Wiley 1999). Na stronie <http://www.securingjava.com> dostępna jest jej wersja elektroniczna.

#### **java.lang.SecurityManager 1.0**

- **void checkPermission(Permission p) 1.2**

Sprawdza, czy bieżąca polityka bezpieczeństwa zawiera określone pozwolenie. Zgłasza wyjątek `SecurityException`, gdy pozwolenie nie zostało przyznane.

#### **java.lang.Class 1.0**

- **ProtectionDomain getProtectionDomain() 1.2**

zwraca domenę ochronną danej klasy lub wartość `null`, w przypadku gdy klasa została załadowana bez takiej domeny.

#### **java.security.ProtectionDomain 1.2**

- **ProtectionDomain(CodeSource source, PermissionCollection collections)**

tworzy domenę ochronną o określonym źródle kodu i pozwoleniach.

- **CodeSource getCodeSource()**

zwraca źródło kodu dla danej domeny ochronnej.

- **boolean implies(Permission p)**

zwraca wartość `true`, gdy domena ochronna zawiera podane pozwolenie.

#### **java.security.CodeSource 1.2**

- **Certificate[] getCertificates()**

zwraca certyfikaty podpisu pliku klasy dla danego źródła kodu.

- **URL getLocation()**

zwraca adres URL lokalizacji danego źródła kodu.

### **9.3.2. Pliki polityki bezpieczeństwa**

*Menedżer polityki* czyta zawartość plików polityki, na którą składają się instrukcje określające przyporządkowanie pozwoleń do źródeł kodu. Typową zawartość pliku polityki przedstawiono poniżej.

```
grant codeBase "http://www.horstmann.com/classes"
{
    permission java.io.FilePermission "/tmp/*", "read,write";
}
```

Plik polityki o takiej zawartości pozwala odczytywać pliki w katalogu */tmp* wszystkim klasom załadowanym z lokalizacji *http://www.horstmann.com/classes* i zapisywać je.

Pliki polityki umieszczamy w określonych miejscach struktury katalogów. Domyślnie określone są dwa takie miejsca:

- plik *java.policy* w katalogu domowym platformy Java,
- plik *.java.policy* w katalogu domowym użytkownika (zwróćmy uwagę na kropkę rozpoczęającą nazwę pliku).



Położenie tych plików możemy określić wcześniej za pomocą pliku konfiguracyjnego *java.security* znajdującego się w katalogu *jre/lib/security*. Domyślnie zawiera on położenie plików polityki określone w poniższy sposób:

```
policy.url.1=file:$(java.home)/lib/security/java.policy
policy.url.1=file:$(user.home)/.java.policy
```

Administrator systemu może zmienić zawartość pliku konfiguracyjnego i wyspecyfikować na przykład adresy URL plików polityki znajdujących się na innym serwerze, które nie będą mogły być edytowane przez użytkowników. Plik konfiguracyjny może zawierać dowolną liczbę adresów URL plików polityki (o kolejnych numerach). Na podstawie ich zawartości tworzona jest kombinacja pozwoleń.

Jeśli chcemy przechowywać pliki polityki poza systemem plików, możemy stworzyć klasę pochodną klasy *Policy*, która będzie zbierać pozwolenia. Następnie musimy zmodyfikować wiersz

```
policy.provider=sun.security.provider.PolicyFile
```

w pliku konfiguracyjnym *java.security*.

Testując tworzone programy, nie będziemy chcieli ciągle zmieniać zawartości standardowych plików polityki. Zamiast tego podamy wprost nazwy plików polityki dla każdej z aplikacji. W tym celu umieścimy pozwolenia w osobnym pliku polityki o nazwie, na przykład, *MyApp.policy*. Mamy następnie dwie możliwości zastosowania tego pliku. Możemy skonfigurować właściwość systemową wewnątrz metody *main* naszej aplikacji:

```
System.setProperty("java.security.policy", "MyApp.policy");
```

Albo uruchomić maszynę wirtualną Java w następujący sposób:

```
java -Djava.security.policy=MyApp.policy MyApp
```

W przypadku apletów skorzystamy z polecenia:

```
appletviewer -J-Djava.security.policy=MyApplet.policy MyApplet.html
```

(Z opcji *-J* programu *appletviewer* korzystamy w celu przekazania dowolnego argumentu maszynie wirtualnej Java).

W powyższych przykładach zawartość pliku polityki kombinowana jest z innymi wykorzystywanymi plikami polityki. Natomiast jeśli określając plik polityki, dodamy drugi znak równości, jak pokazano poniżej

```
java -Djava.security.policy==MyApp.policy MyApp
```

to dla naszej aplikacji zostanie wykorzystana *wyłącznie* zawartość podanego pliku polityki, a zawartość standardowych plików polityki zostanie zignorowana.



Typowym problemem podczas testowania aplikacji jest pozostawienie w bieżącym katalogu pliku `.java.policy`, który zawiera większość pozwoleń, a czasami nawet `AllPermission`. Jeśli więc podczas testowania aplikacji wydaje się, że ignoruje ona ograniczenia zdefiniowane w pliku polityki, to należy sprawdzić, czy nie pozostawiliśmy w bieżącym katalogu pliku `.java.policy`. W przypadku systemu Unix o pomyłkę taką szczególnie łatwo, ponieważ domyślnie pliki o nazwie rozpoczętającej się od znaku kropki nie są uwzględniane przy wyświetlanie zawartości katalogów.

Domyślnie dla aplikacji Java nie jest zainstalowany żaden menedżer bezpieczeństwa. Aby rozpocząć korzystanie z plików polityki, musimy zainstalować menedżera bezpieczeństwa, na przykład dodając w kodzie metody `main` poniższe wywołanie:

```
System.setSecurityManager(new SecurityManager());
```

lub skorzystać z opcji `-Djava.security.manager`, uruchamiając maszynę wirtualną Java:

```
java -Djava.security.manager  
-Djava.security.policy=MyApp.policy MyApp
```

W pozostałej części tego podrozdziału zajmiemy się szczegółowym przedstawieniem opisu pozwoleń w plikach polityki. Omówimy wyczerpująco format plików polityki z wyjątkiem certyfikatów kodu, którym poświęcimy uwagę w dalszej części tego rozdziału.

Plik polityki zawiera sekwencję zapisów grant, z których każdy posiada następującą postać:

```
grant codesource  
{  
    permission_1;  
    permission_2;  
    . . .  
}
```

Źródło kodu codesource składa się z bazy kodu, czyli ścieżki określającej katalog, w którym znajduje się kod (baza kodu może zostać pominięta, jeśli dany zapis dotyczy kodu pochodzącego z dowolnego źródła), oraz nazw wiarygodnych sygnataruszy kodu (które mogą zostać pominięte, jeśli podpisy kodu nie są wymagane przez dany zapis).

Baza kodu podana jest w następującej postaci:

```
codeBase "uri"
```

Jeśli adres URL kończy się znakiem ukośnika `/`, to adres ten odnosi się do katalogu. W przeciwnym przypadku przyjmuje się, że odnosi się do nazwy pliku JAR, na przykład

```
grant codeBase "www.horstmann.com/classes/" { . . . }  
grant codeBase "www.horstmann.com/classes/MyApp.jar" { . . . }
```

Baza kodu określona jest zawsze za pomocą adresu URL, który zawiera znaki ukośnika nawet w przypadku systemu Windows, na przykład

```
grant codeBase "file:C:/myapps/classes/" { . . . }
```



Jak ogólnie wiadomo, adresy URL dla lokalizacji korzystających z protokołu http **rozpoczynają się od podwójnego znaku ukośnika (http://)**. Sytuacja komplikuje się jednak w przypadku adresów URL określających położenie w systemie plików. Okazuje się, że podczas odczytywania plików polityki dopuszczalne są dwie formy takich adresów, `file:///localFile` oraz `file:localFile`. Znak ukośnika poprzedzający literę oznaczającą dysk logiczny w systemie Windows jest opcjonalny. Akceptowane są więc wszystkie poniższe formy adresu URL:

```
file:C:/dir/filename.ext  
file:/C:/dir/filename.ext  
file://C:/dir/filename.ext  
file:///C:/dir/filename.ext
```

Sprawdziliśmy, że również adres postaci `file:///C:/dir/filename.ext` jest akceptowany, czego nie potrafimy już wytlumaczyć.

Pozwolenia opisywane są w pliku polityki w następującej formie:

```
permission className targetName, actionPerformed;
```

Parametr `className` stanowi pełną nazwę klasy pozwoleń (na przykład `java.io.FilePermission`). Parametr celu `targetName` zależy od rodzaju pozwoleń i określa na przykład nazwę pliku lub katalogu w przypadku pozwoleń dotyczących operacji w systemie plików bądź nazwę hosta i numer portu w przypadku pozwoleń dotyczących operacji sieciowych. Również postać parametru akcji `actionList` zależy od rodzaju pozwoleń i zawiera listę akcji, na przykład `read` lub `connect`, rozdzielonych znakiem przecinka. Niektóre z pozwoleń nie wykorzystują w ogóle parametrów `targetName` i `actionList`. Tabela 9.2 prezentuje wartości tych parametrów dla standardowych pozwoleń.

**Tabela 9.2.** Pozwolenia i ich parametry

Pozwolenie	Parametr celu	Parametr akcji
<code>java.io.FilePermission</code>	nazwa pliku lub katalogu (patrz szczegółowy opis w tekście)	<code>read,write,execute,delete</code>
<code>java.net.SocketPermission</code>	nazwa hosta i numer portu (patrz szczegółowy opis w tekście)	<code>accept,connect,listen,resolve</code>
<code>java.util.PropertyPermission</code>	nazwa właściwości (patrz szczegółowy opis w tekście)	<code>read,write</code>
<code>java.lang.RuntimePermission</code>	<code>createClassLoader</code> <code>getClassLoader</code> <code>setContextClassLoader</code> <code>enableContextClassLoaderOverride</code> <code>createSecurityManager</code> <code>setSecurityManager</code> <code>exitVM</code> <code>getenv.variableName</code> <code>shutdownHooks</code>	

**Tabela 9.2.** Pozwolenia i ich parametry — ciąg dalszy

Pozwolenie	Parametr celu	Parametr akcji
	setFactory setIO modifyThread modifyThreadGroup getProtectionDomain readFileDescriptor writeFileDescriptor loadLibrary. <i>libraryName</i> accessClassInPackage. <i>packageName</i> defineClassInPackage. <i>packageName</i> accessDeclaredMembers. <i>className</i> queuePrintJob stopThread getStackTrace setDefaultUncaughtExceptionHandler preferences usePolicy	
java.awt.AWTPermission	showWindowWithoutWarningBanner accessClipboard accessEventQueue createRobot fullScreenExclusive listenToAllAWTEvents readDisplayPixels replaceKeyboardFocusManager watchMousePointer setWindowAlwaysOnTop setAppletStub	
java.net.NetPermission	setDefaultAuthenticator specifyStreamHandler requestPasswordAuthentication setProxySelector getProxySelector setCookieHandler getCookieHandler setResponseCache getResponseCache	
java.lang.reflect.ReflectPermission	suppressAccessChecks	
java.io.SerializablePermission	enableSubclassImplementation enableSubstitution	
java.security.SecurityPermission	createAccessControlContext getDomainCombiner getPolicy setPolicy getProperty. <i>keyName</i> setProperty. <i>keyName</i> insertProvider. <i>providerName</i> removeProvider. <i>providerName</i> setSystemScope	

**Tabela 9.2.** Pozwolenia i ich parametry — ciąg dalszy

Pozwolenie	Parametr celu	Parametr akcji
	setIdentityPublicKey setIdentityInfo addIdentityCertificate removeIdentityCertificate printIdentity clearProviderProperties.providerName putProviderProperty.providerName removeProviderProperty.providerName getSignerPrivateKey setSignerKeyPair	
java.security.AllPermission		
javax.audio.AudioPermission	play record	
javax.security.AuthPermission	doAs doAsPrivileged getSubject getSubjectFromDomainCombiner setReadOnly modifyPrincipals	
	modifyPublicCredentials modifyPrivateCredentials refreshCredential destroyCredential createLoginContext.contextName getLoginConfiguration setLoginConfiguration refreshLoginConfiguration	
java.util.logging.LoggingPermission	control	
java.sql.SQLPermission	setLog	

Jak widać w tabeli 9.2, większość pozwoleń dotyczy pojedynczych operacji. W takim przypadku parametrem celu jest dana operacja i możemy założyć, że parametr akcji posiada domyślną wartość permit. Klasy pozwoleń tego typu stanowią pochodne klasy `BasicPermission` (patrz rysunek 9.7). Natomiast parametr celu dla pozwoleń związanych z systemem plików, operacjami sieciowymi i właściwościami jest bardziej skomplikowany i dla tego omówimy go szczegółowo.

Parametr celu w przypadku pozwoleń dotyczących systemu plików może przyjmować następujące postaci:

- `file` plik,
- `directory/` katalog,
- `directory/*` wszystkie pliki w katalogu,
- `*` wszystkie pliki w bieżącym katalogu,

- directory/-* wszystkie pliki w katalogu lub podkatalog,
- wszystkie pliki w bieżącym katalogu lub podkatalog,
- <>ALL FILES>> wszystkie pliki systemu plików.

Na przykład poniższe pozwolenie umożliwia dostęp do wszystkich plików znajdujących się w katalogu */myapp* i w dowolnym z jego podkatalogów.

```
permission java.io.FilePermission "/myapp/-", "read.write.delete";
```

Odwrotny ukośnik używany w nazwach plików systemu Windows oznaczamy za pomocą pary takich ukośników.

```
permission java.io.FilePermission "c:\\\\myapp\\\\-", "read.write.delete";
```

Parametry celu pozwoleń dotyczących operacji sieciowych składają się z nazwy hosta i zakresu portów. Nazwa hosta może być podana w następujący sposób:

- |                                      |  |
|--------------------------------------|--|
| <i>hostname</i> lub <i>IPaddress</i> | pojedynczy host,   |
| <i>localhost</i> lub pusty łańcuch   | lokalny host,  |
| <i>*.domainSuffix</i>                | dowolny host, którego domena kończy się przyrostkiem <i>domainSuffix</i> , |
| *                                    | dowolny host.  |

Zakresy portów są opcjonalne i mogą mieć postać:

- |               |   |
|---------------|---|
| <i>:n</i>     | pojedynczy port,                              |
| <i>:n-</i>    | wszystkie porty o numerze <i>n</i> i wyższym, |
| <i>:-n</i>    | wszystkie porty o numerze <i>n</i> i niższym, |
| <i>:n1-n2</i> | wszystkie porty podanego zakresu.             |

Oto przykład:

```
permission java.net.SocketPermission "*.*.horstmann.com:8000-8999", "connect";
```

Parametr celu pozwoleń dotyczących właściwości przyjmuje jedną z podanych niżej dwu postaci:

- |                         |  |
|-------------------------|--|
| <i>property</i>         | określona właściwość,                    |
| <i>propertyPrefix.*</i> | wszystkie właściwości o danym prefiksie. |

Przykładami wartości parametru celu mogą być "java.home" lub "java.vm.\*".

Poniższe pozwolenie umożliwia odczyt właściwości, których nazwa rozpoczyna się od prefiksu *java.vm*.

```
permission java.util.PropertyPermission "java.vm.*", "read";
```

W plikach polityki możemy także korzystać z właściwości systemowych. Token postaci \${*property*} zastępowany jest wartością właściwości. Na przykład za \${user.home} zostanie

podstawiona nazwa katalogu użytkownika. Poniżej przedstawiamy przykład typowego wykorzystania właściwości systemowych w pozwoleniach.

```
permission java.io.FilePermission "${user.home}" "read,write";
```

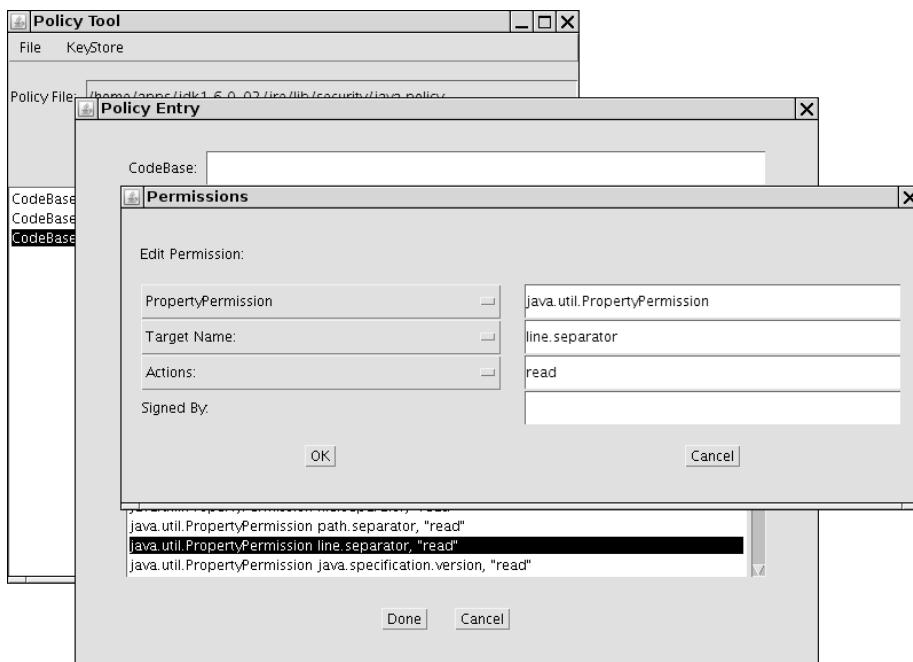
Aby tworzyć pliki polityki niezależne od platformy systemowej, należy korzystać z właściwości `file.separator` zamiast separatorów postaci `/` lub `\`. Dla uproszczenia udostępniono skrócony zapis postaci  `${/}`  równoważny  `${file.separator}` . Na przykład pozwolenie postaci

```
permission java.io.FilePermission "${user.home}${/}" "read,write";
```

umożliwia odczyt i zapis plików w katalogu użytkownika i jego podkatalogach na dowolnej platformie systemowej.



Pakiet JDK zawiera proste narzędzie edycji plików polityki o nazwie `policytool` (patrz rysunek 9.8). Oczywiście narzędzie to nie jest odpowiednie dla użytkowników końcowych, dla których znaczenie poszczególnych pozwoleń stanowi zupełną tajemnicę. Przeznaczone jest raczej dla administratorów, którzy preferują korzystanie z interfejsu graficznego niż ręczną edycję plików. Pożądane byłoby jednak wprowadzenie pewnego zestawu kategorii (odpowiadających na przykład niskiemu, średniemu i wysokiemu poziomowi bezpieczeństwa), które byłyby zrozumiałe również dla użytkowników, którzy nie są ekspertami. Uważamy, że obecnie platforma Java zawiera wszelkie elementy potrzebne do konstruowania precyzyjnych modeli bezpieczeństwa, ale dopracowania wymaga sposób ich udostępnienia użytkownikom końcowym i administratorom.



Rysunek 9.8. Narzędzie `policytool`

### 9.3.3. Tworzenie własnych klas pozwoleń

W podrozdziale tym pokażemy, w jaki sposób tworzyć własne klasy pozwoleń, które mogą być następnie wykorzystywane przez użytkowników w plikach polityki.

Implementując własną klasę pozwoleń, rozszerzamy klasę Permission i dostarczamy następujących metod:

- konstruktora o dwu parametrach klasy String określających parametry celu i akcji,
- String getActions(),
- boolean equals(),
- int hashCode(),
- boolean implies(Permission other).

Najważniejsza z nich jest ostatnia metoda. Pozwolenia są bowiem *uporządkowane* w taki sposób, że bardziej ogólne pozwolenia *implikują* bardziej szczegółowe. Rozpatrzmy poniższy przykład pozwolenia

```
p1 = new FilePermission("/tmp/-", "read,write");
```

Pozwala ono odczytywać i zapisywać dowolne pliki znajdujące się w katalogu /tmp i jego podkatalogach.

A oto przykłady bardziej szczegółowych pozwoleń, które implikuje powyższe pozwolenie:

```
p2 = new FilePermission("/tmp/-", "read");
p3 = new FilePermission("/tmp/aFile", "read,write");
p4 = new FilePermission("/tmp/aDirectory/-", "write");
```

Innymi słowy pozwolenie p1 implikuje pozwolenie p2, jeśli:

- 1.** Cel pozwolenia p1 implikuje cel pozwolenia p2.
- 2.** Akcje pozwolenia p1 implikują akcje pozwolenia p2.

Przedstawimy teraz przykład zastosowania metody `implies`. Gdy konstruktor klasy `FileInputStream` zamierza otworzyć plik do odczytu, sprawdza najpierw, czy posiada odpowiednie pozwolenie. W tym celu przekazuje metodzie `checkPermission` obiekt określający *szczegółowe* pozwolenie:

```
checkPermission(new FilePermission(fileName, "read"));
```

Menedżer bezpieczeństwa sprawdza następnie, czy któryś z obowiązujących pozwoleń implikuje przekazane mu szczegółowe pozwolenie. Jeśli tak, to konstruktor może otworzyć plik.

Obiekt `AllPermissions` implikuje wszystkie inne pozwolenia.

Definiując własną klasę pozwoleń, musimy określić pojęcie implikacji dla jej obiektów. Założymy na przykład, że zdefiniowaliśmy klasę `TVPermission` dla odbiornika telewizyjnego wykorzystującego technologię Java. Pozwolenie w postaci

```
new TVPermission("Tom:2-12:1900-2200", "watch,record")
```

umożliwia Tomowi oglądanie i nagrywanie programów nadawanych na kanałach od 2 do 12 między godziną 19:00 a 22:00. Metodę `implies` musimy zaimplementować w taki sposób, by pozwolenie to implikowało bardziej szczegółowe pozwolenia, na przykład

```
new TVPermission("Tom:4:2000-2100", "watch")
```

### 9.3.4. Implementacja klasy pozwoleń

W kolejnym przykładzie zaimplementujemy nowe pozwolenie, które umożliwiać będzie kontrolę nad umieszczaniem słów *sex*, *drugs* i *C++* w obszarze tekstowym. Wykorzystamy w tym celu własną klasę pozwoleń, a lista zabronionych słów dostarczana będzie za pośrednictwem pliku polityki.

Poniższa klasa pochodna klasy `JTextArea` będzie konsultować się z menedżerem bezpieczeństwa za każdym razem, gdy dodawany będzie nowy tekst.

```
class WordCheckTextArea extends JTextArea
{
    public void append(String text)
    {
        WordCheckPermission p
            = new WordCheckPermission(text, "insert");
        SecurityManager manager = System.getSecurityManager();
        if (manager != null) manager.checkPermission(p);
        super.append(text);
    }
}
```

Jeśli menedżer bezpieczeństwa stwierdzi przyznanie pozwolenia `WordCheckPermission`, to tekst zostanie dodany do obszaru tekstowego. W przeciwnym razie metoda `checkPermission` wyrzuci wyjątek.

Pozwolenie klasy `WordCheckPermission` posiadać będzie dwa rodzaje akcji: `insert` (polegającą na wstawieniu określonego tekstu) oraz `avoid` (pozwalającą wstawić dowolny tekst, który nie zawiera określonych słów). Program korzystający z pozwoleń tej klasy uruchomimy z plikiem polityki o następującej zawartości:

```
grant
{
    permission WordCheckPermission "sex.drugs.C++". "avoid";
};
```

Zezwala on na dodanie dowolnego tekstu, który nie zawiera słów *sex*, *drugs* oraz *C++*.

Projektując klasę `WordCheckPermission`, szczególną uwagę musimy zwrócić na metodę `implies`. Poniżej podajemy zasady, które pozwalają stwierdzić, czy pozwolenie `p1` implikuje pozwolenie `p2`.

- Jeśli `p1` posiada akcję `avoid` i `p2` posiada akcję `insert`, to cel pozwolenia `p2` nie może zawierać słów określonych przez `p1`. Na przykład pozwolenie

```
WordCheckPermission "sex.drugs.C++". "avoid"
```

implikuje pozwolenie

WordCheckPermission "Mary had a little lamb", "insert"

- Jeśli oba pozwolenia p1 i p2 posiadają akcję avoid, to zbiór słów określony przez pozwolenie p2 musi zawierać wszystkie słowa pozwolenia p1. Na przykład pozwolenie

WordCheckPermission "sex.drugs", "avoid"

implikuje pozwolenie

WordCheckPermission "sex.drugs.C++", "avoid"

- Jeśli oba pozwolenia p1 i p2 posiadają akcję insert, to tekst pozwolenia p1 musi zawierać tekst pozwolenia p2. Na przykład pozwolenie

WordCheckPermission "Mary had a little lamb", "insert"

implikuje pozwolenie

WordCheckPermission "a little lamb", "insert"

Implementację klasy WordCheckPermission zawiera listing 9.4.

#### **Listing 9.4.** permission/WordCheckPermission.java

```
package permissions;

import java.security.*;
import java.util.*;

/**
 * Pozwolenie kontrolujące wystąpienie określonych słów.
 */
public class WordCheckPermission extends Permission
{
    private String action;

    /**
     * Tworzy obiekt pozwolenia
     * @param target lista słów oddzielonych przecinkami
     * @param anAction "insert" lub "avoid"
     */
    public WordCheckPermission(String target, String anAction)
    {
        super(target);
        action = anAction;
    }

    public String getActions()
    {
        return action;
    }

    public boolean equals(Object other)
    {
        if (other == null) return false;
        if (!getClass().equals(other.getClass())) return false;
        WordCheckPermission b = (WordCheckPermission) other;
        if (b.action.equals(action)) return true;
        return false;
    }
}
```

```
if (!Objects.equals(action, b.action)) return false;
if ("insert".equals(action)) return Objects.equals(getName(), b.getName());
else if ("avoid".equals(action)) return badWordSet().equals(b.badWordSet());
else return false;
}

public int hashCode()
{
    return Objects.hash(getName(), action);
}

public boolean implies(Permission other)
{
    if (!(other instanceof WordCheckPermission)) return false;
    WordCheckPermission b = (WordCheckPermission) other;
    if (action.equals("insert"))
    {
        return b.action.equals("insert") && getName().indexOf(b.getName()) >= 0;
    }
    else if (action.equals("avoid"))
    {
        if (b.action.equals("avoid")) return
            ↪b.badWordSet().containsAll(badWordSet());
        else if (b.action.equals("insert"))
        {
            for (String badWord : badWordSet())
                if (b.getName().indexOf(badWord) >= 0) return false;
            return true;
        }
        else return false;
    }
    else return false;
}

/**
 * Pobiera słowa określone przez pozwolenie.
 * @return zbiór zabronionych słów
 */
public Set<String> badWordSet()
{
    Set<String> set = new HashSet<>();
    set.addAll(Arrays.asList(getName().split(",")));
    return set;
}
```

---

Zwróćmy uwagę, że cel pozwolenia uzyskujemy, korzystając z metody o nieco mylącej nazwie `getName` udostępnianej przez klasę `Permission`.

Ponieważ pozwolenia opisane są w plikach polityki za pomocą par łańcuchów znaków, to klasa pozwoleń musi potrafić parsować takie łańcuchy. Poniższą metodę wykorzystamy do przekształcenia listy słów pozwolenia posiadającego akcję `avoid` na obiekt klasy `Set`.

```
public Set<String> badWordSet()
{
    Set<String> set = new HashSet<String>();
```

```

        set.addAll(Arrays.asList(getName().split(", ")));
        return set;
    }
}

```

Dzięki temu możemy następnie porównać uzyskane zbiory, korzystając z metod `equals` i `containsAll`. W rozdziale 2 pokazaliśmy, że metoda `equals` przyjmuje, że dwa zbiory są równe, jeśli zawierają te same elementy w dowolnym porządku. Zbiory uzyskane z łańcuchów postaci "sex,drugs,C++" i "C++,drugs,sex" będą więc równe.



Należy upewnić się, że klasa pozwolenia jest klasą publiczną. Procedura ładowająca pliki polityki nie potrafi załadować klas widocznych w obrębie pakietów umieszczonych poza początkową ścieżką dostępu do klas. Jeśli procedura ta nie znajdzie takiej klasy, to nie poinformuje nas o tym.

Program z listingu 9.5 pokazuje sposób działania klasy `WordCheckPermission`. Użytkownik może wpisać dowolny tekst w polu tekstowym i następnie wybrać przycisk `Insert`. Jeśli kontrola pozwolenia przebiegnie pomyślnie, to zostanie on dodany do tekstu w obszarze tekstowym okna programu. W przeciwnym razie pojawi się okno dialogowe informujące o błędzie (patrz rysunek 9.9).

**Rysunek 9.9.**

Program  
*PermissionTest*  
w działaniu



**Listing 9.5.** permissions/PermissionTest.java

```

package permissions;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Demonstruje wykorzystanie
 * własnej klasy pozwoleń WordCheckPermission.
 * @version 1.03 2007-10-06
 * @author Cay Horstmann
 */
public class PermissionTest
{
    public static void main(String[] args)
    {

```

```

System.setProperty("java.security.policy", "permissions/PermissionTest.policy");
System.setSecurityManager(new SecurityManager());
EventQueue.invokeLater(new Runnable()
{
    public void run()
    {
        JFrame frame = new PermissionTestFrame();
        frame.setTitle("PermissionTest");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
});
}

/**
 * Ramka zawierająca pole tekstowe umożliwiające
 * dodawanie tekstu do obszaru tekstowego, pod warunkiem że
 * nie zawiera on zabronionych słów.
 */
class PermissionTestFrame extends JFrame
{
    private JTextField textField;
    private WordCheckTextArea textArea;
    private static final int TEXT_ROWS = 20;
    private static final int TEXT_COLUMNS = 60;

    public PermissionTestFrame()
    {
        textField = new JTextField(20);
        JPanel panel = new JPanel();
        panel.add(textField);
        JButton openButton = new JButton("Insert");
        panel.add(openButton);
        openButton.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                insertWords(textField.getText());
            }
        });
        add(panel, BorderLayout.NORTH);

        textArea = new WordCheckTextArea();
        textArea.setRows(TEXT_ROWS);
        textArea.setColumns(TEXT_COLUMNS);
        add(new JScrollPane(textArea), BorderLayout.CENTER);
        pack();
    }

    /**
     * Próbuje dodać słowa do obszaru tekstowego.
     * Wyświetla okno dialogowe, jeśli próba nie powiedzie się.
     * @param words wstawiane słowa
     */
    public void insertWords(String words)
    {
}

```

```

        try
        {
            textArea.append(words + "\n");
        }
        catch (SecurityException ex)
        {
            JOptionPane.showMessageDialog(this, "I am sorry, but I cannot do that.");
            ex.printStackTrace();
        }
    }

    /**
     * Klasa obszaru tekstowego, której metoda append
     * sprawdza, czy dodawany tekst nie zawiera zabronionych słów.
     */
    class WordCheckTextArea extends JTextArea
    {
        public void append(String text)
        {
            WordCheckPermission p = new WordCheckPermission(text, "insert");
            SecurityManager manager = System.getSecurityManager();
            if (manager != null) manager.checkPermission(p);
            super.append(text);
        }
    }
}

```



Jeśli przyjrzymy się rysunkowi 9.9, to zauważymy, że okno komunikatu opatrzone jest trójkątnym znakiem ostrzegawczym. Jego zadaniem jest ostrzeżenie użytkownika, że okno mogło pojawić się bez pozwolenia. Ostrzeżenie to rozpoczęło swoją karierę w postaci groźnie wyglądającej etykiety "Untrusted Java Applet Window", ale w kolejnych wersjach JDK zostało ostatecznie i obecnie jest praktycznie bezużyteczne. Wyłącza je cel showWindowWithoutWarningBanner pozwolenia `java.awt.AWTPermission`. Pozwolenie to możemy dodać do pliku polityki.

W ten sposób poznaleś możliwości konfigurowania bezpieczeństwa na platformie Java. W większości przypadków wystarczające okazują się standardowe pozwolenia. W szczególnych sytuacjach możesz zdefiniować własne pozwolenia i konfigurować je w taki sam sposób jak pozwolenia standardowe.

### `java.security.Permission` 1.2

- `Permission(String name)`  
tworzy obiekt pozwolenia o podanej nazwie celu.
- `String getName()`  
zwraca nazwę celu pozwolenia.
- `boolean implies(Permission other)`  
sprawdza, czy pozwolenie to implikiuje inne pozwolenie, czyli czy inne pozwolenie opisuje bardziej szczegółowy warunek, który jest konsekwencją warunku opisanego przez dane pozwolenie.

## 9.4. Uwierzytelnianie użytkowników

W Java SE 1.4 wprowadzono usługę Java Authentication and Authorization Service (JAAS). Umożliwia ona uwierzytelnianie użytkowników i przyznawanie im pozwoleń.

JAAS jest interfejsem programowym pozwalającym odseparować aplikacje Java od konkretnej technologii użytej do implementacji usługi uwierzytelniania. Obsługuje między innymi informacje o użytkownikach systemów UNIX, Windows NT, uwierzytelnianie Kerberos oraz uwierzytelniania opartego na certyfikatach.

Po uwierzytelnieniu użytkownika możemy przypisać mu zbiór pozwoleń. Możemy więc wyspecyfikować na przykład, że użytkownik Harry posiada określony zbiór pozwoleń, których nie posiadają pozostała użytkownicy. Składnia w takim przypadku wygląda następująco:

```
grant principal com.sun.security.auth.UnixPrincipal "harry"
{
    permission java.util.PropertyPermission "user.*", "read";
    .
    .
}:
```

Klasa `com.sun.security.auth.UnixPrincipal` umożliwia pobranie nazwy użytkownika systemu Unix, który uruchomił program. Jej metoda `getname` zwraca nazwę użytkownika, którą porównamy z łańcuchem "harry".

Aby menedżer bezpieczeństwa mógł wykonać powyższe polecenie `grant`, należy użyć obiektu `LoginContext`. Poniżej przedstawiamy schemat kodu logowania użytkownika do systemu:

```
try
{
    System.setSecurityManager(new SecurityManager());
    LoginContext context = new LoginContext("Login1"); // zdefiniowany w pliku
                                                       // konfiguracyjnym JAAS
    context.login(); // pobiera uwierzytelniony Subject
    Subject subject = context.getSubject();
    .
    .
    context.logout();
}
catch (LoginException exception) // gdy login nie powiodł się
{
    exception.printStackTrace();
}
```

W wyniku jego wykonania zmienna `subject` reprezentuje użytkownika uwierzytelnionego przez system.

Parametr "Login1" konstruktora `LoginContext` oznacza zapis o tej samej nazwie w pliku konfiguracyjnym JAAS. Poniżej przedstawiamy przykład takiego pliku konfiguracyjnego:

```
Login1
{
    com.sun.security.auth.module.UnixLoginModule required;
    com.whizzbang.auth.module.RetinaScanModule sufficient;
```

```

};

Login2
{
    ...
}

```

Pakiet JDK nie dostarcza oczywiście modułów uwierzytelniania korzystających z biometrii. Pakiet com.sun.security.auth.module zawiera następujące moduły uwierzytelniania:

```

UnixLoginModule
NTLoginModule
Krb5LoginModule
JndiLoginModule
KeyStoreLoginModule

```

Polityka logowania do systemu jest sekwencją modułów logowania, z których każdy oznaczony jest etykietą required, sufficient, requisite lub optional. Znaczenie tych słów kluczowych określa następujący algorytm:

- 1.** Moduły te wykonywane są po kolej, dopóki wykonanie modułu opatrzonego etykietą sufficient zakończy się powodzeniem lub wykonanie modułu oznaczonego jako requisite zakończy się niepowodzeniem bądź osiągnięty zostanie koniec listy modułów.
- 2.** Uwierzytelnianie kończy się pomyślnie, gdy wszystkie moduły oznaczone jako required lub requisite zostaną wykonane pomyślnie lub gdy żaden z tych modułów nie zostanie wykonany bądź co najmniej jeden moduł oznaczony jako sufficient lub optional zostanie wykonany pomyślnie.

Moduł logowania służy uwierzytelnieniu *podmiotu*, który może posiadać wielu *nadzorców*. Nadzorca opisuje niektóre właściwości podmiotu, takie jak nazwa użytkownika, identyfikator grupy czy rola. Nadzorcy zarządzają pozwoleniami i są wymieniani w klauzuli grant. Klasa com.sun.security.auth.UnixPrincipal opisuje nazwę użytkownika systemu Unix, a klasę UnixNumericGroupPrincipal możemy wykorzystać do sprawdzania przynależności użytkowników do grup w tym systemie.

Klauzula grant może sprawdzać nadzorcę w następujący sposób:

```
grant principalClass "principalName"
```

Na przykład:

```
grant com.sun.security.auth.UnixPrincipal "harry"
```

Po zalogowaniu się użytkownika do systemu musimy uruchomić kod, który w osobnym kontekście dostępu sprawdzi jego nadzorców. W tym celu skorzystamy z metody statycznej doAs lub doAsPrivileged, która utworzy nowy obiekt klasy PrivilegedAction. Metoda run tego obiektu sprawdzi nadzorców podmiotu.

Obie wymienione metody wywołują metodę run obiektu implementującego interfejs PrivilegedAction używając pozwoleń nadzorców podmiotu:

```

PrivilegedAction<T> action = new
    PrivilegedAction<T>()
    {

```

```
public T Object run()
{
    // tutaj używa nadzorców do kontroli wykonywanych akcji
    .
    .
};

T result = Subject.doAs(subjectc, action); // lub Subject.doAsPrivileged(subject, action, null);
```

Jeśli sprawdzane akcje mogą wyrzucać sprawdzane wyjątki, to należy zaimplementować interfejs `PrivilegedExceptionAction` zamiast `PrivilegedAction`.

Różnica pomiędzy metodami `doAs` i `doAsPrivileged` jest subtelna. Metoda `doAs` wykonywana jest dla bieżącego kontekstu kontroli dostępu, natomiast metoda `doAsPrivileged` używa nowego kontekstu. Druga z wymienionych metod pozwala oddzielić pozwolenia dla kodu logowania od „logiki biznesowej”. W naszym przykładzie kod logowania posiada pozwolenia:

```
permission javax.security.auth.AuthPermission "createLoginContext.Login1";
permission javax.security.auth.AuthPermission "doAsPrivileged";
```

Uwierzytelniony użytkownik otrzymuje pozwolenie:

```
permission java.util.PropertyPermission "user.*", "read";
```

Gdybyśmy użyli metody `doAs` zamiast `doAsPrivileged`, to również kod logowania otrzymałby to pozwolenie!

Programy z listingu 9.6 i 9.7 demonstrują sposób ograniczenia pozwoleń tylko do określonych użytkowników. Program `AuthTest` po uwierzytelnieniu użytkownika wykonuje prostą akcję polegającą na pobraniu właściwości systemowej.

---

**Listing 9.6.** auth/AuthTest.java

```
package auth;

import java.security.*;
import javax.security.auth.*;
import javax.security.auth.login.*;

/**
 * Program uwierzytelnia użytkownika
 * za pomocą niestandardowej procedury logowania do systemu,
 * a następnie wykonuje SysPropAction,
 * korzystając z pozwoleń przyznanych użytkownikowi.
 * @version 1.01 2007-10-06
 * @author Cay Horstmann
 */
public class AuthTest
{
    public static void main(final String[] args)
    {
        System.setSecurityManager(new SecurityManager());
        try
        {
            LoginContext context = new LoginContext("Login1");
            context.login();
            System.out.println("Authentication successful.");
            Subject subject = context.getSubject();
        }
    }
}
```

```
        System.out.println("subject=" + subject);
        PrivilegedAction<String> action = new SysPropAction("user.home");
        String result = Subject.doAsPrivileged(subject, action, null);
        System.out.println(result);
        context.logout();
    }
    catch (LoginException e)
    {
        e.printStackTrace();
    }
}
```

**Listing 9.7.** auth/SysPropAction.java

```
package auth;

import java.security.*;

/**
 * Akcja wyszukująca właściwość systemową.
 * @version 1.01 2007-10-06
 * @author Cay Horstmann
 */
public class SysPropAction implements PrivilegedAction<String>
{
    private String propertyName;

    /**
     * Tworzy akcję wyszukiwania podanej właściwości.
     * @param propertyName nazwa właściwości (na przykład "user.home")
     */
    public SysPropAction(String propertyName) { this.propertyName = propertyName; }

    public String run()
    {
        return System.getProperty(propertyName);
    }
}
```

Aby uruchomić kod przykładowy, musimy umieścić klasy logowania i klasę akcji w osobnych plikach JAR:

```
javac *.java  
jar cvf login.jar auth/AuthTest*.class  
jar cvf action.jar auth/SysPropAction.class
```

Jeśli przyjrzymy się zawartości pliku polityki zamieszczonego w listingu 9.8, to zobaczymy, że użytkownik systemu Unix o nazwie `harry` posiada pozwolenie na odczyt wszystkich plików. Zmień zatem nazwę `harry` na swoją nazwę użytkownika. Następnie uruchom program za pomocą poniższego polecenia:

```
java -classpath login.jar:action.jar  
-Djava.security.policy=auth/AuthTest.policy  
-Djava.security.auth.login.config=auth/jaas.config  
auth.AuthTest
```

### Listing 9.8. auth/AuthTest.policy

```
grant codebase "file:login.jar"
{
    permission javax.security.auth.AuthPermission "createLoginContext.Login1";
    permission javax.security.auth.AuthPermission "doAsPrivileged";
};

grant principal com.sun.security.auth.UnixPrincipal "harry"
{
    permission java.util.PropertyPermission "user.*", "read";
};
```

Na listingu 9.9 przedstawiona została konfiguracja logowania.

### Listing 9.9. auth/jaas.config

```
Login1
{
    com.sun.security.auth.module.UnixLoginModule required;
};
```

Jeśli przykład ten ma działać w systemie Windows, to należy w pliku polityki *AuthTest.policy* zamienić *UnixPrincipal* na *NTUserPrincipal*, a w pliku konfiguracyjnym *jaas.config* zamiast *UnixLoginModule* wpisać *NTLoginModule*. Uruchamiając program, nazwy plików JAR należy oddzielić znakiem średnika zamiast dwukropka:

```
java -classpath login.jar;action.jar . . .
```

Program AutTest wyświetli wartość właściwości *user.home*. Jeśli jednak zmienisz nazwę użytkownika w pliku *AuthTest.policy*, to zostanie wyrzucony wyjątek, ponieważ nie posiadasz już odpowiedniego pozwolenia.



Powyższe wskazówki należy wykonać ze szczególną starannością. Wprowadzając pozornie nieznaczące zmiany, bardzo łatwo możemy popsuć konfigurację.

#### API javax.security.auth.login.LoginContext 1.4

##### ■ `LoginContext(String name)`

tworzy kontekst logowania. Parametr *name* odpowiada deskryptorowi logowania w pliku konfiguracyjnym JAAS.

##### ■ `void login()`

loguje podmiot lub wyrzuca wyjątek *LoginException*, jeśli logowanie nie powiodło się. Wywołuje metodę *login* menedżerów podanych w pliku konfiguracyjnym JAAS.

##### ■ `void logout()`

wylogowuje podmiot. Wywołuje metodę *logout* menedżerów podanych w pliku konfiguracyjnym JAAS.

- `Subject getSubject()`  
zwraca uwierzytelniony podmiot.

**API `javax.security.auth.Subject 1.4`**

- `Set<Principal> getPrincipals()`  
zwraca nadzorców danego podmiotu.
- `static Object doAs(Subject subject, PrivilegedAction action)`
- `static Object doAs(Subject subject, PrivilegedActionException action)`
- `static Object doAsPrivileged(Subject subject, PrivilegedAction action, AccessControlContext context)`
- `static Object doAsPrivileged(Subject subject, PrivilegedActionException action, AccessControlContext context)`

Wykonują akcję w imieniu podmiotu. Zwracają wartość będącą wynikiem metody `run`. Metoda `doAsPrivileged` wykonuje akcję w podanym kontekście kontroli dostępu. Możemy dostarczyć jej „migawkowego kontekstu” uzyskanego wcześniej przez wywołanie metody statycznej `AccessController.getContext()` lub wartości `null`, gdy chcemy wykonać kod w nowym kontekście.

**API `java.security.PrivilegedAction 1.4`**

- `Object run()`  
metodę tę należy zdefiniować w celu wykonania określonego kodu w imieniu podmiotu.

**API `java.security.PrivilegedExceptionAction 1.4`**

- `Object run()`  
metodę tę należy zdefiniować w celu wykonania określonego kodu w imieniu podmiotu. Metoda może wyrzucać wyjątek.

**API `java.security.Principal 1.1`**

- `String getName()`  
zwraca nazwę nadzorcy.

### 9.4.1. Moduły JAAS

W niniejszym podrozdziale przedstawimy przykład wykorzystania interfejsu JAAS, który ilustruje:

- sposób implementacji własnego modułu logowania,
- sposób implementacji *uwierzytelniania opartego na rolach*.

Implementacja własnego modułu logowania może okazać się przydatna na przykład wtedy, gdy aplikacja przechowuje w bazie danych informację o użytkownikach. Nawet jeśli domyślny moduł logowania jest wystarczający w większości przypadków, to analiza implementacji własnego modułu pozwala lepiej zrozumieć opcje pliku konfiguracyjnego JAAS.

Uwierzytelnianie oparte na rolach staje się istotne w sytuacji, gdy zachodzi konieczność zarządzania dużą liczbą użytkowników. Umieszczanie nazw wszystkich użytkowników w pliku polityki jest wtedy mało praktyczne. Zamiast tego moduł logowania powinien tworzyć odworowanie użytkowników na role takie jak `admin` czy `HR`, a pozwolenia powinny być określone dla tych ról.

Jednym z zadań modułu logowania jest wypełnienie zbioru nadzorców uwierzytelnianego podmiotu. Jeśli moduł logowania obsługuje role, to dodaje wtedy obiekty `Principal` opisujące role. JDK nie dostarcza klasy realizującej to zadanie, wobec czego napisaliśmy własną (listing 9.10). Klasa ta przechowuje parę złożoną z opisu i wartości, na przykład `role=admin`. Metoda `getName` tej klasy zwraca tę parę, dzięki czemu pozwolenia oparte na rolach możemy umieszczać w pliku polityki:

```
grant principal SimplePrincipal "role=admin" { . . . }
```

**Listing 9.10.** *jaas/SimplePrincipal.java*

```
package jaas;

import java.security.*;
import java.util.*;

/**
 * Nadzorca (na przykład "role=HR" lub "username=harry").
 */
public class SimplePrincipal implements Principal
{
    private String descr;
    private String value;

    /**
     * Tworzy obiekt SimplePrincipal przechowujący opis i wartość.
     * @param roleName nazwa roli
     */
    public SimplePrincipal(String descr, String value)
    {
        this.descr = descr;
        this.value = value;
    }

    /**
     * Zwraca nazwę roli nadzorcy
     * @return nazwa roli
     */
    public String getName()
    {
        return descr + "=" + value;
    }

    public boolean equals(Object otherObject)
    {
```

---

```

        if (this == otherObject) return true;
        if (otherObject == null) return false;
        if (getClass() != otherObject.getClass()) return false;
        SimplePrincipal other = (SimplePrincipal) otherObject;
        return Objects.equals(getName(), other.getName());
    }

    public int hashCode()
    {
        return Objects.hashCode(getName());
    }
}

```

---

Nasz moduł logowania wyszukuje użytkowników, hasła i role w pliku tekstowym zawierającym wiersze takie jak przedstawione poniżej:

```

harry|secret|admin
carl|guessme|HR

```

Oczywiście w praktyce moduł logowania przechowywałby raczej takie informacje w bazie danych lub w katalogu.

Kod modułu SimpleLoginModule przedstawiony został na listingu 9.11. Metoda checkLogin sprawdza, czy nazwa użytkownika i hasło zgadzają się z zapisanymi w pliku. Jeśli tak, to do zbioru nadzorców uwierzytelnianego podmiotu dodawane są dwa obiekty SimplePrincipal:

```

Set<Principal> principals = subject.getPrincipals();
principals.add(new SimplePrincipal("username", username));
principals.add(new SimplePrincipal("role", role));

```

**Listing 9.11.** jaas/SimpleLoginModule.java

---

```

package jaas;

import java.io.*;
import java.nio.file.*;
import java.security.*;
import java.util.*;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.security.auth.spi.*;

/**
 * Moduł uwierzytelniający użytkowników na podstawie nazw, hasel i roli
 * wczytanych z pliku tekstowego.
 */
public class SimpleLoginModule implements LoginModule
{
    private Subject subject;
    private CallbackHandler callbackHandler;
    private Map<String, ?> options;

    public void initialize(Subject subject, CallbackHandler callbackHandler,
                          Map<String, ?> sharedState, Map<String, ?> options)
    {

```

```

        this.subject = subject;
        this.callbackHandler = callbackHandler;
        this.options = options;
    }

    public boolean login() throws LoginException
    {
        if (callbackHandler == null) throw new LoginException("no handler");

        NameCallback nameCall = new NameCallback("username: ");
        PasswordCallback passCall = new PasswordCallback("password: ", false);
        try
        {
            callbackHandler.handle(new Callback[] { nameCall, passCall });
        }
        catch (UnsupportedCallbackException e)
        {
            LoginException e2 = new LoginException("Unsupported callback");
            e2.initCause(e);
            throw e2;
        }
        catch (IOException e)
        {
            LoginException e2 = new LoginException("I/O exception in callback");
            e2.initCause(e);
            throw e2;
        }

        try
        {
            return checkLogin(nameCall.getName(), passCall.getPassword());
        }
        catch (IOException ex)
        {
            LoginException ex2 = new LoginException();
            ex2.initCause(ex);
            throw ex2;
        }
    }

    /**
     * Sprawdza, czy uwierzytelnienie jest poprawne. Jeśli tak, to podmiot
     * uzyskuje nadzorców na podstawie nazwy i roli.
     * @param username nazwa użytkownika
     * @param password tablica znaków zawierająca hasło
     * @return true jeśli uwierzytelnienie jest poprawne
     */
    private boolean checkLogin(String username, char[] password) throws
    ↳LoginException, IOException
    {
        try (Scanner in = new Scanner(Paths.get("") + options.get("pwfile")))
        {
            while (in.hasNextLine())
            {
                String[] inputs = in.nextLine().split("\\|");
                if (inputs[0].equals(username) && Arrays.equals(inputs[1].toCharArray(),
                ↳password))
                {

```

```

        String role = inputs[2];
        Set<Principal> principals = subject.getPrincipals();
        principals.add(new SimplePrincipal("username", username));
        principals.add(new SimplePrincipal("role", role));
        return true;
    }
}
return false;
}

public boolean logout()
{
    return true;
}

public boolean abort()
{
    return true;
}

public boolean commit()
{
    return true;
}
}

```

Pozostała część implementacji klasy SimpleLoginModule jest dość oczywista. Metoda `initialize` otrzymuje:

- uwierzytelniany Subject,
- obiekt obsługi pobierający informację o logowaniu,
- mapę `sharedState` używaną do komunikacji pomiędzy modułami logowania,
- mapę `options` zawierającą pary złożone z nazwy i wartości pochodzące z konfiguracji logowania.

Nasz moduł skonfigurujemy następująco:

```
SimpleLoginModule required pwfile= "password.txt";
```

Moduł logowania uzyskuje wartość `pwfile` z mapy `options`.

Moduł logowania nie zajmuje się uzyskaniem nazwy użytkownika i hasła. Zadanie to wykonuje osobny obiekt obsługi. Taki podział zadań umożliwia użycie tego samego modułu logowania niezależnie od tego, czy logowanie wykorzystuje informacje uzyskane za pośrednictwem okna dialogowego, prostej konsoli czy z pliku konfiguracyjnego.

Obiekt obsługi podajemy, tworząc `LoginContext`, na przykład:

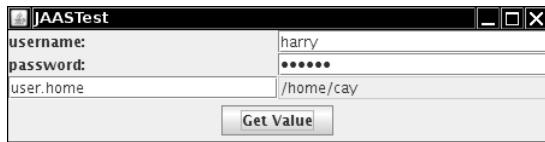
```
LoginContext context = new LoginContext("Login1",
    new com.sun.security.auth.callback.DialogCallbackHandler());
```

Klasa `DialogCallbackHandler` tworzy proste okno dialogowe umożliwiające pobranie nazwy użytkownika i hasła. Klasa `com.sun.security.auth.callback.TextCallbackHandler` pobiera te informacje z konsoli.

Jednak w naszym programie zastosowaliśmy własne okno do pobrania nazwy użytkownika i hasła (patrz rysunek 9.10) i utworzyliśmy prostszy obiekt obsługi, który przechowuje i zwraca wprowadzone informacje (patrz listing 9.12).

**Rysunek 9.10.**

Własny moduł logowania



**Listing 9.12.** jaas/SimpleCallbackHandler.java

```
package jaas;

import javax.security.auth.callback.*;

/**
 * Prosty obiekt obsługi prezentujący nazwę użytkownika i jego hasło.
 */
public class SimpleCallbackHandler implements CallbackHandler
{
    private String username;
    private char[] password;

    /**
     * Konstruktor.
     * @param username nazwa użytkownika
     * @param password tablica znaków zawierająca hasło
     */
    public SimpleCallbackHandler(String username, char[] password)
    {
        this.username = username;
        this.password = password;
    }

    public void handle(Callback[] callbacks)
    {
        for (Callback callback : callbacks)
        {
            if (callback instanceof NameCallback)
            {
                ((NameCallback) callback).setName(username);
            }
            else if (callback instanceof PasswordCallback)
            {
                ((PasswordCallback) callback).setPassword(password);
            }
        }
    }
}
```

Obiekt ten dysponuje jedną metodą handle, która przetwarza tablicę obiektów Callback. Szereg wstępnie zdefiniowanych klas, takich jak na przykład NameCallback i PasswordCallback, implementuje interfejs Callback. Możemy również sami implementować takie klasy, na przykład RetinaScanCallback. Kod metody jest niezbyt estetyczny, ponieważ musi analizować typ obiektu implementującego interfejs Callback:

```
public void handle(Callback[] callbacks)
{
    for (Callback callback : callbacks)
    {
        if (callback instanceof NameCallback) . . .
        else if (callback instanceof PasswordCallback) . . .
        else . . .
    }
}
```

Moduł logowania przygotowuje tablicę obiektów Callback potrzebnych do uwierzytelniania:

```
NameCallback nameCall = new NameCallback("username: ");
PasswordCallback passCall = new PasswordCallback("password: ", false);
callbackHandler.handle(new Callback[] { nameCall, passCall });
```

Następnie pobiera z nich informacje.

Program przedstawiony na listingu 9.13 wyświetla okno umożliwiające wprowadzenie informacji potrzebnej do zalogowania użytkownika oraz nazwy właściwości systemowej. Jeśli użytkownik zostanie zalogowany, to wartość tej właściwości zostaje pobrana przez PrivilegedAction. Jak pokazuje zawartość pliku polityki przedstawiona na listingu 9.14, tylko użytkownikom o roli admin przyznawane jest pozwolenie odczytu właściwości.

#### **Listing 9.13. jaas/JAATest.java**

```
package jaas;

import java.awt.*;
import javax.swing.*;

/**
 * Program uwierzytelnia użytkownika
 * za pomocą własnego modułu logowania,
 * a następnie wykonuje SysPropAction, korzystając
 * z pozwoleń użytkownika.
 * @version 1.01 2012-06-10
 * @author Cay Horstmann
 */
public class JAATest
{
    public static void main(final String[] args)
    {
        System.setSecurityManager(new SecurityManager());
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                JFrame frame = new JAASFrame();
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setTitle("JAATest");
            }
        });
    }
}
```

```
        frame.setVisible(true);
    }
}:
}
}
```

---

**Listing 9.14.** *jaas/JAASTest.policy*

```
grant codebase "file:login.jar"
{
    permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
    permission java.awt.AWTPermission "accessEventQueue";
    permission javax.security.auth.AuthPermission "createLoginContext.Login1";
    permission javax.security.auth.AuthPermission "doAsPrivileged";
    permission javax.security.auth.AuthPermission "modifyPrincipals";
    permission java.io.FilePermission "jaas/password.txt", "read";
};

grant principal jaas.SimplePrincipal "role=admin"
{
    permission java.util.PropertyPermission "*", "read";
};
```

---

Podobnie jak w poprzednich podrozdziałach musimy oddzielić kod logowania od kodu akcji. Utworzymy dwa pliki JAR:

```
javac *.java
jar cvf login.jar JAAS*.class Simple*.class
jar cvf action.jar SysPropAction.class
```

Program uruchomimy w następujący sposób:

```
java -classpath login.jar:action.jar
-Djava.security.policy=JAASTest.policy
-Djava.security.auth.login.config=jaas.config
JAASTest
```

Na listingu 9.15 przedstawiona została konfiguracja logowania.

**Listing 9.15.** *jaas/jaas.config*

```
Login1
{
    jaas.SimpleLoginModule required pwfile="jaas/password.txt" debug=true;
};
```

---



Możliwe jest zastosowanie bardziej złożonego, dwufazowego protokołu, gdzie logowanie zostaje zatwierdzone, jeśli wszystkie moduły występujące w konfiguracji logowania zakończyły się pomyślnie. Więcej informacji na ten temat można znaleźć na stronie <http://docs.oracle.com/javase/7/docs/technotes/guides/security/jaas/JAASLMDevGuide.html>.

**API javax.security.auth.callback.CallbackHandler 1.4**

- `void handle(Callback[] callbacks)`  
obsługuje przekazane obiekty `Callback`, prowadzi dialog z użytkownikiem (jeśli to konieczne) i umieszcza uzyskane informacje w tych obiektach.

**API javax.security.auth.callback.NameCallback 1.4**

- `NameCallback(String prompt)`
- `NameCallback(String prompt, String defaultValue)`  
tworzy obiekt `NameCallback` o tekście zachęty `prompt` i nazwie `defaultValue`.
- `void setName(String name)`
- `String getName()`  
konfiguruje lub zwraca nazwę pobieraną przez ten obiekt.
- `String getPrompt()`  
zwraca tekst zachęty używany podczas pobierania nazwy.
- `String getDefaultName()`  
zwraca nazwę domyślną.

**API javax.security.auth.callback.PasswordCallback 1.4**

- `PasswordCallback(String prompt, boolean echoOn)`  
tworzy `PasswordCallback` o tekście zachęty `prompt`, korzystając z wartości znacznika `echoOn`.
- `void setPassword(char[] password)`
- `char[] getPassword()`  
konfiguruje lub zwraca hasło pobierane przez ten obiekt.
- `String getPrompt()`  
zwraca tekst zachęty używany podczas pobierania hasła.
- `boolean isEchoOn()`  
zwraca znacznik `echo` używany podczas pobierania hasła.

**API javax.security.auth.spi.LoginModule 1.4**

- `void initialize(Subject subject, CallbackHandler handler, Map<String,?> sharedState, Map<String,?> options)`  
inicjuje moduł logowania dla uwierzytelniania podmiotu `subject`. Do pobrania informacji związanej z logowaniem używa obiektu `handler`. Mapa `sharedState`

wykorzystywana jest do komunikacji z innymi modułami logowania, a mapa options zawiera pary złożone z nazwy i wartości podane w konfiguracji logowania dla tego modułu.

- `boolean login()`  
przeprowadza proces uwierzytelniania i dostarcza nadzorców podmiotu. Zwraca wartość `true`, jeśli logowanie się powiodło.
- `boolean commit()`  
metoda stosowana w scenariuszach wymagających zatwierdzenia dwufazowego. Wywoływana, gdy wszystkie moduły logowania zakończyły się pomyślnie. Zwraca wartość `true`, jeśli operacja zakończyła się pomyślnie.
- `boolean abort()`  
wywoływana, gdy niepowodzenie innego modułu logowania oznacza przerwanie procesu logowania. Zwraca wartość `true`, jeśli operacja zakończyła się pomyślnie.
- `boolean logout()`  
wylogowuje podany podmiot. Zwraca wartość `true`, jeśli operacja zakończyła się pomyślnie.

## 9.5. Podpis cyfrowy

Jak już wspomnieliśmy wcześniej, technologia Java wzbudziła na początku zainteresowanie przede wszystkim dzięki możliwości tworzenia i wykonywania appletów. W praktyce jednak szybko okazało się, że, ze względu na restrykcyjny model bezpieczeństwa zastosowany w JDK 1.0, implementacja appletów, które posiadałyby bogatą funkcjonalność, była praktycznie niemożliwa. Szczególnie bolesne ograniczenia te były dla użytkowników dobrze zabezpieczonych intranetów korporacyjnych, gdzie załadowanie appletu związane było z minimalnym ryzykiem. Na szczęście firma Sun szybko zrozumiała, że warunkiem powodzenia technologii appletów będzie możliwość zastosowania *różnych* poziomów bezpieczeństwa w zależności od źródła pochodzenia appletów. Jeśli applet pochodzi z wiarygodnego źródła i można sprawdzić, że nie został on zmodyfikowany, to użytkownik może zdecydować o przyznaniu mu dodatkowych pozwoleń.

Aby przyznać appletowi dodatkowe pozwolenia, musimy określić:

- Źródło pochodzenia appletu.
- Czy applet nie został zmodyfikowany lub uszkodzony.

W ciągu ostatnich 50 lat matematycy i informatycy opracowali zaawansowane algorytmy zapewniające integralność danych i umożliwiające używanie podpisu cyfrowego. Pakiet `java.security` zawiera implementację wielu z nich. Aby korzystać z tych algorytmów, nie jest na szczęście wymagana dokładna znajomość ich matematycznych podstaw. W kolejnych podrozdziałach pokażemy sposób, w jaki można wykorzystać skróty wiadomości do wykrywania modyfikacji w plikach danych oraz podpisy cyfrowe w celu identyfikacji sygnatariuszy.

## 9.5.1. Skróty wiadomości

Skrót wiadomości stanowi cyfrowy „odcisk palca” bloku danych. Korzystając na przykład z algorytmu SHA1 (*secure hash algorithm #1*), otrzymujemy sekwencję 160 bitów (20 bajtów) niezależnie od długości wyjściowego bloku danych. Podobnie jak w przypadku odcisków palca, mamy nadzieję, że nie znajdziemy dwóch bloków danych o tym samym skrócie SHA1. Oczywiście nie jest to prawdą, ponieważ istnieje „jedynie”  $2^{160}$  różnych skrótów SHA1. Liczba ta jest jednak tak duża, że prawdopodobieństwo wystąpienia dwóch bloków o tym samym skrócie jest znikome. Jak znikome? James Walsh podaje w książce *In True Odds: How Risks Affect Your Everyday Life* (Merritt Publishing, 1996), że szansa śmierci na skutek uderzenia pioruna jest jak jeden do 30 000. Wybierz teraz jeszcze 9 osób — kolegów z firmy lub uczelni. Prawdopodobieństwo tego, że Ty i *wszystkie wybrane osoby* zginiecie na skutek uderzenia pioruna jest wyższe niż prawdopodobieństwo, że sfałszowana wiadomość będzie posiadać taki sam skróty SHA1 co oryginalna wiadomość.

Skrót wiadomości posiada dwie istotne właściwości.

- Nawet zmiana pojedynczego bitu danych powoduje zmianę skrótu.
- Fałszerz będący w posiadaniu danej wiadomości nie może utworzyć sfałszowanej wiadomości o takim samym skrócie.

Oczywiście druga z wymienionych właściwości jest znowu kwestią odpowiednio małego prawdopodobieństwa. Założymy, że pewien miliarder zpisał następujący testament.

„Po mojej śmierci majątek powinien zostać podzielony równo pomiędzy moje dzieci. Jednak mój syn George nie otrzyma w spadku nic”.

Założymy też, że skróty tej wiadomości wyglądają, jak poniżej:

```
2D 8B 35 F3 BF 49 CD B1 94 04 E0 66 21 2B 5E 57 70 49 E1 7E
```

Miliarder zdeponował testament u pewnego adwokata, a jego skróty powierzył innemu adwokatowi. Założymy teraz, że George chce przekupić pierwszego z adwokatów, aby zmienił testament, tak by inny z synów miliardera — Bill — został pozbawiony spadku. Oczywiście zmiana ta spowoduje, że skróty będące wyglądają inaczej:

```
2A 33 0B 4B B3 FE CC 1C 9D 5C 01 A7 0 51 0B 49 AC 8F 98 92
```

Czy jednak George nie może zmienić treści testamentu w taki sposób, by skróty pozostały niezmienione? Nawet gdyby posiadał miliard komputerów, z których każdy sprawdzałby milion wiadomości na sekundę, to nawet w czasie, jaki upłynął od powstania Ziemi, nie zdąłby odnaleźć odpowiedniej wiadomości.

W celu wyznaczania skrótów wiadomości opracowano szereg algorytmów. Do najbardziej znanych należy wspomniany już algorytm SHA1 opracowany przez Narodowy Instytut Standardów i Technologii USA oraz MD5, algorytm wymyślony przez Ronaldą Rivesta z MIT. Szczegóły działania algorytmów można poznać, korzystając na przykład z książki *Cryptography and Network Security*, wyd. czwarte, autorstwa Williama Stallingsa (Prentice Hall 2005). W ostatnim czasie w działaniu algorytmu MD5 wykryto drobne regularności i wobec tego niektórzy kryptografowie zalecają raczej stosowanie algorytmu SHA1 (więcej informacji na ten temat na stronie <http://www.rsa.com/rsalabs/node.asp?id=2834>).

W języku Java zaimplementowano oba wymienione algorytmy. Klasa MessageDigest stanowi fabrykę obiektów hermetyzujących algorytmy skrótu. Posiada ona metodę statyczną getInstance zwracającą obiekt klasy będącej rozszerzeniem klasy MessageDigest. Klasa MessageDigest spełnia więc podwójną rolę:

- klasy fabryki,
- klasy bazowej dla wszystkich algorytmów skrótu.

Poniżej przedstawiamy sposób, w jaki należy za jej pomocą pobrać obiekt algorytmu SHA1:

```
MessageDigest alg = MessageDigest.getInstance("SHA-1");
```

(Aby uzyskać obiekt algorytmu MD5, należy użyć łańcucha "MD5" jako parametru metody getInstance).

Po uzyskaniu obiektu reprezentującego algorytm przekazujemy mu stopniowo wszystkie bajty wiadomości, korzystając z metody update. Poniższy przykład ilustruje sposób przekazania bajtów pliku w celu wyznaczenia jego skrótu:

```
InputStream in = . . . ;
int ch;
while ((ch = in.read()) != -1)
    alg.update((byte)ch);
```

Jeśli bajty wiadomości zostały umieszczone w tablicy, to możemy przekazać całą wiadomość, korzystając z pojedynczego wywołania metody update:

```
byte[] bytes = . . . ;
alg.update(bytes);
```

Po przekazaniu całej wiadomości wywołujemy metodę digest. Metoda ta uzupełnia wiadomość zgodnie z wymaganiami algorytmu, wykonuje obliczenia i zwraca skrót w postaci tablicy bajtów.

```
byte[] hash = alg.digest();
```

Program, którego kod źródłowy zawiera listing 9.16, wyznacza skrót wiadomości przy użyciu algorytmu SHA1 lub MD5. Uruchamiamy go w poniższy sposób

```
java hash.Digest hash/input.txt
```

lub

```
java hash.Digest hash/input.txt MD5
```

---

**Listing 9.16. hash/Digest.java**

---

```
package hash;

import java.io.*;
import java.nio.file.*;
import java.security.*;

/**
 * Program wyznaczający skrót zawartości pliku.
 * @version 1.20 2012-06-16
```

```

 * @author Cay Horstmann
 */
public class Digest
{
    /**
     * @param args args[0] to nazwa pliku, a args[1] to (opcjonalny) algorytm (SHA-1 lub MD5)
     */
    public static void main(String[] args) throws IOException,
        GeneralSecurityException
    {
        String aliname = args.length >= 2 ? args[1] : "SHA-1";
        MessageDigest alg = MessageDigest.getInstance(alinkname);
        byte[] input = Files.readAllBytes(Paths.get(args[0]));
        byte[] hash = alg.digest(input);
        String d = "";
        for (int i = 0; i < hash.length; i++)
        {
            int v = hash[i] & 0xFF;
            if (v < 16) d += "0";
            d += Integer.toString(v, 16).toUpperCase() + " ";
        }
        System.out.println(d);
    }
}

```

### java.security.MessageDigest 1.1

- static MessageDigest getInstance(String algorithm)  
zwraca obiekt MessageDigest implementujący podany algorytm. Wyrzuca wyjątek NoSuchAlgorithmException, jeśli podany algorytm nie jest dostępny.
- void update(byte input)
- void update(byte[] input)
- void update(byte[] input, int offset, int len)  
Przekazują algorytmowi bajty wiadomości.
- byte[] digest()  
wyznacza skrót, zwraca go w postaci tablicy bajtów i resetuje algorytm.
- void reset()  
resetuje algorytm skrótu.

## 9.5.2. Podpiswanie wiadomości

W poprzednim podrozdziale pokazaliśmy, w jaki sposób wyznaczyć skrót wiadomości. Jeśli wiadomość zostanie zmodyfikowana, to skrót przestanie do niej pasować. Jeśli odbiorca otrzyma wiadomość i skrót różnymi drogami, to może sprawdzić, czy wiadomość została zmodyfikowana. Jednak jeśli fałszerz przejmie i wiadomość, i jej skrót, to bez trudu może zmodyfikować wiadomość i wyznaczyć nowy, dopowiadający jej skrót. Algorytmy wyznaczania

skrótu są powszechnie dostępne, a ich użycie nie wymaga zastosowania jakiegokolwiek tajnego klucza. Dlatego też odbiorca nie dowie się nigdy, że wiadomość została zmodyfikowana. Problem ten rozwiązuje zastosowanie podpisów cyfrowych.

Aby zrozumieć sposób działania podpisu cyfrowego, musimy najpierw wyjaśnić kilka pojęć z zakresu *kryptografii klucza publicznego*. Wprowadza ona pojęcie klucza *publicznego* i klucza *prywatnego*. Pierwszy z nich udostępniamy publicznie wszystkim zainteresowanym, natomiast drugi zabezpieczamy starannie jedynie do własnego użytku. Klucze te są powiązane pewną matematyczną zależnością, ale dokładna natura tego związku nie jest dla nas istotna. (Jeśli jesteś zainteresowany tym zagadnieniem, możesz zajrzeć na stronę *Handbook of Applied Cryptography*, którą znajdziesz pod adresem <http://www.cacr.math.uwaterloo.ca/hac/>).

Klucze są bardzo długie i skomplikowane. Poniżej przedstawiamy parę kluczy algorytmu DSA (Digital Structure Algorithm).

#### Klucz publiczny:

```
p: fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae16
17ae01f35b91a47e6df63413c5e12ed0899bcd132acd50d99151bdc43ee7375
92e17
q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5
g: 678471b27a9cf44ee91a49c5147db1a9aaaf244f05a434d6486931d2d1427
1b9e35030b71fd73da179069b32e2935630e1c2062354d0da20a6c416e50be79
4ca4
y: c0b6e67b4ac098eb1a32c5f8c4c1f0e7e6fb9d832532e27d0bdbab9ca2d2a
8123ce5a8018b8161a760480fadd040b927281ddb22cb9bc4df596d7de4d1b97
7d50
```

#### Klucz prywatny:

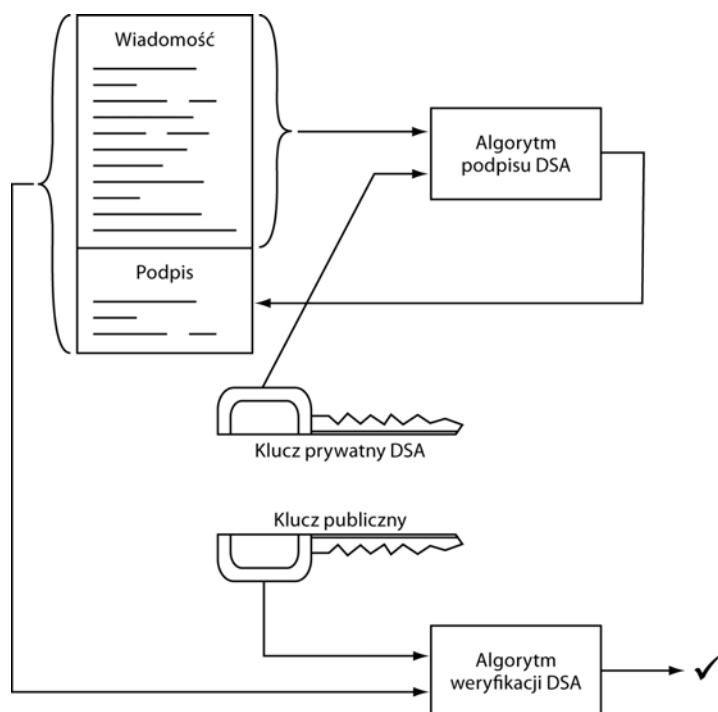
```
p: fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae16
17ae01f35b91a47e6df63413c5e12ed0899bcd132acd50d99151bdc43ee73759
2e17
q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5
g: 678471b27a9cf44ee91a49c5147db1a9aaaf244f05a434d6486931d2d1427
1b9e35030b71fd73da179069b32e2935630e1c2062354d0da20a6c416e50be79
4ca4
x: 146c09f881656cc6c51f27ea6c3a91b85ed1d70a
```

Uważa się, że praktycznie nie jest możliwe wyznaczenie jednego klucza na podstawie drugiego. Jeśli ktoś posiada Twój klucz publiczny, na jego podstawie nie będzie w stanie za swojego życia obliczyć klucza prywatnego, nawet gdyby dysponował wszystkimi komputerami na świecie.

Choć trudno w to uwierzyć, to dotychczas nie znaleziono algorytmu, który pozwalałby ustalić klucz prywatny na podstawie klucza publicznego. Jeśli klucze są odpowiednio długie, to zastosowanie podejścia polegającego na przeglądaniu kolejnych wartości klucza prywatnego nie umożliwi jego uzyskania w sensownym czasie. Oczywiście można spróbować skonstruować bardziej inteligentny algorytm wyszukiwania klucza prywatnego. Algorytm szyfrowania RSA (algorytm szyfrowania utworzony przez Rivesta, Shamira i Adlemana) bazuje na przykład na rozkładzie na czynniki dużych liczb. Przez ostatnie 20 lat wielu matematyków bez rezultatu poszukiwało dobrego algorytmu takiego rozkładu. Dlatego też przyjmuje się w praktyce, że klucze, których moduł ma długość 2000 i więcej bitów, są bezpieczne. Równie bezpieczny jest algorytm szyfrowania DSA.

Rysunek 9.11 pokazuje jak proces ten przebiega w praktyce.

**Rysunek 9.11.**  
Weryfikacja podpisu cyfrowego za pomocą algorytmu DSA



Założmy, że Alice pragnie wysłać wiadomość Bobowi, który z kolei chce mieć pewność, że wiadomość rzeczywiście pochodzi od Alice, a nie od fałszerza. Alice tworzy wiadomość i *podpisuje* ją za pomocą swojego klucza prywatnego. Bob, korzystając z kopii klucza publicznego, *weryfikuje* podpis Alice. Jeśli weryfikacja przebiegnie pomyślnie, to Bob może być pewny, że:

- Oryginalna wiadomość nie została zmodyfikowana.
- Wiadomość została podpisana przez Alice, która posiada klucz prywatny pasujący do klucza publicznego użytego przez Boba do weryfikacji.

Łatwo możemy zauważyć, dlaczego tak istotne jest właściwe zabezpieczenie klucza prywatnego. Jeśli ktoś ukradnie klucz Alice bądź władze zmuszą ją do ujawnienia klucza, to złodziej lub agent rządowy bez trudu może podszyć się pod Alice, wysyłając wiadomości w jej imieniu.

### 9.5.3. Weryfikacja podpisu

Pakiet JDK zawiera program keytool umożliwiający tworzenie i zarządzanie certyfikatami z wiersza poleceń. Docelowo jego funkcjonalność będzie prawdopodobnie wykorzystywana przez bardziej przyjazne oprogramowania wyposażone w graficzny interfejs użytkownika. Na razie jednak musi wystarczyć nam program keytool w jego obecnej postaci. Wykorzystamy go, aby pokazać sposób, w jaki Alice może podpisać dokument i przesłać go Bobowi, który z kolei sprawdzi, że dokument został podpisany rzeczywiście przez Alice, a nie przez oszusta.

Program keytool zarządza *składnicami kluczy*, bazami certyfikatów i parami klucz prywatny/klucz publiczny. Każdy klucz umieszczony w składnicy posiada *alias*. Poniżej przedstawiamy sposób, w jaki Alice może utworzyć składnicę kluczy alice.store i wygenerować parę kluczy o aliasie alice.

```
keytool -genkeypair -keystore alice.certs -alias alice
```

Tworząc nową lub otwierając istniejącą składnicę, zostaniemy poproszeni o podanie hasła. Dla potrzeb naszego przykładu użyjemy hasła secret. Jednak w przypadku każdego innego zastosowania należy wybrać lepsze hasło i chronić plik składnicy, ponieważ zawiera on klucz prywatny.

Tworząc klucz, zostaniemy poproszeni o podanie następujących informacji:

```
Enter keystore password: secret
Reenter new password: secret
What is your first and last name?
[Unknown]: Alice Lee
What is the name of your organizational unit?
[Unknown]: Engineering Department
What is the name of your organization?
[Unknown]: ACME Software
What is the name of your City or Locality?
[Unknown]: San Francisco
What is the name of your State or Province
[Unknown]: CA
What is the two-letter country code for this unit?
[Unknown]: US
Is <CN=Alice Lee, OU=Engineering Department, O=ACME Software,
L=San Francisco, ST=CA, C=US> correct?
[no]: yes
```

Program keytool korzysta w celu identyfikacji posiadaczy kluczy i wydawców certyfikatów z unikalnych nazw standardu X.500 zbudowanych z komponentów *Common Name* (CN), *Organizational Unit* (OU), *Organization* (O), *Location* (L), *State* (ST) i *Country* (C).

Następnie musimy podać hasło klucza lub wybrać klawisz *Enter*, akceptując w ten sposób hasło składnicy jako hasło klucza.

Założymy, że Alice chce przekazać Bobowi swój klucz publiczny. W tym celu musi utworzyć plik certyfikatu:

```
keytool -exportcert -keystore alice.certs -alias alice -file alice.cer
```

Uzyskany certyfikat Alice przesyła Bobowi, który może obejrzeć jego zawartość w poniższy sposób:

```
keytool -printcert -file alice.cer
```

Zawartość certyfikatu może wyglądać następująco:

```
Owner: CN=Alice Lee, OU=Engineering Department, O=ACME Software,
L=Cupertino, ST=CA, C=US
```

```
Issuer: CN=Alice Lee, OU=Engineering Department, O=ACME Software,
L=Cupertino, ST=CA, C=US
```

Serial Number: 470835ce

Valid from: Sat Oct 06 18:26:38 PDT 2007 until: Fri Jan 04 17:26:38 PST 2008

Certificate fingerprints:

MD5: BC:18:15:27:85:69:48:B1:5A:C3:0B:1C:C6:11:B7:81

SHA1: 31:0A:A0:B8:C2:8B:3B:B6:85:7C:EF:C0:57:E5:94:95:61:47:6D:34

Signature algorithm name: SHA1withDSA

Version: 3

Jeśli Bob chce sprawdzić, czy otrzymał właściwy certyfikat, może zadzwonić do Alice i zweryfikować podpis przez telefon.

 Niektórzy wydawcy certyfikatów publikują podpisy certyfikatów na stronach internetowych. Na przykład aby sprawdzić certyfikat Verisign umieszczony w katalogu *jre/lib/security/cacerts*, użyjemy opcji *-list*:

```
keytool -list -keystore jre/lib/security/cacerts
```

Hasło tej składnicy to changeit. Jeden z certyfikatów umieszczonych w tej składnicy przedstawia się następująco:

Owner: OU=VeriSign Trust Network, OU="(c) 1998 VeriSign, Inc. - For authorized use only", OU=Class 1 Public Primary Certification Authority - G2, O="VeriSign, Inc.", C=US

Issuer: OU=VeriSign Trust Network, OU="(c) 1998 VeriSign, Inc. - For authorized use only", OU=Class 1 Public Primary Certification Authority - G2, O="VeriSign, Inc.", C=US

Serial number: 4cc7eaaa983e71d39310f83d3a899192

Valid from: Mon May 18 02:00:00 CEST 1998 until: Wed Aug 02 01:59:59 CEST 2028

Certificate fingerprints:

MD5: DB:23:3D:F9:69:FA:4B:B9:95:80:44:73:5E:7D:41:83

SHA1: 27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47

Jego wiarygodność możemy sprawdzić na stronie <http://www.verisign.com/repository/root.html>.

Po sprawdzeniu wiarygodności certyfikatu Bob może umieścić go w swojej składnicy kluczów.

```
keytool -importcert -keystore bob.certs -alias alice -file alice.cer
```

 Nie należy umieszczać w składnicy certyfikatów, do których wiarygodności mamy wątpliwości. Po umieszczeniu certyfikatu w składnicy każdy program, który korzysta z jej zawartości, zakłada, że każdy z certyfikatów może być wykorzystywany do weryfikacji podpisu.

Alice może teraz wysyłać Bobowi podpisane dokumenty. Program jarsigner umożliwia podpisowywanie i weryfikację podpisów plików JAR. Alice umieści najpierw dokument w pliku JAR.

```
jar cvf document.jar document.txt
```

Następnie skorzysta z programu jarsigner i podpisze plik JAR. W tym celu musi podać składnicę kluczów, określić podpisywany plik JAR oraz alias klucza.

```
jarsigner -keystore alice.certs document.jar alice
```

Natomiast Bob sprawdzi podpis otrzymanego pliku przy użyciu opcji `-verify` programu `jarsigner`.

```
jarsigner -verify -keystore bob.certs document.jar
```

Jak łatwo zauważyc, Bob nie musi podawać aliasu klucza. Program `jarsigner` odnajduje nazwę X.500 właściciela klucza i poszukuje odpowiedniego certyfikatu w składnicy.

Jeśli plik JAR nie został uszkodzony lub zmodyfikowany, to weryfikacja podpisu przebiegnie pomyślnie i program `jarsigner` wyświetli komunikat:

```
jar verified.
```

W przeciwnym razie pojawi się komunikat o błędzie.

## 9.5.4. Problem uwierzytelniania

Załóżmy, że otrzymaliśmy od Alice wiadomość podpisana za pomocą jej klucza prywatnego w sposób, który właśnie pokazaliśmy. Jeśli nie posiadamy jeszcze jego klucza publicznego, to możemy poprosić ją o przesłanie jego kopii lub na przykład pobrać klucz z jej strony internetowej. Dysponując kluczem publicznym, możemy sprawdzić, czy autorem otrzymanej wiadomości jest naprawdę Alice i czy wiadomość nie została zmodyfikowana. Przypuśćmy teraz, że otrzymaliśmy wiadomość od nieznanej nam osoby, która twierdzi, że jest przedstawicielem znanego producenta oprogramowania i proponuje nam uruchomienie programu demonstracyjnego dołączonego do wiadomości. Za pomocą załączonej kopii klucza publicznego możemy też przekonać się, że podpis wiadomości jest wiarygodny, a wiadomość nie została zmodyfikowana.

Musimy jednak zachować ostrożność, ponieważ *nadal nie wiemy, kto jest autorem wiadomości*. Każdy może wygenerować parę kluczy, podpisać wiadomość za pomocą klucza prywatnego i wysłać ją wraz z kluczem publicznym. Problem ustalenia tożsamości nadawcy nazywamy *problemem uwierzytelniania*.

Rozwiążanie tego problemu jest na ogół bardzo proste. Jeśli nieznany nam nadawca znany jest innej osobie, której ufamy, to może przekazać jej klucz publiczny (pod warunkiem, że także ma do niej zaufanie). Następnie wspólny znajomy przekaże nam klucz publiczny, potwierdzając, że jego właściciel rzeczywiście pracuje dla znanego producenta oprogramowania (patrz rysunek 9.12). W ten sposób wspólny znajomy poręcza tożsamość nieznanej nam osoby.

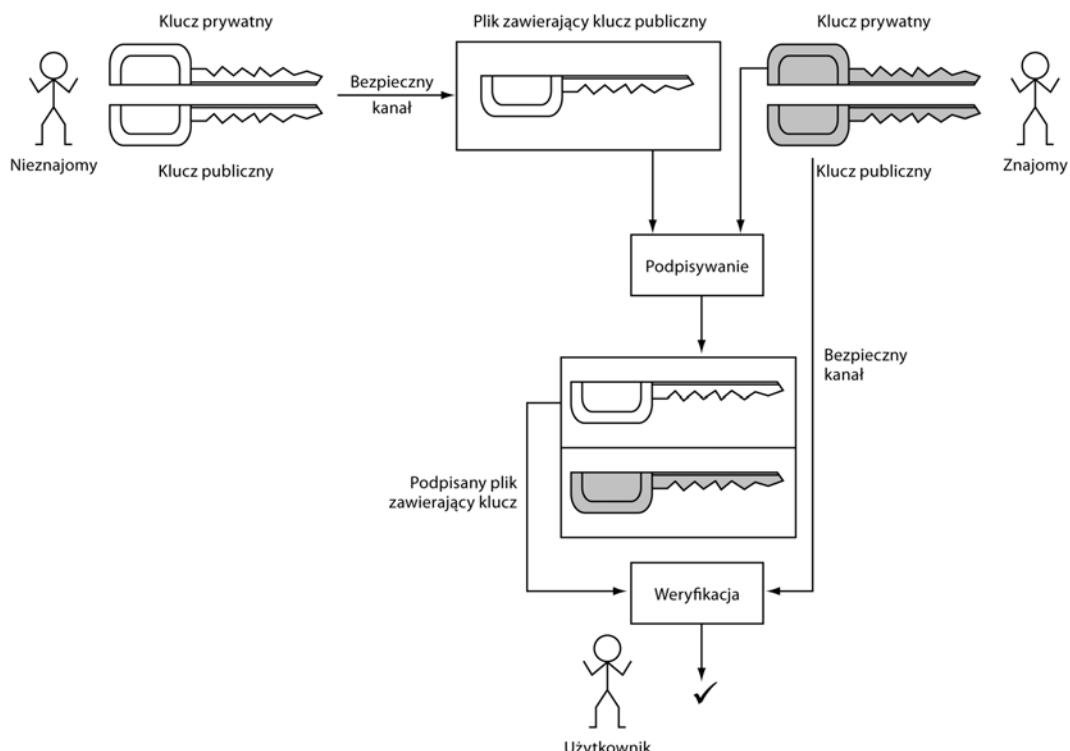
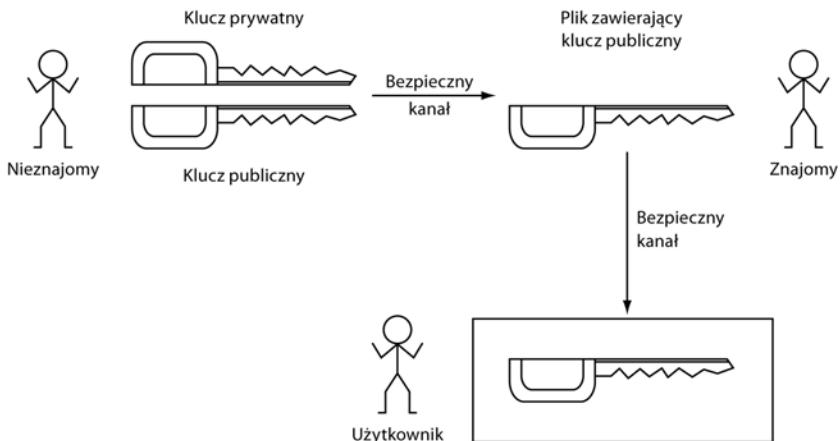
W praktyce wspólny znajomy nie musi się z nami nawet spotykać. Wystarczy, że przekaże nam plik zawierający klucz publiczny nieznajomego opatrzony swoim podpisem (patrz rysunek 9.13).

Otrzymując plik zawierający klucz publiczny, weryfikujemy podpis znajomego. Ponieważ ufamy mu, to zakładamy, że przed złożeniem swojego podpisu sprawdził wiarygodność nieznajomego.

Najczęściej jednak okazuje się, że z nieznanym nam nadawcą wiadomości nie mamy wspólnych znajomych. Niektóre z modeli uwierzytelniania zakładają, że w każdym przypadku możliwe jest utworzenie „łańcucha zaufania” wzajemnych znajomości. W praktyce nie zawsze

**Rysunek 9.12.**

Uwierzytelnianie przez zaufanego pośrednika

**Rysunek 9.13.** Uwierzytelnianie z wykorzystaniem podpisu zaufanego pośrednika

jednak jest to możliwe. Możemy znać i ufać na przykład Alice i wiedzieć, że ufa ona Bobowi, a jednak sami nie musimy ufać Bobowi. Inne modele zakładają istnienie instytucji publicznego zaufania. Przykładem może być tutaj firma Verisign, Inc. (<http://www.verisign.com>).

W praktyce często spotykana jest sytuacja, w której podpis cyfrowy poręczony jest przez kilka instytucji, dla których musimy określić stopień naszego zaufania. Możemy na przykład obdarzyć dużym zaufaniem firmę Verisign, ponieważ spotkaliśmy jej logo na stronach wielu

witryn internetowych lub zapoznaliśmy się z jej profesjonalnymi procedurami stosowanymi w procesie certyfikacji i tworzenia kluczy.

Jednak stosując uwierzytelnianie w praktyce, powinniśmy kierować się realizmem. Prezes firmy Verisign, nie spotyka się przecież osobiście z każdym jej klientem, którego klucz publiczny poręcza firma. Klienci firmy Verisign wypełniają najczęściej formularz zamieszczony na stronie internetowej. W formularzu tym musimy podać imię i nazwisko zamawiającego, organizację, w której pracuje, kraj, którego jest obywatelem oraz adres poczty elektronicznej. Następnie klucz (lub opis sposobu jego uzyskania) wysyłany jest pocztą elektroniczną do zamawiającego. Możemy więc być pewni, że adres poczty elektronicznej jest prawdziwy, ale zamawiający mógł podać dowolne personalia i nazwę organizacji. W przypadku identyfikatorów klasy 1. przydzielanych przez firmę Verisign informacja ta nie jest sprawdzana. Natomiast w przypadku identyfikatora klasy 3. Verisign wymaga stawienia się zamawiającego przed notariuszem oraz sprawdza kondycję finansową reprezentowanej przez niego firmy. Inne instytucje związane z uwierzytelnianiem stosują odmienne procedury. Dlatego też niezwykle istotne jest, aby za każdym razem gdy otrzymamy uwierzytelnioną wiadomość, dokładnie rozumieć, na czym polega uwierzytelnienie w danym przypadku.

### 9.5.5. Podpiswanie certyfikatów

W podrozdziale 9.5.3., „Weryfikacja podpisu”, pokazaliśmy sposób, w jaki Alice używa samodzielnie podписанego certyfikatu, aby udostępnić Bobowi swój klucz publiczny. Bob musiał upewnić się o jego wiarygodności, porównując telefonicznie podpis z właścicielem certyfikatu czyli Alice.

Załóżmy teraz, że Alice chce wysłać Cindy podpisaną wiadomość. Jednak Cindy nie zamierza sprawdzać telefonicznie wiarygodności każdego podpisu. Musi więc istnieć osoba bądź organizacja, której Cindy zaufa w przypadku weryfikacji podpisów. Dla potrzeb naszego przykładu założymy, że będzie to Departament Zasobów Informacyjnych firmy ACME Software.

Departament ten jest między innymi odpowiedzialny za prowadzenie *ośrodka certyfikacji*. Każdy pracownik firmy ACME posiada klucz publiczny tego ośrodka w swojej składnicy kluczy, zainstalowany przez administratora systemu, który dokładnie sprawdził odcisk klucza. Ośrodek certyfikacji podpisuje klucze swoich pracowników. Gdy instalują oni klucze swoich kolegów z firmy, składnica kluczy zakłada ich wiarygodność, ponieważ są podpisane zaufanym kluczem.

Aby zasymulować ten proces, utworzymy dodatkową składnicę kluczy o nazwie acmesoft → .certs. Utworzmy parę kluczy i wyeksportujmy klucz publiczny:

```
keytool -genkeypair -keystore acmesoft.certs -alias acmeroot  
keytool -exportcert -keystore acmesoft.certs -alias acmeroot  
-file acmeroot.cer
```

Klucz publiczny zostaje wyeksportowany do certyfikatu podписанego nim samym. Certyfikat ten umieścimy następnie w składnicy kluczy każdego pracownika.

```
keytool -importcert -keystore cindy.store -alias acmeroot  
-file acmeroot.cer
```

Aby Alice mogła wysyłać podpisane wiadomości do Cindy i innych pracowników firmy ACME Software, musi najpierw dostarczyć swój certyfikat do Departamentu Zasobów Informacyjnych, który go podpisze. W tym celu użyjemy własnej klasy CertificateSigner (jej kod znajdziesz w przykładach dołączonych do tej książki), ponieważ keytool nie udostępnia takiej funkcjonalności. Upoważniony pracownik firmy ACME Software zweryfikuje tożsamość Alice i podpisze jej certyfikat w następujący sposób:

```
java CertificateSigner -keystore acmesoft.certs -alias acmeroot  
-infile alice.cer -outfile alice_signedby_acmeroot.cer
```

Program CertificateSigner musi posiadać dostęp do składnicy kluczy firmy ACME Software, a uruchamiający go pracownik powinien znać hasło dostępu do składnicy. Właściwy przebieg tej operacji jest niezwykle istotny z punktu widzenia bezpieczeństwa.

Alice przekaże plik *alice\_signedby\_acmeroot.cer* Cindy i innym pracownikom firmy ACME Software lub firma ta umieści go w ogólnie dostępnym katalogu. Przypomnijmy, że plik ten zawiera klucz publiczny należący do Alice oraz poręczenie tego faktu przez firmę ACME Software.

Teraz Cindy może już umieścić podpisany certyfikat w swojej składnicy kluczy:

```
keytool -importcert -keystore cindy.certs -alias alice -file  
→alice_signedby_acmeroot.cer
```

Składnica automatycznie zweryfikuje, że klucz został podpisany za pomocą klucza firmy, który już znajduje się w składnicy. Dlatego też Cindy *nie* będzie nawet proszona o weryfikację podpisu certyfikatu.

Jeśli Cindy umieści w swojej składnicy certyfikat firmy oraz certyfikaty pracowników przesyłających jej dokumenty, to nie musi już więcej zajmować się zawartością składnicy.

## 9.5.6. Żądania certyfikatu

W poprzednim rozdziale zasymulowaliśmy działanie ośrodka certyfikacji za pomocą składnicy kluczy i klasy CertificateSigner. W praktyce większość takich ośrodków używa nieco bardziej zaawansowanych narzędzi i nieco innych formatów certyfikatów. W tym podrozdziale zajmiemy się omówieniem dodatkowych kroków, których podjęcie jest wymagane dla właściwej współpracy z takimi narzędziami.

Jako przykład wykorzystamy pakiet OpenSSL. Jest on wstępnie zainstalowany w wielu systemach Linux oraz w systemie Mac OS X. Dostępna jest również wersja Cygwin. Pakiet ten można również załadować z witryny <http://www.openssl.org>.

Aby utworzyć ośrodek certyfikacji, uruchamiamy skrypt CA. Jego dokładne położenie zależy od konkretnego systemu operacyjnego. W przypadku Ubuntu wywołujemy go następująco:

```
/usr/lib/ssl/misc/CA.pl -newca
```

Skrypt ten tworzy w katalogu bieżącym podkatalog o nazwie *demoCA*. Katalog ten zawiera parę kluczy korzenia organizacji oraz przechowuje certyfikaty i listy odwołań certyfikatów.

Następnie należy zainportować klucz publiczny do składnic kluczy platformy Java dla wszystkich użytkowników. Jednak ośrodek certyfikacji przechowuje klucze w formacie Privacy Enhanced Mail (PEM), a nie w formacie DER akceptowanym przez składnice kluczy. Dlatego też należy skopiować plik *demoCA/cacert.pem* jako plik *acmeroot.pem* i otworzyć go w edytorze tekstu. Następnie usunąć całą zawartość poprzedzającą wiersz

```
-----BEGIN CERTIFICATE-----
```

oraz znajdująca się za wierszem

```
-----END CERTIFICATE-----
```

Teraz możemy już zainportować *acmeroot.pem* do każdej składnicy kluczy w zwykły sposób:

```
keytool -importcert -keystore cindy.certs -alias alice -file acmeroot.pem
```

Szkoda tylko, że program keytool nie potrafi sam wykonać tej prostej operacji edycji pliku *acmeroot.pem*.

Aby podpisać klucz publiczny Alice, musimy wygenerować *żądanie certyfikatu*, które zawiera certyfikat w formacie PEM:

```
keytool -certreq -keystore alice.store -alias alice -file alice.pem
```

Aby podpisać certyfikat wywołujemy

```
openssl ca -in alice.pem -out alice_signedby_acmeroot.pem
```

Tak jak poprzednio usuwamy całą zawartość pliku *alice\_signedby\_acmeroot.pem* znajdująca się poza znacznikami BEGIN CERTIFICATE/END CERTIFICATE. Następnie importujemy go do składnicy kluczy:

```
keytool -importcert -keystore cindy.certs -alias alice -file  
→alice_signedby_acmeroot.pem
```

Takie same kroki podejmujemy, jeśli chcemy, aby certyfikat został podpisany przez publiczny ośrodek certyfikacji, na przykład firmę VeriSign.

## 9.6. Podpisywanie kodu

Jednym z najważniejszych zastosowań uwierzytelniania jest podpisywanie kodu wykonywalnych programów. Jeśli załadujemy program z sieci, to słusznie możemy obawiać się szkód, które może wyrządzić jego uruchomienie. Program mógł zostać na przykład zainfekowany wirusem. Jeśli będziemy natomiast pewni pochodzenia programu i tego, że jego kod nie został zmodyfikowany, to możemy uruchamiać go bez obaw. Gdy program został napisany w języku Java, to możemy także wykorzystać informację o źródle jego pochodzenia, przyznając mu prawa wykonywania różnych operacji. Możemy ograniczyć wykonanie apletu wyłącznie do piaskownicy lub zdefiniować inny zbiór praw i ograniczeń. Na przykład gdy załadujemy program edytora tekstu, możemy zezwolić mu na dostęp do drukarki oraz podkatalogu zawierającego dokumenty, ale nie na tworzenie połączeń sieciowych, by nie mógł bez naszej wiedzy wysyłać przez sieć tworzonych dokumentów.

Z poprzednich podrozdziałów wiemy już, jak zaimplementować ten złożony schemat.

## 9.6.1. Podpisywanie plików JAR

W niniejszym podrozdziale zajmiemy się podpisywaniem appletów działających z modułem plugin technologii Java. Omówimy dwa rodzaje scenariuszy:

- Podpisywanie appletów intranetowych.
- Podpisywanie appletów dostępnych przez Internet.

W pierwszym scenariuszu administrator systemu instaluje pliki polityki oraz certyfikaty na lokalnych maszynach. Za każdym razem, gdy moduł plugin ładuje podpisany applet, sprawdzane są podpisy kodu apletu w składnicy kluczy oraz pozwolenia w pliku polityki. Instalacja certyfikatów i plików polityki wykonywana jest tylko jeden raz dla każdej lokalnej maszyny. Użytkownicy mogą uruchamiać korporacyjne applety poza piaskownicą. Za każdym razem, gdy w sieci korporacyjnej udostępniany jest nowy applet, musi on zostać podpisany i umieszczony na serwerze. Natomiast konfiguracja poszczególnych maszyn nie musi być aktualizowana. Taki scenariusz jest dość atrakcyjnym rozwiązaniem i powodem, dla którego aplikacje korporacyjne coraz częściej tworzone są w technologii appletów Java.

Drugi scenariusz przewiduje uzyskanie certyfikatów przez producenta apletu od firm zajmujących się wydawaniem certyfikatów (na przykład Verisign). Gdy użytkownik odwiedza stronę zawierającą podpisany applet, przeglądarka otwiera okno dialogowe zawierające informacje o producencie apletu i umożliwia użytkownikowi uruchomienie apletu z pełnymi prawami bądź jedynie w piaskownicy. Scenariusz ten omówimy szczegółowo w podrozdziale 9.6.2, „Certyfikaty twórców oprogramowania”.

Teraz zajmiemy się sposobami tworzenia plików polityki określających prawa appletów pochodzących z różnych źródeł. Tworzenie takich plików nie jest zadaniem użytkowników appletów, lecz administratorów intranetów przygotowujących dystrybucję appletów.

Przypuśćmy, że firma ACME Software zamierza udostępnić swoim pracownikom programy, który wymagają dostępu do lokalnego systemu plików. Ponieważ program taki nie może pracować w piaskownicy, konieczna jest instalacja plików polityki na lokalnych komputerach pracowników.

Jak pokazaliśmy już wcześniej w tym rozdziale, firma ACME może identyfikować pochodzenie programów na podstawie ich bazy kodu. W praktyce jednak oznacza to konieczność modyfikacji plików polityki za każdym razem, gdy kod apletu przenoszony jest na inny serwer. Dlatego też lepszym rozwiązaniem będzie *podpisanie* pliku JAR zawierającego kod apletu.

Najpierw firma ACME wygeneruje certyfikat korzenia:

```
keytool -genkeypair -keystore acmesoft.certs -alias acmeroot
```

Oczywiście składnica kluczy zawierająca prywatny klucz korzenia musi znajdować się w bezpiecznym miejscu. Dlatego utworzymy kolejną składnicę, *client.certs*, która będzie przechowywać publiczne certyfikaty, i umieścimy w niej publiczny certyfikat acmeroot.

```
keytool -exportcert -keystore acmesoft.certs -alias acmeroot -file acmeroot.cer
keytool -importcert -keystore client.certs -alias acmeroot -file acmeroot.cer
```

Zanim podpiszemy plik JAR, musimy umieścić w nim klasy apletu w zwykły sposób.

```
javac FileReadApplet.java  
jar cvf FileReadApplet.jar *.class
```

Następnie zaufana osoba w firmie ACME użyje programu jarsigner, któremu poda nazwę pliku JAR i alias prywatnego klucza:

```
jarsigner -keystore acmesoft.certs FileReadApplet.jar acmeroot
```

Podpisany aplet jest już gotowy do umieszczenia na serwerze.

Przejdźmy teraz do konfiguracji komputerów pracowników. Na każdym z nich musi zostać umieszczony plik polityki.

W pliku tym musimy określić położenie składnicy kluczy. W tym celu na początku pliku polityki dodamy następujący wiersz:

```
keystore "keystoreURL", "keystoreType";
```

Pierwszy z parametrów może być pełnym łańcuchem URL lub określonym względem położenia pliku polityki. Jeśli składnica została utworzona za pomocą programu keytool, to drugi z parametrów będzie łańcuchem "JKS".

```
keystore "clients.certs", "JKS";
```

Do klauzuli grant w pliku polityki dodamy frazę signedBy "alias", na przykład:

```
grant signedBy "acmeroot"  
{  
    . . .  
};
```

Od tego momentu każdy podpisany kod, którego podpis można zweryfikować za pomocą klucza publicznego związanego z podanym aliasem, otrzymuje powysze pozwolenie.

Możemy to sprawdzić, korzystając z kodu apletu zamieszczonego na listingu 9.17. Próbuje on przeczytać zawartość lokalnego pliku. Domyślana polityka bezpieczeństwa zezwala apletom jedynie na odczyt plików położonych w ich bazie kodu i jej podkatalogach. Uruchamiając aplet za pomocą programu appletviewer, możemy przekonać się, że aplet rzeczywiście może czytać pliki położone w jego bazie kodu, ale nie w innych katalogach.

---

**Listing 9.17.** signed/FileReadApplet.java

```
package signed;  
  
import java.awt.*;  
import java.awt.event.*;  
import java.io.*;  
import java.nio.file.*;  
import javax.swing.*;  
  
/**  
 * Aplet ten może zostać uruchomiony poza "piaskownicą" (ang. sandbox)  
 * i czytać lokalne pliki, jeśli udzielimy mu odpowiedniego pozwolenia.  
 * @version 1.12 2012-06-10  
 * @author Cay Horstmann  
 */
```

```
public class FileReadApplet extends JApplet
{
    private JTextField fileNameField;
    private JTextArea fileText;

    public void init()
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                fileNameField = new JTextField(20);
                JPanel panel = new JPanel();
                panel.add(new JLabel("File name:"));
                panel.add(fileNameField);
                JButton openButton = new JButton("Open");
                panel.add(openButton);
                ActionListener listener = new ActionListener()
                {
                    public void actionPerformed(ActionEvent event)
                    {
                        loadFile(fileNameField.getText());
                    }
                };
                fileNameField.addActionListener(listener);
                openButton.addActionListener(listener);

                add(panel, "North");
                fileText = new JTextArea();
                add(new JScrollPane(fileText), "Center");
            }
        });
    }

    /**
     * Ładuje zawartość pliku do obszaru tekstowego.
     * @param filename nazwa pliku
     */
    public void loadFile(String filename)
    {
        fileText.setText("");
        try
        {
            fileText.append(new String(Files.readAllBytes(Paths.get(filename))));
        }
        catch (IOException ex)
        {
            fileText.append(ex + "\n");
        }
        catch (SecurityException ex)
        {
            fileText.append("I am sorry, but I cannot do that.\n");
            fileText.append(ex + "\n");
            ex.printStackTrace();
        }
    }
}
```

Utwórzmy teraz plik polityki *applets.policy* o następującej zawartości:

```
keystore "client.certs", "JKS";
grant signedBy "acmeroot"
{
    permission java.lang.RuntimePermission "usePolicy";
    permission java.io.FilePermission "/etc/*", "read";
}
```

Pozwolenie *usePolicy* unieważnia domyślne ograniczenia dla podpisanych appletów. Apletom podpisany przez acmeroot zezwalamy na odczyt plików w katalogu */etc*. (Użytkownicy systemu Windows: proszę zastąpić ten katalog innym, na przykład *C:\Windows*).

Uruchamiając program *appletviewer*, poinformujemy go, by wykorzystał utworzony plik polityki

```
appletviewer -J-Djava.security.policy=applets.policy
FileReadApplet.html
```

Teraz aplet może czytać już pliki z katalogu */etc*, co potwierdza poprawne działanie mechanizmu podpisywania.



Jeśli uruchomienie apletu sprawia problemy, można użyć opcji *-J-Djava.security.debug=polocy*, która spowoduje wyświetlenie szczegółowych informacji o konfiguracji polityki bezpieczeństwa.

Podpisany aplet możemy przetestować uruchamiając go w przeglądarce (patrz rysunek 9.14). W tym celu należy umieścić plik pozwoleń i składnicę kluczy w odpowiednim katalogu. W systemie UNIX lub Linux będzie nim podkatalog *java/deployment* domowego katalogu użytkownika. W systemach Windows Vista i Windows 7 będzie nim katalog *C:\Users\your>LoginName\AppData\Sun\Java\Deployment*. W dalszej części podrozdziału będziemy się odwoływać do tych katalogów za pomocą wspólnej nazwy *deploydir*.

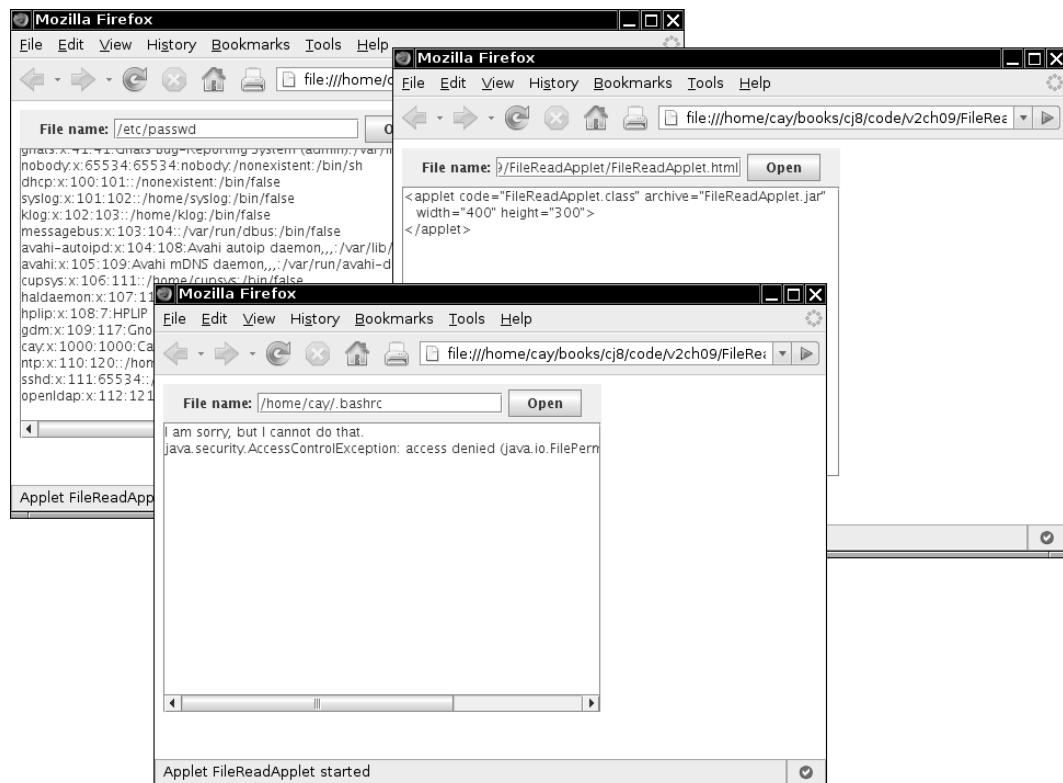
Pliki *applet.policy* i *client.certs* kopujemy do katalogu *deploydir/security*. W tym katalogu zmieniamy nazwę pliku *applets.policy* na *java.policy*. (Upewnij się, czy w ten sposób nie usuniesz już wcześniej istniejącego pliku *java.policy*. Jeśli taki już istnieje, dodaj do niego zawartość pliku *applet.policy*).



Więcej informacji na temat konfigurowania bezpieczeństwa klientów Java znajdziesz w podrozdziałach „Deployment Configuration File and Properties” i „Java Control Panel” przewodnika <http://docs.oracle.com/javase/7/docs/technotes/guides/deployment/deployment-guide/overview.html>.

Uruchom ponownie swoją przeglądarkę i załaduj plik *FileReadApplet.html*. Nie powinieneś przy tym być proszony o zaakceptowanie żadnego certyfikatu. Sprawdź, czy możesz załadować dowolny plik z katalogu */etc* oraz z katalogu, z którego załadowałeś aplet, ale nie z innych katalogów.

Po zakończeniu testu pamiętaj, aby wyczyścić zawartość katalogu *deploydir/security*. Usuń pliki *java.policy* i *client.certs*. Uruchom ponownie przeglądarkę. Jeśli teraz załadowajesz aplet, nie powinieneś mieć możliwości odczytu plików z lokalnego systemu plików. Zostaniesz również poproszony o podanie certyfikatu. Certyfikaty bezpieczeństwa omawiamy w następnym podrozdziale.



Rysunek 9.14. Program FileReadApplet w działaniu

## 9.6.2. Certyfikaty twórców oprogramowania

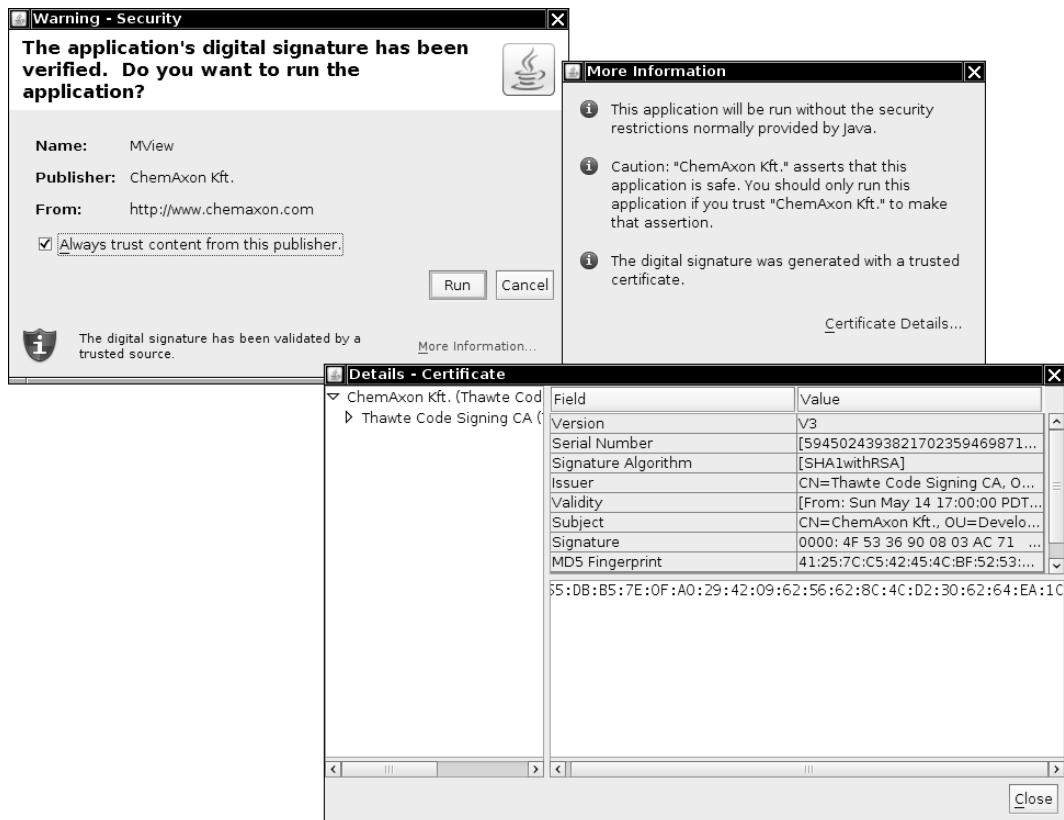
Omówiliśmy dotąd scenariusz, w którym apłyty działają w sieci intranetowej, a administrator określa politykę bezpieczeństwa ich wykonania. Scenariusz ten możliwy jest jedynie w przypadku apletów, dla których potrafimy określić źródło ich pochodzenia.

Załóżmy teraz, że, przeglądając strony internetowe, natrafiliśmy na applet nieznanego producenta (patrz rysunek 9.15). Aplet ten podpisany jest za pomocą certyfikatu *twórcy oprogramowania* wydanego na przykład przez firmę Verisign. Okno dialogowe poinformuje nas o tożsamości twórcy apletu i wydawcy jego certyfikatu. Następnie możemy wybrać wykonanie apletu:

- z pełnymi prawami,
- bądź w piaskownicy (nazwa przycisku *Cancel* jest w tym przypadku nieco myląca, ponieważ nie powoduje przerwania działania apletu, lecz ogranicza jego wykonanie do piaskownicy).

Jakie fakty mogą wpływać na naszą decyzję? Wiemy, że:

- Wydawcą certyfikatu jest firma Thawte.
- Aplet został podpisany za pomocą klucza, dla którego został wydany certyfikat i kod apletu nie został zmodyfikowany.



Rysunek 9.15. Uruchamianie podpisaneego appletu

- Certyfikat został podpisany przez firmę Thawte, co zostało zweryfikowane za pomocą klucza publicznego firmy znajdującego się w pliku *cacerts*.

Czy na tej podstawie możemy wnioskować, że wykonanie kodu będzie bezpieczne? Czy możemy zaufać twórcy oprogramowania, jeśli cała nasza wiedza o nim sprowadza się do jego nazwy (ChemAxon Kft.) i faktu, że zakupił on certyfikat w firmie Thawte? Oczywiście firma Thawte zadała sobie trud ustalenia, że za nazwą ChemAxon Kft. nie ukrywa się oszust. Jednak żaden z wydawców certyfikatów nie przeprowadza wszechstronnego audytu statusu i kompetencji twórców oprogramowania.

W sytuacji gdy producent appletu nie jest znany użytkownikowi, podjęcie decyzji o przyznaniu appletowi wszelkich praw lokalnej aplikacji jest niezwykle trudne. W przypadku gdy producent appletu jest powszechnie znaną firmą, łatwiej możemy obdarzyć go zaufaniem, biorąc pod uwagę jego dotychczasowe dokonania.

Sytuacja, w której program domaga się przyznania mu wszelkich praw, z gruntu nie jest właściwa i naraża na niebezpieczeństwo przede wszystkim niedoświadczonych użytkowników.

Czy gdyby każdy program objął szczegółowo pozwolenia, które należy mu przyznać, to zmieniłoby to znacząco sytuację użytkownika? Jak pokazaliśmy wcześniej, wymaga to tech-

nicznej wiedzy o poszczególnych pozwoleniach. Od większości użytkowników nie można więc wymagać, by decydowali o tym, czy należy danemu apletowi przydzielić na przykład prawo dostępu do kolejki AWT, czy też nie.

Dlatego pozostajemy sceptycznie nastawieni do możliwości korzystania z aplików podpisywanych za pomocą certyfikatów wystawianych twórcom oprogramowania. Lepiej będzie, jeśli aplikacje dostępne przez internet będą ograniczały swoje działanie do piaskownicy, a jej implementacja będzie ciągle udoskonalana. Interfejs programowy Web Start omówiony w rozdziale 10. książki *Java 2. Podstawy* uważamy za krok we właściwym kierunku.

## 9.7. Szyfrowanie

Omówiliśmy dotąd jedną z najważniejszych technik kryptograficznych zaimplementowanych na platformie Java, jaką jest uwierzytelnianie za pomocą podpisu cyfrowego. Kolejnym istotnym aspektem problematyki bezpieczeństwa jest *szyfrowanie*. Po uwierzytelnieniu informacja staje się natychmiast dostępna. Podpis cyfrowy umożliwia sprawdzenie, że nie została ona zmodyfikowana. Po zastosowaniu szyfrowania informacja nie jest dostępna tak długo, aż nie odszyfrujemy jej, korzystając z odpowiedniego klucza.

Uwierzytelnianie jest zupełnie wystarczające w przypadku kodu programów. Natomiast szyfrowanie staje się niezbędne, gdy apliki bądź aplikacje przesyłają poufne informacje, takie jak na przykład numery kart kredytowych.

Do niedawna wykorzystanie dobrych algorytmów szyfrowania utrudnione było przez ograniczenia patentowe eksportowe nakładane przez władze USA. Sytuacja ta uległa zmianie i, począwszy od wersji Java SE 1.4, algorytmy takie stanowią część standardowej biblioteki.

### 9.7.1. Szyfrowanie symetryczne

Platforma Java udostępnia klasę `Cipher`, która jest klasą bazową dla wszystkich algorytmów szyfrowania. Obiekt reprezentujący algorytm szyfrowania pobieramy, korzystając z metody `getInstance` tej klasy:

```
Cipher cipher = Cipher.getInstance(algorithmName);
```

lub

```
Cipher cipher = Cipher.getInstance(algorithmName, providerName);
```

Pakiet JDK dostarcza algorytmów szyfrowania, dla których wartość parametru `providerName` określającego nazwę dostawcy algorytmu jest łańcuchem "SunJCE". Wartość ta przyjmowana jest domyślnie, jeśli nie podamy innej wartości tego parametru.

Pierwszy z parametrów metody podaje nazwę algorytmu, na przykład "AES" lub "DES/CBC/  
PKCS5Padding".

DES (*Data Encryption Standard*) jest już nieco przestarzałym algorytmem szyfrowania wykorzystującym klucz o długości 56 bitów. Obecnie jest już nawet możliwe złamanie go za

pomocą ataków opartych na metodzie pełnego przeglądu (patrz [http://w2.eff.org/Privacy/Crypto/Crypto\\_misc/DESCracker/](http://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/)). Dużo lepszym rozwiązaniem jest więc wykorzystanie jego następcy — algorytmu AES (*Advanced Encryption Standard*). Więcej informacji na temat tego algorytmu znajdziesz na stronie <http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.

Po uzyskaniu obiektu reprezentującego algorytm szyfrowania musimy określić tryb jego pracy oraz klucz.

```
int mode = . . . ;
Key key = . . . ;
cipher.init(mode, key);
```

Tryb pracy określamy za pomocą jednej z poniższych wartości:

```
Cipher.ENCRYPT_MODE
Cipher.DECRYPT_MODE
Cipher.WRAP_MODE
Cipher.UNWRAP_MODE
```

Ostatnie dwie wartości wykorzystywane są do szyfrowania jednego klucza innym. Przykład ich zastosowania pokażemy w następnym podrozdziale.

Następnie możemy już cyklicznie wywoływać metodę update, szyfrując w ten sposób kolejne bloki danych:

```
int blockSize = cipher.blockSize();
byte[] inBytes = new byte[blockSize];
. . . // umieszcza dane w tablicy inBytes
int outputSize = cipher.outputSize(inLength);
byte[] outBytes = new byte[outputSize];
int outLength = cipher.update(inBytes, 0, outputSize, outBytes);
. . . // zapisuje zawartość tablicy outBytes
```

Po zakończeniu szyfrowania należy wywołać jeden raz metodę doFinal. Jeśli pozostał jeszcze do zaszyfrowania blok danych, którego rozmiar jest mniejszy od blockSize, to wywołamy ją w następujący sposób:

```
outBytes = cipher.doFinal(inBytes, 0, inLength);
```

Jeśli natomiast zaszyfrowane zostały już wszystkie dane, to wywołanie będzie miało poniższą postać.

```
outBytes = cipher.doFinal();
```

Wywołanie metody doFinal jest niezbędne ze względu na konieczność *dopełnienia* ostatniego bloku. Algorytm DES wykorzystuje blok o rozmiarze 8 bajtów. Założmy, że ostatni blok szyfrowanej informacji zawiera mniej niż 8 bajtów. Moglibyśmy sami dopełnić go wyzerowanymi bajtami i zaszyfrować. Jednak po odszyfrowaniu uzyskana informacja różniłaby się od oryginalnej właśnie o te kilka zerowych bajtów. Dlatego też w algorytmach szyfrowania zastosowano standardowe sposoby dopełniania końcowych bloków. Jednym z nich jest PKCS#5 (*Public Key Cryptography Standard*) opracowany przez firmę RSA Security Inc. (<ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf>). Dopełnia on ostatni blok bajtami, których wartość odpowiada ich liczbie. Innymi słowy, jeśli L jest ostatnim (niekompletnym) blokiem, to jest on dopełniany w następujący sposób:

```

L 01           if length(L) = 7
L 02 02       if length(L) = 6
L 03 03 03    if length(L) = 5
.
L 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07 07

```

Jeśli długość szyfrowanej informacji dzieli się bez reszty przez 8, to blok postaci

```
08 08 08 08 08 08 08 08
```

dołączany jest na końcu szyfrowanej informacji. Podczas odszyfrowywania przyjmuje się, że ostatni bajt uzyskanej informacji oznacza liczbę bajtów, które należy usunąć.

## 9.7.2. Generowanie klucza

Wyjaśnimy teraz, w jaki sposób uzyskać klucz szyfrowania. Każdy szyfr używa innego formatu kluczy, a dodatkowo musimy upewnić się, że klucz zostanie wygenerowany losowo. W tym celu wykonujemy następujące kroki:

- 1.** Tworzymy obiekt KeyGenerator.
- 2.** Inicjujemy generator za pomocą źródła wartości losowych. Jeśli długość szyfrowanego bloku może być różna, to określamy również wymaganą wartość.
- 3.** Wywołujemy metodę generateKey.

Poniżej przedstawiamy sposób tworzenia klucza dla algorytmu AES.

```

KeyGenerator keygen = KeyGenerator.getInstance("AES");
SecureRandom random = new SecureRandom(); //patrz niżej
keygen.init(random);
Key key = keygen.generateKey();

```

Klucz możemy również utworzyć na podstawie określonego zbioru danych (na przykład hasła lub zależności czasowych naciśnięć klawiszy). Tworzymy wtedy klucz klasy SecretKeySpec (implementującej interfejs SecretKey) w następujący sposób:

```

byte[] keyData = . . . ; //16 bajtów dla DES
SecretKey key = new SecretKeySpec(keyData, "AES");

```

Generując klucze, powinniśmy dostarczyć *prawdziwie losowej* wartości. Generator liczb losowych w postaci klasy Random jest inicjowany na podstawie aktualnego czasu i daty i nie jest wystarczający do celów kryptograficznych. Jeśli założymy, że wykorzystuje on zegar działający z dokładnością do 0,1 sekundy, to w ciągu dnia występuje co najwyżej 864 000 wartości posiewu generatora. Jeśli falsożer wie, w jakim dniu wygenerowano klucz (co może wydedukować na przykład na podstawie daty ważności klucza), to wygenerowanie wszystkich par kluczy możliwych w danym dniu nie będzie przedstawiać dla niego większej trudności.

Klasa SecureRandom umożliwia generowanie liczb losowych, które są dużo „bezpieczniejsze” od tworzonych przez klasę Random. Nadal musimy oczywiście dostarczyć wartość posiewu do zainicjowania generatora. Najlepiej byłoby uzyskać ją jako przypadkową wartość pobraną z urządzenia w rodzaju generatora białego szumu. Innym sposobem będzie poproszenie

użytkownika o wpisanie przypadkowego ciągu z klawiatury, przy czym każdy z wpisanych znaków dostarczy jedynie dwu bitów wartości posiewu. Po zebraniu ich w tablicy przekażemy ją metodzie `setSeed`.

```
SecureRandom secrand = new SecureRandom();
byte[] b = new byte[20];
//tablicę wypełniamy prawdziwie losowymi bitami
secrand.setSeed(b);
```

Jeśli nie dostarczymy wartości posiewu, to generator liczb losowych wyznaczy własne 20 bajtów posiewu, uruchamiając wątki, zawieszając ich wykonanie i mierząc dokładny czas, w którym ich wykonywanie zostało wznowione.



Opisany algorytm posiewu *nie* został uznany za bezpieczny. W przeszłości algorytmy, które bazowały na innych zależnościach czasowych systemu (na przykład czasie dostępu do dysku), okazywały się nie w pełni losowe.

Przykładowy program, którego kod źródłowy umieściliśmy na końcu bieżącego podrozdziału, stanowi ilustrację wykorzystania algorytmu szyfrowania AES (patrz listing 9.18). Metoda `crypt` przedstawiona na listingu 9.19 zostanie ponownie wykorzystana w innych przykładach. Aby użyć programu, należy najpierw wygenerować klucz szyfrowania w następujący sposób:

```
java aes.AESTest -genkey secret.key
```

#### **Listing 9.18. aes/AESTest.java**

```
package aes;

import java.io.*;
import java.security.*;
import javax.crypto.*;

/**
 * Program testujący szyfr AES. Uruchamianie:
 * java AESTest -genkey keyfile
 * java AESTest -encrypt plaintext encrypted keyfile
 * java AESTest -decrypt encrypted decrypted keyfile
 * @author Cay Horstmann
 * @version 1.01 2012-06-10
 */
public class AESTest
{
    public static void main(String[] args)
        throws IOException, GeneralSecurityException, ClassNotFoundException
    {
        if (args[0].equals("-genkey"))
        {
            KeyGenerator keygen = KeyGenerator.getInstance("AES");
            SecureRandom random = new SecureRandom();
            keygen.init(random);
            SecretKey key = keygen.generateKey();
            try (ObjectOutputStream out = new ObjectOutputStream(new
                FileOutputStream(args[1])))
            {
                out.writeObject(key);
            }
        }
    }
}
```

```
        }
    else
    {
        int mode;
        if (args[0].equals("-encrypt")) mode = Cipher.ENCRYPT_MODE;
        else mode = Cipher.DECRYPT_MODE;

        try (ObjectInputStream keyIn = new ObjectInputStream(new
            →FileInputStream(args[3]));
            InputStream in = new FileInputStream(args[1]);
            OutputStream out = new FileOutputStream(args[2]))
        {
            Key key = (Key) keyIn.readObject();
            Cipher cipher = Cipher.getInstance("AES");
            cipher.init(mode, key);
            Util.crypt(in, out, cipher);
        }
    }
}
```

**Listing 9.19.** aes/Util.java

```
package aes;

import java.io.*;
import java.security.*;
import javax.crypto.*;

public class Util
{
    /**
     * Szyfruje bajty strumienia wejściowego i wysyła je do strumienia wyjściowego.
     * @param in strumień wejściowy
     * @param out the strumień wyjściowy
     * @param cipher algorytm szyfrowania
     */
    public static void crypt(InputStream in, OutputStream out, Cipher cipher) throws
    →IOException,
        GeneralSecurityException
    {
        int blockSize = cipher.getBlockSize();
        int outputSize = cipher.getOutputSize(blockSize);
        byte[] inBytes = new byte[blockSize];
        byte[] outBytes = new byte[outputSize];

        int inLength = 0;
        ;
        boolean more = true;
        while (more)
        {
            inLength = in.read(inBytes);
            if (inLength == blockSize)
            {
                int outLength = cipher.update(inBytes, 0, blockSize, outBytes);
                out.write(outBytes, 0, outLength);
            }
        }
    }
}
```

```

        else more = false;
    }
    if (inLength > 0) outBytes = cipher.doFinal(inBytes, 0, inLength);
    else outBytes = cipher.doFinal();
    out.write(outBytes);
}
}

```

Klucz ten zostanie zapisany w pliku *secret.key*.

Korzystając z utworzonego klucza, możemy teraz zaszyfrować zawartość dowolnego pliku *plaintextFile*:

```
java AESTest -encrypt plaintextFile encryptedFile secret.key
```

oraz odszyfrować plik *encryptedFile*, tak jak poniżej.

```
java AESTest -decrypt encryptedFile decryptedFile secret.key
```

Sposób działania programu jest przejrzysty. Opcja *-genkey* powoduje wygenerowanie nowego klucza i zapisanie go w pliku. Operacja ta trwa dość długo ze względu na czasochlonność procesu inicjalizacji bezpiecznego generatora losowego. Opcje *-encrypt* i *-decrypt* wywołują tę samą metodę *crypt*, która z kolei korzysta z metod *update* i *doFinal*. Metoda *update* wywoływana jest tak długo, jak bloki danych posiadają pełną długość, a metoda *doFinal* wywoływana jest dla niepełnego bloku końcowego lub po zakończeniu szyfrowania.

#### **java.crypto.Cipher 1.4**

- static Cipher getInstance(String algorithm)
- static Cipher getInstance(String algorithm, String provider)

Zwracają obiekt klasy *Cipher* reprezentujący podany algorytm szyfrowania. Wyrzucają wyjątek *NoSuchAlgorithmException*, jeśli dany algorytm nie jest dostępny.

- int getBlockSize()  
zwraca rozmiar (w bajtach) bloku stosowanego przez algorytm szyfrowania lub wartość 0, jeśli algorytm nie używa bloków.
- int getOutputSize(int inputLength)  
zwraca wielkość buforu wyjściowego niezbędnego, w przypadku gdy szyfrowana informacja posiada podaną wielkość. Metoda ta bierze pod uwagę także bajty już buforowane przez obiekt reprezentujący algorytm szyfrowania.
- void init(int mode, Key key)  
inicjuje algorytm szyfrowania. Parametr *mode* przyjmuje jedną z wartości *ENCRYPT\_MODE*, *DECRYPT\_MODE*, *WRAP\_MODE* lub *UNWRAP\_MODE*,
- byte[] update(byte[] in)
- byte[] update(byte[] in, int offset, int length)

- int update(byte[] in, int offset, int length, byte[] out)

Szyfrują blok danych. Pierwsze dwie metody zwracają zaszyfrowane dane, a trzecia zwraca liczbę bajtów umieszczonych w tablicy out.

- byte[] doFinal()
- byte[] doFinal(byte[] in)
- byte[] doFinal(byte[] in, int offset, int length)
- int doFinal(byte[] in, int offset, int length, byte[] out)

Szyfrują ostatni blok danych i wymiatają zawartość bufora wykorzystywanego przez algorytm. Pierwsze trzy metody zwracają zaszyfrowane dane, a czwarta liczbę bajtów umieszczonych w tablicy out.

#### java.crypto.KeyGenerator 1.4

- static KeyGenerator getInstance(String algorithm)  
zwraca obiekt reprezentujący podany algorytm. Wyrzuca wyjątek NoSuchAlgorithmException, jeśli dany algorytm nie jest dostępny.
- void init(SecureRandom random)
- void init(int keySize, SecureRandom random)  
Inicjują generator klucza.
- SecretKey generateKey()  
generuje nowy klucz.

#### javax.crypto.spec.SecretKeySpec 1.4

- SecretKeySpec(byte[] key, String algorithm)  
tworzy specyfikację klucza.

### 9.7.3. Strumienie szyfrujące

Biblioteka JCE dostarcza wygodnych w użyciu klas strumieni, które automatyzują proces szyfrowania i odszyfrowywania danych. Poniżej prezentujemy przykład szyfrowania danych zapisywanych w pliku za pomocą strumienia szyfrującego:

```
Cipher cipher = . . . ;
cipher.init(Cipher.ENCRYPT_MODE, key);
CipherOutputStream out = new CipherOutputStream(
    new FileOutputStream(outputFileName), cipher);
byte[] bytes = new byte[BLOCKSIZE];
int inLength = getData(bytes); //pobiera dane ze źródła
while (inLength != -1)
{
```

```

        out.write(bytes, 0, inLength);
        inLength = getData(bytes); //pobiera kolejne dane ze źródła
    }
    out.flush();
}

```

W podobny sposób możemy wykorzystać klasę CipherInputStream do odczytu i odszyfrowania danych z pliku:

```

Cipher cipher = . . . ;
cipher.init(Cipher.DECRYPT_MODE, key);
CipherInputStream in = new CipherInputStream(
    new FileInputStream(inputFileName), cipher);
byte[] bytes = new byte[BLOCKSIZE];
int inLength = in.read(bytes);
while (inLength != -1)
{
    putData(bytes, 0, inLength);
    inLength = in.read(bytes); //umieszcza dane w tablicy
}

```

Klasy strumieni szyfrujących uwalniają programistę od konieczności wywoływania metod update i doFinal.

#### javax.security.CipherInputStream 1.4

- `CipherInputStream(InputStream in, Cipher cipher)`  
tworzy strumień wejściowy odczytujący dane ze strumienia in i szyfrujący bądź odszyfrowujący je za pomocą podanego algorytmu.
- `int read()`
- `int read(byte[] b, int off, int len)`  
Odczytuje dane ze strumienia wejściowego, automatycznie je szyfrując bądź odszyfrowując.

#### javax.security.CipherOutputStream 1.4

- `CipherOutputStream(OutputStream out, Cipher cipher)`  
tworzy strumień wyjściowy zapisujący dane do strumienia out i szyfrujący bądź odszyfrowujący je za pomocą podanego algorytmu.
- `void write(int ch)`
- `void write(byte[] b, int off, int len)`  
Zapisują dane do strumienia wyjściowego, automatycznie je szyfrując bądź odszyfrowując.
- `void flush()`  
wymija bufor algorytmu szyfrowania oraz wykonuje operację dopełnienia, jeśli jest wymagana.

## 9.7.4. Szyfrowanie kluczem publicznym

Algorytm szyfrowania AES, którego przykład zastosowania pokazaliśmy w poprzednim podrozdziale, jest *algorytmem z kluczem symetrycznym*. Oznacza to, że ten sam klucz wykorzystywany jest do szyfrowania i odszyfrowania informacji. Jak łatwo się domyślić, piętą achillesową takiego algorytmu jest dystrybucja kluczy. Jeśli na przykład Alice wyśle zaszyfrowaną wiadomość Bobowi, to będzie on potrzebował do jej odszyfrowania tego samego klucza, którego użyła Alice do zaszyfrowania wiadomości. Jeśli Alice zmieni klucz, to, wysyłając kolejną wiadomość Bobowi, będzie musiała, korzystając z bezpiecznego kanału, przesyłać mu także nowy klucz. Skoro jednak Alice szyfruje wiadomości przesyłane Bobowi, to w praktyce może to oznaczać, że nie dysponuje bezpiecznym kanałem.

Kryptografia klucza publicznego rozwiązuje ten problem. Stosując szyfrowanie z kluczem publicznym, Bob posiada parę kluczy składającą się z klucza publicznego i odpowiadającego mu klucza prywatnego. Bob może udostępnić każdemu swój klucz publiczny, ale klucz prywatny musi przechowywać w bezpiecznym miejscu. Alice zaszyfruje wiadomość wysyłaną Bobowi za pomocą jego klucza.

W rzeczywistości nie jest to takie proste. Wszystkie znane dotąd algorytmy szyfrowania przy użyciu klucza publicznego są *znacznie* wolniejsze niż algorytmy z kluczem symetrycznym, takie jak DES czy AES. Szyfrowanie dużej ilości informacji za pomocą algorytmów z kluczem publicznym jest więc niepraktyczne. Problemu tego możemy jednak łatwo uniknąć, tworząc kombinację algorytmu szyfrowania kluczem publicznym z szybkim algorytmem szyfrowania symetrycznego w następujący sposób.

- 1.** Alice tworzy dowolny klucz algorytmu szyfrowania symetrycznego i wykorzystuje go do zaszyfrowania wiadomości.
- 2.** Alice szyfruje klucz symetryczny, korzystając z klucza publicznego Boba.
- 3.** Alice przesyła Bobowi zaszyfrowaną wiadomość i zaszyfrowany klucz symetryczny.
- 4.** Bob odszyfrowuje klucz symetryczny za pomocą swojego klucza prywatnego.
- 5.** Uzyskany klucz symetryczny Bob wykorzystuje do odszyfrowania wiadomości.

Jedynie Bob może odszyfrować klucz symetryczny, ponieważ tylko on posiada klucz prywatny. W ten sposób mało wydajny algorytm szyfrowania kluczem publicznym wykorzystywany jest jedynie do zaszyfrowania klucza symetrycznego.

Do najczęściej stosowanych algorytmów szyfrowania kluczem publicznym należy algorytm RSA opracowany przez Rivesta, Shamira i Adlemana. Do października 2000 roku algorytm ten był chroniony patentem przyznanym firmie RSA Security Inc. Licencje udzielane na jego wykorzystanie nie były tanie — stanowiły typowo 3% dochodu ze sprzedaży produktu zawierającego implementację algorytmu, lecz nie mniej niż 50 tysięcy USD w ciągu roku. Obecnie algorytm stanowi własność publiczną.

Aby użyć algorytmu RSA, musimy dysponować parą klucz publiczny-klucz prywatny. Wykorzystamy w tym celu obiekt klasy KeyPairGenerator, jak poniżej:

```
KeyPairGenerator pairgen = KeyPairGenerator.getInstance("RSA");
SecureRandom random = new SecureRandom();
pairgen.initialize(KEYSIZE, random);
```

```
KeyPair keyPair = pairgen.generateKeyPair();
Key publicKey = keyPair.getPublic();
Key privateKey = keyPair.getPrivate();
```

Program, którego kod źródłowy zamieszczono w listingu 9.20, dysponuje trzema opcjami. Opcja `-genkey` powoduje wygenerowanie pary kluczy. Opcja `-encrypt` generuje klucz AES i *szyfruje* go za pomocą klucza publicznego.

```
Key key = . . . ; //klucz algorytmu AES
Key publicKey = . . . ; //klucz publiczny algorytmu RSA
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.WRAP_MODE, publicKey);
byte[] wrappedKey = cipher.wrap(key);
```

**Listing 9.20.** rsa/RSATest.java

```
package rsa;

import java.io.*;
import java.security.*;
import javax.crypto.*;

/**
 * Program wykorzystujący algorytm szyfrowania RSA.
 * Uruchamianie:
 * java RSATest -genkey public private
 * java RSATest -encrypt plaintext encrypted public
 * java RSATest -decrypt encrypted decrypted private
 * @author Cay Horstmann
 * @version 1.01 2012-06-10
 */
public class RSATest
{
    private static final int KEYSIZE = 512;

    public static void main(String[] args)
        throws IOException, GeneralSecurityException, ClassNotFoundException
    {
        if (args[0].equals("-genkey"))
        {
            KeyPairGenerator pairgen = KeyPairGenerator.getInstance("RSA");
            SecureRandom random = new SecureRandom();
            pairgen.initialize(KEYSIZE, random);
            KeyPair keyPair = pairgen.generateKeyPair();
            try (ObjectOutputStream out = new ObjectOutputStream(new
                FileOutputStream(args[1])))
            {
                out.writeObject(keyPair.getPublic());
            }
            try (ObjectOutputStream out = new ObjectOutputStream(new
                FileOutputStream(args[2])))
            {
                out.writeObject(keyPair.getPrivate());
            }
        }
        else if (args[0].equals("-encrypt"))
        {
            KeyGenerator keygen = KeyGenerator.getInstance("AES");
```

```

SecureRandom random = new SecureRandom();
keygen.init(random);
SecretKey key = keygen.generateKey();

// szyfruje klucz AES za pomocą klucza publicznego RSA
try (ObjectInputStream keyIn = new ObjectInputStream(new
    → FileInputStream(args[3]));
    DataOutputStream out = new DataOutputStream(new
    → FileOutputStream(args[2]));
    InputStream in = new FileInputStream(args[1]) )
{
    Key publicKey = (Key) keyIn.readObject();
    Cipher cipher = Cipher.getInstance("RSA");
    cipher.init(Cipher.WRAP_MODE, publicKey);
    byte[] wrappedKey = cipher.wrap(key);
    out.writeInt(wrappedKey.length);
    out.write(wrappedKey);

    cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.ENCRYPT_MODE, key);
    Util.crypt(in, out, cipher);
}
}
else
{
    try (DataInputStream in = new DataInputStream(new FileInputStream(args[1]));
        ObjectInputStream keyIn = new ObjectInputStream(new FileInputStream(args[3]));
        OutputStream out = new FileOutputStream(args[2]));
    {
        int length = in.readInt();
        byte[] wrappedKey = new byte[length];
        in.read(wrappedKey, 0, length);

        // odszyfrowuje za pomocą klucza prywatnego RSA
        Key privateKey = (Key) keyIn.readObject();

        Cipher cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.UNWRAP_MODE, privateKey);
        Key key = cipher.unwrap(wrappedKey, "AES", Cipher.SECRET_KEY);

        cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.DECRYPT_MODE, key);

        Util.crypt(in, out, cipher);
    }
}
}

```

Następnie tworzy plik zawierający:

- długość zaszyfrowanego klucza,
  - zaszyfrowany klucz,
  - zawartość pliku zaszyfrowanego algorytmem AES

Opcja `-decrypt` umożliwia odszyfrowanie takiego pliku. Testując działanie programu, powiniśmy najpierw wygenerować parę kluczy:

```
java rsa.RSATest -genkey public.key private.key
```

Następnie zaszyfrować plik

```
java rsa.RSATest -encrypt plaintextFile encryptedFile public.key
```

i odszyfrować go, sprawdzając później, czy uzyskany plik jest wierną kopią pliku wyjściowego.

```
java rsa.RSATest -decrypt encryptedFile decryptedFile private.key
```

Tym przykładem kończymy omówienie zagadnień bezpieczeństwa na platformie Java. Pokazaliśmy w jaki sposób maszyna wirtualna razem z menedżerem bezpieczeństwa umożliwiają określenie pozwoleń dla programów, a także sposoby użycia biblioteki Java do uwierzytelniania i szyfrowania. Nie przedstawiliśmy natomiast kilku bardziej zaawansowanych bądź specjalistycznych zagadnień, do których należą między innymi:

- Interfejs programowy GSS (*Generic Security Services*), który obsługuje protokół Kerberos (a także inne protokoły bezpiecznej wymiany wiadomości). Wprowadzenie do tego zagadnienia jest dostępne na stronie <http://docs.oracle.com/javase/7/docs/technotes/guides/security/jgss/tutorials/index.html>.
- Obsługa warstwy SASL (*Simple Authentication and Security Layer*) używanej przez protokoły LDAP i IMAP. Implementację wykorzystania SASL przez własną aplikację najlepiej rozpocząć od lektury na stronie <http://docs.oracle.com/javase/7/docs/technotes/guides/security/sasl/sasl-refguide.html>.
- Obsługa warstwy SSL (*Secure Socket Layer*). Zastosowanie tej warstwy w przypadku protokołu HTTP jest przezroczyste dla programistów i sprowadza się do wykorzystania adresów URL rozpoczynających się od przedrostka https. Implementację wykorzystania SASL przez własną aplikację najlepiej rozpocząć od lektury dokumentacji interfejsu JSEE (Java Secure Socket Extension) na stronie <http://docs.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html>.

Po omówieniu zagadnień bezpieczeństwa na platformie Java w rozdziale 10. zajmiemy się przetwarzaniem rozproszonym.

# 10

## Skrypty, kompilacja i adnotacje

W tym rozdziale:

- Skrypty na platformie Java.
- Interfejs kompilatora.
- Stosowanie adnotacji.
- Składnia adnotacji.
- Adnotacje standardowe.
- Przetwarzanie adnotacji w kodzie źródłowym.
- Inżynieria kodu bajtowego.

W tym rozdziale omówimy trzy techniki przetwarzania kodu. Interfejs skryptów pozwala wywoływać kod napisany w językach skryptów takich jak JavaScript lub Groovy. Natomiast interfejs kompilatora pozwala nam kompilować kod Java z poziomu własnej aplikacji. Z kolei procesor adnotacji działa na kodzie źródłowym w języku Java lub plikach klas zawierających adnotacje. Jak zobaczymy, przetwarzanie adnotacji znajduje wiele zastosowań, od prostej diagnostyki kodu, po „inżynierię kodu bajtowego”, umieszczanie kodu bajtowego w plikach klas czy nawet uruchamianie programów.

### 10.1. Skrypty na platformie Java

Język skryptów pozwala uniknąć tradycyjnego cyklu edycja, kompilacja, konsolidacja, wykonanie dzięki interpretacji tekstu skryptu w trakcie jego wykonania. Języki skryptów mają wiele zalet:

- szybkie tworzenie nowych rozwiązań sprzyjające eksperymentowaniu,
- modyfikacja zachowania programu podczas jego działania,
- personalizacja programu przez użytkowników.

Z drugiej strony, większość języków skryptów nie posiada cech ułatwiających tworzenie skomplikowanych aplikacji, na przykład kontroli zgodności typów, hermetyzacji czy modularności.

Dlatego kusząca wydaje się możliwość połączenia zalet języków skryptów i tradycyjnych języków programowania. Interfejs skryptów pozwala nam zrealizować tę możliwość na platformie Java. Umożliwia on kodowi Java wywoływanie skryptów napisanych w językach skryptów takich jak JavaScript, Groovy, Ruby czy nawet tak egzotycznych językach jak Scheme czy Haskell. (Odwrotna możliwość, czyli dostęp do kodu w języku Java z poziomu skryptów, zależy od intencji twórców danego języka skryptów. Większość języków skryptów wykonywanych przez maszynę wirtualną Java dysponuje taką możliwością).

W kolejnych podrozdziałach pokażemy, jaki silnik wybrać dla poszczególnych języków, jak uruchamiać skrypty, i w jaki sposób wykorzystać zaawansowane możliwości oferowane przez niektóre silniki skryptów.

### 10.1.1. Wybór silnika skryptów

Silnik skryptów jest biblioteką pozwalającą wykonać skrypty napisane w określonym języku. Podczas uruchamiania maszyna wirtualna Java wykrywa dostępne silniki skryptów. Możemy uzyskać ich wyliczenie, tworząc obiekt klasy ScriptEngineManager i wywołując jego metodę getEngineFactories. Każda z otrzymanych fabryk silników możemy zapytać o nazwę obsługiwanej silnika, typy MIME i rozszerzenia nazw plików. Typowe wartości zostały przedstawione w tabeli 10.1.

**Tabela 10.1. Właściwości fabryk silników skryptów**

Silnik	Nazwy	Typy MIME	Rozszerzenia nazw plików
Rhino (dołączony do Java SE)	js, rhino, JavaScript, javascript, ECMAScript, ecmascript	application/javascript, application/ecmascript, text/javascript, text/ecmascript	
Groovy	groovy	brak	groovy
SISC Scheme	scheme, sisc	brak	scc, sce, scm, shp

Zwykle wiemy, jaki silnik jest nam potrzebny, i możemy zażądać go za pomocą nazwy, typu MIME lub rozszerzenia. Na przykład:

```
ScriptEngine engine = manager.getEngineByName("JavaScript");
```

Java SE 7 zawiera wersję Rhino interpretera JavaScript stworzonego przez fundację Mozilla. Możemy ją rozszerzać o dodatkowe języki, dostarczając odpowiednich plików JAR na ścieżce klas. W ogólnym przypadku potrzebne są dwa zbiory plików JAR. Sam język skryptów jest implementowany przez pojedynczy plik JAR lub zbiór takich plików. Silnik przystosowujący język do interfejsu skryptów zwykle wymaga osobnego pliku JAR. Witryna <http://java.net/projects/scripting> udostępnia silniki dla wielu języków skryptów. Na przykład dla języka Groovy ścieżka dostępu do klas powinna udostępniać pliki *groovy/lib/\** (z witryny <http://groovy.codehaus.org>) oraz plik *groovy-engine.jar* (z witryny <http://java.net/projects/scripting>).

**API** javax.script.ScriptEngineManager 6

- List<ScriptEngineFactory> getEngineFactories()
   
zwraca listę wszystkich wykrytych fabryk silników.
- ScriptEngine getEngineByName(String name)
- ScriptEngine getEngineByExtension(String extension)
- ScriptEngine getEngineByMimeType(String mimeType)
   
zwraca silnik skryptów na podstawie podanej nazwy, rozszerzenia nazw plików bądź typu MIME.

**API** javax.script.ScriptEngineFactory 6

- List<String> getNames()
- List<String> getExtensions()
- List<String> getMimeTypes()
   
zwraca nazwy, rozszerzenia nazw plików i typy MIME charakteryzujące daną fabrykę.

## 10.1.2. Wykonywanie skryptów i wiązania zmiennych

Gdy mamy już odpowiedni silnik, możemy wywołać skrypt w następujący sposób:

```
Object result = engine.eval(scriptString);
```

Jeśli skrypt został zapisany w pliku, należy otworzyć obiekt Reader i wywołać

```
Object result = engine.eval(reader);
```

Ten sam silnik może służyć do wywoływania wielu skryptów. Jeśli skrypt zdefiniuje pewne zmienne, funkcje lub klasy, to większość silników skryptów zachowa te definicje do późniejszego wykorzystania. Na przykład sekwencja wywołań

```
engine.eval("n = 1728");
Object result = engine.eval("n + 1");
```

zwróci wartość 1729.



Za pomocą poniższego wywołania możemy przekonać się, czy można bezpiecznie wykonywać współbieżnie skrypty w wielu wątkach:

```
Object param = factory.getParameter("THREADING");
```

Wywołanie to zwraca jedną z następujących wartości:

- null: współbieżne wykonanie nie jest bezpieczne,
- "MULTITHREADED": współbieżne wykonanie nie jest bezpieczne. Efekty działania skryptu w jednym wątku mogą być widoczne w innych wątkach.

- "THREAD-ISOLATED": tak jak "MULTITHREADED", ale dodatkowo dla każdego wątku utrzymywane są osobne wiązania zmiennych.
- "STATELESS": tak jak "THREAD-ISOLATED", ale skrypty nie zmieniają wiązań zmiennych.

Często zachodzi potrzeba dodania wiązania zmiennej do silnika. Wiązanie składa się z nazwy i związanego z nią obiektu Java. Rozważmy na przykład poniższe wywołania:

```
engine.put(k, 1728);
Object result = engine.eval("k + 1");
```

Kod skryptu odczytuje definicję k z wiązania w „zakresie silnika”. Jest to szczególnie ważne, ponieważ większość języków skryptów ma dostęp do obiektów Java i to często przy użyciu dużo prostszej składni niż w języku Java. Na przykład:

```
engine.put(b, new JButton());
engine.eval("f.text = 'Ok'");
```

I odwrotnie, możemy również pobierać zmienne, które zostały powiązane za pomocą instrukcji skryptu:

```
engine.eval("n = 1728");
Object result = engine.get("n");
```

Oprócz zakresu silnika istnieje również zakres globalny. Każde wiązanie, które dodamy do obiektu ScriptEngineManager, będzie widoczne dla wszystkich silników.

Zamiast dodawać poszczególne wiązania w zakresie silnika lub globalnym, możemy zebrać je za pomocą obiektu typu Bindings i przekazać razem metodzie eval:

```
Bindings scope = engine.createBindings();
scope.put(b, new JButton());
engine.eval(scriptString, scope);
```

Jest to przydatne w sytuacji, gdy zbiór wiązań nie powinien przetrwać do kolejnych wywołań metody eval.



Czasami przydatne są inne zakresy niż tylko zakres silnika i globalny. Na przykład kontener Web może potrzebować zakresów żądania i sesji. W takim przypadku jesteśmy zdani sami na siebie. Musimy stworzyć klasę, która implementuje interfejs ScriptContext, zarządzając kolekcją zakresów. Każdy zakres jest identyfikowany za pomocą liczby całkowitej, a zakresy o niższej wartości identyfikatora powinny być przeszukiwane jako pierwsze. (Biblioteka standardowa dostarcza klasę SimpleScriptContext, ale ona zarządza jedynie zakresem globalnym i zakresami silników).

### javax.script.ScriptEngine 6

- Object eval(String script)
- Object eval(Reader reader)
- Object eval(String script, Bindings bindings)

- Object eval(Reader reader, Bindings bindings)  
wykonuje skrypt podany w postaci łańcucha lub obiektu odczytu pliku, dla podanych wiązań.
- Object get(String key)
- void put(String key, Object value)  
zwraca lub konfiguruje wiązanie w zakresie silnika.
- Bindings createBindings()  
tworzy pusty obiekt wiązania dla danego silnika.

**API** javax.script.ScriptEngineManager 6

- Object get(String key)
- void put(String key, Object value)  
zwraca lub konfiguruje wiązanie w globalnym zakresie.

**API** javax.script.Bindings 6

- Object get(String key)
- void put(String key, Object value)  
zwraca lub umieszcza wiązanie w zakresie reprezentowanym przez dany obiekt Bindings.

### 10.1.3. Przekierowanie wejścia i wyjścia

Standardowe wejście i wyjście skryptu możemy przekierować, wywołując metody setReader i setWriter kontekstu skryptu. Na przykład:

```
StringWriter writer = new StringWriter();
engine.getContext().setWriter(new PrintWriter(writer, true));
```

W ten sposób wszystkie dane wyprowadzane przez JavaScript za pomocą funkcji print lub println zostaną wysłane do obiektu writer.



Metodzie setWriter możemy przekazać dowolny obiekt Writer, ale silnik Rhino wyrzuca wyjątek, jeśli obiekt ten nie jest typu PrintWriter.

Metody setReader i setWriter mają wpływ jedynie na standardowe wejście i wyjście silnika skryptów. Jeśli na przykład wykonamy poniższy kod JavaScript

```
println("Hello");
java.lang.System.out.println("World");
```

to przekierowane zostanie jedynie wyjście pierwszego wywołania.

W przypadku silnika Rhino nie istnieje pojęcie standardowego wejścia i wobec tego wywołanie metody setReader nie przynosi efektu.

**API** *javax.script.ScriptEngine* 6

- `ScriptContext getContext()`

zwraca domyślny kontekst skryptu dla danego silnika.

**API** *javax.script.ScriptContext* 6

- `Reader getReader()`
- `void setReader(Reader reader)`
- `Writer getWriter()`
- `void setWriter(Writer writer)`
- `Writer getErrorWriter()`
- `void setErrorWriter(Writer writer)`

zwraca lub konfiguruje obiekt wejścia lub obiekt wyjścia dla standardowego wejścia i wyjścia błędów.

#### 10.1.4. Wywoływanie funkcji i metod skryptów

Wiele silników skryptów pozwala wywołać funkcję języka skryptów bez konieczności wykonania całego skryptu. Jest to przydatne, gdy na przykład chcemy umożliwić użytkownikowi implementację pewnej usługi w wybranym przez niego języku skryptów.

Silnik, który oferuje taką możliwość, musi implementować interfejs `Invocable`. W szczególności interfejs ten implementuje silnik `Rhino`.

Aby wywołać funkcję, wywołujemy metodę `invokeFunction`, podając nazwę funkcji i jej parametry:

```
if (engine implements Invocable)
    ((Invocable) engine).invokeFunction("aFunction", param1, param2);
```

Jeśli język skryptów jest obiektowy, to możemy wywołać metodę skryptu:

```
((Invocable) engine).invokeMethod(implicitParam, "aMethod", explicitParam1,
    ↴explicitParam2);
```

W tym przypadku obiekt `implicitParam` spełnia rolę zastępcy obiektu w języku skryptów. Obiekt ten musi być wynikiem wcześniejszego wywołania silnika skryptów.



Jeśli silnik skryptów nie implementuje interfejsu `Invocable`, nadal istnieje możliwość wywołania metody w sposób niezależny od konkretnego języka skryptów. Metoda `getMethodCallSyntax` interfejsu `ScriptEngineFactory` tworzy łańcuch, który możemy przekazać następnie metodzie `eval`. Jednak w tym wypadku wszystkie parametry metody muszą być związane z nazwami, podczas gdy metodę `invokeMethod` możemy wywołać dla dowolnych wartości.

Możemy nawet posunąć się o krok dalej i zażądać, aby silnik skryptów implementował interfejs języka Java. Wtedy możemy wywoływać funkcje i metody skryptu, posługując się składnią wywołań metod w języku Java.

Szczegóły zależą od konkretnego silnika skryptów, ale zwykle należy dostarczyć osobną funkcję dla każdej metody interfejsu. Rozważmy na przykład poniższy interfejs w języku Java:

```
public interface Greeter
{
    String greet(String whom);
}
```

Dla Rhino dostarczymy poniższą funkcję:

```
function greet(x) { return "Hello, " + x + "!"; }
```

Kod ten należy najpierw przekazać metodzie eval. Następnie musimy użyć wywołania:

```
Greeter g = ((Invocable) engine).getInterface(Greeter.class);
```

Od tego momentu możemy używać zwykłych wywołań w języku Java:

```
String result = g.greet("World");
```

W rezultacie wywołana zostanie metoda greet w języku JavaScript. Rozwiązanie takie przypomina nieco wywoływanie metod zdalnych, omówione w rozdziale 11.

W przypadku obiektowego języka skryptów możemy używać klas skryptu za pośrednictwem odpowiedniego interfejsu. Rozważmy na przykład poniższy kod JavaScript, który definiuje klasę SimpleGreeter.

```
function SimpleGreeter(salutation) { this.salutation = salutation; }
SimpleGreeter.prototype.greet = function(whom) { return this.salutation + ", " +
➥ whom + "!"; }
```

Klasa ta może służyć do tworzenia różnych rodzajów powitań (takich jak Hello, Goodbye i tym podobnych).



Więcej informacji na temat definiowania klas w języku JavaScript znajdziesz w książce *JavaScript — The Definitive Guide*, wyd. piąte, autorstwa Davida Flanagan (O'Reilly 2006).

Po przekazaniu definicji klasy JavaScript metodzie eval wywołujemy metodę getInterface:

```
Object goodbyeGreeter = engine.eval("new SimpleGreeter('Goodbye')");
Greeter g = ((Invocable) engine).getInterface(goodbyeGreeter, Greeter.class);
```

Jeśli teraz w języku Java użyjemy wywołania g.greet("World"), to w efekcie zostanie wywołana metoda greet obiektu goodbyeGreeter języka JavaScript. Wynikiem jej działania będzie łańcuch "Goodbye, World!".

Podsumowując, interfejs Invocable przydaje się, gdy chcemy wywoływać kod skryptów w języku Java, nie wnikając w szczegóły składni języka skryptów.

**API** *javax.script.Invocable* 6

- `Object invokeFunction(String name, Object... parameters)`
- `Object invokeMethod(Object implicitParameter, String name, Object... explicitParameters)`  
wywołuje funkcję lub metodę o podanej nazwie, której przekazuje podane parametry.
- `<T> T getInterface(Class<T> iface)`  
zwraca implementację podanego interfejsu, która implementuje jego metody za pomocą funkcji języka skryptów.
- `<T> T getInterface(Object implicitParameter, Class<T> iface)`  
zwraca implementację podanego interfejsu, która implementuje jego metody za pomocą metod podanego obiektu.

## 10.1.5. Kompilacja skryptu

Niektóre silniki skryptów mogą kompilować kod skryptów do postaci pośredniej umożliwiającej efektywniejsze wykonanie. Silniki te implementują interfejs `Compilable`. Poniższy przykład pokazuje sposób komplikacji skryptu zapisanego w pliku:

```
Reader reader = new FileReader("myscript.js");
CompiledScript script = null;
if (engine implements Compilable)
    CompiledScript script = ((Compilable) engine).compile(reader);
```

Po skompilowaniu skryptu możemy go wykonać. Poniższy kod wykonuje skompilowany skrypt, jeśli komplikacja się powiodła, lub oryginalny skrypt w przypadku, gdy silnik nie obsługuje komplikacji.

```
if (script != null)
    script.eval();
else
    engine.eval(reader);
```

Oczywiście komplikacja skryptu ma sens przede wszystkim wtedy, gdy chcemy go wykonywać wielokrotnie.

**API** *javax.script.Compilable* 6

- `CompiledScript compile(String script)`
- `CompiledScript compile(Reader reader)`  
komplikują skrypt przekazany jako łańcuch lub obiekt odczytu.

**API** *javax.script.CompiledScript* 6

- `Object eval()`

- Object eval(Bindings bindings)

wykonują skrypt.

## 10.1.6. Przykład: skrypty i graficzny interfejs użytkownika

Omówione możliwości interfejsu skryptów na platformie Java zilustrujemy przykładowym programem pozwalającym użytkownikowi specyfikować obsługę zdarzeń w wybranym przez niego języku skryptów.

Przyjrzyjmy się zatem programowi przedstawionemu na listingu 10.1, który dodaje możliwość użycia skryptów dla dowolnej klasy ramki. Domyslnie wczytuje klasę ButtonFrame przedstawioną na listingu 10.2, która przypomina rozwiązanie z przykładu obsługi zdarzeń przedstawionego w książce *Java 2. Podstawy*. Istnieją jednak dwie różnice:

- każdy komponent ma skonfigurowaną właściwość name,
- brak obiektów obsługi zdarzeń.

**Listing 10.1.** *script/ScriptTest.java*

```
package script;

import java.awt.*;
import java.beans.*;
import java.io.*;
import java.lang.reflect.*;
import java.util.*;
import javax.script.*;
import javax.swing.*;

/**
 * @version 1.01 2012-01-28
 * @author Cay Horstmann
 */
public class ScriptTest
{
    public static void main(final String[] args)
    {
        EventQueue.invokeLater(new Runnable()
        {
            public void run()
            {
                try
                {
                    ScriptEngineManager manager = new ScriptEngineManager();
                    String language;
                    if (args.length == 0)
                    {
                        System.out.println("Available factories: ");
                        for (ScriptEngineFactory factory : manager.getEngineFactories())
                            System.out.println(factory.getEngineName());
                    }
                    language = "js";
                }
            }
        });
    }
}
```

```

        }
    else language = args[0];

    final ScriptEngine engine = manager.getEngineByName(language);
    if (engine == null)
    {
        System.err.println("No engine for " + language);
        System.exit(1);
    }

    final String frameClassName = args.length < 2 ? "buttons1.ButtonFrame"
    ↪: args[1];

    JFrame frame = (JFrame) Class.forName(frameClassName).newInstance();
    InputStream in = frame.getClass().getResourceAsStream("init."
    ↪language);
    if (in != null) engine.eval(new InputStreamReader(in));
    getComponentBindings(frame, engine);

    final Properties events = new Properties();
    in = frame.getClass().getResourceAsStream(language + ".properties");
    events.load(in);

    for (final Object e : events.keySet())
    {
        String[] s = ((String) e).split("\\.");
        addListener(s[0], s[1], (String) events.get(e), engine);
    }
    frame.setTitle("ScriptTest");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
}
catch (ReflectiveOperationException | IOException
    | ScriptException | IntrospectionException ex)
{
    ex.printStackTrace();
}
}
};

}

/**
 * Zbiera wszystkie nazwane komponenty w kontenerze.
 * @param c komponent
 * @param engine silnik skryptów
 */
private static void getComponentBindings(Component c, ScriptEngine engine)
{
    String name = c.getName();
    if (name != null) engine.put(name, c);
    if (c instanceof Container)
    {
        for (Component child : ((Container) c).getComponents())
            getComponentBindings(child, engine);
    }
}
}

```

```

    /**
     * Dodaje obiekt nasłuchujący do obiektu,
     * który wykonuje skrypt.
     * @param beanName nazwa ziarnka, do którego należy dodać obiekt nasłuchujący
     * @param eventName nazwa typu obiektu nasłuchującego, np. "action" lub "change"
     * @param scriptCode kod skryptu do wykonania
     * @param engine silnik wykonujący skrypt
     * @param bindings wiązania potrzebne do wykonania skryptu
     * @throws IntrospectionException
     */
    private static void addListener(String beanName, String eventName, final String
        ↳scriptCode,
        final ScriptEngine engine) throws ReflectiveOperationException, IntrospectionException
    {
        Object bean = engine.get(beanName);
        EventSetDescriptor descriptor = getEventSetDescriptor(bean, eventName);
        if (descriptor == null) return;
        descriptor.getAddListenerMethod().invoke(bean,
            Proxy.newProxyInstance(null, new Class[] { descriptor.getListenerType() },
                new InvocationHandler()
            {
                public Object invoke(Object proxy, Method method, Object[] args)
                    throws Throwable
                {
                    engine.eval(scriptCode);
                    return null;
                }
            }));
    }

    private static EventSetDescriptor getEventSetDescriptor(Object bean, String eventName)
        throws IntrospectionException
    {
        for (EventSetDescriptor descriptor : Introspector.getBeanInfo(bean.getClass())
            .getEventSetDescriptors())
            if (descriptor.getName().equals(eventName)) return descriptor;
        return null;
    }
}

```

**Listing 10.2.** buttons1/ButtonFrame.java

```

package buttons1;

import javax.swing.*;

public class ButtonFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;

    private JPanel panel;
    private JButton yellowButton;
    private JButton blueButton;
    private JButton redButton;

    public ButtonFrame()
    
```

```

    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        panel = new JPanel();
        panel.setName("panel");
        add(panel);

        yellowButton = new JButton("Yellow");
        yellowButton.setName("yellowButton");
        blueButton = new JButton("Blue");
        blueButton.setName("blueButton");
        redButton = new JButton("Red");
        redButton.setName("redButton");

        panel.add(yellowButton);
        panel.add(blueButton);
        panel.add(redButton);
    }
}

```

Obiekty obsługujące zdarzeń definiujemy w pliku właściwości. Każda definicja właściwości ma postać

*nazwaKomponentu.nazwaZdarzenia = kodSkryptu*

Jeśli wybierzemy język JavaScript, to obiekty obsługujące zdarzeń konfigurujemy w pliku *js.properties* w następujący sposób:

```

yellowButton.action=panel.background = java.awt.Color.YELLOW
blueButton.action=panel.background = java.awt.Color.BLUE
redButton.action=panel.background = java.awt.Color.RED

```

Przykłady kodu dla tej książki zawierają również odpowiednie pliki dla języków Groovy i SISC Scheme.

Program rozpoczyna swoje działanie od załadowania silnika dla języka skryptów określonego w wierszu wywołania programu. Jeśli żaden język nie został podany, program domyślnie używa języka JavaScript.

Później program przetwarza skrypt *init.język*, jeśli taki istnieje. Rozwiązywanie takie wydaje się dobrym pomysłem w ogólnym przypadku. Co więcej, interpreter języka Scheme wymaga dodatkowej inicjalizacji, której nie ma sensu umieszczać w każdym skrypcie obsługi zdarzeń.

Następnie rekurencyjnie przeglądamy wszystkie komponenty podrzędne i dodajemy wiązania (*nazwa, obiekt*) w zakresie silnika.

Program wczytuje potem plik *język.properties*. Dla każdej właściwości tworzymy zastępczy obiekt obsługi zdarzeń, który umożliwia wykonanie kodu skryptu. Szczegóły tego procesu są nieco techniczne. W ich zrozumieniu może pomóc lektura odpowiedniego podrozdziału poświęconego obiektom zastępczym w rozdziale 6. książki *Java 2. Podstawy* oraz podrozdziału omawiającego zdarzenia JavaBeans w rozdziale 8. niniejszej książki. Sedno tego fragmentu kodu sprowadza się jednak do tego, że każdy obiekt obsługi zdarzeń używa powyższego wywołania:

```
engine.eval(scriptCode);
```

Przyjrzyjmy się bliżej sytuacji dla komponentu yellowButton. Przetwarzając poniższy wiersz skryptu

```
yellowButton.action=panel.background = java.awt.Color.YELLOW
```

odnajdujemy komponent JButton o nazwie "yellowButton". Następniełączamy do niego obiekt obsługujący zdarzeń ActionListener posiadający metodę actionPerformed, która wykonuje skrypt

```
panel.background = java.awt.Color.YELLOW
```

Silnik skryptów zawiera wiązanie nazwy "panel" z obiektem typu JPanel. W momencie zajścia zdarzenia wykonana zostaje metoda setBackground tego obiektu, co powoduje zmianę koloru.

Omawiany program uruchamiamy, wywołując

```
java script.ScriptTest
```

W tym przypadku obsługa zdarzeń będzie się domyślnie odbywać w języku JavaScript.

Dla języka Groovy program wywołujemy w poniższy sposób:

```
java -classpath .:groovy/lib/*:jsr223-engines/groovy/build/groovy-engine.jar
→ script.ScriptTest groovy
```

W tym przypadku groovy jest katalogiem, w którym zainstalowaliśmy Groovy, a jsr223-engines jest katalogiem zawierającym adaptery silnika pobrane z witryny <http://java.net/projects/scripting>.

Aby wypróbować nasz przykład z językiem Scheme, powinieneś go załadować z witryny <http://sisc-scheme.org/> i wywołać program w następujący sposób:

```
java -classpath .:sisc/*:jsr223-engines/scheme/build/scheme-engine.jar script.ScriptTest
→scheme
```

Aplikacja ta demonstruje zastosowanie skryptów w programowaniu graficznego interfejsu użytkownika na platformie Java. Można nawet pójść o krok dalej i opisać interfejs użytkownika w pliku XML w sposób pokazany w rozdziale 2. W takim przypadku nasz program stałby się interpreterem graficznego interfejsu użytkownika, którego wygląd byłby zdefiniowany w języku XML, a zachowanie w języku skryptów. Można tu dopatrzeć się pewnego podobieństwa do dynamicznych stron HTML czy środowisk wykorzystujących skrypty po stronie serwera.

## 10.2. Interfejs kompilatora

W poprzednich podrozdziałach pokazaliśmy sposób korzystania z kodu skryptów na platformie Java. Teraz zajmiemy się innym scenariuszem: programami w języku Java, które kompilują kod w języku Java. Istnieje wiele kategorii programów narzędziowych, które muszą wywoływać kompilator Java:

- środowiska tworzenia aplikacji,
- programy uczące języka Java,
- narzędzie automatyzujące proces tworzenia aplikacji i jej testowania,
- narzędzia wykorzystujące szablony uzupełniane niewielkimi porcjami kodu Java, na przykład JavaServer Pages (JSP).

W przeszłości aplikacje wywoływały kompilator Java, wykorzystując nieudokumentowane klasy biblioteki *jdk/lib/tools.jar*. Począwszy od wersji Java SE 6, dostępny jest publicznie odpowiedni interfejs i używanie biblioteki *tools.jar* nie jest już konieczne. W tym podrozdziale omówimy interfejs programowy kompilatora.

## 10.2.1. Kompilacja w najprostszy sposób

Wywołanie kompilatora jest bardzo proste. Oto przykład:

```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
OutputStream outStream = ...;
OutputStream errStream = ...;
int result = compiler.run(null, outStream, errStream, "-sourcepath", "src",
    ➔"Test.java");
```

Jeśli wartość zmiennej *result* będzie równa 0, komplikacja zakończyła się pomyślnie.

Kompilator wysyła informacje o postępie komplikacji oraz komunikaty o błędach do dostarczonych mu strumieni. Jeśli wywołując kompilator, użyjemy *null* jako wartości parametrów określających te strumienie, to kompilator będzie wysyłał informacje do strumieni *System.out* i *System.err*. Pierwszym parametrem metody *run* jest strumień wejściowy. Ponieważ kompilator nie pobiera żadnych danych z konsoli, to zawsze nadajemy temu parametrowi wartość *null*. (Metoda *run* jest dziedziczona po bardziej ogólnym interfejsie *Tool*, który umożliwia działanie narzędzi wczytujących dane wejściowe).

Pozostałe parametry metody *run* reprezentują argumenty przekazywane kompilatorowi, gdy wywołujemy go w wierszu poleceń. Określają one opcje kompilatora oraz nazwy plików.

## 10.2.2. Stosowanie zadań kompilacji

Jeszcze większą kontrolę nad przebiegiem procesu komplikacji umożliwia nam obiekt *CompilationTask*. W szczególności możemy:

- sterować źródłem kodu, na przykład dostarczając go w postaci łańcucha zamiast pliku;
- kontrolować lokalizację plików klas, na przykład umieszczając je w bazie danych;
- nasłuchiwać błędów i ostrzeżeń w trakcie komplikacji;
- uruchamiać kompilator w tle.

Lokalizacją plików źródłowych i plików klas zarządza JavaFileManager. Obiekt ten jest odpowiedzialny za ustalenie instancji klasy JavaFileObject dla plików źródłowych i plików klas. Instancja JavaFileObject może reprezentować plik na dysku lub dostarczać innego mechanizmu odczytu i zapisu zawartości.

Aby nasłuchiwać komunikatów o błędach kompilacji, instalujemy obiekt DiagnosticListener. Otrzymuje on obiekt Diagnostic za każdym razem, gdy kompilator wysyła ostrzeżenie lub komunikat o błędzie. Klasa DiagnosticCollector implementuje ten interfejs. Zbiera ona wszystkie informacje wysyłane przez kompilator, co umożliwia ich przeglądanie po zakończeniu kompilacji.

Obiekt Diagnostic zawiera informacje o lokalizacji błędu (nazwę pliku, numer wiersza i kolumny) oraz jego opis.

Obiekt CompilationTask uzyskujemy, wywołując metodę getTask klasy JavaCompiler. Musimy przy tym określić:

- Obiekt Writer do zapisu informacji wysyłanych przez kompilator, które nie są raportowane za pomocą obiektów Diagnostic, lub podać wartość null, jeśli kompilator ma używać w tym celu strumienia System.err.
- Obiekt JavaFileManager lub podać wartość null, jeśli kompilator ma używać własnego, standardowego menedżera plików.
- Obiekt DiagnosticListener.
- Łącuch opcji lub wartość null w przypadku braku opcji.
- Nazwy klas dla przetwarzania adnotacji lub wartości null, jeśli nie będą wykorzystywane (przetwarzaniem adnotacji zajmiemy się w dalszej części tego rozdziału).
- Instancje JavaFileObject dla plików źródłowych.

Ostatnie trzy z wymienionych argumentów należy podać jako obiekty Iterable. Na przykład sekwencję opcji możemy określić w następujący sposób:

```
Iterable<String> options = Arrays.asList("-g", "-d", "classes");
```

Alternatywnie możemy użyć dowolnej klasy kolekcji.

Jeśli chcemy, aby kompilator czytał pliki źródłowe z dysku, możemy zażądać od StandardJavaFileManager, aby na podstawie nazw plików lub obiektów File utworzył odpowiednie instancje JavaFileObject. Na przykład:

```
StandardJavaFileManager fileManager = compiler.getStandardFileManager(null, null,
    null);
Iterable<JavaFileObject> fileObjects =
    fileManager.getJavaFileObjectsFromStrings(fileName);
```

Jeśli jednak chcemy, aby kompilator czytał kod z innego źródła niż pliki dyskowe, musimy stworzyć własną klasę pochodną klasy JavaFileObject. Listing 10.3 przedstawia kod klasy udostępniającej kod źródłowy umieszczony w obiektach klasy StringBuilder. Klasa ta rozszerza klasę SimpleJavaFileObject i zastępuje metodę getCharContent własną wersją zwracającą zawartość obiektu StringBuilder. Klasę tę wykorzystujemy w naszym programie przykładowym, który dynamicznie tworzy kod klasy w języku Java i kompiluje go.

**Listing 10.3.** compiler/StringBuilderJavaSource.java

```

package compiler;

import java.net.*;
import javax.tools.*;

/**
 * Umieszcza kod w obiekcie typu StringBuilder.
 * @version 1.00 2007-11-02
 * @author Cay Horstmann
 */
public class StringBuilderJavaSource extends SimpleJavaFileObject
{
    private StringBuilder code;

    /**
     * Tworzy nowy obiekt StringBuilderJavaSource
     * @param name nazwa pliku źródłowego reprezentowanego przez ten obiekt
     */
    public StringBuilderJavaSource(String name)
    {
        super(URI.create("string:/// " + name.replace('.', '/') + Kind.SOURCE.extension),
              Kind.SOURCE);
        code = new StringBuilder();
    }

    public CharSequence getCharContent(boolean ignoreEncodingErrors)
    {
        return code;
    }

    public void append(String str)
    {
        code.append(str);
        code.append('\n');
    }
}

```

Klasa CompilationTask implementuje interfejs Callable<Boolean>. Możemy przekazać ją obiektowi do wykonania w osobnym wątku lub po prostu wywołać metodę call. Jeśli metoda call zwróci wartość Boolean.FALSE, oznacza to, że komplikacja nie powiodła się.

```

Callable<Boolean> task = new JavaCompiler.CompilationTask(null, fileManager,
    ↳ diagnostics, options, null, fileObjects);
if (!task.call())
    System.out.println("Compilation failed");

```

Jeśli kompilator ma tworzyć pliki klas na dysku, to wystarczy mu w tym celu standardowy obiekt klasy JavaFileManager. Jednak nasz przykładowy program będzie generować pliki klas w tablicach bajtów i potem wczytywać je z pamięci przy użyciu specjalnej procedury ładowania klas. Listing 10.4 definiuje klasę implementującą interfejs JavaFileObject. Jej metoda openOutputStream zwraca strumień ByteArrayOutputStream, w którym kompilator będzie umieszczać tworzony kod bajtowy.

**Listing 10.4.** compiler/ByteArrayJavaClass.java

```

pac
kage compiler;

import java.io.*;
import java.net.*;
import javax.tools.*;

/**
 * Klasa przechowująca kod bajtowy w tablicy bajtów.
 * @version 1.00 2007-11-02
 * @author Cay Horstmann
 */
public class ByteArrayJavaClass extends SimpleJavaFileObject
{
    private ByteArrayOutputStream stream;

    /**
     * Tworzy nowy obiekt ByteArrayJavaClass
     * @param name nazwa pliku klasy reprezentowanego przez ten obiekt
     */
    public ByteArrayJavaClass(String name)
    {
        super(URI.create("bytes://" + name), Kind.CLASS);
        stream = new ByteArrayOutputStream();
    }

    public OutputStream openOutputStream() throws IOException
    {
        return stream;
    }

    public byte[] getBytes()
    {
        return stream.toByteArray();
    }
}

```

Jednak poinformowanie menedżera plików wykorzystywanego przez kompilator o tym, że powinien używać naszej implementacji interfejsu JavaFileObject, nie jest takie proste. Biblioteka języka Java nie udostępnia klasy implementującej interfejs StandardJavaFileManager. Musimy zatem utworzyć klasę pochodną klasy ForwardingJavaFileManager, która deleguje wszystkie wywołania danego menedżera plików. W naszym przypadku musimy jedynie zmienić zachowanie metody getJavaFileForOutput. Osiagniemy to, stosując poniższy schemat:

```

JavaFileManager fileManager = compiler.getStandardFileManager(diagnostics, null,
    null);
fileManager = new ForwardingJavaFileManager<JavaFileManager>(fileManager)
{
    public JavaFileObject getJavaFileForOutput(Location location, final String
        className,
        Kind kind, FileObject sibling) throws IOException
    {
        return specjalizowany obiekt implementujący interfejs JavaFileObject
    }
};

```

Podsumowując: jeśli chcemy wywołać kompilator w zwykły sposób, tak aby czytał i zapisywał pliki dyskowe, używamy metody `run` obiektu `JavaCompiler`. Możemy przy tym przechwytywać ostrzeżenia i komunikaty o błędach komplikacji, ale wtedy musimy sami je parsować.

Jeśli natomiast potrzebujemy dokładniejszej kontroli nad obsługą plików lub raportowaniem błędów, używamy interfejsu `CompilationTask`. Jest on dość skomplikowany, ale pozwala kontrolować wszystkie aspekty procesu komplikacji.

#### `javax.tools.Tool` 6

- `int run(InputStream in, OutputStream out, OutputStream err, String... arguments)`

uruchamia program narzędziowy z użyciem podanych strumieni wejścia, wyjścia i błędów, a także argumentów uruchomienia. Zwraca 0 w przypadku poprawnego wykonania lub inną wartość w przypadku błędu.

#### `javax.tools.JavaCompiler` 6

- `StandardJavaFileManager getStandardFileManager(DiagnosticListener<? super JavaFileObject> diagnosticListener, Locale locale, Charset charset)`

zwraca standardowego menedżera plików używanego przez kompilator. W przypadku menedżera stosującego domyślny sposób raportowania błędów, domyślny lokalizator i zbiór znaków dostarczamy `null` jako wartości parametrów wywołania.

- `JavaCompiler.CompilationTask getTask(Writer out, JavaFileManager fileManager, DiagnosticListener<? super JavaFileObject> diagnosticListener, Iterable<String> options, Iterable<String> classesForAnnotationProcessing, Iterable<? extends JavaFileObject> sourceFiles)`

zwraca zadanie komplikacji dla podanych plików źródłowych. Sposób stosowania został omówiony w ostatnim podrozdziale.

#### `javax.tools.StandardJavaFileManager` 6

- `Iterable<? extends JavaFileObject> getJavaFileObjectsFromStrings(Iterable<String> fileNames)`
- `Iterable<? extends JavaFileObject> getJavaFileObjectsFromFiles(Iterable<? extends File> files)`

przekształca sekwencję nazw plików lub obiektów `File` w sekwencję instancji `JavaFileObject`.

#### `javax.tools.JavaCompiler.CompilationTask` 6

- `Boolean call()`  
wykonuje zadanie komplikacji.

**API** javax.tools.DiagnosticCollector<S> 6

- DiagnosticCollector()
 

tworzy pusty obiekt DiagnosticCollector.
- List<Diagnostic<? extends S>> getDiagnostics()
 

wzwraca zebrane informacje.

**API** javax.tools.Diagnostic<S> 6

- S getSource()
 

wzwraca obiekt źródła związany z tą informacją diagnostyczną.
- Diagnostic.Kind getKind()
 

wzwraca typ informacji diagnostycznej reprezentowany przez jedną z wartości ERROR, WARNING, MANDATORY\_WARNING, NOTE lub OTHER.
- String getMessage(Locale locale)
 

wzwraca komunikat opisujący problem reprezentowany przez obiekt Diagnostic. Jeśli ma zostać użyty domyślny lokalizator, metodzie przekazujemy wartość null.
- long getLineNumber()
- long getColumnNumber()
- wzwraca numer wiersza lub kolumny, w której wystąpił problem reprezentowany przez obiekt Diagnostic.

**API** javax.tools.SimpleJavaFileObject 6

- CharSequence getCharContent(boolean ignoreEncodingErrors)
 

metodę tę zastępujemy, tworząc obiekt reprezentujący kod źródłowy. Metoda ta wzwraca kod źródłowy w języku Java.
- OutputStream openOutputStream()
 

metodę tę zastępujemy, tworząc obiekt reprezentujący plik klasy. Metoda ta wzwraca strumień, w którym kompilator zapisuje kod bajtowy.

**API** javax.tools.ForwardingJavaFileManager<M extends JavaFileManager> 6

- protected ForwardingJavaFileManager(M fileManager)
 

tworzy obiekt JavaFileManager, który deleguje wszystkie wywołania do podanego menedżera plików.
- FileObject getFileForOutput(JavaFileManager.Location location, String className, JavaFileObject.Kind kind, FileObject sibling)
 

wywołanie tej metody przechwytyjemy, gdy chcemy zastąpić obiekt zapisu plików klas własnym obiektem. Parametr kind przyjmuje jedną z wartości SOURCE, CLASS, HTML lub OTHER.

## 10.2.3. Przykład: dynamiczne tworzenie kodu w języku Java

Dynamiczne strony WWW wykonane w technologii JSP pozwalają łączyć HTML z niewielkimi porcjami kodu w języku Java, na przykład:

```
<p>The current date and time is <b><%= new java.util.Date() %></b>.</p>
```

Silnik JSP dynamicznie kompiluje kod Java do postaci serwletu. W naszym przykładowym programie będziemy natomiast generować dynamicznie kod Swing. Koncepcja jego działania sprowadza się do wykorzystania narzędzia tworzącego układ komponentów interfejsu użytkownika oraz określenia sposobu ich zachowania w osobnym pliku. Listing 10.5 przedstawia bardzo prosty przykład klasy ramki, a listing 10.6 zawiera kod akcji przycisków. Zwróćmy uwagę, że konstruktor klasy ramki wywołuje abstrakcyjną metodę addEventHandlers. Nasz generator kodu utworzy klasę pochodną implementującą metodę addEventHandlers, która doda osobny obiekt nasłuchujący na podstawie każdego wiersza pliku *action.properties*. (Czytelnikowi pozostawiamy rozbudowę kodu o obsługę innych zdarzeń).

**Listing 10.5.** buttons2/ButtonFrame.java

---

```
package buttons2;
import javax.swing.*;

/**
 * @version 1.00 2007-11-02
 * @author Cay Horstmann
 */
public abstract class ButtonFrame extends JFrame
{
    public static final int DEFAULT_WIDTH = 300;
    public static final int DEFAULT_HEIGHT = 200;

    protected JPanel panel;
    protected JButton yellowButton;
    protected JButton blueButton;
    protected JButton redButton;

    protected abstract void addEventHandlers();

    public ButtonFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        panel = new JPanel();
        add(panel);

        yellowButton = new JButton("Yellow");
        blueButton = new JButton("Blue");
        redButton = new JButton("Red");

        panel.add(yellowButton);
        panel.add(blueButton);
        panel.add(redButton);

        addEventHandlers();
    }
}
```

**Listing 10.6.** buttons2/action.properties

---

```
yellowButton=panel.setBackground(java.awt.Color.YELLOW);
blueButton=panel.setBackground(java.awt.Color.BLUE);
redButton=panel.setBackground(java.awt.Color.RED);
```

---

Klasę pochodną umieścimy w pakiecie o nazwie `x`, mając nadzieję, że nie jest on używany nigdzie indziej w programie. Wygenerowany kod będzie mieć następującą postać:

```
package x;
public class Frame extends NazwaKlasyBazowej {
    protected void addEventHandlers() {
        nazwaKomponentu.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent) {
                kod obiektu obsługi
            } });
        // powtarza dla innych obiektów obsługi zdarzeń ...
    }
}
```

Metoda `buildSource` programu przedstawionego na listingu 10.7 tworzy powyższy kod i umieszcza go w obiekcie `StringBuilderJavaSource`. Obiekt ten zostaje następnie przekazany kompilatorowi języka Java.

**Listing 10.7.** compiler/CompilerTest.java

---

```
package compiler;

import java.awt.*;
import java.io.*;
import java.util.*;
import java.util.List;
import javax.swing.*;
import javax.tools.*;
import javax.tools.JavaFileObject.*;

/**
 * @version 1.00 2007-10-28
 * @author Cay Horstmann
 */
public class CompilerTest
{
    public static void main(final String[] args) throws IOException,
    ↳ClassNotFoundException
    {
        JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();

        final List<ByteArrayJavaClass> classFileObjects = new ArrayList<>();

        DiagnosticCollector<JavaFileObject> diagnostics = new DiagnosticCollector<>();

        JavaFileManager fileManager = compiler.getStandardFileManager(diagnostics, null,
        ↳null);
        fileManager = new ForwardingJavaFileManager<JavaFileManager>(fileManager)
        {
            public JavaFileObject getJavaFileForOutput(Location location, final
            ↳String className,
```

```

        Kind kind, FileObject sibling) throws IOException
    {
        if (className.startsWith("x."))
        {
            ByteArrayJavaClass fileObject = new ByteArrayJavaClass(className);
            classFileObjects.add(fileObject);
            return fileObject;
        }
        else return super.getJavaFileForOutput(location, className, kind, sibling);
    }
};

String frameClassName = args.length == 0 ? "buttons2.ButtonFrame" : args[0];
JavaFileObject source = buildSource(frameClassName);
JavaCompiler.CompilationTask task = compiler.getTask(null, fileManager, diagnostics,
    null,
    null, Arrays.asList(source));
Boolean result = task.call();

for (Diagnostic<? extends JavaFileObject> d : diagnostics.getDiagnostics())
    System.out.println(d.getKind() + ": " + d.getMessage(null));
fileManager.close();
if (!result)
{
    System.out.println("Compilation failed.");
    System.exit(1);
}

EventQueue.invokeLater(new Runnable()
{
    public void run()
    {
        try
        {
            Map<String, byte[]> byteCodeMap = new HashMap<>();
            for (ByteArrayJavaClass cl : classFileObjects)
                byteCodeMap.put(cl.getName().substring(1), cl.getBytes());
            ClassLoader loader = new MapClassLoader(byteCodeMap);
            JFrame frame = (JFrame) loader.loadClass("x.Frame").newInstance();
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setTitle("CompilerTest");
            frame.setVisible(true);
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
});
});

/*
 * Tworzy kod źródłowy klasy pochodnej implementującej metodę addEventHandlers.
 * @return obiekt klasy StringBuilderJavaSource zawierający kod źródłowy
 */
static JavaFileObject buildSource(String superClassClassName)
    throws IOException, ClassNotFoundException

```

```

{
    StringBuilderJavaSource source = new StringBuilderJavaSource("x.Frame");
    source.append("package x;\n");
    source.append("public class Frame extends " + superClass + " {\n");
    source.append("protected void addEventHandlers() {\n");
    final Properties props = new Properties();

    props.load(Class.forName(superClass).getResourceAsStream("action.properties"));
    for (Map.Entry<Object, Object> e : props.entrySet())
    {
        String beanName = (String) e.getKey();
        String eventCode = (String) e.getValue();
        source.append(beanName + ".addActionListener(new java.awt.event.ActionListener()\n" +
        " {\n" +
        "     public void actionPerformed(java.awt.event.ActionEvent event)\n" +
        "     {\n" +
        "         " + eventCode + "\n" +
        "     }\n" +
        " }\n" );
        source.append("} } );\n");
    }
    source.append("} }");
    return source;
}
}

```

Dla każdej klasy pakietu  $x$  konstruujemy obiekt `ByteArrayJavaClass` za pomocą metody `getJavaFileForOutput` menedżera `ForwardingJavaFileManager`. Obiekty te przechwytyują pliki klas generowane podczas komplikacji klasy `x.Frame`. Wspomniana metoda umieszcza każdy taki obiekt na liście, co ułatwia nam późniejsze dotarcie do wygenerowanego kodu. Zwróćmy uwagę, że podczas komplikacji klasy `x.Frame` generowany jest plik głównej klasy i osobny plik klasy dla każdego obiektu nasłuchującego.

Po zakończeniu komplikacji tworzymy mapę wiązań nazw klas z tablicami kodu bajtowego. Prosta procedura ładowania klas (przedstawiona na listingu 10.8) ładuje klasy umieszczone na tej mapie.

#### **Listing 10.8.** compiler/MapClassLoader.java

```

package compiler;

import java.util.*;

/**
 * Klasa implementująca procedurę ładowania klas
 * umieszczonych na mapie. Kluczami mapy są nazwy klas,
 * a wartościami tablice kodu bajtowego.
 * @version 1.00 2007-11-02
 * @author Cay Horstmann
 */
public class MapClassLoader extends ClassLoader
{
    private Map<String, byte[]> classes;

    public MapClassLoader(Map<String, byte[]> classes)
    {
        this.classes = classes;
    }
}

```

```
protected Class<?> findClass(String name) throws ClassNotFoundException
{
    byte[] classBytes = classes.get(name);
    if (classBytes == null) throw new ClassNotFoundException(name);
    Class<?> cl = defineClass(name, classBytes, 0, classBytes.length);
    if (cl == null) throw new ClassNotFoundException(name);
    return cl;
}
```

---

Po załadowaniu skompilowanej klasy tworzymy i wyświetlamy ramkę tej klasy.

```
ClassLoader loader = new MapClassLoader(byteCodeMap);
Class<?> cl = loader.loadClass("x.Frame");
Frame frame = (JFrame) cl.newInstance();
frame.setVisible(true);
```

Klikając przyciski, możemy zmieniać tło. Aby przekonać się, że akcje zostały dynamicznie skompilowane, zmień jeden z wierszy pliku *action.properties*, na przykład w poniższy sposób:

```
yellowButton=panel.setBackground(java.awt.Color.YELLOW);
yellowButton.setEnabled(false);
```

Uruchom program ponownie, a przekonasz się, że kliknięcie przycisku *Yellow* spowoduje jego wyłączenie. Przyjrzyj się również katalogom zawierającym zwykle kod po komplikacji. Nie znajdziesz w nich żadnego pliku źródłowego ani pliku klas dla klas umieszczonej w pakiecie x. Przykład ten demonstruje zatem sposób dynamicznej komplikacji dla kodu źródłowego i plików klas umieszczonej w pamięci zamiast w plikach.

## 10.3. Stosowanie adnotacji

Adnotacje są znacznikami umieszczanymi w kodzie źródłowym w celu dalszego ich przetwarzania przez programy narzędziowe. Znaczniki te mogą być przetwarzane na poziomie kodu źródłowego lub kompilator może dołączyć je do plików klas.

Adnotacje nie zmieniają sposobu komplikacji programów. Kompilator generuje zawsze ten sam kod maszyny wirtualnej, niezależnie od tego, czy zastosowano adnotacje czy nie.

Aby korzystać z adnotacji, należy wybrać program narzędziowy do ich przetwarzania, wstawić w kodzie adnotacje zrozumiałe dla tego programu i następnie go wywołać.

Adnotacje posiadają szeroki zakres potencjalnych zastosowań, co początkowo może nieco utrudniać zrozumienie samej koncepcji. Do najważniejszych zastosowań należą:

- Automatyczne generowanie plików pomocniczych, takich jak deskryptory wdrożeń lub klasy informacyjne ziarnek.
- Automatyczne generowanie kodu do testowania, zapisu informacji w dziennikach, semantyki transakcji i tym podobnych.

W rozdziale zaczniemy omówienie adnotacji od podstawowych koncepcji, a następnie zastosujemy je w konkretnym przykładzie: za pomocą adnotacji oznaczymy metody jako procedury obsługi zdarzeń komponentów AWT i pokażemy procesor adnotacji, który przetworzy adnotacje i dołączy metody do odpowiednich komponentów. Następnie omówimy szczegółowo zasady składni adnotacji. Rozdział zakończymy dwoma zaawansowanymi przykładami przetwarzania adnotacji. Pierwszy z nich będzie przetwarzać adnotacje na poziomie kodu źródłowego. Drugi przykład będzie używać biblioteki Apache Bytecode Engineering Library do przetwarzania plików klas i „wstrzyknięcia” dodatkowego kodu bajtowego do metod oznaczonych adnotacjami.

Oto przykład prostej adnotacji:

```
public class MyClass
{
    ...
    @TestCase public void checkRandomInsertions()
}
```

Adnotacja `@TestCase` dotyczy metody `checkRandomInsertions`.

W języku Java adnotacja używana jest jak *modyfikator* i umieszczana przed elementem, którego dotyczy, *bez użycia znaku średnika* (przez modyfikator rozumiemy słowo kluczowe takie jak `public` czy `static`). Nazwa każdej adnotacji poprzedzana jest symbolem `@`, podobnie jak komentarze Javadoc. Jednak komentarze Javadoc umieszczone są wewnątrz ograniczników `/**...*/`, podczas gdy adnotacje stanowią część kodu.

Sama w sobie adnotacja `@TestCase` nic jeszcze nie znaczy. Aby była użyteczna, potrzebne jest właściwe narzędzie. Na przykład narzędzie testowania JUnit 4 (dostępne pod adresem <http://junit.org>) wywołuje podczas testowania klasy wszystkie metody opatrzone adnotacją `@Test`. Inne narzędzie może usunąć wszystkie metody testujące z pliku klasy, zanim zostanie on dostarczony do użytkownika.

Adnotacje mogą posiadać *elementy*, na przykład:

```
@Test(timeout="10000")
```

Elementy te mogą być przetwarzane przez narzędzia, które odczytują adnotacje. Możliwe są inne formy elementów. Omówimy je w dalszej części rozdziału.

Oprócz metod adnotacjami możemy oznaczać również klasy, pola i zmienne lokalne. Adnotację możemy umieścić wszędzie tam, gdzie modyfikator taki jak `public` lub `static`.

Dla każdej adnotacji musi zostać zdefiniowany *interfejs adnotacji*. Metody tego interfejsu odpowiadają elementom adnotacji. Na przykład adnotacja `Test` wykorzystywana przez narzędzie JUnit jest zdefiniowana za pomocą następującego interfejsu:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test
{
    Long timeout() default 0L;
    ...
}
```

Deklaracja @interface tworzy prawdziwy interfejs w języku Java. Narzędzia, które przetwarzają adnotacje, otrzymują obiekty implementujące interfejs adnotacji. Narzędzie takie wywoła metodę timeout, aby pobrać element timeout wybranej adnotacji Test.

Adnotacje Target i Retention są *metaadnotacjami*. Oznaczają one adnotację Test jako stosowaną wyłącznie do metod i zachowywaną podczas ładowania pliku klasy przez maszynę wirtualną. Metaadnotacje omówimy szczegółowo w punkcie 10.5.3.

Powyżej przedstawiliśmy podstawowe koncepcje związane z metadanymi w programach i adnotacjami. W następnym podrozdziale zajmiemy się konkretnym przykładem przetwarzania adnotacji.

### 10.3.1. Przykład: adnotacje obsługi zdarzeń

Jednym z bardziej żmudnych zadań stojących przed programistą tworzącym interfejs użytkownika jest połączenie obiektów nasłuchujących ze źródłami zdarzeń. Wiele obiektów nasłuchujących ma następującą postać:

```
myButton.addActionListener(new
    ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        doSomething();
    }
});
```

W niniejszym podrozdziale zaprojektujemy adnotacje, które pozwolą uniknąć ręcznej realizacji tego zadania. Adnotacja zdefiniowana na listingu 10.9 będzie stosowana w następujący sposób:

```
@ActionListenerFor(source="myButton") void doSomething() { . . . }
```

---

**Listing 10.9.** runtimeAnnotations/ActionListenerInstaller.java

```
package runtimeAnnotations;

import java.awt.event.*;
import java.lang.reflect.*;

/**
 * @version 1.00 2004-08-17
 * @author Cay Horstmann
 */
public class ActionListenerInstaller
{
    /**
     * Przetwarza wszystkie adnotacje ActionListenerFor danego obiektu.
     * @param obj obiekt, którego metody mogą posiadać adnotacje ActionListenerFor
     */
    public static void processAnnotations(Object obj)
    {
        try
        {
```

```

Class<?> cl = obj.getClass();
for (Method m : cl.getDeclaredMethods())
{
    ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
    if (a != null)
    {
        Field f = cl.getDeclaredField(a.source());
        f.setAccessible(true);
        addListener(f.get(obj), obj, m);
    }
}
catch (ReflectiveOperationException e)
{
    e.printStackTrace();
}

/**
 * Dodaje obiekt nasłuchujący wywołującą daną metodę.
 * @param source źródło zdarzeń, dla którego instalowany jest obiekt nasłuchujący
 * @param param niejawny parametr metody wywoływanej przez obiekt nasłuchujący
 * @param m metoda wywoływana przez obiekt nasłuchujący
 */
public static void addListener(Object source, final Object param, final Method m)
throws ReflectiveOperationException
{
    InvocationHandler handler = new InvocationHandler()
    {
        public Object invoke(Object proxy, Method mm, Object[] args) throws Throwable
        {
            return m.invoke(param);
        }
    };
    Object listener = Proxy.newProxyInstance(null,
        new Class[] { java.awt.event.ActionListener.class }, handler);
    Method adder = source.getClass().getMethod("addActionListener",
        ↳ActionListener.class);
    adder.invoke(source, listener);
}
}

```

Programista nie będzie już musiał wywoływać metody `addActionListener`, a jedynie oznaczy odpowiednie metody adnotacjami. Na listingu 10.10 przedstawiony został program `ButtonFrame` pochodzący z rozdziału 8. książki *Java 2. Podstawy*, ale zaimplementowany z użyciem naszej adnotacji.

#### **Listing 10.10. *buttons3/ButtonFrame.java***

```

package buttons3;

import java.awt.*;
import javax.swing.*;
import runtimeAnnotations.*;

```

```
/*
 * Ramka zawierająca panel z przyciskami
 * @version 1.00 2004-08-17
 * @author Cay Horstmann
 */
public class ButtonFrame extends JFrame
{
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;

    private JPanel panel;
    private JButton yellowButton;
    private JButton blueButton;
    private JButton redButton;

    public ButtonFrame()
    {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

        panel = new JPanel();
        add(panel);

        yellowButton = new JButton("Yellow");
        blueButton = new JButton("Blue");
        redButton = new JButton("Red");

        panel.add(yellowButton);
        panel.add(blueButton);
        panel.add(redButton);

        ActionListenerInstaller.processAnnotations(this);
    }

    @ActionListenerFor(source = "yellowButton")
    public void yellowBackground()
    {
        panel.setBackground(Color.YELLOW);
    }

    @ActionListenerFor(source = "blueButton")
    public void blueBackground()
    {
        panel.setBackground(Color.BLUE);
    }

    @ActionListenerFor(source = "redButton")
    public void redBackground()
    {
        panel.setBackground(Color.RED);
    }
}
```

---

Zdefiniowania wymaga również interfejs adnotacji. Jego kod przedstawiony został na listingu 10.11.

**Listing 10.11.** runtimeAnnotations/ActionListenerFor.java

---

```
package runtimeAnnotations;

import java.lang.annotation.*;

/**
 * @version 1.00 2004-08-17
 * @author Cay Horstmann
 */

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ActionListenerFor
{
    String source();
}
```

---

Oczywiście same adnotacje nic jeszcze nie znaczą. Zostają jedynie umieszczone w pliku źródłowym. Kompilator lokuje je w pliku klasy, który zostaje załadowany przez maszynę wirtualną. Potrzebny więc będzie mechanizm, który przeanalizuje adnotacje i zainstaluje obiekty nasłuchujące. Zadaniem tym zajmie się klasa `ActionListenerInstaller`. Konstruktor klasy `ButtonFrame` wywoła:

```
ActionListenerInstaller.processAnnotations(this);
```

Metoda statyczna `processAnnotations` tworzy wyliczenie wszystkich metod obiektu, który został jej przekazany. Dla każdej metody pobiera obiekt adnotacji `ActionListenerFor` i przetwarza go.

```
Class<?> cl = obj.getClass();
for (Method m : cl.getDeclaredMethods())
{
    ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
    if (a != null) . . .
}
```

Użyliśmy tutaj metody `getAnnotation` zdefiniowanej przez interfejs `AnnotatedElement`. Klasy `Method`, `Constructor`, `Field`, `Class` i `Package` implementują ten interfejs.

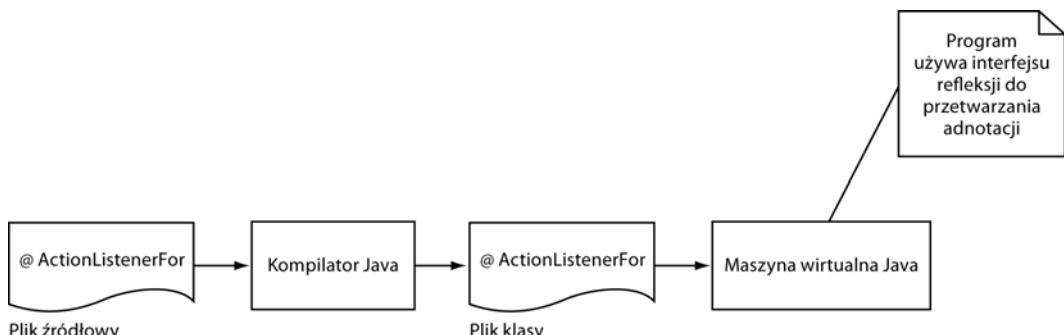
Nazwa pola źródłowego umieszczona jest w obiekcie adnotacji. Nazwę tę pobieramy, wywołując metodę `source`, a następnie wyszukujemy pasujące pole.

```
String fieldName = a.source();
Field f = cl.getDeclaredField(fieldName);
```

Przykład ten ilustruje ograniczenie naszej adnotacji. Element źródłowy musi być nazwą pola, a nie może być zmienną lokalną.

Pozostała część kodu jest dość oczywista. Dla każdej metody oznaczonej adnotacją tworzymy obiekt pośredni, który implementuje interfejs `ActionListener` i którego metoda `actionPerformed` wywołuje metodę oznaczoną adnotacją. (Więcej informacji na temat tworzenia obiektów pośrednich znajduje się w rozdziale 6. książki *Java 2. Podstawy*). Szczegóły nie są tutaj istotne. Podstawową obserwacją jest to, że funkcjonalność adnotacji została ustalona przez metodę `processAnnotations`.

Rysunek 10.1 przedstawia sposób obsługi adnotacji w naszym przykładzie.



**Rysunek 10.1.** Przetwarzanie adnotacji podczas działania programu

W tym przykładzie adnotacje zostały przetworzone podczas działania programu. Możliwe jest także przetwarzanie adnotacji na poziomie kodu źródłowego. Generator kodu źródłowego mógłby w naszym przykładzie utworzyć kod instalujący obiekty nasłuchujące. Alternatywnie adnotacje mogłyby zostać przetworzone na poziomie kodu bajtowego. Edytor kodu bajtowego mógłby wstrzyknąć wywołanie metody addActionListener do konstruktora ramki. Może wydawać się to skomplikowane, ale istnieją biblioteki, które znacznie ułatwiają to zadanie. Przykład pokażemy w podrozdziale 10.7., „Inżynieria kodu bajtowego”.

Nasz przykład nie jest oczywiście poważnym narzędziem dla programistów tworzących interfejs użytkownika. Metoda pomocnicza dodająca obiekt nasłuchujący byłaby w tym przypadku równie wygodna co adnotacja. (Klasa `java.beans.EventHandler` wykorzystuje takie rozwiązanie. Łatwo można ją zaadaptować, aby dostarczała metodę, która dodaje obiekt obsługi zdarzeń, zamiast tylko go tworzyć).

Nasz przykład ilustruje zastosowanie mechanizmu adnotacji w programach i sposób ich analizy. Mamy nadzieję, że konkretny przykład lepiej przygotuje Czytelnika do lektury kolejnych podrozdziałów omawiających w szczegółach składnię adnotacji.

#### API `java.lang.AnnotatedElement 5.0`

- `boolean isAnnotationPresent(Class<? extends Annotation> annotationType)`  
zwraca wartość true, jeśli element posiada adnotację podanego typu.
- `<T extends Annotation> T getAnnotation(Class<T> annotationType)`  
zwraca adnotację podanego typu lub wartość null, gdy element nie posiada takiej adnotacji.
- `Annotation[] getAnnotations()`  
zwraca wszystkie adnotacje danego elementu, w tym dziedziczone. Jeśli element nie posiada żadnej adnotacji, to metoda zwraca tablicę o długości 0.
- `Annotation[] getDeclaredAnnotations()`  
zwraca wszystkie adnotacje zadeklarowane dla danego elementu z wyłączeniem adnotacji dziedziczonych. Jeśli element nie posiada żadnej adnotacji, to metoda zwraca tablicę o długości 0.

## 10.4. Składnia adnotacji

W niniejszym podrozdziale omówimy wyczerpująco składnię adnotacji.

Adnotacja zdefiniowana jest przez interfejs adnotacji:

```
modyfikatory @interface NazwaAdnotacji
{
    deklaracja_elementu1
    deklaracja_elementu2
    .
}
}
```

Każda deklaracja elementu posiada postać:

*typ nazwaElementu();*

lub:

*typ nazwaElementu() default wartość;*

Na przykład poniższa adnotacja ma dwa elementy, assignedTo i severity.

```
public @interface BugReport
{
    String assignedTo() default "[none]";
    int severity() = 0;
}
```

Każda adnotacja posiada format:

*@NazwaAdnotacji(nazwaElementu1=wartość1, nazwaElementu2=wartość2, . . .)*

na przykład:

*@BugReport(assignedTo="Harry", severity=10)*

Kolejność elementów nie ma znaczenia. Adnotacja:

*@BugReport(severity=10, assignedTo="Harry")*

jest identyczna z poprzednią.

Domyślna wartość podana w deklaracji jest używana, jeśli wartość elementu nie została podana.  
Na przykład dla adnotacji:

*@BugReport(severity=10)*

wartością elementu assignedTo będzie łańcuch "[none]".



Wartości domyślne nie są przechowywane wraz z adnotacją, lecz są wyznaczane dynamicznie. Na przykład, jeśli zmienimy wartość domyślną elementu assignedTo na łańcuch "[]" i skompilujemy ponownie interfejs BugReport, to adnotacja *@BugReport(severity=10)* użyje nowej wartości domyślnej nawet w tych plikach klas, które zostały skompilowane przed zmianą wartości domyślnej.

Zapis adnotacji można uprościć, stosując dwa skróty.

Jeśli dla adnotacji nie podajemy żadnych elementów, ponieważ adnotacja nie posiada elementów albo wszystkie elementy używają wartości domyślnych, to zapisując taką adnotację, nie musimy używać nawiasów. Na przykład zapis:

```
@BugReport
```

jest równoważny zapisowi:

```
@BugReport(assignedTo="[none]", severity=0)
```

Skróconą postać adnotacji nazywamy *adnotacją znacznikową*.

Drugim ze wspomnianych skrótów jest *adnotacja pojedynczej wartości*. Jeśli element adnotacji posiada specjalną nazwę `value` i nie podajemy żadnego innego elementu, to możemy pominąć nazwę elementu i znak `=`. Na przykład, gdyby zdefiniować interfejs adnotacji `ActionListenerFor` z poprzedniego podrozdziału w następujący sposób:

```
public @interface ActionListenerFor
{
    String value();
}
```

to adnotację można wtedy zapisać jako:

```
@ActionListenerFor("yellowButton")
```

zamiast:

```
@ActionListenerFor(value="yellowButton")
```

Wszystkie interfejsy adnotacji stanowią rozszerzenie interfejsu `java.lang.annotation.Annotation`. Interfejs ten jest zwykłym interfejsem języka Java, a *nie* interfejsem adnotacji. Na końcu podrozdziału zamieszczony został opis metod tego interfejsu.

Interfejsy adnotacji nie mogą być rozszerzane. Innymi słowy, wszystkie interfejsy adnotacji stanowią bezpośrednie rozszerzenie interfejsu `java.lang.annotation.Annotation`.

Nigdy nie tworzymy klas implementujących interfejsy adnotacji. Maszyna wirtualna sama generuje klasy i obiekty pośrednie, gdy jest to konieczne. Na przykład dla adnotacji `ActionListenerFor` maszyna wirtualna wykonuje operację, której schemat przedstawiony został poniżej:

```
return Proxy.newProxyInstance(classLoader, ActionListenerFor.class,
new
    InvocationHandler()
{
    public Object invoke(Object proxy, Method m, Object[] args) throws
        Throwable
    {
        if (m.getName().equals("source")) return wartość adnotacji source;
        ...
    }
});
```

Deklaracje elementów w interfejsie adnotacji są w istocie deklaracjami metod. Metody należące do interfejsu adnotacji nie mogą posiadać parametrów i klauzuli throws, a także nie mogą być generyczne.

Typem elementu adnotacji może być:

- typ podstawowy (int, short, long, byte, char, double, float lub boolean),
- String,
- Class (z opcjonalnym parametrem typu, na przykład Class<? extends MyClass>),
- typ enum,
- typ adnotacji,
- tablica wymienionych wyżej typów.

A oto przykład zawierający poprawne deklaracje elementów:

```
public @interface BugReport
{
    enum Status { UNCONFIRMED, CONFIRMED, FIXED, NOTABUG };
    boolean showStopper() default false;
    String assignedTo() default "[none]";
    Class<?> testCase() default Void.class;
    Status status() default Status.UNCONFIRMED;
    Reference ref() default @Reference(); //typ adnotacji
    String[] reportedBy();
}
```

Ponieważ adnotacje przetwarzane są przez kompilator, to wszystkie wartości elementów muszą być stałymi znanimi w momencie komplikacji, na przykład:

```
@BugReport(showStopper=true, assignedTo="Harry", testCase=MyTestCase.class,
    ↳status=BugReport.Status.CONFIRMED, . . .)
```



Element adnotacji nigdy nie może przyjmować wartości null. Niedopuszczalna jest nawet wartość domyślna null. W praktyce jest to dość niewygodne i wymaga stosowania innych wartości domyślnych takich jak "" czy Void.class.

Jeśli wartość elementu jest tablicą, to jej wartości ujmujemy w nawiasy klamrowe:

```
@BugReport(. . . reportedBy={"Harry", "Carl"})
```

Nawiasy te możemy pominać, gdy element posiada jedną wartość:

```
@BugReport(. . . reportedBy="Joe") // równoważny zapis {"Joe"}
```

Ponieważ element adnotacji może być inną adnotacją, to możemy tworzyć dowolnie złożone adnotacje, na przykład:

```
@BugReport(ref=@Reference(id="3352627"), . . .)
```



Błądem jest wprowadzanie cyklicznych zależności adnotacji. Na przykład, jeśli adnotacja BugReport posiada element będący typu adnotacji Reference, to adnotacja Reference nie powinna mieć elementu będącego typu adnotacji BugReport.

Adnotacje możemy dodawać do:

- pakietów,
- klas (w tym enum),
- interfejsów (w tym interfejsów adnotacji),
- metod,
- konstruktorów,
- pól instancji (w tym stałych enum),
- zmiennych lokalnych,
- zmiennych parametrów.

Adnotacje zmiennych lokalnych mogą być jednak przetwarzane tylko na poziomie kodu źródłowego. Pliki klas nie opisują zmiennych lokalnych i dlatego podczas komplikacji adnotacje zmiennych lokalnych są tracone. Z podobnych powodów również adnotacje pakietów nie są zachowywane poza kodem źródłowym.



Adnotację pakietu umieszczamy w pliku *package-info.java*, który zawiera tylko instrukcję package poprzedzoną adnotacją.

Ten sam element może posiadać wiele adnotacji pod warunkiem, że należą one do różnych typów. Dla tego samego elementu możemy więc użyć adnotacji danego typu co najwyżej raz. Na przykład komplikacja poniższego fragmentu kodu zakończy się błędem:

```
@BugReport(showStopper=true, reportedBy="Joe")
@BugReport(reportedBy={"Harry", "Carl"})
void myMethod()
```

Jeśli jest to problemem, to możemy zaprojektować adnotację, której wartością jest tablica prostszych adnotacji:

```
@BugReports({
    @BugReport(showStopper=true, reportedBy="Joe"),
    BugReport(reportedBy={"Harry", "Carl"})
})
void myMethod()
```

#### API *java.lang.annotation.Annotation 5.0*

- `Class<? extends Annotation> annotationType()`  
zwraca obiekt `Class` reprezentujący interfejs adnotacji dla danego obiektu adnotacji.  
Zauważmy, że wywołanie metody `getClass` dla obiektu adnotacji zwróci rzeczywistą klasę obiektu, a nie interfejs.
- `boolean equals(Object other)`  
zwraca wartość `true`, gdy `other` jest obiektem implementującym ten sam interfejs adnotacji co dany obiekt adnotacji i gdy ich elementy są równe.

- `int hashCode()`  
zwraca skrót zgodny z metodą `equals`, wyznaczany na podstawie nazwy interfejsu adnotacji i wartości elementów.
- `String toString()`  
zwraca łańcuch zawierający nazwę interfejsu adnotacji i wartości elementów, na przykład `@BugReport(assignedTo=[none], severity=0)`.

## 10.5. Adnotacje standardowe

Java SE definiuje szereg interfejsów adnotacji w pakietach `java.lang`, `java.lang.annotation` i `javax.annotation`. Cztery z nich są metaadnotacjami, które opisują zachowanie się interfejsów adnotacji. Pozostałe są typowymi adnotacjami, których możemy używać w kodzie źródłowym tworzonych programów. Zostały one przedstawione w tabeli 10.2. Omówimy je szczegółowo w kolejnych dwóch podrozdziałach.

**Tabela 10.2.** Adnotacje standardowe

Interfejs adnotacji	Adnotacja stosowana dla	Znaczenie
<code>Deprecated</code>	Dowolnego elementu	Oznacza element, który nie jest zalecany.
<code>SuppressWarnings</code>	Dowolnego elementu z wyjątkiem pakietów i adnotacji	Wyłącza ostrzeżenia podanego typu.
<code>Override</code>	Metod	Sprawdza, czy dana metoda przesłania metodę klasy bazowej.
<code>PostConstruct</code> <code>PreDestroy</code>	Metod	Oznaczona metoda powinna zostać wywołana natychmiast po utworzeniu obiektu lub przed jego usunięciem.
<code>Resource</code>	Klas, interfejsów, metod, pól	Dla klasy lub interfejsu: oznacza jako zasób używany w innym miejscu. Dla metody lub pola: oznacza do „wstrzygnięcia”.
<code>Resources</code>	Klas, interfejsów	Tablica zasobów.
<code>Generated</code>	Dowolnego elementu	Oznacza, że kod źródłowy został wygenerowany przez narzędzie.
<code>Target</code>	Adnotacji	Określa elementy, dla których można zastosować adnotację.
<code>Retention</code>	Adnotacji	Określa, jak długo zachowywana jest adnotacja.
<code>Documented</code>	Adnotacji	Określa, czy adnotacja powinna zostać uwzględniona w dokumentacji elementu.
<code>Inherited</code>	Adnotacji	Jeśli adnotacja zostanie zastosowana dla klasy, to zostanie automatycznie odziedziczona przez klasy pochodne.

## 10.5.1. Adnotacje kompilacji

Adnotacji @Deprecated używamy dla elementów, których użycie nie jest już zalecane. Kompilator generuje ostrzeżenia w przypadku napotkania takich elementów. Adnotacja ta spełnia tę samą rolę co znacznik @deprecated w Javadoc.

Adnotacja @SuppressWarnings informuje kompilator, aby przestał generować ostrzeżenia podanego typu, na przykład:

```
@SuppressWarnings("unchecked")
```

Adnotacja @Override może być stosowana tylko dla metod. Kompilator sprawdza wtedy, czy metoda oznaczona tą adnotacją rzeczywiście przesyła metodę klasy bazowej. Jeśli zadeklarujemy na przykład klasę:

```
public MyClass
{
    @Override public boolean equals(MyClass other);
    ...
}
```

wystąpi błąd kompilacji. Metoda equals zadeklarowana w klasie MyClass *nie* przesyła bowiem metody equals klasy Object, która posiada parametr klasy Object, a nie MyClass.

Adnotacja @Generated została przewidziana dla narzędzi generowania kodu. Jej zastosowanie pozwala odróżnić kod źródłowy wygenerowany przez narzędzie od kodu napisanego przez programistę. Dzięki temu na przykład edytor kodu może ukrywać kod wygenerowany, a generator kodu usuwać starsze wersje wygenerowanego kodu. Każda taka adnotacja musi zawierać unikalny identyfikator generatora kodu. Opcjonalnym elementem jest łańcuch określający datę (w formacie ISO8601) i łańcuch komentarza. Na przykład:

```
@Generated("com.horstmann.beanproperty", "2008-01-04T12:08:56.235-0700");
```

## 10.5.2. Adnotacje zarządzania zasobami

Adnotacje @PostConstruct i @PreDestroy są używane przez środowiska, które kontrolują cykl życia obiektów, na przykład kontenery WWW czy serwery aplikacji. Metody oznaczone tymi adnotacjami powinny zostać wywołane natychmiast po stworzeniu obiektu lub przed jego usunięciem.

Adnotację @Resource wprowadzono z myślą o wstrzykiwaniu zasobów. Rozważmy na przykład aplikację WWW, która korzysta z bazy danych. Oczywiście informacje związane z dostępem do bazy nie powinny być umieszczone w kodzie takiej aplikacji. Zamiast tego kontener WWW będzie używać interfejsu użytkownika w celu określenia parametrów połączenia oraz nazwy JNDI dla źródła danych. Aplikacja WWW będzie wtedy odwoływać się do źródła danych w poniższy sposób:

```
@Resource(name="jdbc/mydb")
private DataSource source;
```

Gdy tworzony jest obiekt zawierający pole source, kontener „wstrzykuje” referencję źródła danych.

### 10.5.3. Metaadnotacje

Metaadnotacja `@Target` ogranicza elementy, dla których może zostać użyta adnotacja, na przykład:

```
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface BugReport
```

W tabeli 10.3 przedstawione zostały wszystkie możliwe wartości. Należą one do typu wyliczeniowego `ElementType`. W nawiasach klamrowych możemy podać dowolną liczbę typów elementów.

**Tabela 10.3.** Typy elementów używane przez metaadnotację `@Target`

Typ elementu	Adnotacja stosowana dla
ANNOTATION_TYPE	Deklaracji typu adnotacji
PACKAGE	Pakietów
TYPE	Klas (w tym <code>enum</code> ) i interfejsów (w tym typów adnotacji)
METHOD	Metod
CONSTRUCTOR	Konstruktorów
FIELD	Pól (w tym stałych <code>enum</code> )
PARAMETER	Parametrów metod lub konstruktorów
LOCAL_VARIABLE	Zmiennych lokalnych

Adnotacja bez ograniczenia `@Target` może być stosowana dla dowolnego elementu. Kompilator sprawdza, czy adnotacje stosowane są dla dozwolonych elementów. Jeśli zastosujemy adnotację `@BugReport` dla pola, to wystąpi błąd komilacji.

Metaadnotacja `@Retention` określa sposób zachowywania adnotacji, który określamy za pomocą co najmniej jednej wartości spośród podanych w tabeli 10.4. Domyślana wartość to `RetentionPolicy.CLASS`.

**Tabela 10.4.** Sposoby zachowywania adnotacji określone za pomocą metaadnotacji `@Retention`

Sposób	Opis
SOURCE	Adnotacje nie są zachowywane w plikach klas.
CLASS	Adnotacje są zachowywane w plikach klas, ale nie są ładowane przez maszynę wirtualną.
RUNTIME	Adnotacje są zachowywane w plikach klas i są ładowane przez maszynę wirtualną. Są wtedy dostępne za pomocą interfejsu programowego refleksji.

Na listingu 10.11 dla adnotacji `@ActionListenerFor` zadeklarowaliśmy `RetentionPolicy.RUNTIME`, ponieważ do przetwarzania adnotacji użyliśmy refleksji. W kolejnych dwóch podrozdziałach pokażemy przykłady przetwarzania adnotacji na poziomie kodu źródłowego oraz na poziomie plików klas.

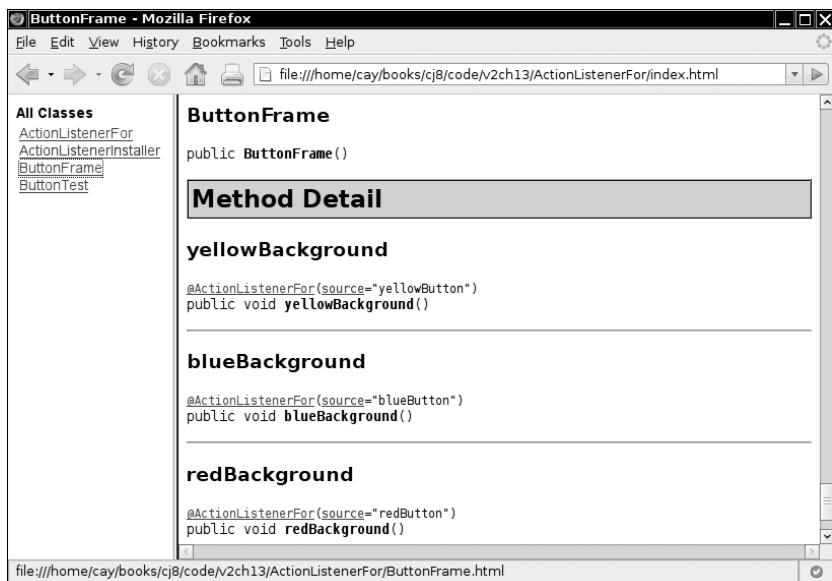
Metaadnotacja `@Documented` stanowi wskazówkę dla narzędzi dokumentujących takich jak Javadoc. Adnotacje, dla których zastosowano tę metaadnotację, powinny być traktowane z punktu widzenia dokumentacji kodu tak jak inne modyfikatory, takie jak `protected` lub `static`. Użycie pozostałych adnotacji nie jest odnotowywane w dokumentacji. Założymy, że zadeklarowaliśmy adnotację `@ActionListenerFor` w następujący sposób:

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ActionListenerFor
```

W takim przypadku dokumentacja każdej metody oznaczonej adnotacją zawiera tę adnotację, co pokazane zostało na rysunku 10.2.

Rysunek 10.2.

Adnotacje  
w dokumentacji



Jeśli adnotacja ma charakter tymczasowy (na przykład `@BugReport`), to zwykle jej zastosowanie nie wymaga udokumentowania.



Dozwolone jest zastosowanie adnotacji do niej samej. Na przykład adnotacja `@Documented` sama jest oznaczona jako `@Documented`. Dzięki temu dokumentacja Javadoc adnotacji może informować o tym, czy dana adnotacja uwzględniana jest w dokumentacji.

Metaadnotacja `@Inherited` stosowana jest tylko dla adnotacji klas. Jeśli klasa posiada adnotację zdeklarowaną jako `@Inherited`, to adnotację tę posiadać będą automatycznie jej wszystkie klasy pochodne. Możliwość ta ułatwia tworzenie adnotacji, które działają w taki sam sposób jak interfejsy znacznikowe takie jak na przykład `Serializable`.

Adnotacja `Serializable` byłaby nawet bardziej wskazana niż interfejs znacznikowy `Serializable` nieposiadający metod. Klasa jest serializowalna, ponieważ istnieje odpowiedni mechanizm zapisu i odczytu jej pól, a nie ze względu na zastosowanie jakichkolwiek zasad

projektowania obiektowego. Fakt ten jest lepiej reprezentowany przez adnotację niż dziedziczenie interfejsu. Oczywiście interfejs Serializable wprowadzono już w JDK 1.1, na długo zanim pojawiły się adnotacje.

Przypuśćmy, że definiujemy dziedziczoną adnotację @Persistent wskazującą, że obiekt pewnej klasy może zostać zapisany w bazie danych. Wtedy wszystkie klasy pochodne tej klasy zostaną automatycznie oznaczone jako trwałe.

```
@Inherited @Persistent {}
@Persistent class Employee { . . . }
class Manager extends Employee { . . . } // również @Persistent
```

Gdy mechanizm trwałości będzie wyszukiwał obiekty do zapisania w bazie danych, to uwzględnii zarówno obiekty klasy Employee, jak i Manager.

## 10.6. Przetwarzanie adnotacji w kodzie źródłowym

Jednym z zastosowań adnotacji jest automatyczne generowanie plików zawierających dodatkowe informacje o programach. W przeszłości edycja Enterprise Edition platformy Java przyprawiała programistów o ból głowy ilością dodatkowego kodu. Obecnie Java EE 5 używa adnotacji w celu znakomitego uproszczenia modelu programowania.

W niniejszym podrozdziale przedstawimy tę technikę na prostszym przykładzie. Napiszemy program, który będzie automatycznie tworzył klasę informacyjną ziarnka. Właściwości ziarnka oznaczymy adnotacją i uruchomimy program narzędziowy, który będzie parsował plik źródłowy, analizował adnotacje i tworzył plik źródłowy klasy informacyjnej ziarnka.

Przypomnijmy sobie z rozdziału 8., że klasa informacyjna ziarnka udostępnia precyzyjniejszy opis ziarnka niż może go uzyskać proces automatycznej introspekcji. Klasa informacyjna ziarnka zawiera listę wszystkich właściwości ziarnka. Właściwości mogą posiadać opcjonalne edytory. Typowym przykładem klasy informacyjnej ziarnka jest klasa ChartBeanBean → Info przedstawiona w rozdziale 8.

Aby wyeliminować ręczne tworzenie klas informacyjnych ziarnek, stworzymy adnotację @Property. Możemy użyć jej do oznaczenia metod set lub get właściwości, na przykład:

```
@Property String getTitle() { return title; }
```

lub:

```
@Property(editor="TitlePositionEditor")
public void setTitlePosition(int p) { titlePosition = p; }
```

Na listingu 10.12 przedstawiona została definicja adnotacji @Property. Zwróćmy uwagę, że zachowywana jest ona tylko na poziomie kodu źródłowego (RetentionPolicy.SOURCE). Nie jest więc umieszczana w plikach klas i tym samym nie jest dostępna dla mechanizmu refleksji.

### **Listing 10.12. sourceAnnotations/Property.java**

```
package sourceAnnotations;
import java.lang.annotation.*;
```

```

@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Property
{
    String editor() default "";
}

```



Większy sens miałoby zadeklarowanie elementu editor jako elementu typu Class. Jednak procesor adnotacji nie przetwarza adnotacji typu Class, ponieważ znaczenie klasy może zależeć od szeregu czynników zewnętrznych (takich jak ścieżka dostępu do klas czy procedura ładowająca klasy), dlatego też do określenia nazwy klasy edytora użyliśmy typu String.

Aby automatycznie wygenerować klasę informacyjną ziarnka dla klasy o nazwie *BeanClass*, musimy wykonać następujące zadania:

- 1.** Zapisać plik źródłowy *BeanClassBeanInfo.java*. Zadeklarować klasę *BeanClassBeanInfo* jako klasę pochodną klasy *SimpleBeanInfo* i przesłonić jej metodę *getPropertyDescriptor*.
- 2.** Dla każdej metody oznaczonej adnotacją uzyskać nazwę właściwości poprzez usunięcie przedrostka get lub set i zamianę dużych liter na małe w pozostałym fragmencie.
- 3.** Dla każdej właściwości zapisać instrukcję tworzącą *PropertyDescriptor*.
- 4.** Jeśli właściwość posiada edytor, zapisać wywołanie metody *setPropertyEditorClass*.
- 5.** Zapisać kod zwracający tablicę wszystkich deskryptorów właściwości.

Na przykład adnotacja:

```

@Property(editor="TitlePositionEditor")
public void setTitlePosition(int p) { titlePosition = p; }

```

umieszczona w klasie *ChartBean* zostanie przetłumaczona na:

```

public class ChartBeanBeanInfo extends java.beans.SimpleBeanInfo
{
    public java.beans.PropertyDescriptor[] getProperties()
    {
        java.beans.PropertyDescriptor titlePositionDescriptor
            = new java.beans.PropertyDescriptor("titlePosition", ChartBean.class);
        titlePositionDescriptor.setPropertyEditorClass(TitlePositionEditor.class);
        ...
        return new java.beans.PropertyDescriptor[]
        {
            titlePositionDescriptor,
            ...
        }
    }
}

```

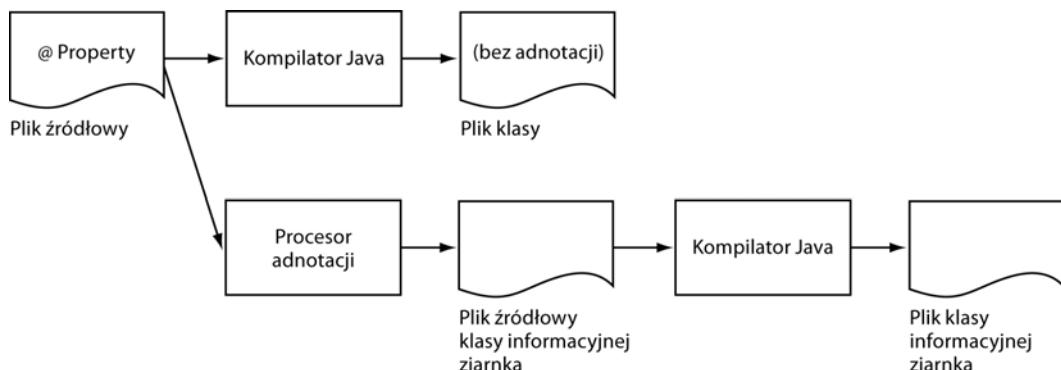
(Kod szablonu został pogrubiony.)

Zadanie to jest łatwe do wykonania, jeśli potrafimy odnaleźć wszystkie metody oznaczone adnotacją @Property.

W wersji Java SE 6 do kompilatora możemy dodawać *procesory adnotacji*. (W Java SE 5 w tym samym celu stosowany był osobny program o nazwie *apt*). Aby uruchomić przetwarzanie adnotacji, wywołujemy

```
javac -processor NazwaKlasyProcesora1, NazwaKlasyProcesora2, ... pliki źródłowe
```

Kompilator wyszuka adnotacje w plikach źródłowych. Następnie użyje odpowiednich procesorów adnotacji. Procesory te wykonywane są po kolej. Jeśli procesor adnotacji utworzy nowy plik źródłowy, to proces zostaje powtórzony. Dopiero gdy kolejna runda przetwarzania nie utworzy żadnych nowych plików źródłowych, nastąpi kompilacja wszystkich plików źródłowych. Rysunek 10.3 ilustruje sposób przetwarzania adnotacji @Property.



**Rysunek 10.3.** Przetwarzanie adnotacji w kodzie źródłowym

Nie będziemy szczegółowo omawiać interfejsu przetwarzania adnotacji, lecz damy obraz jego możliwości na przykładzie programu przedstawionego na listingu 10.13.

**Listing 10.13.** sourceAnnotations/BeanInfoAnnotationProcessor.java

```

package sourceAnnotations;

import java.beans.*;
import java.io.*;
import java.util.*;
import javax.annotation.processing.*;
import javax.lang.model.*;
import javax.lang.model.element.*;
import javax.tools.*;
import javax.tools.Diagnostic.*;

/**
 * Klasa procesora analizującego adnotacje Property.
 * @version 1.11 2012-01-26
 * @author Cay Horstmann
 */
@SupportedAnnotationTypes("sourceAnnotations.Property")
@SupportedSourceVersion(SourceVersion.RELEASE_7)
public class BeanInfoAnnotationProcessor extends AbstractProcessor

```

```

    {
        @Override
        public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv)
        {
            for (TypeElement t : annotations)
            {
                Map<String, Property> props = new LinkedHashMap<>();
                String beanClassName = null;
                for (Element e : roundEnv.getElementsAnnotatedWith(t))
                {
                    String mname = e.getSimpleName().toString();
                    String[] prefixes = { "get", "set", "is" };
                    boolean found = false;
                    for (int i = 0; !found && i < prefixes.length; i++)
                        if (mname.startsWith(prefixes[i]))
                    {
                        found = true;
                        int start = prefixes[i].length();
                        String name = Introspector.decapitalize(mname.substring(start));
                        props.put(name, e.getAnnotation(Property.class));
                    }

                    if (!found) processingEnv.getMessager().printMessage(Kind.ERROR,
                            "@Property must be applied to getXxx, setXxx, or isXxx method", e);
                    else if (beanClassName == null)
                        beanClassName = ((TypeElement) e.getEnclosingElement()).getQualifiedName()
                                .toString();
                }
            try
            {
                if (beanClassName != null) writeBeanInfoFile(beanClassName, props);
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }
        }
        return true;
    }

    /**
     * Zapisuje plik źródłowy klasy BeanInfo.
     * @param beanClassName nazwa klasy ziarnka
     * @param props mapa nazw właściwości i ich adnotacji
     */
    private void writeBeanInfoFile(String beanClassName, Map<String, Property> props)
        throws IOException
    {
        JavaFileObject sourceFile = processingEnv.getFiler().createSourceFile(
            beanClassName + "BeanInfo");
        PrintWriter out = new PrintWriter(sourceFile.openWriter());
        int i = beanClassName.lastIndexOf(".");
        if (i > 0)
        {
            out.print("package ");
            out.print(beanClassName.substring(0, i));
            out.println(";");
        }
    }
}

```

```

out.print("public class ");
out.print(beanClassName.substring(i + 1));
out.println("BeanInfo extends java.beans.SimpleBeanInfo");
out.println("{");
out.println("    public java.beans.PropertyDescriptor[] getPropertyDescriptors()");
out.println("    {");
out.println("        try");
out.println("        {");
for (Map.Entry<String, Property> e : props.entrySet())
{
    out.print("            java.beans.PropertyDescriptor ");
    out.print(e.getKey());
    out.println("Descriptor");
    out.print("            = new java.beans.PropertyDescriptor(\"");
    out.print(e.getKey());
    out.print("\", \"");
    out.print(beanClassName);
    out.println(".class');");
    String ed = e.getValue().editor().toString();
    if (!ed.equals(""))
    {
        out.print("            ");
        out.print(e.getKey());
        out.print("Descriptor.setPropertyEditorClass(");
        out.print(ed);
        out.println(".class);");
    }
}
out.println("        return new java.beans.PropertyDescriptor[]");
out.print("    }");
boolean first = true;
for (String p : props.keySet())
{
    if (first) first = false;
    else out.print(",");
    out.println();
    out.print("        ");
    out.print(p);
    out.print("Descriptor");
}
out.println();
out.println("    }");
out.println("}");
out.println("    }");
out.println("    catch (java.beans.IntrospectionException e)");
out.println("    {");
out.println("        e.printStackTrace();");
out.println("        return null;");
out.println("    }");
out.println("}");
out.println("}");
out.close();
}
}

```

Procesor adnotacji implementuje interfejs Processor, zwykle rozszerzając klasę AbstractProcessor. Należy przy tym określić, które adnotacje obsługuje nasz procesor. W tym celu stosujemy również adnotacje:

```
@SupportedAnnotationTypes("com.horstmann.annotations.Property")
public class BeanInfoAnnotationProcessor extends AbstractProcessor
```

Możemy określić wybrane typy adnotacji, używając przy tym znaku gwiazdki, na przykład "com.horstmann.\*" (wszystkie adnotacje pakietu com.horstmann i dowolnego pakietu pod-rzędnego), czy nawet "\*" (wszystkie adnotacje).

Klasa BeanInfoAnnotationProcessor posiada jedną metodę publiczną process, która wywoływana jest dla każdego pliku. Metoda ta posiada dwa parametry, zbiór adnotacji przetwarzanych w danej rundzie oraz referencję obiektu RoundEnv zawierającego informacje o bieżącej rundzie przetwarzania.

Metoda process przegląda metody opatrzone adnotacją. Dla każdej metody ustala nazwę właściwości, usuwając przedrostek get, set lub is, i zmienia następną literę nazwy na małą. Poniżej przedstawiamy schemat działania metody.

```
public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment
    roundEnv)
{
    for (TypeElement t : annotations)
    {
        Map<String, Property> props = new LinkedHashMap<String, Property>();
        for (Element e : roundEnv.getElementsAnnotatedWith(t))
        {
            props.put(property name, e.getAnnotation(Property.class));
        }
    }
    zapisuje plik źródłowy klasy informacyjnej ziarnka
    return true;
}
```

Metoda process powinna zwrócić wartość true, jeśli obsłużyła wszystkie przedstawione jej adnotacje. Innymi słowy, gdy adnotacje te nie wymagają obsługi przez pozostałe procesory.

Kod zapisujący plik źródłowy jest sekwencją instrukcji out.print. Zwróćmy uwagę, że plik wyjściowy tworzymy w następujący sposób:

```
JavaFileObject sourceFile = processingEnv.getFiler().createSourceFile(beanClassName
    + "BeanInfo");
PrintWriter out = new PrintWriter(sourceFile.openWriter());
```

Klasa AbstractProcessor posiada pole processingEnv zdefiniowane jako protected i umożliwiające dostęp do różnych usług przetwarzania. Interfejs Filer jest odpowiedzialny za tworzenie nowych plików i umożliwia śledzenie ich powstawania, ponieważ mogą wymagać przetworzenia w kolejnej rundzie.

Gdy procesor adnotacji wykrywa błąd, to używa obiektu Messager do komunikacji z użytkownikiem. Używamy go, na przykład, do przekazania komunikatu o błędzie, gdy nazwa metody oznaczonej adnotacją @Property nie zaczyna się od przedrostka get, set lub is:

```
if (!found) processingEnv.getMessager().printMessage(Kind.ERROR,
    "@Property must be applied to getXxx, setXxx, or isXxx method", e);
```

W przykładach kodu dla tej książki umieściliśmy plik *ChartBean.java* zawierający adnotacje. Najpierw jednak skompilujemy procesor adnotacji:

```
javac sourceAnnotations/BeanInfoAnnotationProcessor.java
```

Następnie uruchomimy

```
javac -processor sourceAnnotations.BeanInfoAnnotationProcessor chart/ChartBean.java
i sprawdzimy, że został automatycznie wygenerowany plik ChartBeanBeanInfo.java.
```

Aby uzyskać więcej informacji o przetwarzaniu adnotacji, możemy dodać opcję `XprintRounds`, wywołując kompilator. Uzyskamy następujące informacje:

```
Round 1:
    input files: {com.horstmann.corejava.ChartBean}
    annotations: [com.horstmann.annotations.Property]
    last round: false
Round 2:
    input files: {com.horstmann.corejava.ChartBeanBeanInfo}
    annotations: []
    last round: false
Round 3:
    input files: {}
    annotations: []
    last round: true
```

Przykład ten ilustruje wykorzystanie adnotacji umieszczonych w kodzie źródłowym do generowania innych plików. Pliki te nie muszą być plikami źródłowymi w języku Java. Procesory adnotacji mogą generować deskryptory XML, pliki właściwości, skrypty powłoki, dokumentację w formacie HTML i tak dalej.



Wydawać się może, że adnotacje mogą również umożliwić automatyczne generowanie nie trywialnych metod `set` i `get`. Na przykład adnotacja:

```
@Property private String title;
```

mogłaby spowodować wygenerowanie metod:

```
public String getTitle() { return title; }
public void setTitle(String title) { this.title = title; }
```

jednak metody te muszą zostać dodane do *tej samej klasy*. Operacja taka wymaga edytowania pliku źródłowego, a nie wygenerowania kolejnego pliku. Przekracza to możliwości programu `apt`. Możliwe byłoby stworzenie specjalizowanego narzędzia, ale prawdopodobnie wykraczałoby ono poza zastosowania adnotacji. Adnotacje pomyślano bowiem jako sposób opisu kodu, a nie dyrektywy służące jego edycji.

## 10.7. Inżynieria kodu bajtowego

Pokazaliśmy dotąd sposób, w jaki adnotacje mogą być przetwarzane podczas działania programu lub na poziomie kodu źródłowego. Trzecia możliwość polega na przetwarzaniu adnotacji na poziomie kodu bajtowego. Jeśli adnotacje nie zostaną usunięte z kodu źródłowego, to są obecne w plikach klas. Format plików klas jest udokumentowany (patrz <http://docs.oracle.com/javase/specs/jvms/se7/html>). Format ten jest dość skomplikowany i jego przetwarzanie bez odpowiednich bibliotek nie jest łatwe. Jedną z takich bibliotek jest `BCEL` (*Bytecode Engineering Library*), dostępna na stronie <http://jakarta.apache.org/bcel>.

W niniejszym podrozdziale użyjemy biblioteki BCEL, aby dodać komunikaty dziennika do kodu metod oznaczonych adnotacjami. Jeśli metoda będzie oznaczona adnotacją:

```
@LogEntry(logger="loggerName")
```

to na początku metody wstawimy kody bajtowe odpowiadające instrukcji:

```
Logger.getLogger(loggerName).entering(nazwaKlasy, nazwaMetody);
```

Na przykład, jeśli oznaczymy metodę hashCode należącą do klasy Item jako:

```
@LogEntry(logger="global") public int hashCode()
```

to za każdym razem, gdy wywołana zostanie ta metoda, wyświetlany będzie komunikat podobny do poniższego:

```
Aug 17, 2004 9:32:59 PM Item hashCode
FINER: ENTRY
```

Aby osiągnąć taki efekt, musimy:

- 1.** załadować kod bajtowy z pliku klasy,
- 2.** zlokalizować wszystkie metody,
- 3.** dla każdej metody sprawdzić, czy jest oznaczona za pomocą adnotacji LogEntry,
- 4.** jeśli tak, to na początku metody wstawić kod bajtowy odpowiadający poniższym instrukcjom:

```
ldc nazwaDziennika
invokestatic java/util/logging/Logger.getLogger:(Ljava/lang/String;)
    ↳Ljava/util/logging/Logger;
ldc nazwaKlasy
ldc nazwaMetody
invokevirtual java/util/logging/Logger.entering:(Ljava/lang/String;Ljava/
    ↳lang/String;)V
```

Wstawienie tych kodów może wydawać się kłopotliwe, ale dzięki zastosowaniu biblioteki BCEL staje się łatwe. Nie będziemy tutaj szczegółowo omawiać procesu analizy i wstawiania kodu bajtowego. Najważniejsze jest to, że program przedstawiony na listingu 10.14 edytuje plik klasy i wstawia odpowiednie wywołanie na początku metody oznaczonej adnotacją LogEntry.

---

**Listing 10.14. bytecodeAnnotations/EntryLogger.java**

---

```
package bytecodeAnnotations;

import java.io.*;
import org.apache.bcel.*;
import org.apache.bcel.classfile.*;
import org.apache.bcel.generic.*;

/**
 * Dodaje kod zapisu komunikatów dziennika
 * do wszystkich metod oznaczonych adnotacją LogEntry.
 * @version 1.10 2007-10-27
 * @author Cay Horstmann
 */
```

```

public class EntryLogger
{
    private ClassGen cg;
    private ConstantPoolGen cpg;

    /**
     * Dodaje kod zapisu komunikatów dziennika do podanej klasy
     * @param args nazwa pliku klasy
     */
    public static void main(String[] args)
    {
        try
        {
            if (args.length == 0) System.out.println("USAGE: java
                -bytecodeAnnotations.EntryLogger classname");
            else
            {
                JavaClass jc = Repository.lookupClass(args[0]);
                ClassGen cg = new ClassGen(jc);
                EntryLogger el = new EntryLogger(cg);
                el.convert();
                String f = Repository.lookupClassFile(cg.getClassName()).getPath();
                System.out.println("Dumping " + f);
                cg.getJavaClass().dump(f);
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }

    /**
     * Tworzy obiekt EntryLogger, który dodaje kod zapisu komunikatów dziennika
     * do wszystkich metod oznaczonych adnotacją należących do podanej klasy
     * @param cg klasa
     */
    public EntryLogger(ClassGen cg)
    {
        this.cg = cg;
        cpg = cg.getConstantPool();
    }

    /**
     * Przekształca klasę, dodając wywołania zapisu do dziennika.
     */
    public void convert() throws IOException
    {
        for (Method m : cg.getMethods())
        {
            AnnotationEntry[] annotations = m.getAnnotationEntries();
            for (AnnotationEntry a : annotations)
            {
                if (a.getAnnotationType().equals("LbytecodeAnnotations/LogEntry;"))
                {
                    for (ElementValuePair p : a.getElementValuePairs())
                    {
                        if (p.getNameString().equals("logger"))

```

```

        {
            String loggerName = p.getValue().stringifyValue();
            cg.replaceMethod(m, insertLogEntry(m, loggerName));
        }
    }
}
}

/**
 * Wstawia kod na początku metody.
 * @param m metod
 * @param loggerName nazwa wywoływanego obiektu zapisu do dziennika
 */
private Method insertLogEntry(Method m, String loggerName)
{
    MethodGen mg = new MethodGen(m, cg.getClassName(), cpg);
    String className = cg.getClassName();
    String methodName = mg.getMethod().getName();
    System.out.printf("Adding logging instructions to %s.%s%n", className, methodName);

    int getLoggerIndex = cpg.addMethodref("java.util.logging.Logger", "getLogger",
        "(Ljava/lang/String;)Ljava/util/logging/Logger;");
    int enteringIndex = cpg.addMethodref("java.util.logging.Logger", "entering",
        "(Ljava/lang/String;Ljava/lang/String;)V");

    InstructionList il = mg.getInstructionList();
    InstructionList patch = new InstructionList();
    patch.append(new PUSH(cpg, loggerName));
    patch.append(new INVOKESTATIC(getLoggerIndex));
    patch.append(new PUSH(cpg, className));
    patch.append(new PUSH(cpg, methodName));
    patch.append(new INVOKEVIRTUAL(enteringIndex));
    InstructionHandle[] ihs = il.getInstructionHandles();
    il.insert(ihs[0], patch);

    mg.setMaxStack();
    return mg.getMethod();
}
}

```



Czytelnikom zainteresowanym inżynierią kodu bajtowego polecamy lekturę podręcznika biblioteki BCEL dostępnego pod adresem <http://jakarta.apache.org/bcel/manual.html>.

Do skompilowania i uruchomienia programu EntryLogger potrzebna jest biblioteka BCEL w wersji 6.0 lub nowszej. (W chwili oddawania tego rozdziału do druku wersja ta była jeszcze tworzona. Jeśli dotąd nie została ukończona, sprawdź zawartość katalogu trunk w repozytorium Subversion).

Poniżej pokazujemy sposób dodania instrukcji zapisu komunikatów dziennika do pliku *Item.java* przedstawionego na listingu 10.15.

```
javac Item.java
javac -classpath .:bcel-numerWersji.jar EntryLogger.java
java -classpath .:bcel-numerWersji.jar EntryLogger Item
```

**Listing 10.15.** set/Item.java

---

```
package set;

import java.util.*;
import bytecodeAnnotations.*;

/**
 * Produkt z opisem i numerem.
 * @version 1.01 2012-01-26
 * @author Cay Horstmann
 */
public class Item
{
    private String description;
    private int partNumber;

    /**
     * Konstruktor.
     * @param aDescription opis produktu
     * @param aPartNumber numer produktu
     */
    public Item(String aDescription, int aPartNumber)
    {
        description = aDescription;
        partNumber = aPartNumber;
    }

    /**
     * Zwraca opis produktu.
     * @return opis
     */
    public String getDescription()
    {
        return description;
    }

    public String toString()
    {
        return "[descripion=" + description + ", partNumber=" + partNumber + "]";
    }

    @LogEntry(logger = "global")
    public boolean equals(Object otherObject)
    {
        if (this == otherObject) return true;
        if (otherObject == null) return false;
        if (getClass() != otherObject.getClass()) return false;
        Item other = (Item) otherObject;
        return Objects.equals(description, other.description) && partNumber ==
            other.partNumber;
    }

    @LogEntry(logger = "global")
```

```
public int hashCode()
{
    return Objects.hash(description, partNumber);
}
```

---

Przed i po zmodyfikowaniu pliku klasy Item warto wywołać:

```
javap -c Item
```

Dzięki temu możemy zauważyc instrukcje wstawione na początku metod hashCode, equals i compareTo.

```
public int hashCode();
Code:
 0: ldc      #85; //String global
 2: invokestatic #80; // Method java/util/logging/
   ↳Logger.getLogger:(Ljava/lang/String;)Ljava/util/logging/Logger;
 5: ldc      #86; //String Item
 7: ldc      #88; //String hashCode
 9: invokevirtual #84; // Method java/util/logging/
   ↳Logger.entering:(Ljava/lang/String; Ljava/lang/String;)V
12: bipush13
14: aload_0
15: getfield      #2; //Field description:Ljava/lang/String;
18: invokevirtual #15; // Method java/lang/String.hashCode:()I
21: imul
22: bipush 17
24: aload_0
25: getfield      #3; //Field partNumber:I
28: imul
29: iadd
30: ireturn
```

Program SetTest przedstawiony na listingu 10.16 wstawia obiekty Item do zbioru. Jeśli uruchomimy go dla zmodyfikowanego pliku klasy, to pojawią się komunikaty dziennika.

```
Aug 18, 2004 10:57:59AM Item hashCode
FINER: ENTRY
Aug 18, 2004 10:57:59AM Item hashCode
FINER: ENTRY
Aug 18, 2004 10:57:59AM Item hashCode
FINER: ENTRY
Aug 18, 2004 10:57:59AM Item equals
FINER: ENTRY
[[description=Toaster, partNumber=1729], [description=Microwave, partNumber=4104]]
```

---

**Listing 10.16. set/SetTest.java**

---

```
package set;

import java.util.*;
import java.util.logging.*;

/**
 * @version 1.02 2012-01-26
 * @author Cay Horstmann
 */
```

```

public class SetTest
{
    public static void main(String[] args)
    {
        Logger.getLogger(Logger.GLOBAL_LOGGER_NAME).setLevel(Level.FINEST);
        Handler handler = new ConsoleHandler();
        handler.setLevel(Level.FINEST);
        Logger.getLogger(Logger.GLOBAL_LOGGER_NAME).addHandler(handler);

        Set<Item> parts = new HashSet<>();
        parts.add(new Item("Toaster", 1279));
        parts.add(new Item("Microwave", 4104));
        parts.add(new Item("Toaster", 1279));
        System.out.println(parts);
    }
}

```

Zauważmy, że na skutek dwukrotnego wstawienia tego samego elementu wywołana została metoda `equals`.

Przykład ten ilustruje duże możliwości inżynierii kodu bajtowego. Adnotacje zostały wykorzystane do wstawienia dyrektyw w programie. Na ich podstawie narzędzie edycji kodu bajtowego modyfikuje następnie instrukcje maszyny wirtualnej.

## 10.7.1. Modyfikacja kodu bajtowego podczas ładowania

W poprzednim podrozdziale przedstawiliśmy narzędzie umożliwiające modyfikacje plików klas, jednak dodanie jeszcze jednego narzędzia do procesu tworzenia programu może być czasami mało wygodne. Atrakcyjną alternatywą będzie w tym wypadku odroczenie inżynierii kodu bajtowego do momentu *załadowania kodu klasy*.

Przed pojawiением się Java SE 5.0 rozwiązywanie takie wymagało implementacji własnej procedury ładowającej klasy. Obecnie interfejs `Instrumentation` umożliwia połączenie własnego obiektu przekształcającego kod bajtowy. Obiekt ten musi zostać zainstalowany, zanim wywołana będzie metoda `main` programu. Wymaganie to możemy spełnić, definiując *agenta* czyli bibliotekę ładowaną w celu monitorowania programu. Kod agenta może przeprowadzić odpowiednią inicjalizację wewnętrz metody `premain`.

W celu stworzenia agenta należy wykonać następujące kroki:

- 1** Zaimplementować klasę posiadającą metodę:

```
public static void premain(String arg, Instrumentation instr)
```

Metoda ta zostaje wywołana, gdy ładowany jest agent. Agentowi można podać w wierszu poleceń pojedynczy argument, który przekazywany jest za pomocą parametru `arg`. Parametr `instr` można wykorzystać do instalacji różnych punktów zaczepienia.

- 2** Utworzyć plik manifestu, który konfiguruje atrybut `Premain-Class`, na przykład:

```
Premain-Class: bytecodeAnnotations.EntryLoggingAgent
```

**3.** Umieścić kod agenta i plik manifestu w pliku JAR, na przykład:

```
javac -classpath .:bcel-numerWersji.jar bytecodeAnnotations.EntryLoggingAgent
jar cvfm EntryLoggingAgent.jar EntryLoggingAgent.mf
bytecodeAnnotations/Entry*.class
```

Aby uruchomić program razem z agentem, należy użyć następujących opcji:

```
java -javaagent:PlikJARAgenta=argumentAgenta . . .
```

Na przykład, aby uruchomić program SetTest razem z agentem zapisującym komunikaty dziennika, wywołamy:

```
javac SetTest.java
java -javaagent:EntryLoggingAgent.jar=set.Item -classpath .:bcel-numerWersji.jar
set.SetTest
```

Argument Item jest nazwą klasy, którą powinien zmodyfikować agent.

Na listingu 10.17 przedstawiony został kod agenta. Agent instaluje obiekt przekształcający plik klasy. Obiekt ten sprawdza najpierw, czy nazwa klasy zgadza się z argumentem agenta. Jeśli tak, to używa klasy EntryLogger przedstawionej w poprzednim podrozdziale do modyfikacji kodu bajtowego. Jednak modyfikacje te nie są zapisywane w pliku, lecz wykonywane podczas ładowania kodu przez maszynę wirtualną (patrz rysunek 10.4). Innymi słowy technika ta wykonuje modyfikację kodu bajtowego „w locie”.

**Listing 10.17.** bytecodeAnnotations/EntryLoggingAgent.java

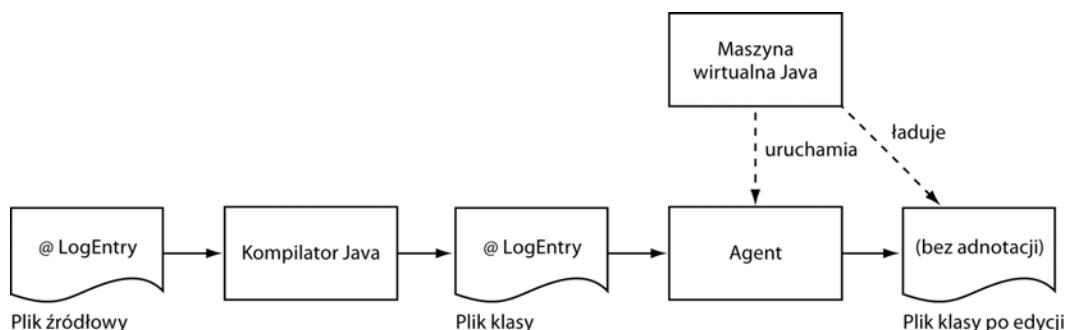
```
package bytecodeAnnotations;

import java.lang.instrument.*;
import java.io.*;
import java.security.*;
import org.apache.bcel.classfile.*;
import org.apache.bcel.generic.*;

/**
 * @version 1.00 2004-08-17
 * @author Cay Horstmann
 */
public class EntryLoggingAgent
{
    public static void premain(final String arg, Instrumentation instr)
    {
        instr.addTransformer(new ClassFileTransformer()
        {
            public byte[] transform(ClassLoader loader, String className, Class<?> cl,
                                   ProtectionDomain pd, byte[] data)
            {
                if (!className.equals(arg)) return null;
                try
                {
                    ClassParser parser = new ClassParser(new ByteArrayInputStream(data),
                                                       className
                                                       + ".java");
                    JavaClass jc = parser.parse();
                    ClassGen cg = new ClassGen(jc);
                    cg.addAnnotation("Ljavassist/instrument/Transformer;" +
                                    ".transform(Ljava/lang/Object;Ljava/lang/String;L" +
                                    "java/lang/Class;L" +
                                    "java/lang/ProtectionDomain;[B)L[B;");

                    byte[] transformedData = cg.toBytecode();
                    return transformedData;
                }
                catch (Exception e)
                {
                    System.out.println("Error transforming class " + className);
                    e.printStackTrace();
                }
            }
        });
    }
}
```

```
        EntryLogger el = new EntryLogger(cg);
        el.convert();
        return cg.getJavaClass().getBytes();
    }
    catch (Exception e)
    {
        e.printStackTrace();
        return null;
    }
}
});
```



**Rysunek 10.4.** Modyfikacja klas podczas ładowania

W rozdziale omówiliśmy sposoby:

- umieszczania adnotacji w programach Java,
  - projektowania własnych interfejsów adnotacji,
  - implementacji narzędzi, które wykorzystują adnotacje.

Omówiliśmy w tym rozdziale trzy technologie związane z przetwarzaniem kodu: skrypty, komplikację programów Java i przetwarzanie adnotacji. Wykorzystanie pierwszych dwóch jest stosunkowo proste. Natomiast tworzenie narzędzi przetwarzających adnotacje jest zadaniem na tyle bardziej skomplikowanym, że nie musi się go podejmować każdy programista. Przykłady narzędzi pokazane w tym rozdziale mają dać Czytelnikowi ogólne pojęcie na temat możliwości wykorzystania adnotacji, a może nawet zachęcić do próby stworzenia własnych narzędzi.

W kolejnym, ostatnim już rozdziale książki zajmiemy się interfejsem programowym pozwalającym używać w programach Java metod macierzystych zaimplementowanych w języku C/C++.



# 11

## Obiekty rozproszone

W tym rozdziale:

- Role klienta i serwera.
- Wywołania zdalnych metod.
- Model programowania RMI.
- Parametry metod zdalnych i wartości zwracane.
- Aktywacja zdalnych obiektów.

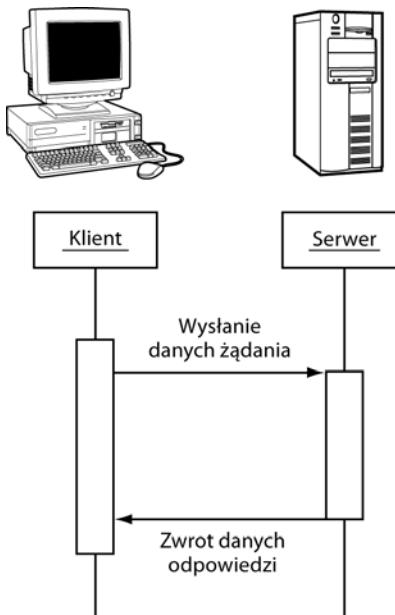
Co pewien czas społeczność programistów powraca do idei wszechobecnych obiektów jako cudownego środka, który okaże się panaceum na wszystkie problemy. Idea ta opiera się na wykorzystaniu współpracujących ze sobą obiektów, które istnieć mogą w dowolnych miejscach. Obiekty te komunikują się ze sobą za pośrednictwem sieci i zestawu standardowych protokołów. Na przykład istnieć może obiekt klienta, który umożliwia użytkownikowi sformułowanie żądania uzyskania pewnych danych. Obiekt ten przesyła następnie obiektowi istniejącemu na serwerze komunikat zawierający szczegóły żądania. Obiekt serwera uzyskuje dane, komunikując się z bazą danych lub innymi obiektami, po czym odsyła je klientowi. Koncepcyjnie zatem cały proces zdaje się wyglądać zupełnie prosto, ale aby efektywnie używać rozproszonych obiektów należy poznać sposób ich działania.

W bieżącym rozdziale skoncentrujemy się na wykorzystaniu protokołu *RMI (Remote Method Invocation)* do komunikacji między dwoma maszynami wirtualnymi Java (które mogą działać na różnych komputerach). Kiedyś RMI uważano za technologię przydatną programistom aplikacji, ale obecnie jest interesująca jedynie jako przykład prostego systemu rozproszonych obiektów.

## 11.1. Role klienta i serwera

Podstawowa idea programowania rozproszonego jest prosta. Klient wysyła żądanie poprzez sieć do serwera. Serwer przetwarza odebrane żądanie i odsyła klientowi odpowiedź. Proces ten pokazano na rysunku 11.1.

**Rysunek 11.1.**  
Przesyłanie obiektów między klientem i serwerem



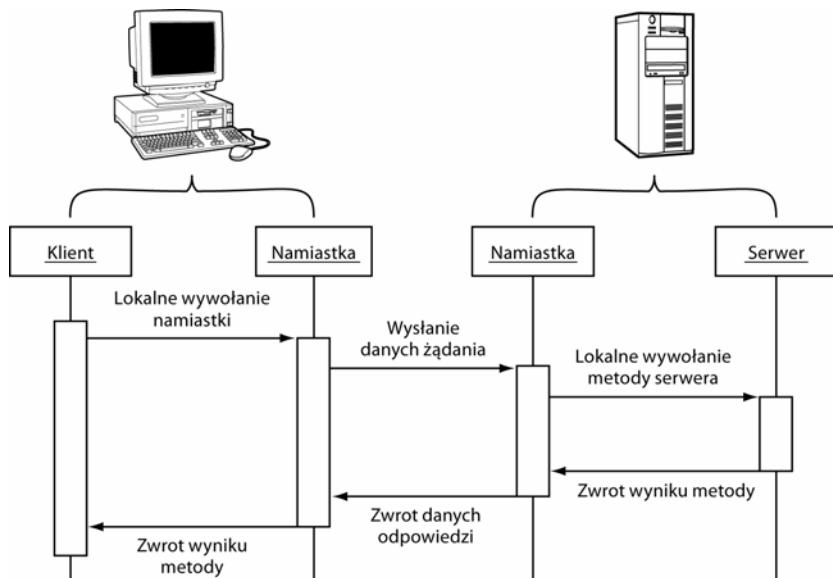
Na wstępie chcielibyśmy od razu zaznaczyć, że wspomniane żądania i odpowiedzi są czym innym niż żądania i odpowiedzi, których używają aplikacje WWW. Klientem w naszym przypadku nie jest przeglądarka internetowa, lecz dowolna aplikacja wykonująca dowolnie skomplikowaną logikę biznesową. Aplikacja klienta nie musi wcale wchodzić w interakcję z użytkownikiem. Jeśli jednak interakcje te mają miejsce, to mogą równie dobrze odbywać się za pośrednictwem wiersza poleceń, jak i interfejsu graficznego Swing. Protokół przekazywania żądań i odpowiedzi pozwala na przesyłanie dowolnych obiektów, podczas gdy tradycyjne aplikacje WWW ograniczają się do protokołu HTTP w przypadku żądań i wysyłania odpowiedzi w formacie HTML.

Interesować nas będzie zatem mechanizm, który umożliwia programistom tworzącemu klienta zwykłe wywołanie metody zamiast zajmowania się szczegółami przesyłania danych w sieci i parsowania odpowiedzi. Rozwiążanie polega na zainstalowaniu w kliencie obiektu stanowiącego *namiastkę* obiektu serwera. Namiastka jest obiektem maszyny wirtualnej klienta zachowującym się jak zdalny obiekt. Klient używa namiastki, wykonując zwykłe wywołania metod. Namiastka kontaktuje się z serwerem, używając odpowiedniego protokołu sieciowego.

Podobnie programista tworzący obiekt serwera nie musi zajmować się kwestiami komunikacji z klientem. Także i w tym przypadku rozwiązanie polega na zainstalowaniu kolejnej namiastki, tym razem na serwerze. Namiastka klienta komunikuje się z namiastką serwera i wywołuje zwykłe metody obiektu implementującego usługę (patrz rysunek 11.2).

**Rysunek 11.2.**

Zdalne wywołanie metody przy użyciu namiastek



W jaki sposób namiastki komunikują się między sobą? Zależy to od konkretnej technologii implementacji. Spotykane są trzy rozwiązania:

- CORBA (*Common Object Request Broker Architecture*) — umożliwia wywoływanie metod obiektów zaimplementowanych w dowolnym języku programowania. CORBA używa protokołu IIOP (Internet Inter-ORB) do komunikacji pomiędzy obiektami.
- Architektura usług sieciowych stanowiąca kolekcję różnych protokołów, czasami określanych wspólną nazwą WS-\*. Nie jest ona związana z konkretnym językiem programowania, ale używa do komunikacji formatu opartego na XML-u. Formatem przesyłania obiektów jest SOAP (*Simple Object Access Protocol*).
- RMI (*Remote Method Invocation*) — umożliwia wywoływanie metod rozproszonych obiektów Java.

Technologie CORBA i SOAP są zupełnie niezależne od języka programowania. Programy klienta i serwera mogą być implementowane w C, C++, C#, Javie lub dowolnym innym języku programowania. Dla każdego obiektu należy dostarczyć opisu interfejsu określającego sygnatury metod i typy danych. Opis ten powstaje w specjalnym języku IDL (*Interface Definition Language*) w przypadku CORBA lub WSDL (*Web Services Description Language*) w przypadku usług sieciowych.

Wielu programistów uważało przez ostatnie kilka lat, że CORBA jest rozwiązaniem z przeszłością. Jednak z czasem CORBA zyskała niezbyt dobrą reputację wynikającą, czasami zasłużenie, ze skomplikowanej implementacji i problemów ze współpracą różnych implementacji i osiągnięta umiarkowany sukces.

Pojawieniu się usług sieciowych towarzyszył początkowo podobny entuzjazm, ponieważ miały być rozwiązaniem dużo prostszym oraz dodatkowo korzystającym z technologii WWW i XML, które same zdążyły już osiągnąć sukces. Z czasem jednak rozwiązanie to również stało się skomplikowane na skutek wyposażenia w większość możliwości, które oferuje CORBA. Jego

zaletą jest wykorzystanie formatu XML, co ułatwia analizę i wyszukiwanie błędów. Jednak równocześnie konieczność przetwarzania języka XML stanowi wąskie gardło z punktu widzenia efektywności. Obecnie rozwiązania WS-\* nie są już darzone takim entuzjazmem, a ich reputacja pogarsza się ze względu na coraz bardziej skomplikowaną implementację i problemy ze zgodnością.

Jeśli komunikujące się programy zostały zaimplementowane w języku Java, to zalety specyfikacji CORBA lub WS-\* przestają być istotne. Firma Sun rozwinięła dużo prostszy mechanizm wywoływania zdalnych metod RMI (*Remote Method Invocation*), przeznaczony do komunikacji pomiędzy aplikacjami platformy Java.

Warto poznać mechanizm RMI, nawet gdy nie planuje się jego wykorzystania we własnych programach. Umożliwia to poznanie podstawowych mechanizmów programowania rozproszonych aplikacji na przykładzie stosunkowo prostej architektury.

## 11.2. Wywołania zdalnych metod

Kluczową koncepcją przetwarzania rozproszonego jest *wywoływanie zdalnych metod*. Pewien kod (zwany dalej *klientem*) wywołuje metodę obiektu znajdującego się na innym komputerze (*obiektu zdalnego*). Oczywiście, aby było to możliwe, istnieć musi odpowiedni mechanizm, który zapewni przesłanie parametrów metody, jej odpowiednie wywołanie i przesłanie rezultatu działania z powrotem.

Zanim przejdziemy do szczegółowego omówienia tego procesu, chcielibyśmy zaznaczyć, że terminologia klient-serwer odnosi się wyłącznie do pojedynczego wywołania metody. Tylko z punktu widzenia danego wywołania metody zdalnej komputer, który ją wywołuje, jest klientem, a komputer, w którym znajduje się obiekt wywoływanej metody — serwerem. Ta sytuacja może ulec zmianie z każdym kolejnym wywołaniem metody zdalnej. Dotychczasowy serwer stanie się klientem, jeśli tylko wywoła zdalną metodę obiektu znajdującego się na innej maszynie.

### 11.2.1. Namiastka i szeregowanie parametrów

Jeśli kod klienta zamierza wywołać zdalną metodę zdalnego obiektu, to wywołuje w rzeczywistości zwykłą metodę języka Java obiektu zastępczego zwanego *namiastką*.

```
Warehouse centralWarehouse = pobranie obiektu namiastki;  
double price = centralWarehouse.getPrice("Blackwell Toaster");
```

Namiastka znajduje się zawsze na kliencie, a nie na serwerze. Potrafi ona skontaktować się z serwerem poprzez sieć i przesyła mu parametry wywołania zdalnej metody w postaci bloku bajtów. Każdy z parametrów reprezentowany jest za pomocą kodu niezależnego od rodzaju maszyny. Proces kodowania parametrów nazywa się *szeregowaniem parametrów*. Jego zadaniem jest przekształcenie parametrów na format odpowiedni do ich przesłania między maszynami wirtualnymi Java. W przypadku protokołu RMI obiekty są serializowane przy użyciu mechanizmu serializacji omówionego w rozdziale 1. Natomiast protokół SOAP używa w tym celu języka XML.

Podsumujmy: metoda namiastki tworzy na maszynie klienta blok informacji składający się z:

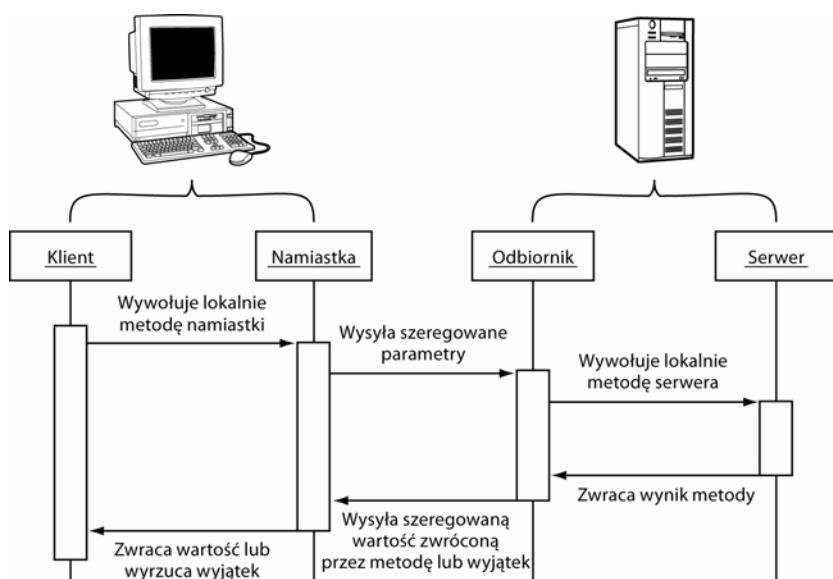
- identyfikatora zdalnego obiektu,
- opisu metody, która ma być wywołana,
- szeregowanych parametrów.

Namiastka wysyła tę informację do serwera. Po stronie serwera obiekt odbiorcy wykonuje dla każdego zdalnego wywołania metody następujące działania.

1. Lokalizuje zdalny obiekt, którego metodę należy wywołać.
2. Wywołuje tę metodę przekazując jej dostarczone parametry.
3. Pobiera jej wynik lub wyjątek.
4. Wysyła informację będącą szeregowanym wynikiem działania metody do namiastki na maszynie klienta.

Namiastka rozszerowuje wynik lub wyjątek otrzymany z serwera. Wartość tą zwróci następnie metoda namiastki. Jeśli zdalna metoda wyrzuciła wyjątek, to namiastka także go wyrzuca. Rysunek 11.3 obrazuje przepływ informacji w procesie wywołania metody zdalnej.

**Rysunek 11.3.**  
Szeregowanie  
parametrów



Cały ten dość skomplikowany proces jest wykonywany automatycznie i zwykle w sposób niewidoczny dla programisty.

Szczegóły implementacji zdalnych obiektów i pozyskiwania namiastek zależą od technologii zastosowanej dla rozproszonych obiektów. W kolejnych podrozdziałach przyjrzymy się bliżej implementacji RMI.

## 11.3. Model programowania RMI

Omawiając model programowania RMI, posłużymy się bardzo prostym przykładem. Zdalny obiekt będzie reprezentował magazyn, a klient będzie mógł wysłać zapytanie o cenę wybranego produktu. W kolejnych podrozdziałach pokażemy sposób implementacji i uruchomienia programów serwera i klienta.

### 11.3.1. Interfejsy i implementacje

Klient musi wiedzieć, jakie operacje są dozwolone na zdalnych obiektach rezydujących na serwerze. Możliwości te wyraża się za pomocą interfejsów współdzielonych przez klienta i serwer. Na przykład interfejs przedstawiony na listingu 11.1 opisuje usługę dostarczaną przez zdalny obiekt reprezentujący magazyn:

**Listing 11.1.** warehouse1/Warehouse.java

```
import java.rmi.*;  
  
/**  
 * Zdalny interfejs prostego magazynu.  
 * @version 1.0 2007-10-09  
 * @author Cay Horstmann  
 */  
public interface Warehouse extends Remote  
{  
    double getPrice(String description) throws RemoteException;  
}
```



W tym rozdziale nie stosujemy pakietów. Tworzenie zdalnych aplikacji samo w sobie jest wystarczająco skomplikowane i dlatego postanowiliśmy uniknąć dodatkowych komplikacji związanych z katalogami pakietów.

Interfejsy zdalnych obiektów muszą zawsze rozszerzać interfejs `Remote` zdefiniowany w pakiecie `java.rmi`. Wszystkie metody tych interfejsów muszą także deklarować możliwość wyrzucenia wyjątku `RemoteException`. Powodem takiej deklaracji jest mniejsza niezawodność zdalnych wywołań metod — zawsze istnieje możliwość, że mogą one zawieść. Na przykład serwer jest chwilowo niedostępny lub pojawia się problem z komunikacją w sieci. Klient musi być przygotowany na wystąpienie takich sytuacji i dlatego w przypadku wywołań zdalnych metod musimy zawsze obsługiwać wyjątek `RemoteException`.

Po stronie serwera musimy zaimplementować klasę, która wykonuje metody wyspecyfikowane przez interfejs (patrz listing 11.2).

**Listing 11.2.** warehouse1/WarehouseImpl.java

```
import java.rmi.*;  
import java.rmi.server.*;  
import java.util.*;
```

```

/**
 * Klasa implementacji zdalnego interfejsu Warehouse.
 * @version 1.01 2012-01-26
 * @author Cay Horstmann
 */
public class WarehouseImpl extends UnicastRemoteObject implements Warehouse
{
    private Map<String, Double> prices;

    public WarehouseImpl() throws RemoteException
    {
        prices = new HashMap<>();
        prices.put("Blackwell Toaster", 24.95);
        prices.put("ZapXpress Microwave Oven", 49.95);
    }

    public double getPrice(String description) throws RemoteException
    {
        Double price = prices.get(description);
        return price == null ? 0 : price;
    }
}

```



Konstruktor klasy `WarehouseImpl` deklaruje możliwość wyrzucenia wyjątku `RemoteException`, ponieważ klasa bazowa wyrzuca ten wyjątek, gdy nie może połączyć się z usługą sieciową odpowiedzialną za nadzorowanie tworzenia obiektów serwera.

Klasa spełnia funkcję serwera zdalnych metod, jeśli stanowi rozszerzenie klasy `UnicastRemoteObject`. Konstruktor tej klasy umożliwia zdalny dostęp do obiektów. Dziedziczenie po klasie `UnicastRemoteObject` jest zatem najłatwiejszym sposobem implementacji usług sieciowych i wykorzystamy je we wszystkich przykładach zamieszczonych w tym rozdziale.

W niektórych przypadkach możemy nie chcieć tworzyć klasy serwera, która stanowi rozszerzenie klasy `UnicastRemoteObject`, ponieważ nasza klasa dziedziczy już po innej klasie. W takiej sytuacji zmuszeni jesteśmy do „ręcznego” tworzenia obiektów serwera i przekazywania ich metodzie statycznej `exportObject`. Zamiast tworzyć klasę pochodną klasy `UnicastRemoteObject`, wywołamy:

```
UnicastRemoteObject.exportObject(this, 0);
```

wewnętrz konstruktora zdalnego obiektu. Drugi z parametrów metody `exportObject` posiada wartość 0, co oznacza, że do nasłuchiwanego połączeń od klientów może być wykorzystywany dowolny port.



Termin „unicast” odnosi się do faktu, że zdalny obiekt zostaje zlokalizowany na podstawie wywołania skierowanego pod pojedynczy adres IP i numer portu. Tylko taki mechanizm lokalizacji zdalnych obiektów jest obsługiwany na platformie Java SE. Bardziej zaawansowane systemy rozproszonych obiektów (na przykład JINI) umożliwiają wyszukiwanie określone terminem „multicast”. W takim przypadku zdalne obiekty mogą znajdować się na wielu różnych serwerach.

## 11.3.2. Rejestr RMI

Aby skorzystać z usług zdalnego obiektu znajdującego się na serwerze, klient musi użyć jego lokalnej namiastki. W jaki sposób? Najczęściej spotykanym rozwiązaniem jest wywołanie zdalnej metody innego obiektu serwera, która zwróci obiekt namiastki. Powstaje jednak typowy problem „kury i jajka”. Pierwszy z obiektów serwera musi być zlokalizowany w jakiś inny sposób. Biblioteka RMI dostarcza w tym celu *usługę rejestracji początkowej*.

Program serwera rejestruje co najmniej jeden zdalny obiekt, korzystając z usługi rejestracji początkowej. Obiekt zdalny zostaje zarejestrowany przez podanie odpowiedniego adresu URL oraz referencji obiektu implementacji.

Adresy URL mechanizmu RMI rozpoczynają się przedrostkiem `rmi:`, po którym następuje opcjonalna nazwa serwera, opcjonalny numer portu i nazwa zdalnego obiektu. Oto przykład:

```
rmi://regserver.mycompany.com:99/central_warehouse
```

Domyślnie przyjmowany jest serwer `localhost` i port o numerze 1099. Serwer informuje rejestr, że należy związać podaną nazwę z danym obiektem.

A oto kod rejestrujący obiekt klasy `WarehouseImpl` w rejestrze RMI na tym samym serwerze:

```
WarehouseImpl centralWarehouse = new WarehouseImpl();
Context namingContext = new InitialContext();
namingContext.bind("rmi:central_warehouse", centralWarehouse);
```

Program przedstawiony na listingu 11.3 tworzy i rejestruje obiekt klasy `WarehouseImpl`.

**Listing 11.3.** warehouse1/WarehouseServer.java

```
import java.rmi.*;
import javax.naming.*;

/**
 * Program tworzy instancję zdalnego obiektu
 * reprezentującego magazyn,
 * rejestruje ją w serwisie nazw i oczekuje wywołań metod obiektu przez klientów.
 * @version 1.12 2007-10-09
 * @author Cay Horstmann
 */
public class WarehouseServer
{
    public static void main(String[] args) throws RemoteException, NamingException
    {
        System.out.println("Constructing server implementation...");
        WarehouseImpl centralWarehouse = new WarehouseImpl();

        System.out.println("Binding server implementation to registry...");
        Context namingContext = new InitialContext();
        namingContext.bind("rmi:central_warehouse", centralWarehouse);

        System.out.println("Waiting for invocations from clients...");
    }
}
```



Z powodów bezpieczeństwa aplikacja może rejestrować obiekty, tylko korzystając z usługi rejestru na tej samej maszynie. Zapobiega to modyfikowaniu informacji rejestru przez klientów o złych zamiarach. Natomiast dowolny klient może korzystać z usługi odnajdywania obiektów.

Klient może uzyskać wyliczenie wszystkich zarejestrowanych obiektów RMI, używając poniższego wywołania:

```
Enumeration<NameClassPair> e = namingContext.list("rmi://regserver.mycompany.com");
```

NameClassPair jest klasą pomocniczą, która zawiera zarówno nazwę obiektu, jak i nazwę jego klasy. Na przykład poniższy kod wyświetla nazwy wszystkich zarejestrowanych obiektów:

```
while (e.hasMoreElements()) System.out.println(e.nextElement().getName());
```

Klient pobiera namiastkę zdalnego obiektu, określając serwer i nazwę zdalnego obiektu w następujący sposób:

```
String url = "rmi://regserver.mycompany.com/central_warehouse";
Warehouse centralWarehouse = (Warehouse) namingContext.lookup(url);
```



Ponieważ zagwarantowanie unikalności nazw w rejestrze zawsze stanowi problem, powyższej metody nie należy używać jako podstawowego sposobu lokalizowania obiektów na serwerze. Należy jedynie umieszczać w rejestrze niewielką liczbę obiektów, które potrafią zlokalizować inne obiekty.

Kod przedstawiony na listingu 11.4 implementuje klienta, który pobiera namiastkę zdalnego obiektu i wywołuje jego metodę getPrice. Rysunek 11.4 ilustruje przepływ sterowania w omawianej sytuacji. Klient uzyskuje namiastkę Warehouse i wywołuje jej metodę getPrice. Namiastka kontaktuje się z serwerem i powoduje wywołanie metody getPrice obiektu WarehouseImpl.

#### **Listing 11.4.** warehouse1/WarehouseClient.java

```
import java.rmi.*;
import java.util.*;
import javax.naming.*;

/**
 * Klient wywołujący zdальną metodę.
 * @version 1.0 2007-10-09
 * @author Cay Horstmann
 */
public class WarehouseClient
{
    public static void main(String[] args) throws NamingException, RemoteException
    {
        Context namingContext = new InitialContext();

        System.out.print("RMI registry bindings: ");
        Enumeration<NameClassPair> e = namingContext.list("rmi://localhost/");
        while (e.hasMoreElements())
            System.out.println(e.nextElement().getName());

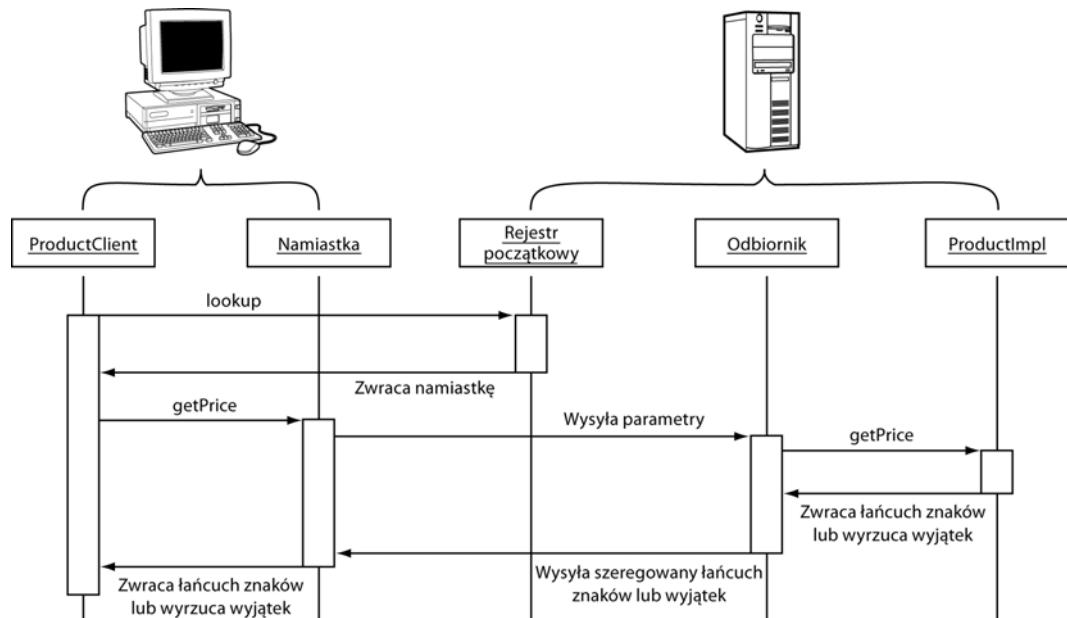
        String url = "rmi://localhost/central_warehouse";
        Warehouse centralWarehouse = (Warehouse) namingContext.lookup(url);
        System.out.println("Price of a central warehouse: " + centralWarehouse.getPrice());
    }
}
```

```

Warehouse centralWarehouse = (Warehouse) namingContext.lookup(url);

String descr = "Blackwell Toaster";
double price = centralWarehouse.getPrice(descr);
System.out.println(descr + ": " + price);
}
}

```



Rysunek 11.4. Wywołanie zdalnej metody getPrice

#### API `java.naming.initialContext 1.3`

- `InitialContext()`

tworzy kontekst nazw, który może być używany do dostępu do rejestru RMI.

#### API `java.naming.Context 1.3`

- `static Object lookup(String name)`

zwraca obiekt o podanej nazwie. Zgłasza wyjątek `NamingException`, jeśli nazwa nie jest związana z żadnym obiektem.

- `static void bind(String name, Object obj)`

rejestruje obiekt `obj` pod nazwą `name`. Zgłasza wyjątek `NameAlreadyBoundException`, jeśli obiekt jest już zarejestrowany.

- `static void unbind(String name)`

wyrejestrowuje obiekt o danej nazwie. Wyrejestrowanie nieistniejącej nazwy jest zabronione.

- static void rebind(String name, Object obj)  
rejestruje obiekt obj pod nazwą name, zastępując inny obiekt, jeśli zarejestrowany został pod tą samą nazwą.
- NamingEnumeration<NameClassPair> list(String name)  
zwraca wyliczenie obiektów o pasującej nazwie. Aby uzyskać wyliczenie wszystkich obiektów RMI, należy użyć nazwy "rmi:".

**API javax.naming.NameClassPair 1.3**

- String getName()  
zwraca nazwę obiektu.
- String getClassName()  
zwraca nazwę klasy, do której należy obiekt.

**API java.rmi.Naming 1.1**

- static Remote lookup(String url)  
zwraca zdalny obiekt określony przez adres URL. Wyrzuca wyjątek NotBound, jeśli obiekt taki nie został zarejestrowany.
- static void bind(String name, Remote obj)  
rejestruje obiektu obj pod nazwą name. Wyrzuca wyjątek AlreadyBoundException, jeśli obiekt jest już zarejestrowany.
- static void unbind(String name)  
wyrejestrowuje obiekt o danej nazwie. Wyrzuca wyjątek NotBound, jeśli obiekt o takiej nazwie nie jest zarejestrowany.
- static void rebind(String name, Remote obj)  
rejestruje obiekt obj pod nazwą name, zastępując inny obiekt, jeśli zarejestrowany został pod tą samą nazwą.
- static String[] list(String url)  
zwraca tablicę nazw znajdujących się w rejestrze zlokalizowanym pod określonym adresem URL.

### 11.3.3. Przygotowanie wdrożenia

Wdrożenie aplikacji korzystającej z RMI może okazać się kłopotliwe, ponieważ składa się z wielu etapów, na których możemy popełnić błąd, a komunikaty o błędach nie są zbyt wyczerpujące. Z naszego doświadczenia wynika, że warto porządnie przetestować wdrożenie takiej aplikacji w realistycznych warunkach, rozdzielając klasy przeznaczone dla klienta i serwera.

Klasy serwera i klienta umieścimy zatem w osobnych katalogach:

```
server/
  WarehouseServer.class
  Warehouse.class
  WarehouseImpl.class
```

```
client/
  WarehouseClient.class
  Warehouse.class
```

Wdrażając aplikacje RMI, zwykle dostarczamy dynamicznie klas działającym programom. Przykładem może być rejestr RMI. Pamiętajmy, że jedna instancja tego rejestru obsługuje wiele różnych aplikacji RMI. Rejestr RMI musi mieć dostęp do plików klas dla interfejsów usług, które są rejestrowane. W momencie uruchomienia rejestru nie są znane żądania rejestracji, które mogą się pojawić podczas jego działania. Dlatego też rejestr RMI musi dynamicznie ładować pliki klas dla wszystkich zdalnych interfejsów, które dotąd nie były mu znane.

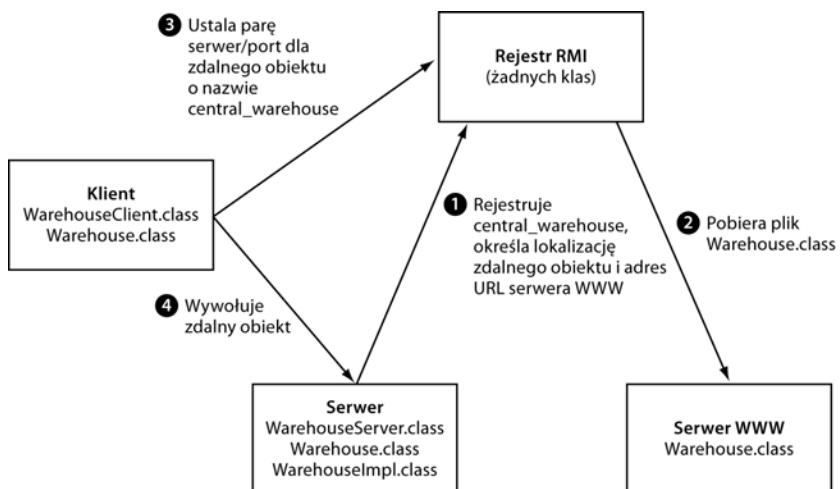
Dynamicznie dostarczane pliki klas są dystrybuowane przez standardowe serwery WWW. W naszym przykładzie serwer musi udostępnić rejestrowi RMI plik *Warehouse.class*, który umieścimy w osobnym katalogu o nazwie *download*.

```
download/
  Warehouse.class
```

Zawartość tego katalogu będzie udostępniana przez serwer WWW.

Podczas wdrażania aplikacji serwer, rejestr RMI, serwer WWW i klient mogą zostać rozlokowane na czterech różnych komputerach (patrz rysunek 11.5). W celach testowych wystarczy nam do tego jeden komputer.

**Rysunek 11.5.**  
Wywołania serwera  
podczas działania  
aplikacji *Warehouse*



Ze względów bezpieczeństwa usługa rmiregistry będąca częścią JDK może być wywoływana jedynie przez kod działający na tej samej maszynie. Innymi słowy, serwer i proces rmiregistry muszą zostać uruchomione na tym samym komputerze. W ogólnym przypadku architektura RMI zezwala na implementację rejestrów RMI obsługujących wiele serwerów.

Do przetestowania naszej aplikacji wykorzystamy serwer WWW o nazwie NanoHTTPD dostępny pod adresem <http://elonen.iki.fi/code/nanohttpd>. Ten niewielki serwer został zaimplementowany za pomocą jednego pliku źródłowego w języku Java. Otwórz nowe okno wiersza poleceń, przejdź do katalogu *download* i umieść w nim plik *NanoHTTPD.java*. Skompiluj ten plik źródłowy i uruchom serwer WWW, wywołując

```
java NanoHTTPD 8080
```

Parametr wywołania określa oczywiście numer portu wykorzystywany przez serwer. Jeśli na Twoim komputerze port 8080 jest już używany, wybierz inny dostępny port.

Następnie otwórz kolejne okno wiersza poleceń, przejdź do katalogu, który *nie zawiera żadnych plików klas*, i uruchom rejestr RMI:

```
rmiregistry
```



Przed uruchomieniem rejestrów RMI upewnij się, że zmienna środowiska CLASSPATH nie jest zainicjowana, oraz sprawdź, czy bieżący katalog nie zawiera żadnych plików klas. W przeciwnym razie rejestr RMI może odnaleźć przypadkowe pliki klas, których użycie zamiast klas, które powinien załadować z innego źródła. Wyjaśnienie takiego zachowania znajdziesz na stronie <http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/codebase.html>. W skrócie: dla każdego obiektu namiastki określona jest baza kodu, z której została załadowana. Baza ta jest używana do załadowania powiązanych klas. Jeśli rejestr RMI znajdzie klasę w lokalnym katalogu, to do załadowania wspomnianych klas użycie niewłaściwej bazy kodu.

Teraz możemy już uruchomić serwer. Otwieramy trzecie okno wiersza poleceń, przechodzimy do katalogu *server* i wydajemy polecenie

```
java -Djava.rmi.server.codebase=http://localhost:8080/ WarehouseServer
```

Właściwość *java.rmi.server.codebase* określa adres URL, pod którym dostępne są pliki klas. Program serwera przekazuje ten adres URL rejestrowi RMI.

Spójrzmy w okno, w którym działa serwer NanoHTTPD. Powinniśmy zobaczyć w nim komunikat, który świadczy o tym, że plik *Warehouse.class* został udostępniony rejestrowi RMI.



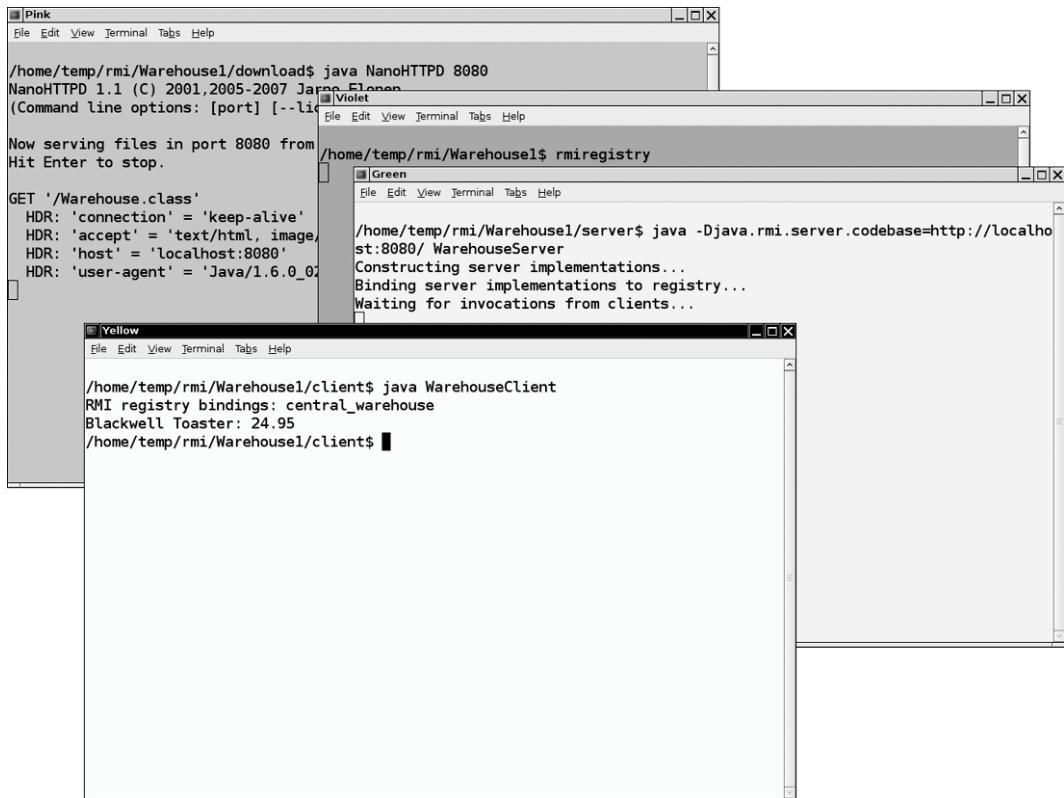
Bardzo istotne jest, aby pamiętać o zakończeniu adresu URL wskazującego pliki klas za pomocą ukośnika (/).

Zwrócmy uwagę, że program serwera nie kończy swojego działania. Może się to wydawać dziwne, ponieważ program ten jedynie tworzy obiekt i rejestruje go. Metoda *main* kończy swoje działanie zgodnie z przewidywaniami, zaraz po zarejestrowaniu obiektu. Jednak gdy tworzymy obiekt klasy pochodnej klasy *UnicastRemoteObject*, to uruchamiany jest nowy wątek, który powoduje, że program jest dalej wykonywany i umożliwia nawiązywanie połączeń przez klientów.

Na koniec otworzymy czwarte okno wiersza poleceń, przejdźmy do katalogu *client* i wydajmy polecenie

```
java WarehouseClient
```

Ujrzymy krótki komunikat wskazujący, że zdalna metoda została pomyślnie wywołana (patrz rysunek 11.6).



Rysunek 11.6. Testowanie aplikacji RMI



Jeśli chcesz przetestować tylko podstawową logikę działania programu, możesz umieścić pliki klas klienta i serwera w tym samym katalogu. Następnie możesz uruchomić rejestr RMI, serwer i klienta w tym katalogu. Ponieważ jednak ładowanie klas przez RMI jest często źródłem niepotrzebnej frustracji, powyżej zdecydowaliśmy się przedstawić poprawną konfigurację dynamicznych klas pozwalającą uniknąć błędów.

### 11.3.4. Rejestrowanie aktywności RMI

Jeśli uruchomimy serwer z następującą opcją

```
-Djava.rmi.server.logCalls=true WarehouseServer &
```

to serwer będzie wypisywać na konsoli wszystkie wywołania zdalnych metod. Wypróbuj tę opcję, daje ona niezły pogląd na działanie RMI.

Dodatkowe komunikaty o działaniu RMI możemy uzyskać, konfigurując odpowiednio standar-dowy mechanizm dziennika na platformie Java (patrz rozdział 11. książki *Java 2. Podstawy*).

W tym celu należy utworzyć plik *logging.properties* o następującej zawartości:

```
handlers=java.util.logging.ConsoleHandler
.level=FINE
java.util.logging.ConsoleHandler.level=FINE
java.util.logging.ConsoleHandler.formatter= java.util.logging.SimpleFormatter
```

Zamiast od razu konfigurować globalny poziom rejestrowania jako FINEST, możemy dostroić poziom indywidualnie dla poszczególnych rejestratorów. Rejestratory RMI przedstawione zostały w tabeli 11.1. Aby na przykład śledzić aktywność związaną z ładowaniem klas, możemy skonfigurować

```
sun.rmi.loader.level=FINE
```

**Tabela 11.1.** Rejestratory RMI

Nazwa rejestratora	Rejestrowana aktywność
sun.rmi.server.call	Zdalne wywołania po stronie serwera
sun.rmi.server.ref	Zdalne referencje po stronie serwera
sun.rmi.client.call	Zdalne wywołania po stronie klienta
sun.rmi.client.ref	Zdalne referencje po stronie klienta
sun.rmi.dgc	Rozproszone odzyskiwanie nieużytków
sun.rmi.loader	RMIClassLoader
sun.rmi.transport.misc	Warstwa transportowa
sun.rmi.transport.tcp	Połączenia TCP
sun.rmi.transport.proxy	Tunelowanie HTTP

Usługę rejestracji RMI należy uruchomić z opcją

```
-J-Djava.util.logging.config.file=directory/logging.properties
```

Klienta i serwer uruchamiamy z opcją

```
-Djava.util.logging.config.file=directory/logging.properties
```

Poniżej przedstawiamy przykładowe komunikaty ilustrujące problem z załadunkiem klasy — usługa rejestracji początkowego RMI nie może odnaleźć klasy Warehouse, ponieważ serwer zakończył swoje działanie.

```
FINE: RMI TCP Connection(1)-127.0.1.1: (port 1099) op = 80
Oct 13, 2007 4:43:30 PM sun.rmi.server.LoaderHandler loadProxyClass
FINE: RMI TCP Connection(1)-127.0.1.1: interfaces = [java.rmi.Remote, Warehouse],
↳codebase =
"http://localhost:8080/"
Oct 13, 2007 4:43:30 PM sun.rmi.server.LoaderHandler loadProxyClass
FINE: RMI TCP Connection(1)-127.0.1.1: proxy class resolution failed
java.lang.ClassNotFoundException: Warehouse
```

## 11.4. Parametry zdalnych metod i wartości zwracane

W momencie wywołania zdalnej metody jej parametry muszą zostać przekazane z maszyny wirtualnej klienta na maszynę wirtualną serwera. Po zakończeniu wywołania wartość zwrocona przez metodę musi zostać przekazana w przeciwnym kierunku. Gdy pewna wartość jest przekazywana pomiędzy maszynami wirtualnymi, możemy wyróżnić dwa przypadki: przekazywanie obiektów zdalnych i przekazywanie obiektów, które zdalne nie są. Na przykład założmy, że klient serwera `WarehouseServer` przekazuje referencję `Warehouse` (czyli namiastki pozwalającej na wywoływanie zdalnych metod) innej metodzie zdalnej. Jest to przykład przekazywania obiektu zdalnego. Jednak większość parametrów metod będzie zwykłymi obiektami Java, a nie namiastkami zdalnych obiektów. Przykładem może być parametr `String` metody `getPrice` naszej pierwszej przykładowej aplikacji.

### 11.4.1. Przekazywanie obiektów zdalnych

Gdy referencja zdalnego obiektu jest przekazywana z jednej maszyny wirtualnej na drugą, zarówno nadawca, jak i odbiorca używają referencji tego samego bytu. Referencja ta nie jest adresem w pamięci (który ma sens jedynie dla określonej, jednej maszyny wirtualnej), lecz składa się z adresu sieciowego i unikalnego identyfikatora zdalnego obiektu. Informacja ta jest hermetyzowana przez obiekt namiastki.

Koncepcyjnie przekazywanie zdalnej referencji przypomina przekazywanie referencji lokalnych obiektów w obrębie tej samej maszyny wirtualnej. Musimy jednak zawsze pamiętać, że wywołanie metody dla zdalnej referencji jest znacznie wolniejsze i potencjalnie bardziej zawodne niż wywołanie metody dla lokalnej referencji.

### 11.4.2. Przekazywanie obiektów, które nie są zdalne

Rozważmy parametr `String` metody `getPrice`. Łąncuch znaków musi zostać skopiowany z klienta na serwer. Nietrudno sobie wyobrazić, w jaki sposób łańcuch znaków może zostać przetransportowany przez sieć. Mechanizm RMI pozwala również kopiować bardziej złożone obiekty, jeśli tylko są one *serializowalne*. RMI używa mechanizmu serializacji omówionego w rozdziale 1. do przesyłania obiektów w sieci. Oznacza to, że dowolna klasa implementująca interfejs `Serializable` może być użyta jako parametr zdalnej metody lub wartość przez nią zwieracana.

Przekazywanie parametrów na zasadzie serializacji ma subtelny wpływ na semantykę zdalnych metod. Gdy przekazujemy obiekty lokalnej metodzie, to w rzeczywistości zostają przekazane ich *referencje*. Jeśli metoda zmodyfikuje przekazane obiekty, to wprowadzone zmiany będą widoczne dla kodu wywołującego metodę. Jeśli jednak zdalna metoda zmodyfikuje parametr poddany wcześniej serializacji, to zmiany zajdą jedynie w kopii oryginalnego obiektu i w związku z tym kod wywołujący zdalną metodę nigdy się o nich nie dowie.

Podsumowując, istnieją dwa mechanizmy przekazywania wartości pomiędzy maszynami wirtualnymi.

- Obiekty klas implementujących interfejs Remote są przekazywane jako zdalne referencje.
- Obiekty klas implementujących interfejs Serializable, ale nie Remote, są kopowane przy użyciu serializacji.

Wszystko to odbywa się automatycznie i nie wymaga interwencji programisty. Pamiętajmy jednak, że serializacja dużych obiektów może być powolna, a zdalne metody nie mogą modyfikować serializowanych parametrów. Oba problemów możemy oczywiście uniknąć, przekazując zdalne referencje. Ale nic za darmo: wywołania metod dla zdalnych referencji są znacznie kosztowniejsze niż wywołania lokalnych metod. Znajomość tych kosztów pozwala na dokonywanie bardziej świadomych decyzji podczas projektowania zdalnych usług.



Odzyskiwanie zasobów zdalnych obiektów odbywa się automatycznie, podobnie jak w przypadku lokalnych obiektów. Jednak sam mechanizm jest znacznie bardziej skomplikowany. Gdy lokalny mechanizm odzyskiwania ustali, że nie istnieją już żadne lokalne przypadki wykorzystania zdalnej referencji, powiadamia rozproszony mechanizm odzyskiwania zasobów, że dany klient nie używa już referencji tego serwera. Gdy serwer nie jest już używany przez żadnego klienta, jego zasoby mogą zostać odzyskane.

Nasz kolejny przykład będzie ilustracją przekazywania zdalnych i serializowalnych obiektów. Zmienimy w tym celu interfejs Warehouse w sposób pokazany na listingu 11.5. Na podstawie listy słów kluczowych magazyn zwróci produkt najlepiej pasujący do opisu.

#### **Listing 11.5.** warehouse2/Warehouse.java

```
import java.rmi.*;
import java.util.*;

/**
 * Zdalny interfejs prostego magazynu.
 * @version 1.0 2007-10-09
 * @author Cay Horstmann
 */
public interface Warehouse extends Remote
{
    double getPrice(String description) throws RemoteException;
    Product getProduct(List<String> keywords) throws RemoteException;
}
```

Parametr metody getProduct ma typ List<String>. Wartość parametru musi należeć do serializowalnej klasy implementującej interfejs List<String>, na przykład ArrayList<String>. (Nasz przykładowy klient przekazuje wartość uzyskiwaną poprzez wywołanie Arrays.asList. Metoda ta gwarantuje zwrócenie serializowalnej listy).

Typem zwracanym przez metodę getProduct jest Product, który hermetyzuje opis, cenę i lokalizację produktu (patrz listing 11.6).

**Listing 11.6.** warehouse2/Product.java

```
import java.io.*;  
  
public class Product implements Serializable  
{  
    private String description;  
    private double price;  
    private Warehouse location;  
  
    public Product(String description, double price)  
    {  
        this.description = description;  
        this.price = price;  
    }  
  
    public String getDescription()  
    {  
        return description;  
    }  
  
    public double getPrice()  
    {  
        return price;  
    }  
  
    public Warehouse getLocation()  
    {  
        return location;  
    }  
  
    public void setLocation(Warehouse location)  
    {  
        this.location = location;  
    }  
}
```

---

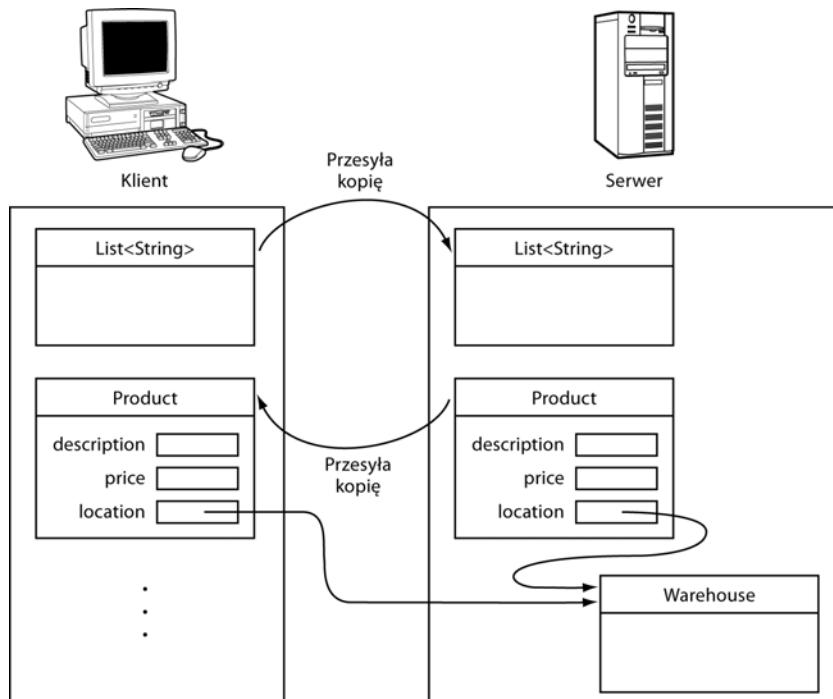
Zwróćmy uwagę, że klasa `Product` jest serializowalna. Serwer tworzy obiekt klasy `Product`, a klient otrzymuje jego kopię (patrz rysunek 11.7).

Zauważmy jednak pewien szczegół. Klasa `Product` ma pole instancji typu `Warehouse`, czyli zdalnego interfejsu. Serializacja obiektu `Warehouse` nie ma sensu, ponieważ może on zawierać sporo informacji związanych z jego stanem. Zamiast serializowanego obiektu klient otrzymuje namiastkę zdalnego obiektu `Warehouse`. Namiastka ta może różnić się od namiastki centralnej `Warehouse`, dla której wywołano metodę `getProduct`. W naszej implementacji będziemy mieć dwa rodzaje produktów: tosty i książki, które zlokalizowane będą w różnych magazynach.

### 11.4.3. Dynamiczne ładowanie klas

Z działaniem naszego kolejnego przykładowego programu wiąże się jeszcze jedna subtelnosć. Serwer na podstawie otrzymanej listy słów kluczowych zwraca instancję klasy `Product`. Oczywiście podczas komplikacji klienta został wykorzystany plik tej klasy, `Product.class`.

**Rysunek 11.7.**  
Kopiowanie lokalnego parametru i obiektu wynikowego



Jednak gdy serwer nie znajdzie towaru odpowiadającego słowom kluczowym, domyślnie zwraca produkt, który zadowoli każdego: książkę o języku Java. Reprezentuje ją obiekt klasy Book, będącej klasą pochodną klasy Product.

Jednak podczas komplikacji klienta klasa Book nie musiała być wcale dostępna. Natomiast podczas działania klient musi potrafić wykonać metody klasy Book przesyłającą metody klasy Product. Wobec tego klient musi dysponować możliwością załadowania dodatkowych klas podczas swojego działania. Używa w tym celu takiego samego mechanizmu jak rejestr RMI. Klas są udostępniane przez serwer WWW, klasa serwera RMI komunikuje klientowi odpowiedni adres URL, a klient używa go do wysłania żądania HTTP, które załaduje pliki klas.

Z załadowaniem kodu z innej maszyny wiążą się nierozerwalnie kwestie zapewnienia bezpieczeństwa. Dlatego też aplikacja RMI dynamicznie ładująca klasy musi korzystać z menedżera bezpieczeństwa. (Rozdział 9. zawiera więcej informacji na temat ładowania klas i menedżerów bezpieczeństwa).

Programy wykorzystujące RMI powinny instalować menedżera bezpieczeństwa kontrolującego aktywność dynamicznie ładowanych klas. Instalujemy go za pomocą poniższego wywołania:

```
System.setSecurityManager(new SecurityManager());
```



Gdy wszystkie klasy dostępne są lokalnie, to nie musimy korzystać z menedżera bezpieczeństwa. Jeśli potrafimy przewidzieć wszystkie pliki klas wykorzystywane przez tworzony program, to możemy umieścić je lokalnie. Często zdarza się jednak, że program serwera lub klienta ewoluje i z czasem dodawane są nowe klasy. Możemy wtedy skorzystać z dynamicznego ładowania klas. Zawsze jednak, gdy ładujemy kod z innego źródła, powinniśmy użyć menedżera bezpieczeństwa.

Domyślnie SecurityManager zabrania całemu kodowi klienta nawiązywania połączeń w sieci. Program klienta musi jednak nawiązać połączenia z trzema lokalizacjami:

- serwerem WWW, który ładuje zdalne klasy;
- rejestrem RMI;
- zdalnymi obiekty.

Aby umożliwić klientowi wykonanie tych połączeń, musimy utworzyć odpowiedni *plik polityki*. (Pliki takie omawialiśmy szczegółowo w rozdziale 9.). Poniżej prezentujemy zawartość pliku polityki, który zezwala aplikacji na wykonywanie połączeń z portami o numerze większym lub równym 1024. (Domyślny port RMI posiada numer 1099, a zdalne obiekty również używają portów o numerach powyżej 1024).

```
grant
{
    permission java.net.SocketPermission
        "*:1024-65535". "connect";
}:
```

Menedżerowi bezpieczeństwa polecamy wczytanie pliku polityki, nadając właściwości `java.security.policy` nazwę pliku. Na przykład:

```
System.setProperty("java.security.policy", "rmi.policy");
```

Alternatywnie możemy też skonfigurować tę właściwość w wierszu polecień:

```
-Djava.security.policy=rmi.policy
```

Zanim uruchomimy przykładową aplikację, musimy najpierw zakończyć działanie rejestru RMI, serwera WWW i programu serwera wykorzystywanych w poprzednim przykładzie. Następnie otwieramy cztery okna wiersza poleceń i wykonujemy kroki opisane poniżej.

- 1 Kompilujemy pliki źródłowe interfejsu, implementacji, klienta i serwera.

```
javac *.java
```

- 2 Tworzymy trzy katalogi, *client*, *server* i *download*, i umieszczamy w nich następujące pliki:

```
client/
    WarehouseClient.class
    Warehouse.class
    Product.class
    client.policy
server/
    Warehouse.class
    Product.class
    Book.class
    WarehouseImpl.class
    WarehouseServer.class
    server.policy
download/
    Warehouse.class
    Product.class
    Book.class
```

3. W pierwszym oknie wiersza poleceń przechodzimy do katalogu, który *nie* zawiera żadnych plików klas. Uruchamiamy w nim rejestr RMI.
4. W drugim oknie wiersza poleceń przechodzimy do katalogu *download* i uruchamiamy serwer NanoHTTPD.
5. W trzecim oknie wiersza poleceń przechodzimy do katalogu *server* i uruchamiamy program serwera.

```
java -Djava.rmi.server.codebase=http://localhost:8080/ WarehouseServer
```

6. W czwartym oknie wiersza poleceń przechodzimy do katalogu *client* i uruchamiamy klienta.

```
java WarehouseClient
```

Listing 11.7 prezentuje kod klasy Book. Zwróćmy uwagę, że zastąpiła ona metodę `getDescription` klasy bazowej własną wersją zwracającą numer ISBN. Wykonanie programu klienta powoduje właśnie wyświetlenie kodu ISBN, co dowodzi, że klasa Book została załadowana dynamicznie. Listing 11.8 przedstawia implementację magazynu. Obiekt reprezentujący magazyn posiada referencję zapasowego magazynu. Jeśli produkt nie został znaleziony w magazynie, przeszukiwany jest magazyn zapasowy. Listing 11.9 przedstawia program serwera. W rejestrze RMI zostaje umieszczony jedynie centralny magazyn. Zauważmy zatem, że zdalna referencja magazynu zapasowego może zostać przekazana klientowi, mimo że obiekt magazynu zapasowego nie został umieszczony w rejestrze RMI. Sytuacja ta ma miejsce, gdy żadne słowo kluczowe nie zostaje dopasowane i wobec tego serwer zwraca klientowi książkę *Core Java* (której pole `location` zawiera referencję magazynu zapasowego).

#### **Listing 11.7. warehouse2/Book.java**

---

```
/*
 * Reprezentuje książkę posiadającą numer ISBN.
 * @version 1.0 2007-10-09
 * @author Cay Horstmann
 */
public class Book extends Product
{
    private String isbn;

    public Book(String title, String isbn, double price)
    {
        super(title, price);
        this.isbn = isbn;
    }

    public String getDescription()
    {
        return super.getDescription() + " " + isbn;
    }
}
```

---

#### **Listing 11.8. warehouse2/WarehouseImpl.java**

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
```

```
/*
 * Klasa będąca implementacją zdalnego interfejsu Warehouse.
 * @version 1.01 2012-01-26
 * @author Cay Horstmann
 */
public class WarehouseImpl extends UnicastRemoteObject implements Warehouse
{
    private Map<String, Product> products;
    private Warehouse backup;

    /**
     * Konstruktor implementacji.
     */
    public WarehouseImpl(Warehouse backup) throws RemoteException
    {
        products = new HashMap<>();
        this.backup = backup;
    }

    public void add(String keyword, Product product)
    {
        product.setLocation(this);
        products.put(keyword, product);
    }

    public double getPrice(String description) throws RemoteException
    {
        for (Product p : products.values())
            if (p.getDescription().equals(description)) return p.getPrice();
        if (backup == null) return 0;
        else return backup.getPrice(description);
    }

    public Product getProduct(List<String> keywords) throws RemoteException
    {
        for (String keyword : keywords)
        {
            Product p = products.get(keyword);
            if (p != null) return p;
        }
        if (backup != null)
            return backup.getProduct(keywords);
        else if (products.values().size() > 0)
            return products.values().iterator().next();
        else
            return null;
    }
}
```

---

**Listing 11.9.** warehouse2/WarehouseServer.java

```
import java.rmi.*;
import javax.naming.*;

/**
 * Program serwera tworzy instancję zdalnego obiektu,
 * umieszcza go w rejestrze i oczekuje na wywołania klientów.
```

```

* @version 1.12 2007-10-09
* @author Cay Horstmann
*/
public class WarehouseServer
{
    public static void main(String[] args) throws RemoteException, NamingException
    {
        System.setProperty("java.security.policy", "server.policy");
        System.setSecurityManager(new SecurityManager());

        System.out.println("Constructing server implementation...");
        WarehouseImpl backupWarehouse = new WarehouseImpl(null);
        WarehouseImpl centralWarehouse = new WarehouseImpl(backupWarehouse);

        centralWarehouse.add("toaster", new Product("Blackwell Toaster", 23.95));
        backupWarehouse.add("java", new Book("Core Java vol. 2", "0132354799", 44.95));

        System.out.println("Binding server implementation to registry...");
        Context namingContext = new InitialContext();
        namingContext.bind("rmi:central_warehouse", centralWarehouse);

        System.out.println("Waiting for invocations from clients...");
    }
}

```

#### 11.4.4. Zdalne referencje obiektów o wielu interfejsach

Zdalna klasa może implementować wiele interfejsów. Weźmy na przykład zdalny interfejs ServiceCenter zdefiniowany jak poniżej.

```

public interface ServiceCenter extends Remote
{
    int getReturnAuthorization(Product prod) throws RemoteException;
}

```

Przypuśćmy teraz, że klasa WarehouseImpl implementuje ten interfejs, a także interfejs Warehouse. Gdy zdalna referencja takiego obiektu jest przekazywana innej maszynie wirtualnej, odbiorca otrzymuje namiastkę, która posiada dostęp do zdalnych metod zarówno interfejsu ServiceCenter, jak i Warehouse. Aby dowiedzieć się, czy konkretny zdalny obiekt implementuje dany interfejs, możemy użyć operatora instanceof. Przypuśćmy, że otrzymaliśmy zdalny obiekt za pośrednictwem zmiennej typu Warehouse.

```
Warehouse location = product.getLocation();
```

Zdalny obiekt może implementować interfejs ServiceCenter lub nie. Aby się o tym przekonać, wykonujemy następujący test:

```
if (location instanceof ServiceCenter)
```

Jeśli wynik jest pozytywny, możemy rzutować location na typ ServiceCenter i wywołać metodę getReturnAuthorization.

## 11.4.5. Zdalne obiekty i metody equals, hashCode oraz clone

Klasy obiektów umieszczanych w zbiorach muszą zastępować metodę `equals`. W przypadku zbiorów wykorzystujących tablicę z kodowaniem mieszającym muszą one definiować także metodę `hashCode`. Jednak porównanie zdalnych obiektów takich klas okazuje się problematyczne. Aby sprawdzić, czy dwa zdalne obiekty posiadają taką samą zawartość, metoda `equals` musiałaby kontaktować się z serwerem, na którym znajdują się obiekty, i porównać ich zawartość. Jednak komunikacja z serwerem może się nie powieść, a metoda `equals` klasy `Object` nie deklaruje oczywiście możliwości wyrzucenia wyjątku `RemoteException`. Ponieważ metoda klasy pochodnej nie może deklarować wyrzucenia innych wyjątków niż zastępowana przez nią metoda klasy bazowej, to metody `equals` nie możemy zdefiniować jako części zdalnego interfejsu. Podobnie sytuacja przedstawia się w przypadku metody `hashCode`.

Metody `equals` i `hashCode` namiastek zdefiniowano więc w taki sposób, że nie korzystają one z zawartości obiektów, a jedynie z ich położenia. Metoda `equals` uważa dwie namiastki za identyczne, jeśli stanowią referencję do tego samego obiektu serwera. Dwie namiastki, które stanowią referencje do różnych obiektów serwera, nigdy nie są uważane za identyczne, nawet gdy obiekty serwera posiadają identyczną zawartość. Podobnie metoda `hashCode` wylicza kod mieszający jedynie na podstawie identyfikatora obiektu.

Z tych samych powodów zdalne obiekty nie posiadają metody `clone`. Gdyby metoda `clone` musiała komunikować się z serwerem, aby poinformować go o konieczności utworzenia klonu zdalnego obiektu, to musiałaby też posiadać możliwość wyrzucenia wyjątku `RemoteException`. Jednak metoda `clone` klasy `Object` deklaruje jedynie możliwość wyrzucenia wyjątku `CloneNotSupportedException`.

Podsumowując: namiastki możemy umieszczać w zbiorach i tablicach mieszających, ale musimy pamiętać, że sprawdzanie identyczności i wyznaczanie kodu mieszającego nie bierze w tym przypadku pod uwagę zawartości zdalnych obiektów. Natomiast klonowanie zdalnych referencji nie jest w ogóle możliwe.

## 11.5. Aktywacja zdalnych obiektów

W poprzednich przykładach program serwera tworzył instancje zdalnych klas i rejestrował je, tak by ich metody były zdalnie dostępne dla programu klienta. W niektórych przypadkach, zwłaszcza gdy istnieje wiele takich obiektów serwera, tworzenie tych obiektów, zanim klient spróbuje z nich skorzystać, może okazać się nieco nadmiarowe. Mechanizm *aktywacji* obiektów serwera pozwala odroczyć utworzenie zdalnego obiektu, aż do momentu, gdy program klienta wywoła zdalnie jego metodę.

Zaletą takiego rozwiązania jest też to, że nie wymaga ono jakichkolwiek modyfikacji kodu klienta. Klient nadal korzysta z referencji zdalnego obiektu i wywołuje jego metody.

Natomiast program serwera zostaje zastąpiony programem aktywacji obiektów tworzącym *deskryptory aktywacji* dla obiektów, których utworzenie zostaje odroczone oraz wiążą odbiorniki wywołań zdalnych metod z rejestrtem początkowym RMI. Gdy zdalne wywołanie metody

danego obiektu występuje pierwszy raz, to informacja zawarta w deskryptorze aktywacji wykorzystywana jest do utworzenia obiektu.

Obiekty serwera, których utworzenie jest odroczone, muszą należeć do klasy pochodnej klasy `Activatable`, a nie klasy `UnicastRemoteObject`. Oczywiście implementują również jeden lub więcej zdalnych interfejsów. Na przykład:

```
class WarehouseImpl
    extends Activatable
    implements Warehouse
{
    ...
}
```

Ponieważ utworzenie instancji takiej klasy zostaje odroczone w czasie, to musi odbywać się w pewien standardowy sposób. W tym celu klasa posiadać musi konstruktor o dwóch parametrach:

- identyfikatorze aktywacji (który przekazujemy po prostu konstruktorowi klasy bazowej),
- pojedynczym obiekcie zawierającym wszystkie informacje niezbędne do utworzenia instancji opakowane za pomocą `MarshalledObject`.

Jeśli potrzeba więcej parametrów konstrukcji obiektu, to musimy je wszystkie zapakować za pomocą pojedynczego obiektu. W tym celu możemy skorzystać na przykład z tablicy `Object[]` lub klasy `ArrayList`.

Na podstawie deskryptora aktywacji konstruujemy `MarshalledObject`:

```
MarshalledObject<T> param = new MarshalledObject<T>(constructionInfo);
```

Konstruktor obiektu implementacji używa metody `get` klasy `MarshalledObject`, aby uzyskać informacje o konstrukcji.

```
T constructionInfo = param.get();
```

Aby zademonstrować sposób aktywacji, zmodyfikujemy klasę `WarehouseImpl` w taki sposób, aby informacja o konstrukcji była mapą opisów i cen. Informacja ta zostanie obudowana obiektem `MarshalledObject` i rozszeregowana przez konstruktor:

```
public WarehouseImpl(ActivationID id, MarshalledObject<Map<String, Double>> param)
    throws RemoteException, ClassNotFoundException, IOException
{
    super(id, 0);
    prices = param.get();
    System.out.println("Warehouse implementation constructed.");
}
```

Przekazując wartość 0 jako drugi parametr konstruktora klasy bazowej, informujemy RMI, że do nasłuchu wywołań metod danego obiektu powinno samo przydzielić odpowiedni port.

Konstruktor wyświetla informację o utworzeniu obiektu, aby można było zaobserwować fakt jego tworzenia na żądanie.



Obiekty serwera nie muszą koniecznie rozszerzać klasy Activatable. Wystarczy umieścić jedynie wywołanie metody statycznej w konstruktorze klasy serwera.

```
Activatable.exportObject(this, id, 0)
```

Zajmijmy się teraz programem aktywacji. Musimy najpierw zdefiniować grupę aktywacji, która opisuje parametry uruchomienia maszyny wirtualnej zawierającej obiekty serwera. Najważniejszym z parametrów jest polityka bezpieczeństwa.

Deskryptory grupy aktywacji tworzymy w następujący sposób:

```
Properties props = new Properties();
props.put("java.security.policy", "/path/to/server.policy");
ActivationGroupDesc group = new ActivationGroupDesc(props, null);
```

Drugi z parametrów konstruktora służy do przekazania specjalnych opcji, które nie są nam potrzebne w tym przykładzie i dlatego przekazujemy wartość null.

Następnie tworzymy identyfikator grupy:

```
ActivationGroupID id =
ActivationGroupID.getSystem().registerGroup(group);
```

Jesteśmy już gotowi do utworzenia deskryptorów aktywacji. Dla każdego obiektu, który tworzony będzie na żądanie, musimy określić:

- identyfikator grupy aktywacji dla maszyny wirtualnej, która będzie tworzyć obiekt,
- nazwę klasy obiektu (na przykład "WarehouseImpl" lub "com.mycompany.MyClassImpl"),
- adres URL określający źródło plików klas (nie powinien zawierać ścieżek pakietów),
- szeregowaną informację wykorzystywaną przy konstrukcji obiektu.

Na przykład:

```
MarshalledObject param = new MarshalledObject(constructionInfo);
ActivationDesc desc = new ActivationDesc(id, "WarehouseImpl",
"http://myserver.com/download/", param);
```

Utworzony deskryptor musimy przekazać metodzie statycznej Activatable.register. Zwróci ona obiekt pewnej klasy implementującej zdalny interfejs klasy implementacji. Obiekt ten powinniśmy zarejestrować, korzystając z usługi nazw.

```
Warehouse centralWarehouse = (Warehouse) Activatable.register(desc);
namingContext.bind("rmi:central_warehouse", centralWarehouse);
```

W przeciwieństwie do programów serwera z poprzednich przykładów, program aktywacji kończy swoje działanie po zarejestrowaniu odbiorników aktywacji. Obiekty serwera tworzone są w momencie pierwszego wywołania ich metod.

Listingi 11.10 i 11.11 pokazują kod programu aktywacji oraz kod implementacji obiektów aktywowanych na żądanie. Interfejs klasy reprezentującej magazyn oraz program klienta pozostają niezmienione.

**Listing 11.10.** activation/WarehouseActivator.java

```

import java.io.*;
import java.rmi.*;
import java.rmi.activation.*;
import java.util.*;
import javax.naming.*;

/**
 * Program serwera tworzy zdalny obiekt reprezentujący magazyn,
 * rejestruje go w usłudze nazw i czeka na klientów, którzy wywołają metody tego obiektu.
 * @version 1.13 2012-01-26
 * @author Cay Horstmann
 */
public class WarehouseActivator
{
    public static void main(String[] args) throws RemoteException, NamingException,
        ActivationException, IOException
    {
        System.out.println("Constructing activation descriptors...");

        Properties props = new Properties();
        // używa pliku server.policy z bieżącego katalogu
        props.put("java.security.policy", new File("server.policy").getCanonicalPath());
        ActivationGroupDesc group = new ActivationGroupDesc(props, null);
        ActivationGroupID id = ActivationGroup.getSystem().registerGroup(group);

        Map<String, Double> prices = new HashMap<>();
        prices.put("Blackwell Toaster", 24.95);
        prices.put("ZapXpress Microwave Oven", 49.95);

        MarshalledObject<Map<String, Double>> param = new MarshalledObject<Map<String,
        ↳Double>>(
            prices);

        String codebase = "http://localhost:8080/";

        ActivationDesc desc = new ActivationDesc(id, "WarehouseImpl", codebase, param);

        Warehouse centralWarehouse = (Warehouse) Activatable.register(desc);

        System.out.println("Binding activable implementation to registry...");
        Context namingContext = new InitialContext();
        namingContext.bind("rmi:central_warehouse", centralWarehouse);
        System.out.println("Exiting...");
    }
}

```

**Listing 11.11.** activation/WarehouseImpl.java

```

import java.io.*;
import java.rmi.*;
import java.rmi.activation.*;
import java.util.*;

```

```
/*
 * Klasa implementacji zdalnego interfejsu Warehouse.
 * @version 1.0 2007-10-20
 * @author Cay Horstmann
 */
public class WarehouseImpl extends Activatable implements Warehouse
{
    private Map<String, Double> prices;

    public WarehouseImpl(ActivationID id, MarshalledObject<Map<String, Double>> param)
        throws RemoteException, ClassNotFoundException, IOException
    {
        super(id, 0);
        prices = param.get();
        System.out.println("Warehouse implementation constructed.");
    }

    public double getPrice(String description) throws RemoteException
    {
        Double price = prices.get(description);
        return price == null ? 0 : price;
    }
}
```

---

Aby uruchomić program należy wykonać następujące kroki.

- 1** Kompilujemy pliki źródłowe.
- 2** Umieszczamy pliki klas w następujący sposób:

```
client/
    WarehouseClient.class
    Warehouse.class
server/
    WarehouseActivator.class
    Warehouse.class
    WarehouseImpl.class
    server.policy
download/
    Warehouse.class
    WarehouseImpl.class
rmi/
    rmid.policy
```

- 3.** Uruchamiamy usługę rejestracji początkowego RMI w katalogu *rmi* (który nie zawiera żadnych plików klas).
- 4.** Uruchamiamy demona aktywacji RMI w katalogu *rmi*:

```
rmid -J-Djava.security.policy=rmid.policy
```

Program *rmid* nasłuchiwa żądań aktywacji i aktywuje obiekty za pomocą osobnej maszyny wirtualnej. Aby uruchomić maszynę wirtualną, program *rmid* musi uzyskać odpowiednie pozwolenia. Są one wyspecyfikowane w pliku polityki (patrz listing 11.12). Opcja *-J* wykorzystywana jest do przekazania opcji maszynie wirtualnej wykonującej demona aktywacji.

**Listing 11.12.** activation/rmid.policy

```
grant
{
    permission com.sun.rmi.rmid.ExecPermission
        "${java.home}${/}bin${/}java";
    permission com.sun.rmi.rmid.ExecOptionPermission
        "-Djava.security.policy=*";
}
```

**5.** Uruchamiamy serwer nanoHTTPD w katalogu *download*.

**6.** Uruchamiamy program aktywacji. dla katalogu server.

```
java -Djava.rmi.server.codebase=http://localhost:8080/ WarehouseActivator
```

Program kończy swoje działanie po zarejestrowaniu odbiorników aktywacji w usłudze nazw. (Zastanawiać może, po co specyfikujemy kolejny raz bazę kodu, skoro zostaje ona określona również w konstruktorze deskryptora aktywacji. Jednak ta informacja jest przetwarzana wyłącznie przez demona aktywacji RMI. Natomiast rejestr RMI nadal potrzebuje bazy kodu, aby załadować klasy zdalnego interfejsu).

**7.** Uruchamiamy program klienta z katalogu *client*:

```
java WarehouseClient
```

Program klienta wyświetla znane już opisy produktów. Podczas pierwszego jego uruchomienia w oknie serwera pojawią się komunikaty o utworzeniu obiektów na żądanie.

**API** **java.rmi.activation.Activatable 1.2**

- **protected Activatable(ActivationID id, int port)**  
tworzy obiekt klasy Activatable i nasłuchuje żądań aktywacji na podanym porcie. Jeśli jako numer portu podamy 0, to port zostanie przyznany automatycznie.
- **static Remote exportObject(Remote obj, ActivationID id, int port)**  
sprawia, że zdalny obiekt można aktywować. Zwraca odbiornik aktywacji, który musi zostać udostępniony zdalnym wywołaniom. Jeśli jako numer portu podamy 0, to port zostanie przyznany automatycznie.
- **Remote register(ActivationDescriptor desc)**  
rejestruje deskryptor obiektu, który można aktywować i przygotowuje go do zdalnych wywołań metod. Zwraca odbiornik aktywacji, który musi zostać udostępniony zdalnym wywołaniom.

**API** **java.rmi.MarshalledObject 1.2**

- **MarshalledObject(Object obj)**  
tworzy obiekt zawierający serializowane dane innego obiektu.

- `Object get()`  
poddaje dane procesowi odwrotnemu do serializacji i zwraca obiekt.

**API `java.rmi.activation.ActivationGroupDesc 1.2`**

- `ActivationGroupDesc(Properties props, ActivationGroupDesc env)`  
tworzy deskryptor grupy aktywacji specyfikujący właściwości maszyny wirtualnej, kreującej obiekty, które można aktywować. Parametr `env` zawiera ścieżkę dostępu do pliku wykonywalnego maszyny wirtualnej i opcje wiersza poleceń lub wartość `null`, gdy specjalna konfiguracja nie jest wymagana.

**API `java.rmi.activation.ActivationGroup 1.2`**

- `static ActivationSystem getSystem()` zwraca referencję do systemu aktywacji.

**API `java.rmi.activation.ActivationSystem 1.2`**

- `ActivationGroupID registerGroup (ActivationGroupDesc group)`  
rejestruje grupę aktywacji i zwraca jej identyfikator.

**API `java.rmi.activation.ActivationDesc 1.2`**

- `ActivationDesc(ActivationGroupID id, String className, String location, MarshalledObject data)`  
tworzy deskryptor aktywacji.

W rozdziale tym przedstawiliśmy mechanizm RMI, który jest zaawansowanym modelem programowania rozproszonego na platformie Java. W ostatnim rozdziale zajmiemy się innym aspektem programowania w języku Java: współpracą z „macierzystym” kodem napisanym w innym języku programowania i działającym na tej samej maszynie, co aplikacja Java.

# 12

## Metody macierzyste

W tym rozdziale:

- Wywołania funkcji języka C z programów w języku Java.
- Numeryczne parametry metod i wartości zwracane.
- Łańcuchy znaków jako parametry.
- Dostęp do składowych obiektu.
- Sygnatury.
- Wywoływanie metod języka Java.
- Tablice.
- Obsługa błędów.
- Interfejs programowy wywołań języka Java.
- Kompletny przykład: dostęp do rejestru systemu Windows.

Mimo że najlepszym rozwiązaniem są programy tworzone w całości w języku Java, to w praktyce zdarzają się sytuacje, w których musimy napisać lub wykorzystać istniejący już kod w innym języku programowania (kod taki będziemy nazywać *macierzystym*).

Zwłaszcza w początkowym stadium rozwoju platformy Java wielu programistów uważało, że zastosowanie języka C lub C++ pozwala poprawić efektywność działania najważniejszych części aplikacji Java. W praktyce jednak takie rozwiązania rzadko okazywały się uzasadnione. Przykładem może być prezentacja przedstawiona na konferencji JavaOne w 1996 roku. Twórcy biblioteki kryptograficznej z firmy Sun Microsystems pokazali, że implementacja funkcji kryptograficznych wyłącznie w samym języku Java okazała się więcej niż wystarczająca. Chociaż prawdopodobnie nie działała również szybko jak porównywalna implementacja w języku C, w praktyce nie miało to żadnego znaczenia, ponieważ działała znacznie szybciej niż operacje wejścia i wyjścia w sieci, które okazały się prawdziwym wąskim gardłem.

Wykorzystanie kodu macierzystego ma też swoje ujemne strony. Jeśli część naszej aplikacji Java napiszemy w innym języku, musimy wtedy dostarczyć osobną bibliotekę kodu macierzystego dla każdej platformy, na której ma działać nasza aplikacja. Kod napisany w języku C lub C++ nie posiada żadnych zabezpieczeń przed nadpisaniem pamięci na skutek niewłaściwego użycia wskaźników. Łatwo zatem napisać metody macierzyste, które zakłócają pracę aplikacji lub spowodują infekcję systemu operacyjnego.

Dlatego też zalecamy posługiwania się kodem macierzystym tylko wtedy, gdy jest to rzeczywiście niezbędne. Można wymienić trzy przyczyny, dla których implementacja części kodu w języku programowania innym niż Java może okazać się konieczna.

- Program musi korzystać bezpośrednio z usług systemu operacyjnego lub urządzeń, które nie są dostępne na platformie Java.
- Jeśli posiadamy już znaczną ilość kodu napisanego i przetestowanego w innym języku programowania i potrafimy przenieść go na wszystkie wymagane platformy docelowe.
- Na podstawie testów efektywności okazało się, że kod w języku Java działa znacznie wolniej niż jego odpowiednik w innym języku.

Platforma Java udostępnia interfejs programowy JNI (Java Native Interface) umożliwiający współpracę z macierzystym kodem napisanym w języku C. Programowanie z wykorzystaniem JNI będzie przedmiotem naszego zainteresowania w tym rozdziale.



Do implementacji metod macierzystych możemy także wykorzystać język C++. Rozwiązanie takie posiada kilka zalet — lepszą kontrolę zgodności typów i wygodniejszy dostęp do funkcji interfejsu JNI. Należy jednak zauważyć, że JNI nie umożliwia wiązania klas platformy Java z klasami języka C++.

## 12.1. Wywołania funkcji języka C z programów w języku Java

Załóżmy, że posiadamy funkcję w języku C, której z pewnych powodów nie będziemy przenosić na platformę Java. Dla potrzeb ilustracji zagadnienia rozpoczęniemy od prostej funkcji języka C wyświetlającej tekst powitania.

W języku Java oznaczamy metodę macierzystą słowem kluczowym native. Oczywiście musimy umieścić ją w jakiejś klasie. Efekt prezentuje listing 12.1.

Słowo kluczowe native przekazuje do kompilatora informację, że definicja metody będzie dostarczona z zewnątrz. Ponieważ metoda macierzysta nie będzie zawierać żadnego kodu w języku Java, to bezpośrednio po deklaracji jej nagłówka występuje znak średnika. Deklaracja metody macierzystej przypomina więc pod tym względem deklarację metody abstrakcyjnej.

**Listing 12.1.** HelloNative.java

```
/*
 * @version 1.11 2007-10-26
 * @author Cay Horstmann
 */
class HelloNative
{
    public static native void greeting();
}
```



Podobnie jak w poprzednim rozdziale dla zachowania prostoty przykładów nie posługujemy się pakietami.

W powyższym przykładzie zadeklarowaliśmy dodatkowo metodę macierzystą jako metodę statyczną. Metody macierzyste mogą być zwykłymi metodami składowymi klasy, jak i metodami statycznymi. Ponieważ nie chcemy jeszcze zajmować się przekazywaniem parametrów metodom macierzystym, to zadeklarowaliśmy metodę macierzystą jako statyczną.

Możemy już nawet skompilować naszą klasę, ale w przypadku, gdy spróbujemy użyć jej w programie, maszyna wirtualna poinformuje nas, że nie potrafi odnaleźć metody greeting, raportując błąd `UnsatisfiedLinkError`. Musimy zatem zająć się implementacją metody macierzystej i napisać kod odpowiedniej funkcji w języku C. Jej nazwa musi być *dokładnie* taka, jakiej oczekuje platforma Java. Nazwami tymi rzadzą następujące reguły:

1. Używamy pełnych nazw metod w języku Java, na przykład `HelloNative.greeting`. Jeśli klasa znajduje się w pakiecie, to nazwę klasy należy poprzedzić nazwą pakietu, na przykład `com.horstmann.HelloNative.greeting`.
2. Każdy znak kropki zastępujemy znakiem podkreślenia i dodajemy do otrzymanej nazwy przedrostek `Java_`. Na przykład `Java_HelloNative_greeting` lub `Java_com_horstmann_HelloNative_greeting`.
3. Jeśli nazwa klasy zawiera znaki, które nie są literami lub cyframi kodu ASCII — na przykład `' '` czy `'$'` lub też znaki Unicode o kodzie większym niż `'\u007F'`, to zastępujemy je łańcuchem w postaci `_0xxxx`, gdzie `xxxx` jest sekwencją czterech cyfr szesnastkowych występujących w kodzie Unicode danego znaku.



Jeśli przeciążymy identyfikator metody macierzystej, dostarczając wiele wersji metody o tej samej nazwie, to do nazwy metody musimy dołączyć podwójny znak podkreślenia, po którym wystąpią zakodowane typy parametrów. Kodowanie typów parametrów omówimy w dalszej części rozdziału. Jeśli posiadamy na przykład metodę macierzystą `greeting` oraz metodę macierzystą `greeting(int repeat)`, to pierwszą z nich wywołamy za pomocą nazwy `Java_HelloNative_greeting`, a drugą — przy użyciu nazwy `Java_HelloNative_greeting_I`.

Na szczęście nazw tych nie musimy tworzyć „ręcznie” — służy do tego program narzędziowy `javah`. Aby z niego skorzystać, należy najpierw skompilować plik źródłowy pokazany w listingu 12.1.

```
javac HelloNative.java
```

Następnie wywołać program javah, który utworzy plik nagłówkowy w języku C. Program javah znajduje się w katalogu *jdk/bin*. Wywołujemy go, podając nazwę klasy — podobnie jak podczas uruchamiania programu w języku Java:

```
javah HelloNative
```

Wywołanie programu javah spowoduje utworzenie pliku nagłówkowego *HelloNative.h*, którego zawartość pokazuje listing 12.2.

---

**Listing 12.2.** *helloNative/HelloNative.h*

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloNative */

#ifndef _Included_HelloNative
#define _Included_HelloNative
#ifndef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloNative
 * Method:    greeting
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloNative_greeting
(JNIEnv *, jclass);

#endif /* __cplusplus
}
#endif
#endif
```

---

Plik ten zawiera deklarację funkcji `Java_HelloNative_greeting`. (Lańcuchy `JNIEXPORT` i `JNICALL` są zdefiniowane w pliku nagłówkowym *jni.h*. Oznaczają one zależne od kompilatora specyfikatory funkcji eksportowanych z bibliotek ładowanych dynamicznie).

Wystarczy teraz jedynie skopiować prototyp funkcji z pliku nagłówkowego do pliku źródłowego i dodać implementację funkcji, co pokazuje listing 12.3.

---

**Listing 12.3.** *helloNative/HelloNative.c*

```
/**
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */

#include "HelloNative.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_HelloNative_greeting(JNIEnv* env, jclass cl)
{
    printf("Hello Native World!\n");
}
```

---



Do implementacji metod macierzystych możemy wykorzystać także język C++. W tym przypadku musimy jednak zadeklarować funkcje implementujące metody macierzyste jako `extern "C"` (co zapobiega generowaniu przez kompilator kodu specyficznego dla programów w języku C++), na przykład:

```
extern "C"
JNIEXPORT void JNICALL Java_HelloNative_greeting(JNIEnv* env, jclass cl)
{
    cout << "Hello, Native World!" << endl;
}
```

W przypadku tej prostej funkcji możemy zignorować parametry `env` i `cl`. Ich zastosowanie pokażemy w dalszej części rozdziału.

Następnie kod źródłowy w języku C komplikujemy do dynamicznie ładowanej biblioteki. Szczegóły tej operacji zależą od posiadanego kompilatora.

W przypadku kompilatora Gnu C w systemie Linux wykorzystamy poniższe polecenia:

```
gcc -fPIC -I jdk/include/
-I jdk/include/linux -shared -o libHelloNative.so HelloNative.c
```

Natomiast dla kompilatora firmy Sun pracującego w systemie Solaris polecenie te będą miały postać:

```
cc -G -I jdk/include/
-I jdk/include/solaris
-o libHelloNative.so HelloNative.c
```

W systemie Windows możemy użyć na przykład kompilatora Microsoft C++ i wtedy polecenie komplikacji będzie miało poniższą postać:

```
c1 -I jdk\include\ -I jdk\include\win32 -LD HelloNative.c
-FeHelloNative.dll
```

W powyższych przykładach `jdk` oznacza katalog, w którym zainstalowano JDK.



Jeśli kompilator Microsoft C++ wywołujemy w wierszu poleceń, to najpierw należy wywołać plik wsadowy `vcvars32.bat` lub `vsvars32.bat`. Plik ten konfiguruje odpowiednio ścieżki dostępu i zmienne środowiska dla kompilatora uruchamianego z poziomu wiersza poleceń. Plik ten można odnaleźć w katalogu `c:\Program Files\Microsoft Visual Studio.NET 2003\Common7\tools`, `c:\Program Files\Microsoft Visual Studio 8\VC` lub podobnym.

W systemie Windows możemy także posłużyć się darmowym środowiskiem Cygwin (<http://www.cygwin.com>), które zawiera kompilator Gnu C i biblioteki umożliwiające programowanie, jak w systemie Unix. Dla kompilatora Cygwin użyjemy następujących polecień:

```
gcc -mno-cygwin -D __int64="long long"
-I jdk/include/ -I jdk/include/win32
-shared -Wl,--add-stdcall-alias -o HelloNative.dll HelloNative.c
```

Powyższe polecenia należy wprowadzać zawsze w jednym wierszu.



Plik nagłówkowy *jni\_md.h* w systemie Windows zawiera poniższą deklarację typu

```
typedef __int64 jlong;
```

Jest ona specyficzna dla kompilatora firmy Microsoft. Dlatego też korzystając w systemie Windows z kompilatora Gnu, powinniśmy zmodyfikować ten plik, tak jak pokazano poniżej.

```
#ifdef __GNUC__
    typedef long long jlong
#else
    typedef __int64 jlong;
#endif
```

Alternatywnym rozwiązaniem będzie wywołanie kompilatora z opcją `-D __int64="long long"`, co pokazaliśmy, omawiając sposoby kompilacji metody macierzystej.

Ostatnią operacją będzie wywołanie metody `System.loadLibrary` w celu zapewnienia, że biblioteka zostanie załadowana przez maszynę wirtualną, zanim zostanie użyta. Wywołanie to najlepiej umieścić w statycznym blokuinicjalizacji klasy zawierającej metodę macierzystą, co pokazano w listingu 12.4.

**Listing 12.4.** *helloNative/HelloNativeTest.java*

```
/*
 * @version 1.11 2007-10-26
 * @author Cay Horstmann
 */
class HelloNativeTest
{
    public static void main(String[] args)
    {
        HelloNative.greeting();
    }

    static
    {
        System.loadLibrary("HelloNative");
    }
}
```

Rysunek 12.1 przedstawia podsumowanie procesu przetwarzania kodu macierzystego

Po skompilowaniu i uruchomieniu programu w oknie terminala powinien pojawić się napis „Hello, Native World!”.



W systemie Linux musimy dodać bieżący katalog do ścieżki dostępu do bibliotek. W tym celu należy albo skonfigurować odpowiednio zmienną środowiskową `LD_LIBRARY_PATH`:

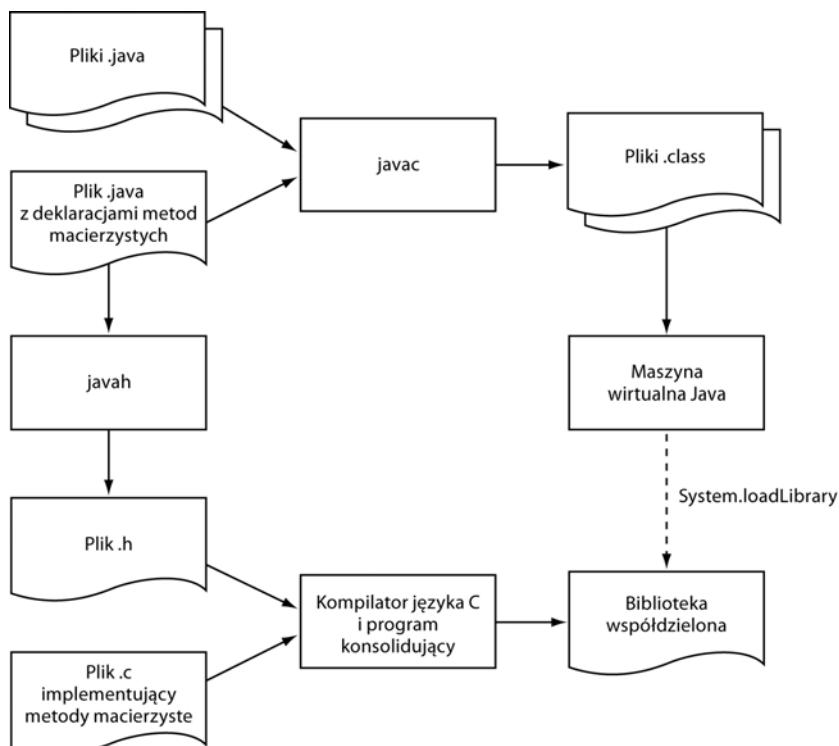
```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

lub właściwość systemową `java.library.path`:

```
java -Djava.library.path=. HelloNativeTest
```

**Rysunek 12.1.**

Przetwarzanie kodu macierzystego



Oczywiście powyższy przykład nie jest szczególnie spektakularny. Jeśli jednak uświadomimy sobie, że napis ten został utworzony za pomocą funkcji języka C, a nie przez kod programu w języku Java, to zrozumiemy, że wykonaliśmy właśnie pierwszy krok na drodze do wspólnego wykorzystania kodu napisanego w obu językach!

Podsumowując, aby użyć metody macierzystej w programie Java należy wykonać następujące kroki:

- 1.** Zadeklarować metodę macierzystą w klasie języka Java.
- 2.** Uruchomić program `javah`, aby uzyskać plik nagłówkowy zawierający deklarację metody w języku C.
- 3.** Zaimplementować metodę macierzystą w języku C.
- 4.** Umieścić kod we współdzielonej bibliotece.
- 5.** Załadować bibliotekę w programie w języku Java.

#### API java.lang.System 1.0

■ `void loadLibrary(String libname)`

Ładuje bibliotekę o podanej nazwie. Biblioteka musi znajdować się w jednym z katalogów określonych przez ścieżkę dostępu do bibliotek. Dokładny sposób ustalenia położenia biblioteki zależy od systemu operacyjnego.



Niektóre z współdzielonych bibliotek zawierających kod macierzysty wymagają wykonania kodu inicjującego. Kod ten umieszczamy wewnątrz metody `JNI_OnLoad`. Podejmie podczas końca pracy maszyny wirtualnej zostanie wywołana metoda `JNI_OnUnLoad`, jeśli tylko ją dostarczymy. Prototypy tych metod przedstawiamy poniżej.

```
jint JNI_OnLoad(JavaVM* vm, void* reserved);
void JNI_OnUnLoad(JavaVM* vm, void* reserved);
```

Metoda `JNI_OnLoad` zwraca najniższą wymaganą wersję maszyny wirtualnej, na przykład `JNI_VERSION_1_2`.

## 12.2. Numeryczne parametry metod i wartości zwracane

Przekazując parametry numeryczne między kodem w języku C i Java, musimy wiedzieć, jakie typy obu języków odpowiadają sobie wzajemnie. Na przykład język C posiada typy `int` i `long`, jednak ich reprezentacja zależy od konkretnej platformy. Na jednych platformach wartości typu `int` reprezentowane są za pomocą 16 bitów, na innych są 32-bitowe. Natomiast na platformie Java wartości typu `int` reprezentowane są *zawsze* za pomocą 32 bitów. Ze względu na te różnice interfejs JNI definiuje więc w języku C odpowiednie typy `jint`, `jlong` itd.

Tabela 12.1 przedstawia typy języka Java i odpowiadające im typy języka C.

**Tabela 12.1.** Typy języka Java i odpowiadające im typy języka C

Język Java	Język C	Bajty
<code>boolean</code>	<code>jboolean</code>	1
<code>byte</code>	<code>jbyte</code>	1
<code>char</code>	<code>jchar</code>	2
<code>short</code>	<code>jshort</code>	2
<code>int</code>	<code>jint</code>	4
<code>long</code>	<code>jlong</code>	8
<code>float</code>	<code>jfloat</code>	4
<code>double</code>	<code>jdouble</code>	8

Typy te zadeklarowane są za pomocą instrukcji `typedef` w pliku nagłówkowym `jni.h` dla danej platformy. Plik ten definiuje także stałe `JNI_FALSE = 0` i `JNI_TRUE = 1`.

## 12.2.1. Wykorzystanie funkcji printf do formatowania liczb

Do momentu udostępnienia Java SE 5.0, platforma Java nie dysponowała funkcją analogiczną do printf w języku C. Założymy na chwilę, że korzystamy z wcześniejszej wersji JDK i chcemy uzyskać podobną funkcjonalność, używając funkcji printf wywoływanej przez metodę macierzystą.

Listing 12.5 prezentuje kod źródłowy klasy Printf1, która korzysta z metody macierzystej do formatowania wyświetlanych liczb zmienoprzecinkowych w polu o podanej szerokości i z podaną dokładnością.

**Listing 12.5.** printf1/Printf1.java

---

```
/*
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */
class Printf1
{
    public static native int print(int width, int precision, double x);

    static
    {
        System.loadLibrary("Printf1");
    }
}
```

---

Zwróćmy uwagę, że, implementując metodę w języku C, musimy zmienić typ parametrów int i double na jint i jdouble, co pokazuje listing 12.6.

**Listing 12.6.** printf1/Printf1.c

---

```
/*
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */

#include "Printf1.h"
#include <stdio.h>

JNIEXPORT jint JNICALL Java_Printf1_print(JNIEnv* env, jclass cl,
    jint width, jint precision, jdouble x)
{
    char fmt[30];
    jint ret;
    sprintf(fmt, "%%.%df", width, precision);
    ret = printf(fmt, x);
    fflush(stdout);
    return ret;
}
```

---

Funkcja Java\_Printf1\_print umieszcza najpierw łańcuch formatujący "%w.pf" w zmiennej fmt, a następnie wywołuje funkcję printf i zwraca liczbę wyświetlonych znaków.

Listing 12.7 zawiera kod źródłowy programu demonstrującego wykorzystanie klasy `Printf1`.

**Listing 12.7.** `printf1/Printf1Test.java`

---

```
/*
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */
class Printf1Test
{
    public static void main(String[] args)
    {
        int count = Printf1.print(8, 4, 3.14);
        count += Printf1.print(8, 4, count);
        System.out.println();
        for (int i = 0; i < count; i++)
            System.out.print("-");
        System.out.println();
    }
}
```

---

## 12.3. Łańcuchy znaków jako parametry

Zajmiemy się teraz omówieniem przekazywania łańcuchów znakowych z metod macierzystych i do nich. Łańcuchy znaków w języku Java są sekwencjami zapisanymi w kodzie UTF-16, natomiast w języku C łańcuchy znaków są sekwencjami bajtów zakończonymi bajtem zero-wym. Reprezentacja łańcuchów znaków w obu językach jest więc różna. Dlatego też interfejs JNI udostępnia dwa zestawy funkcji: jeden, który przekształca łańcuchy języka Java na postać kod „zmodyfikowany UTF-8”, i drugi, który przekształca je na tablice znaków zapisanych w kodzie UTF-16, czyli tablice elementów typu `jchar`. (Kod UTF-8, „zmodyfikowany UTF-8” i UTF-16 zostały omówione w rozdziale 1. Przypomnijmy, że format „zmodyfikowany UTF-8” pozostawia kody znaków ASCII bez zmian, ale znaki należące do Unicode koduje za pomocą sekwencji kilku bajtów).



Standardowy kod UTF-8 i „zmodyfikowany” kod UTF-8 różnią się tylko w przypadku znaków uzupełniających o kodach wyższych od `0xFFFF`. W standardowym kodzie UTF-8 znaki te są kodowane za pomocą 4-bajtowych sekwencji. Natomiast w przypadku kodu „zmodyfikowanego” znak zostaje najpierw zakodowany za pomocą pary surogatów UTF-16, z których każdy zostaje następnie zakodowany za pomocą UTF-8, co w sumie daje 6 bajtów. Przyczyna takiego rozwiązania jest historyczna — specyfikacja JVM powstała, gdy Unicode był ograniczony do 16 bitów.

Jeśli kod w języku C używa już znaków Unicode, to będziemy korzystać z drugiego zestawu funkcji. Jeśli natomiast wszystkie łańcuchy w języku Java ograniczają się wyłącznie do znaków ASCII, to możemy użyć funkcji konwersji do „zmodyfikowanego UTF-8”.

Metoda macierzysta, która posiada parametr typu `String`, otrzymuje w rzeczywistości wartość typu `jstring`. Podobnie jeśli zadeklarowano, że zwraca ona wartość typu `String`, to musi w rzeczywistości zwrócić wartość typu `jstring`. Do tworzenia i odczytu obiektów klasy `jstring`

wykorzystywane są funkcje JNI. Na przykład funkcja NewStringUTF tworzy obiekt klasy `jstring` na podstawie tablicy znaków typu `char` zawierającej znaki w kodzie ASCII lub, ujmując rzecz ogólniej, w „zmodyfikowanym UTF-8”.

Funkcje JNI stosują dość nietypową konwencję wywołań. Poniżej prezentujemy wywołanie funkcji `NewStringUTF`.

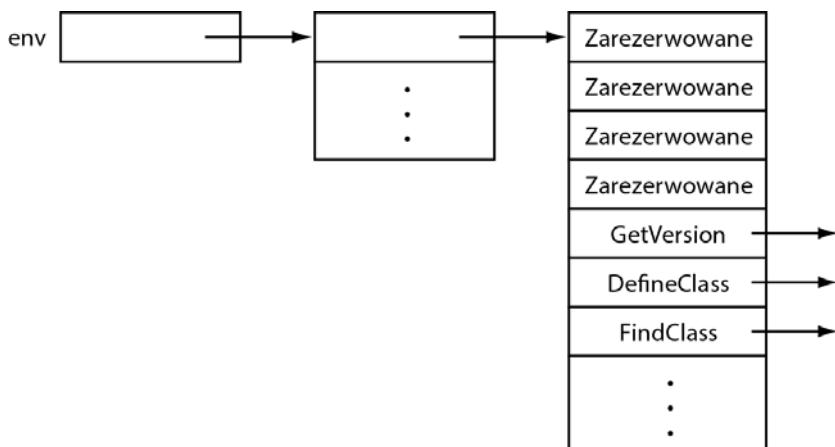
```
JNIEXPORT jstring JNICALL Java_HelloNative_getGreeting
    (JNIEnv* env, jclass cl)
{
    jstring jstr;
    char greeting[] = "Hello, Native World";
    jstr = (*env)->NewStringUTF(env, greeting);
    return jstr;
}
```



Jeśli nie podamy inaczej, to każdy kod przedstawiony w tym rozdziale jest kodem w języku C.

Wszystkie wywołania funkcji JNI wykorzystują wskaźnik `env`, który jest pierwszym parametrem metody macierzystej. Wskaźnik `env` stanowi referencję tablicy wskaźników funkcji (patrz rysunek 12.2). Dlatego też każde wywołanie funkcji JNI musimy poprzedzić przedrostkiem `(*env)->`. Dodatkowo wskaźnik `env` jest także pierwszym parametrem każdej funkcji JNI.

**Rysunek 12.2.**  
Wskaźnik `env`



W języku C++ łatwiej możemy korzystać z funkcji JNI, ponieważ klasa `JNIEnv` w języku C++ posiada wbudowaną metodę, która wyszukuje wskaźnik funkcji. Dzięki temu na przykład funkcję `NewStringUTF` możemy wywołać w następujący sposób:

```
jstr = env->NewStringUTF(greeting);
```

Zwróćmy uwagę, że na liście jej parametrów nie musimy już umieszczać wskaźnika `JNIEnv`.

Funkcja `NewStringUTF` tworzy nowy obiekt klasy `jstring`. Jego zawartość możemy odczytać, korzystając z funkcji `GetStringUTFChars`. Zwraca ona wskaźnik typu `const jbyte*` do znaków łańcucha w „zmodyfikowanym UTF-8”. Zwróćmy uwagę, że specyficzna maszyna

wirtualna może wykorzystywać format UTF do swojej wewnętrznej reprezentacji łańcuchów znaków i wobec tego uzyskany wskaźnik może identyfikować rzeczywisty łańcuch platformy Java. Ponieważ łańcuchy takie są stałe, to specyfikator const należy potraktować poważnie i nie próbować modyfikacji takiego łańcucha. Natomiast jeśli maszyna wirtualna używa do wewnętrznej reprezentacji łańcuchów kodu UTF-16 lub UTF-32, to wywołanie tej funkcji powoduje przydzielenie obszaru pamięci, który wypełniany jest odpowiednikami znaków w „zmodyfikowanym UTF-8”.

Gdy przestajemy korzystać z takiego łańcucha, powinniśmy poinformować o tym maszynę wirtualną, aby mogła zwolnić przydzieloną pamięć. (Procedura odzysku niewykorzystywanych zasobów wykonywana jest w oddzielnym wątku i może przerwać wykonywanie metody macierzystej). W tym celu musimy wywołać więc funkcję `ReleaseStringUTFChars`.

Alternatywnym rozwiązaniem będzie w tym przypadku dostarczenie własnego bufora za pomocą wywołania funkcji `GetStringRegion` lub `GetStringUTFRegion`.

Funkcja `GetStringUTFLength` umożliwia uzyskanie informacji o liczbie znaków potrzebnych do reprezentacji łańcucha w „zmodyfikowanym UTF-8”.



Dokumentacja interfejsu programowego JNI dostępna jest pod adresem <http://docs.oracle.com/javase/7/docs/technotes/guides/jni/>.

#### Dostęp do łańcuchów znakowych platformy Java w języku C

- `jstring NewStringUTF(JNIEnv* env, const char bytes[])`  
zwraca nowy obiekt reprezentujący łańcuch w języku Java utworzony na podstawie łańcucha w „zmodyfikowanym UTF-8” lub wartość NULL, jeśli obiekt nie może być utworzony.
- `jsize GetStringUTFLength(JNIEnv* env, jstring string)`  
zwraca liczbę znaków wymaganych do zakodowania łańcucha w kodzie „zmodyfikowany UTF-8” (bez uwzględnienia zerowego bajtu kończącego łańcucha).
- `const jbyte* GetStringUTFChars(JNIEnv* env, jstring string, jboolean* isCopy)`  
zwraca wskaźnik do reprezentacji łańcucha w kodzie „zmodyfikowany UTF-8” lub wartość NULL, jeśli utworzenie tablicy znaków nie było możliwe. Wskaźnik zachowuje ważność do momentu wywołania metody `ReleaseStringUTFChars`. Parametr `isCopy` wskazuje obiekt `jboolean` posiadający wartość `JNI_TRUE`, jeśli funkcja wykonała kopię oryginalnej reprezentacji łańcucha, a wartość `JNI_FALSE` w przeciwnym razie.
- `void ReleaseStringUTFChars(JNIEnv* env, jstring string, const jbyte bytes[])`  
informuje maszynę wirtualną, że kod macierzysty nie będzie już korzystał z reprezentacji łańcucha umieszczonej w tablicy `bytes` (wskaźnik zwrócony przez funkcję `GetStringUTFChars`).

- void GetStringRegion(JNIEnv\* env, jstring string, jsize start, jsize length, jchar\* buffer)
 

kopiuje sekwencję znaków Unicode łańcucha do bufora dostarczonego przez użytkownika o rozmiarze równym co najmniej  $2 \times \text{length}$ .
- void GetStringUTFRegion(JNIEnv\* env, jstring string, jsize start, jsize length, jchar\* buffer)
 

kopiuje sekwencję znaków łańcucha w kodzie „zmodyfikowany UTF-8” do bufora dostarczonego przez użytkownika. Bufor ten musi mieć wystarczający rozmiar, w najgorszym przypadku równy  $3 \times \text{length}$  bajtów.
- jstring NewString(JNIEnv\* env, const jchar chars[], jsize length)
 

zwraca nowy obiekt łańcucha platformy Java na podstawie dostarczonego łańcucha Unicode lub wartość NULL, jeśli utworzenie obiektu nie powiodło się.

*Parametry:* env wskaźnik interfejsu JNI,  
chars łańcuch wyjściowy,  
length liczba znaków łańcucha.
- jsize GetStringLength(JNIEnv\* env, jstring string)
 

zwraca liczbę znaków łańcucha.
- const jchar\* GetStringChars(JNIEnv\* env, jstring string, jboolean\* isCopy)
 

zwraca wskaźnik do reprezentacji łańcucha w kodzie Unicode lub wartość NULL, jeśli utworzenie tablicy nie było możliwe. Wskaźnik zachowuje ważność do momentu wywołania metody ReleaseStringChars. Parametr isCopy może mieć wartość NULL lub wskazywać obiekt jboolean posiadający wartość JNI\_TRUE, jeśli funkcja wykonała kopię oryginalnej reprezentacji łańcucha i wartość JNI\_FALSE, w przeciwnym razie.
- void ReleaseStringChars(JNIEnv\* env, jstring string, const jchar chars[])
 

informuje maszynę wirtualną, że kod macierzysty nie będzie już korzystał z reprezentacji łańcucha umieszczonej w tablicy chars (wskaźnik zwrocony przez funkcję GetStringChars).

Wykorzystamy teraz przedstawione w poprzednim podrozdziale funkcje interfejsu JNI do utworzenia klasy wywołującej funkcję sprintf języka C. Będziemy z niej korzystać w sposób zaprezentowany w listingu 12.8.

**Listing 12.8.** printf2/Printf2Test.java

```
/*
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */
class Printf2Test
{
    public static void main(String[] args)
    {
        double price = 44.95;
        double tax = 7.75;
```

```
        double amountDue = price * (1 + tax / 100);

        String s = Printf2.sprint("Amount due = %8.2f", amountDue);
        System.out.println(s);
    }
}
```

---

Listing 12.9 prezentuje kod źródłowy klasy zawierającej metodę macierzystą `sprint`.

---

**Listing 12.9.** *printf2/Printf2.java*

```
/*
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */
class Printf2
{
    public static native String sprint(String format, double x);

    static
    {
        System.loadLibrary("Printf2");
    }
}
```

---

Funkcja języka C formatująca liczbę zmiennoprzecinkową musi więc posiadać prototyp w postaci

```
JNIEXPORT jstring JNICALL Java_Printf2_sprint
    (JNIEnv* env, jclass cl, jstring format, jdouble x)
```

Listing 12.10 pokazuje jej implementację. Wykorzystuje ona funkcję `GetStringUTFChars` do odczytania parametru określającego format, funkcję `NewStringUTF` do utworzenia zwracanej wartości oraz funkcję `ReleaseStringUTFChars` do poinformowania maszyny wirtualnej, że dostęp do reprezentacji łańcucha nie będzie już potrzebny.

---

**Listing 12.10.** *printf2/Printf2.c*

```
/*
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */

#include "Printf2.h"
#include <string.h>
#include <stdlib.h>
#include <float.h>

/***
 * @param format łańcuch określający sposób formatowania
 * (na przykład "%8.2f"). Podlańcuchy "%" są pomijane.
 * @return wskaźnik specyfikatora formatu (z pominięciem znaku '%')
 * lub wartość NULL, jeśli nie został odnaleziony.
 */
char* find_format(const char format[])
{

```

```

char* p;
char* q;

p = strchr(format, '%');
while (p != NULL && *(p + 1) == '%' /*pomija %% */)
    p = strchr(p + 2, '%');
if (p == NULL) return NULL;
/* sprawdza czy % jest unikalny */
p++;
q = strchr(p, '%');
while (q != NULL && *(q + 1) == '%' /*pomija %% */)
    q = strchr(q + 2, '%');
if (q != NULL) return NULL; /*% nie jest unikalny */
q = p + strspn(p, "-0#+"); /*pomija znaczniki */
q += strspn(q, "0123456789"); /*pomija specyfikację szerokości */
if (*q == '.') { q++; q += strspn(q, "0123456789"); }
/*pomija specyfikację precyzji */
if (strchr("eEfFgG", *q) == NULL) return NULL;
/*to nie jest format zmiennoprzecinkowy */
return p;
}

JNICALL jstring Java_Printf2_sprint(JNIEnv* env, jclass cl,
jstring format, jdouble x)
{
    const char* cformat;
    char* fmt;
    jstring ret;

    cformat = (*env)->GetStringUTFChars(env, format, NULL);
    fmt = find_format(cformat);
    if (fmt == NULL)
        ret = format;
    else
    {
        char* cret;
        int width = atoi(fmt);
        if (width == 0) width = DBL_DIG + 10;
        cret = (char*) malloc(strlen(cformat) + width);
        sprintf(cret, cformat, x);
        ret = (*env)->NewStringUTF(env, cret);
        free(cret);
    }
    (*env)->ReleaseStringUTFChars(env, format, cformat);
    return ret;
}

```

W przypadku powyższej funkcji uprościliśmy zdecydowanie obsługę błędów. Jeśli format wyświetlanej liczby zmiennoprzecinkowej nie jest w postaci `%w.pc`, gdzie `c` jest jednym ze znaków `e`, `E`, `f`, `g` lub `G`, to po prostu *nie* formatujemy tej liczby. W dalszej części rozdziału pokażemy, w jaki sposób metody macierzyste mogą wyrzucać wyjątki.

## 12.4. Dostęp do składowych obiektu

Dotychczas pokazaliśmy jedynie przykłady implementacji statycznych metod macierzystych, których parametrami były wartości liczbowe i łańcuchy znaków. Przejdziemy teraz do omówienia metod macierzystych, które działają na obiektach. Dla ilustracji zagadnienia zaimplementujemy metodę klasy Employee wprowadzonej w 4. rozdziale książki *Java 2. Podstawy*, korzystając z metody macierzystej. Oczywiście w praktyce nie implementowalibyśmy metody w taki sposób, jednak przykład ten ma ilustrować sposób dostępu do składowych obiektu z metody macierzystej, który może okazać się niezbędny w innych przypadkach.

### 12.4.1. Dostęp do pól instancji

Aby zilustrować sposób dostępu do pól instancji przez metodę macierzystą, zaimplementujemy raz jeszcze metodę `raiseSalary`. Jej implementacja w języku Java była bardzo prosta.

```
public void raiseSalary(double byPercent)
{
    salary *= 1 + byPercent / 100;
}
```

Metodę tę zaimplementujemy teraz jako metodę macierzystą. W przeciwieństwie do poprzednich przykładów metoda ta nie będzie metodą statyczną. Jeśli uruchomimy program `javad`, to uzyskamy poniższy prototyp.

```
JNIEXPORT void JNICALL Java_Employee_raiseSalary
(JNIEnv *, jobject, jdouble);
```

Zwrócmy uwagę na drugi parametr. Nie jest on już typu `jclass`, lecz typu `jobject`. Parametr ten stanowi odpowiednik referencji `this`. Metody statyczne uzyskują referencję do klasy, a pozostałe metody — referencję `this` do obiektu, na rzecz którego są wywoływanie (ukrytego parametru metody).

Ukryty parametr metody wykorzystamy w celu dostępu do składowej `salary` obiektu. W wersji Java 1.0 powiązanie języka Java i C umożliwiał bezpośredni dostęp do składowych obiektu. Wymagało to jednak od maszyny wirtualnej udostępnienia jej wewnętrznych danych. Z tego powodu interfejs JNI wymaga od programistów korzystania ze składowych obiektu za pośrednictwem specjalnych funkcji.

W naszym przykładzie musimy skorzystać z funkcji `GetDoubleField` i `SetDoubleField`, ponieważ składowa `salary` jest typu `double`. Dla składowych innych typów istnieją też odpowiednie funkcje — `GetIntField/GetIntField`, `GetObjectField/SetObjectField` itd. Składnia ich wywołań przedstawia się następująco:

```
x = (*env)->GetXxxField(env, this_obj, fieldID);
(*env)->SetXxxField(env, this_obj, fieldID, x);
```

Parametr `fieldID` jest specjalnego typu `jfieldID`, który identyfikuje składową w strukturze klasy. łańcuch `Xxx` reprezentuje w tym przypadku typ języka Java (`Object`, `Boolean`, `Byte` itd.).

Aby uzyskać fieldID, musimy najpierw pobrać wartość reprezentującą klasę, co możemy osiągnąć na dwa sposoby. Funkcja GetObjectClass zwraca klasę dowolnego obiektu, na przykład:

```
jclass class_Employee = (*env)->GetObjectClass(env, this_obj);
```

Funkcja FindClass pozwala wyspecyfikować nazwę klasy za pomocą łańcucha znaków (zastępując znak kropki w nazwie pakietu znakiem ukośnika).

```
jclass class_String  
= (*env)->FindClass(env, "java/lang/String");
```

Aby uzyskać parametr fieldID, skorzystamy z funkcji GetFieldID. W tym celu musimy przekazać jej nazwę składowej oraz jej *sygnature*, czyli zakodowany typ. Poniżej przedstawiamy sposób uzyskania identyfikatora składowej salary obiektu klasy Employee.

```
jfieldID id_salary  
= (*env)->GetFieldID(env, class_Employee, "salary", "D");
```

Łańcuch "D" oznacza typ double. Reguły tworzenia sygnatur przedstawimy w następnym podrozdziale.

Sposób dostępu do składowych obiektów może wydawać się niepotrzebnie skomplikowany. Jednak chcąc uniknąć bezpośredniego udostępniania składowych obiektów, projektanci interfejsu JNI musieli wprowadzić funkcje pośredniczące. Aby ograniczyć koszt wykonania tych funkcji, wydzielono z nich najkosztowniejszą operację, którą jest wyznaczenie identyfikatora składowej na podstawie jego nazwy. Dzięki temu jeśli wielokrotnie wywołujemy, pobieramy i nadajemy wartość tej samej składowej, to jej identyfikator wyznaczany jest tylko jeden raz.

Zbierzmy uzyskane dotąd informacje. Poniższy kod prezentuje nową implementację metody raiseSalary jako metody macierzystej.

```
JNIEXPORT void JNICALL Java_Employee_raiseSalary  
(JNIEnv* env, jobject this_obj, jdouble byPercent);  
{  
    /* ustala klasę */  
    jclass class_Employee = (*env)->GetObjectClass(env,  
        this_obj);  
  
    /* pobiera identyfikator składowej */  
    jfieldID id_salary = (*env)->GetFieldID(env, class_Employee,  
        "salary", "D");  
  
    /* pobiera wartość składowej */  
    jdouble salary = (*env)->GetDoubleField(env, this_obj,  
        id_salary);  
  
    salary *= 1 + byPercent / 100;  
  
    /* nadaje wartość składowej */  
    (*env)->SetDoubleField(env, this_obj, id_salary, salary);  
}
```



Referencje klas tracą ważność po zakończeniu wykonywania metody macierzystej. Nie należy więc w kodzie programu przechowywać wartości zwracanych przez funkcję GetObjectClass w celu późniejszego ich wykorzystania. Funkcję GetObjectClass musimy wywoływać za każdym razem, gdy wykonywana jest metoda macierzysta. Jeśli z pewnego powodu jest to niemożliwe, to możemy zablokować referencję klasy za pomocą wywołania funkcji NewGlobalRef:

```
static jclass class_X = 0;
static jfieldID id_a;

{
    jclass cx = (*env)->GetObjectClass(env, obj);
    class_X = (*env)->NewGlobalRef(env, cx);
    id_a = (*env)->GetFieldID(env, cls, "a", "...");
}
```

Dzięki temu możemy wykorzystywać teraz referencję klasy oraz identyfikator składowej w kolejnych wywołaniach. Po ich wykonaniu należy pamiętać o poniższym wywołaniu:

```
(*env)->DeleteGlobalRef(env, class_X);
```

Listingi 12.11 i 12.12 prezentują kod źródłowy programu testowego w języku Java oraz klasy Employee. Listing 12.13 zawiera kod źródłowy metody macierzystej raiseSalary w języku C.

**Listing 12.11.** employee/EmployeeTest.java

```
/**
 * @version 1.10 1999-11-13
 * @author Cay Horstmann
 */

public class EmployeeTest
{
    public static void main(String[] args)
    {

        Employee[] staff = new Employee[3];

        staff[0] = new Employee("Harry Hacker", 35000);
        staff[1] = new Employee("Carl Cracker", 75000);
        staff[2] = new Employee("Tony Tester", 38000);

        for (Employee e : staff)
            e.raiseSalary(5);
        for (Employee e : staff)
            e.print();
    }
}
```

**Listing 12.12.** employee/Employee.java

```
/**
 * @version 1.10 1999-11-13
 * @author Cay Horstmann
 */
```

---

```

public class Employee
{
    private String name;
    private double salary;

    public native void raiseSalary(double byPercent);

    public Employee(String n, double s)
    {
        name = n;
        salary = s;
    }

    public void print()
    {
        System.out.println(name + " " + salary);
    }

    static
    {
        System.loadLibrary("Employee");
    }
}

```

---

**Listing 12.13.** employee/Employee.c

---

```

/*
 * @version 1.10 1999-11-13
 * @author Cay Horstmann
 */

#include "Employee.h"

#include <stdio.h>

JNIEXPORT void JNICALL Java_Employee_raiseSalary(JNIEnv* env, jobject this_obj,
                                                 jdouble byPercent)
{
    /* ustala klasę */
    jclass class_Employee = (*env)->GetObjectClass(env, this_obj);

    /* pobiera identyfikator pola */
    jfieldID id_salary = (*env)->GetFieldID(env, class_Employee, "salary", "D");

    /* pobiera wartość pola */
    jdouble salary = (*env)->GetDoubleField(env, this_obj, id_salary);

    salary *= 1 + byPercent / 100;

    /* nadaje wartość polu */
    (*env)->SetDoubleField(env, this_obj, id_salary, salary);
}


```

---

## 12.4.2. Dostęp do pól statycznych

Dostęp do pól statycznych odbywa się w podobny sposób jak dostęp do zwykłych pól obiektów. Korzystamy w tym celu z funkcji GetStaticFieldID i GetStaticXxxField/SetStaticXxxField. Działają one prawie identycznie, jak ich odpowiedniki dla zwykłych pól obiektów, ale z dwoma wyjątkami.

- Ponieważ funkcje te nie działają na obiektach, to zamiast funkcji GetObjectClass korzystamy z funkcji FindClass, która zwraca referencję klasy,
- Stosując funkcję dostępu do pól statycznych, podajemy klasę, a nie obiekt.

Poniżej prezentujemy przykład dostępu do pola System.out.

```
/* ustala klasę */
jclass class_System = (*env)->FindClass(env,
    "java/lang/System");

/* pobiera identyfikator pola */
jfieldID id_out = (*env)->GetStaticFieldID(env,
    class_System, "out", "Ljava/io/PrintStream; ");

/* pobiera wartość pola */
jobject obj_out = (*env)->GetStaticObjectField(env,
    class_System, id_out);
```

### Dostęp do pól instancji

- jfieldID GetFieldID(JNIEnv \*env, jclass cl, const char name[], const char sig[])
 zwraca identyfikator pola.
- Xxx GetXxxField(JNIEnv \*env, jobject jobj, jfieldID id)
 zwraca wartość pola. Typem pola Xxx jest Object, Boolean, Byte, Char, Short, Int, Long, Float lub Double.
- void SetXxxField(JNIEnv \*env, jobject jobj, jfieldID id, Xxx value)
 nadaje nową wartość polu. Typem pola Xxx jest Object, Boolean, Byte, Char, Short, Int, Long, Float lub Double.
- jfieldID GetStaticFieldID(JNIEnv \*env, jclass cl, const char name[], const char sig[])
 zwraca identyfikator statycznej składowej klasy.
- Xxx GetStaticXxxField(JNIEnv \*env, jclass cl, jfieldID id)
 zwraca wartość statycznego pola. Typem pola Xxx jest Object, Boolean, Byte, Char, Short, Int, Long, Float lub Double.
- void SetXxxField(JNIEnv \*env, jclass cl, jfieldID id, Xxx value)
 nadaje nową wartość statycznemu polu. Typem składowej Xxx jest Object, Boolean, Byte, Char, Short, Int, Long, Float lub Double.

## 12.5. Sygnatury

Aby tworzone przez nas metody macierzyste mogły korzystać z pól obiektów platformy Java oraz wywoływać metody tych obiektów, musimy najpierw nauczyć się posługiwać się sygnaturami typów i metod (sygnatury metody opisują jej zestaw parametrów oraz zwracaną wartość). Poniżej przedstawiamy sposób kodowania typów języka Java:

```
B byte
C char
D double
F float
I int
J long
Lclassname; typ będący klasą
S short
V void
Z boolean
```

Typ będący tablicą zapisujemy za pomocą [. Na przykład tablicę łańcuchów znakowych będziemy prezentować jako:

[Ljava/lang/String;

Natomiast tablica float[][] zostanie zapisana jako:

[[F

Sygnatura metody składa się z listy sygnatur jej parametrów ujętej w nawiasy, po której następuje sygnatura typu zwracanego przez tę metodę. Sygnatura metody, której przekazujemy dwa parametry typu int i która zwraca wartość tego typu, będzie miała postać:

(II)I

Metoda printf, którą wykorzystaliśmy w poprzednim przykładzie, będzie miała następującą sygnaturę:

(Ljava/lang/String;)V

Oznacza ona, że jedynym parametrem metody jest łańcuch znaków, a typem metody jest void.

Zwróćmy uwagę, że znak średnika na końcu wyrażenia L kończy określenie typu, a nie jest separatorem między parametrami. Na przykład konstruktor

Employee(java.lang.String, double, java.util.Date)

będzie posiadać sygnaturę w postaci

"(Ljava/lang/String;/;DLjava/util/Date;)V"

Jak łatwo zauważyć, między sygnaturami D oraz Ljava/util/Date; nie występuje żaden znak separatora. Zwróćmy także uwagę, że, tworząc sygnatury, musimy zastąpić znak kropki występujący w nazwach pakietów znakiem ukośnika. Znak V kończący sygnaturę oznacza, że metoda zwraca wartość void. Chociaż w przypadku konstruktorów w języku Java nie określamy zwracanego typu, to jednak na końcu sygnatury musimy umieścić znak V.



Sygnatury składowych i metod klas możemy uzyskać, wywołując program `javap` z opcją `-s` dla pliku klasy. Jeśli więc uruchomimy go w następujący sposób

```
javap -s -private Employee
```

to uzyskamy informację o sygnaturach pól i metod klasy podobną do pokazanej poniżej.

```
Compiled from "Employee.java"
```

```
public class Employee extends java.lang.Object{
    private java.lang.String name;
        Signature: Ljava/lang/String;
    private double salary;
        Signature: D
    public Employee(java.lang.String, double)
        Signature: (Ljava/lang/String/D)V
    public native void raiseSalary(double);
        Signature: (D)V
    public void print():
        Signature: ()V
    static {};
        Signature: ()V
}
```



Nie istnieje żaden racjonalny powód, dla którego zmusza się programistów do stosowania omówionego sposobu kodowania sygnatur. Projektanci interfejsu JNI powinni byli udostępnić funkcję, która wczytywałaby sygnatury zgodne z konwencją języka Java, na przykład w postaci `void(int.java.lang.String)`, a następnie przekodowałaby je na odpowiednią reprezentację wewnętrzną. Po raz kolejny jednak programiści zostali niepotrzebnie zmuszeni do posługiwania się szczegółami związanymi z implementacją maszyny wirtualnej.

## 12.6. Wywoływanie metod języka Java

Jak już pokazaliśmy, metody języka Java mogą wywoływać funkcje języka C — na tym polega działanie metod macierzystych. Czy jest to możliwe w przeciwnym kierunku? A jeśli tak, to po co? Okazuje się, że w praktyce metody macierzyste często muszą żądać pewnej usługi od przekazanego im obiektu. Wywoływanie metod języka Java omówimy najpierw dla zwykłych metod, a później dla metod statycznych.

### 12.6.1. Wywoływanie metod obiektów

Dla zilustrowania sposobu wywoływania metody języka Java przez kod macierzysty przykładową klasę `Printf` rozszerzymy o metodę działającą podobnie do funkcji `fprintf` w języku C. Będzie ona drukować łańcuch znaków, korzystając z dowolnego obiektu `PrintWriter`.

```
class Printf3
{
    public native static void fprintf(PrintWriter out,
        String s, double x);
    . . .
}
```

Utworzymy najpierw docelowy łańcuch str klasy String, korzystając z funkcji sprintf, a następnie wywołamy metodę print klasy PrintWriter z wnętrza funkcji języka C implementującej metodę macierzystą.

W języku C możemy wywołać dowolna metodę języka Java, wywołując funkcję

```
(*env)->CallXxxMethod(env, parametr domyślny, IDmetody,
parametry jawnie)
```

Łańcuch Xxx należy zastąpić przez Void, Int, Object itd., w zależności od typu zwracanego przez wywoływaną metodę. Podobnie jak w przypadku składowych potrzebowaliśmy identyfikatora fieldID, aby wywołać metodę obiektu, tak teraz musimy uzyskać jej identyfikator. W tym celu wywołamy funkcję GetMethodID, której przekażemy klasę, nazwę i sygnaturę metody.

W naszym przykładzie musimy uzyskać identyfikator metody print klasy PrintWriter. Z rozdziału 12. książki *Java 2. Podstawy* pamiętamy, że klasa PrintWriter posiada kilka różnych metod o nazwie print. Dlatego też musimy podać sygnaturę metody, którą chcemy wykorzystać. Będzie to metoda void print(java.lang.String) i zgodnie z tym, co napisaliśmy w poprzednim podrozdziale, będzie ona posiadać sygnaturę "(Ljava/lang/String;)V".

Poniżej prezentujemy kod, który w celu wywołania metody języka Java:

- 1.** Uzyskuje klasę obiektu, na rzecz którego będzie wywołana metoda.
- 2.** Uzyskuje identyfikator metody.
- 3.** Wywołuje metodę, korzystając z funkcji CallXxxMethod.

```
/* ustala klasę */
class_PrintWriter = (*env)->GetObjectClass(env, out);

/* pobiera identyfikator metody */
id_print = (*env)->GetMethodID(env, class_PrintWriter,
    "print", "(Ljava/lang/String;)V");

/* wywołuje metodę */
(*env)->CallVoidMethod(env, out, id_print, str);
```

Listingi 12.14 i 12.15 prezentują kod źródłowy w języku Java programu testowego i klasy Printf3. Listing 12.16 zawiera kod metody macierzystej fprintf w języku C.

#### **Listing 12.14. printf3/Printf3Test.java**

```
import java.io.*;

/**
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */
class Printf3Test
{
    public static void main(String[] args)
    {
        double price = 44.95;
        double tax = 7.75;
        double amountDue = price * (1 + tax / 100);
```

```
    PrintWriter out = new PrintWriter(System.out);
    printf3.fprint(out, "Amount due = %8.2f\n", amountDue);
    out.flush();
}
}
```

---

**Listing 12.15.** printf3/Printf3.java

```
import java.io.*;

/**
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */
class Printf3
{
    public static native void fprintf(PrintWriter out, String format, double x);

    static
    {
        System.loadLibrary("Printf3");
    }
}
```

---

**Listing 12.16.** printf3/Printf3.c

```
/**
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */

#include "Printf3.h"
#include <string.h>
#include <stdlib.h>
#include <float.h>

/***
 * @param format łańcuch określający sposób formatowania
 * (na przykład "%8.2f"). Podłańcuchy "%%%" są pomijane.
 * @return wskaźnik specyfikatora formatu (z pominięciem znaku '%')
 * lub wartość NULL, jeśli nie został odnaleziony.
 */
char* find_format(const char format[])
{
    char* p;
    char* q;

    p = strchr(format, '%');
    while (p != NULL && *(p + 1) == '%') /*pomija %%*/
        p = strchr(p + 2, '%');
    if (p == NULL) return NULL;
    /*sprawdza czy % jest unikalny*/
    p++;
    q = strchr(p, '%');
    while (q != NULL && *(q + 1) == '%') /*pomija %%*/
        q = strchr(q + 2, '%');
    if (q != NULL) return NULL; /*% nie jest unikalny*/
}
```

```

q = p + strspn(p, " -0+#"); /* pomija znaczniki */
q += strspn(q, "0123456789"); /* i specyfikację szerokości pola */
if (*q == '.') { q++; q += strspn(q, "0123456789"); }
/* pomija specyfikację precyzji */
if (strchr("eEfFgG", *q) == NULL) return NULL;
/* to nie jest format zmiennoprzecinkowy */
return p;
}

JNIEXPORT void JNICALL Java_Printf3_fprint(JNIEnv* env, jclass cl,
 jobject out, jstring format, jdouble x)
{
    const char* cformat;
    char* fmt;
    jstring str;
    jclass class_PrintWriter;
    jmethodID id_print;

    cformat = (*env)->GetStringUTFChars(env, format, NULL);
    fmt = find_format(cformat);
    if (fmt == NULL)
        str = format;
    else
    {
        char* cstr;
        int width = atoi(fmt);
        if (width == 0) width = DBL_DIG + 10;
        cstr = (char*) malloc(strlen(cformat) + width);
        sprintf(cstr, cformat, x);
        str = (*env)->NewStringUTF(env, cstr);
        free(cstr);
    }
    (*env)->ReleaseStringUTFChars(env, format, cformat);

    /* wywołuje metodę ps.print(str) */

    /* ustala klasę */
    class_PrintWriter = (*env)->GetObjectClass(env, out);

    /* pobiera identyfikator metody */
    id_print = (*env)->GetMethodID(env, class_PrintWriter, "print",
        "(Ljava/lang/String;)V");

    /* wywołuje metodę */
    (*env)->CallVoidMethod(env, out, id_print, str);
}

```



Identyfikatory składowych i metod wykorzystywane przez JNI są koncepcyjnie podobne do obiektów Method i Field wykorzystywanych przez mechanizm refleksji. Możemy dokonywać konwersji między nimi, posługując się następującymi funkcjami:

```

jobject ToReflectedMethod(JNIEnv* env, jclass class,
    jmethodID methodID); // zwraca obiekt Method
methodID FromReflectedMethod(JNIEnv* env, jobject method);
jobject ToReflectedField(JNIEnv* env, jclass class,
    jfieldID fieldID); // zwraca obiekt Field
fieldID FromReflectedField(JNIEnv* env, jobject field);

```

## 12.6.2. Wywoływanie metod statycznych

Wywoływanie metod statycznych przypomina wywoływanie metod obiektów. Istnieją jednak dwie różnice:

- wykorzystujemy funkcje `GetStaticMethodID` i `CallStaticXxxMethod`,
- wywołując metodę, dostarczamy obiekt reprezentujący klasę, a nie obiekt, na rzecz którego wywoływana jest metoda.

Dla ilustracji zagadnienia spróbujmy wywołać z metody macierzystej metodę statyczną

```
System.getProperty("java.class.path")
```

Wywołanie to zwróci łańcuch określający ścieżkę dostępu do bieżącej klasy.

Najpierw musimy ustalić klasę metody. Ponieważ nie jest łatwo dostępny żaden obiekt klasy `System`, to skorzystamy z metody `FindClass` zamiast `GetObjectClass`.

```
jclass class_System = (*env)->FindClass(env, "java/lang/System");
```

Następnie pobierzemy identyfikator metody statycznej `getProperty`, której sygnatura ma pokazaną poniżej postać (zarówno jej parametr, jak i zwracana wartość są typu `String`).

```
"(Ljava/lang/String;)Ljava/lang/String;"
```

Identyfikator metody uzyskamy więc w następujący sposób:

```
jmethodID id_getProperty = (*env)->GetStaticMethodID(env,
    class_System, "getProperty",
    "(Ljava/lang/String;)Ljava/lang/String;");
```

Możemy już wywołać metodę statyczną `getProperty`. Zwróćmy uwagę, że funkcji `CallStaticObjectMethod` przekazujemy obiekt reprezentujący klasę.

```
jobject obj_ret = (*env)->CallStaticObjectMethod(env,
    class_System, id_getProperty,
    (*env)->NewStringUTF(env, "java.class.path"));
```

Rezultat tego wywołania jest typu `jobject`. Jeśli chcemy manipulować nim jak łańcuchem znaków, to musimy rzutować go na typ `jstring`:

```
jstring str_ret = (jstring) obj_ret;
```



W języku C typy `jstring`, `jclass` oraz typy tablic, które omówimy wkrótce, stanowią ekwiwalent typu `jobject` i dlatego pokazane powyżej rzutowanie nie jest konieczne. Jednak w języku C++ typy te zdefiniowane są jako wskaźniki do klas posiadających określone miejsce w hierarchii dziedziczenia. Dlatego też w języku C++ podstawienie typu `jstring` za typ `jobject` jest dozwolone bez rzutowania, ale podstawienie typu `jobject` za typ `jstring` wymaga wykonania rzutowania.

## 12.6.3. Konstruktory

Metoda macierzysta może utworzyć nowy obiekt platformy Java, wywołując konstruktor danej klasy za pomocą funkcji NewObject.

```
jobject obj_new = (*env)->NewObject(env, klasa, IDmetody, parametry konstruktora);
```

Identyfikator konstruktora będący jednym z parametrów powyższego wywołania uzyskujemy, wywołując funkcję GetMethodID, której przekazujemy nazwę metody w postaci łańcucha "<init>" oraz sygnaturę konstruktora (o zwracanym typie void). Poniżej przedstawiamy sposób utworzenia obiektu klasy FileOutputStream przez metodę macierzystą.

```
const char[] fileName = ". . .";
jstring str_fileName = (*env)->NewStringUTF(env, fileName);
jclass class_FileOutputStream = (*env)->FindClass(env,
    "java/io/FileOutputStream");
jmethodID id_FileOutputStream = (*env)->GetMethodID(env,
    class_FileOutputStream, "<init>", "(Ljava/lang/String;)V");
jobject obj_stream = (*env)->NewObject(env,
    class_FileOutputStream, id_FileOutputStream, str_fileName);
```

Zwróćmy uwagę, że sygnatura konstruktora oznacza, iż typem jego parametru jest java.lang.String, a typem zwracanej wartości — void.

## 12.6.4. Alternatywne sposoby wywoływania metod

Interfejs JNI udostępnia wiele wariantów funkcji służących do wywoływanego metod języka Java z kodu macierzystego. Nie są one tak często wykorzystywane, jak funkcje, które dotąd omówiliśmy, ale czasami bywają przydatne.

Funkcje CallNonvirtualXxxMethod otrzymują jako parametr obiekt, na rzecz którego wywoływana jest metoda, identyfikator metody, obiekt reprezentujący klasę bazową (obiektu, na rzecz którego wywoływana jest metoda) oraz parametry wywołania metody. Funkcje te wywołują wersję metody znajdującej się w podanej klasie z pominięciem zwykłego mechanizmu dynamicznego wywoływanego metod języka Java.

Wszystkie funkcje wywołań posiadają wersje, których nazwy zakończone są przyrostkami „A” i „V” i które otrzymują parametry wywołania metody za pośrednictwem tablicy lub va\_list (zdefiniowanej w pliku nagłówkowym *stdarg.h* języka C).

### API Wywoływanie metod języka Java z języka C

- jmethodID GetMethodID(JNIEnv \*env, jclass cl, const char name[], const char sig[])
   
zwraca identyfikator metody.
- void CallXxxMethod(JNIEnv \*env, jobject obj, jmethodID id, args)
   
■ void CallXxxMethodA(JNIEnv \*env, jobject obj, jmethodID id, jvalue args[])

- void CallXxxMethodV(JNIEnv \*env, jobject obj, jmethodID id, va\_list args)

Wywołują metodę języka Java, która zwraca typ określony przez łańcuch *Xxx* jako Object, Boolean, Byte, Char, Short, Int, Long, Float lub Double. Pierwsza z funkcji posiada zmienną liczbę parametrów, które umieszczamy po identyfikatorze metody. Druga z funkcji otrzymuje parametry metody w tablicy elementów typu *jvalue*, który zdefiniowany jest następująco:

```
typedef union jvalue
{
    jboolean z;
    jbyte b;
    jchar c;
    jshort s;
    jint i;
    jlong j;
    jfloat f;
    jdouble d;
    jobject l;
} jvalue;
```

Trzecia z funkcji dostaje parametry metody za pośrednictwem *va\_list* zdefiniowanej w pliku nagłówkowym *stdarg.h* języka C.

- void CallNonVirtualXxxMethod(JNIEnv \*env, jobject obj, jclass cl, jmethodID id, args)
- void CallNonVirtualXxxMethodA(JNIEnv \*env, jobject obj, jclass cl, jmethodID id, jvalue args[])
- void CallNonVirtualXxxMethodV(JNIEnv \*env, jobject obj, jclass cl, jmethodID id, va\_list args)

Wywołują metodę języka Java z pominięciem dynamicznego mechanizmu wywołania języka Java. łańcuch *Xxx* oznacza typ zwracany przez metodę: Object, Boolean, Byte, Char, Short, Int, Long, Float lub Double. Pierwsza z funkcji posiada zmienną liczbę parametrów, które umieszczamy po identyfikatorze metody. Druga z funkcji otrzymuje parametry metody w tablicy elementów typu *jvalue*, a trzecia — za pośrednictwem *va\_list* zdefiniowanej w pliku nagłówkowym *stdarg.h* języka C.

- jmethodID GetStaticMethodID(JNIEnv \*env, jclass cl, const char name[], const char sig[])
- zwraca identyfikator metody statycznej.
- void CallStaticXxxMethod(JNIEnv \*env, jclass cl, jmethodID id, args)
  - void CallStaticXxxMethodA(JNIEnv \*env, jclass cl, jmethodID id, jvalue args[])
  - void CallStaticXxxMethodV(JNIEnv \*env, jclass cl, jmethodID id, va\_list args)

Wywołują statyczną metodę języka Java, która zwraca typ określony przez łańcuch *Xxx* jako Object, Boolean, Byte, Char, Short, Int, Long, Float lub Double. Pierwsza z funkcji posiada zmienną liczbę parametrów, które umieszczamy po identyfikatorze metody. Druga z funkcji otrzymuje parametry metody w tablicy elementów typu *jvalue*, a trzecia — za pośrednictwem *va\_list* zdefiniowanej w pliku nagłówkowym *stdarg.h* języka C.

- jobject NewObject(JNIEnv \*env, jclass cl, jmethodID id, args)
- jobject NewObjectA(JNIEnv \*env, jclass cl, jmethodID id, jvalue args[])
- jobject NewObjectV(JNIEnv \*env, jclass cl, jmethodID id, va\_list args)

Wywołują konstruktor. Identyfikator konstruktora uzyskujemy, wywołując funkcję GetMethodID, której przekazujemy nazwę metody w postaci łańcucha "<init>" oraz sygnaturę konstruktora (o zwracanym typie void). Pierwsza z funkcji posiada zmienną liczbę parametrów, które umieszczamy po identyfikatorze metody. Druga z funkcji otrzymuje parametry metody w tablicy elementów typu jvalue, a trzecia — za pośrednictwem va\_list zdefiniowanej w pliku nagłówkowym *stdarg.h* języka C.

## 12.7. Tablice

Wszystkie typy tablic w języku Java mają odpowiadające im typy w języku C. Prezentujemy je w tabeli 12.2.

**Tabela 12.2.** Typy tablic w języku Java i odpowiadające im typy języka C

Typ w języku Java	Typ w języku C
boolean[]	jbooleanArray
byte[]	jbyteArray
char[]	jcharArray
int[]	jintArray
short[]	jshortArray
long[]	jlongArray
float[]	jfloatArray
double[]	jdoubleArray
Object[]	jobjectArray



W języku C wszystkie typy tablic są w rzeczywistości synonimami typu jobject. Natomiast w języku C++ tworzą one hierarchię dziedziczenia pokazaną na rysunku 12.3. Typ jarray oznacza natomiast ogólny typ tablicy.

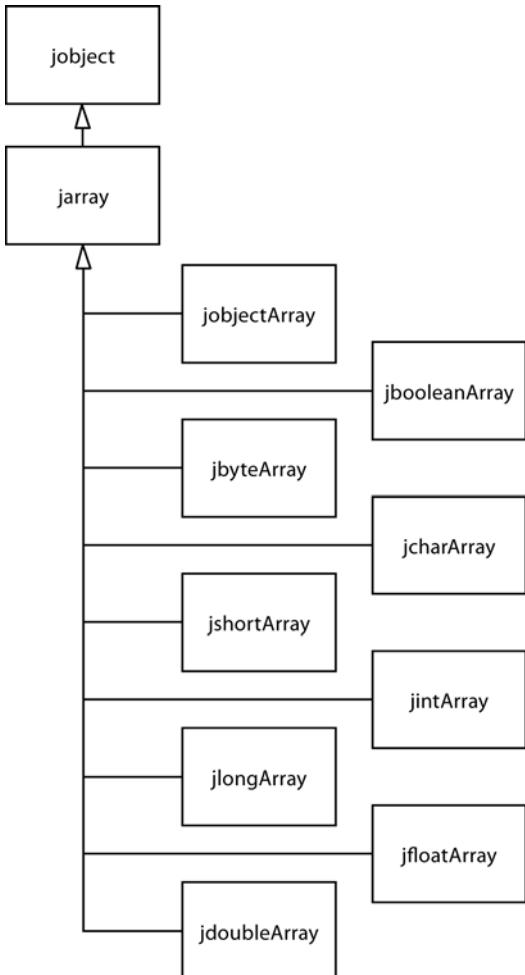
Funkcja GetArrayLength zwraca rozmiar tablicy.

```
jarray = . . . ;
jsize length = (*env)->GetArrayLength(env, array);
```

Sposób dostępu do elementów tablicy zależy od tego, czy przechowuje ona elementy typów prostych (boolean, char i typów numerycznych) czy obiekty. W drugim z wymienionych przypadków korzystamy z funkcji GetObjectArrayElement i SetObjectArrayElement.

**Rysunek 12.3.**

*Hierarchia dziedziczenia typów tablic w języku C++*



```

jobjectArray array = . . . ;
int i, j;
jobject x = (*env)->GetObjectArrayElement(env, array, i);
(*env)->SetObjectArrayElement(env, array, j, x);
  
```

Taki sposób dostępu, mimo że prosty w użyciu, nie jest jednak efektywny. W przypadku obliczeń prowadzonych na wektorach bądź macierzach efektywniejszy jest bezpośredni dostęp do elementów tablicy.

Funkcja `GetXxxArrayElements` zwraca wskaźnik języka C do pierwszego elementu tablicy. Podobnie jak w przypadku łańcuchów znakowych, po zakończeniu korzystania ze wskaźnika należy poinformować o tym maszynę wirtualną, wywołując funkcję `ReleaseXxxArrayElements`. W przypadku bezpośredniego dostępu do elementów tablicy typ reprezentowany przez łańcuch `Xxx` musi być typem prostym. Pamiętajmy jednak, że wskaźnik *może wskazywać kopię tablicy* i wobec tego dokonane zmiany znajdą odzwierciedlenie w oryginalnej tablicy tylko pod warunkiem wywołania odpowiedniej funkcji `ReleaseXxxArrayElements`!



O tym czy tablica jest oryginałem, czy kopią możemy dowiedzieć się, przekazując wskaźnik do zmiennej typu jboolean jako trzeci parametr funkcji GetXxxArrayElements. Jeśli tablica jest kopią, to zmiennej tej zostanie nadana wartość JNI\_TRUE. Jeśli nie jesteśmy zainteresowani tą informacją, to jako trzeci parametr funkcji możemy przekazać wartość NULL.

Poniżej prezentujemy fragment kodu, który mnoży wszystkie elementy tablicy typu double przez wartość stałej. Najpierw pobieramy wskaźnik a do początku tablicy, a następnie wykonujemy bezpośredni dostęp do jej elementów postaci a[i].

```
jdoubleArray array_a = . . . ;
double scaleFactor = . . . ;
double* a = (*env)->GetDoubleArrayElements(env, array_a, NULL);
for (i=0; i < (*env)->GetArrayLength(env, array_a); i++)
    a[i] = a[i] * scaleFactor;
(*env)->ReleaseDoubleArrayElements(env, array_a, 0);
```

To, czy maszyna wirtualna wykonuje kopię oryginalnej tablicy zależy od stosowanego przez nią sposobu zarządzania przydziałem pamięci. Niektóre, tak zwane „kopiąjące”, procedury odzysku pamięci stosowane przez maszyny wirtualne Java przemieszczają obiekty w pamięci, aktualizując ich referencje. W przypadku takiej strategii zarządzania pamięcią konieczne jest wykonanie kopii obiektu, ponieważ procedura odzysku pamięci nie będzie w stanie zlokalizować i zaktualizować wskaźników obiektu w kodzie macierzystym.



Implementacja maszyny wirtualnej firmy Sun stosuje upakowaną reprezentację tablic elementów typu boolean wykorzystującą słowa 32-bitowe. Metoda GetBooleanArrayElements tworzy kopię tablicy, która zawiera nieupakowane elementy typu jboolean.

Jeśli potrzebujemy dostępu do tylko kilku elementów tablicy o sporych rozmiarach, to możemy skorzystać z funkcji GetXxxArrayRegion i SetXxxArrayRegion, które umożliwiają kopiowanie zakresu elementów między tablicą języka Java i tablicą języka C.

Metody macierzyste mogą tworzyć nowe tablice języka Java, korzystając z funkcji NewXxxArray, której należy przekazać rozmiar tworzony tablicy, typ jej elementów oraz wartość początkową taką samą dla wszystkich elementów tablicy (zwykle NULL).

```
jclass class_Employee = (*env)->FindClass(env, "Employee");
jobjectArray array_e = (*env)->NewObjectArray(env, 100,
    class_Employee, NULL);
```

Tablice typów prostych wymagają jedynie określenia ich rozmiaru.

```
jdoubleArray array_d = (*env)->NewDoubleArray(env, 100);
```

Tablica taka wypełniana jest wartością 0.



**Java SE 1.4 wprowadza trzy nowe funkcje interfejsu JNI:**

```
jobject NewDirectByteBuffer(JNIEnv *, void* address, jlong capacity)
void* GetDirectBufferAddress(JNIEnv *, jobject buf)
jlong GetDirectBufferCapacity(JNIEnv *, jobject buf)
```

Bufory o dostępie bezpośrednim wykorzystywane są przez pakiet `java.nio` dla uzyskania lepszej efektywności operacji wejścia/wyjścia przez uniknięcie kopiowania informacji pomiędzy tablicami kodu macierzystego i tablicami platformy Java.

### API Operacje na tablicach języka Java w języku C

- `jsize GetArrayLength(JNIEnv *env, jarray array)`  
zwraca liczbę elementów w tablicy.
- `jobject GetObjectArrayElement(JNIEnv *env, jobjectArray array, jsize index)`  
zwraca wartość elementu tablicy.
- `void GetObjectArrayElement(JNIEnv *env, jobjectArray array, jsize index, jobject value)`  
nadaje nową wartość elementowi tablicy.
- `Xxx* GetXxxArrayElements(JNIEnv env, jarray array, jboolean isCopy)`  
zwraca wskaźnik języka C do elementów tablicy. Typ elementów tablicy `Xxx` może być `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` lub `Double`. Wskaźnik musi zostać przekazany metodzie `ReleaseXxxArrayElements`, gdy nie jest już wykorzystywany. Parametr `isCopy` ma wartość `NULL` lub jest wskaźnikiem zmiennej typu `jboolean`, która otrzyma wartość `JNI_TRUE`, jeśli tablica jest kopią oryginalnej tablicy platformy Java lub wartość `JNI_FALSE` — w przeciwnym razie
- `void ReleaseXxxArrayElements(JNIEnv *env, jarray array, Xxx elems[], jint mode)`  
powiadamia maszynę wirtualną, że wskaźnik uzyskany przez wywołanie metody `GetXxxArrayElements` nie będzie już wykorzystywany. Parametr `mode` może przyjmować wartość `0` (bufor wskazywany przez `elems` może zostać zwolniony po zaktualizowaniu tablicy), `JNI_COMMIT` (zabrania zwolnienia bufora po zaktualizowaniu tablicy) lub `JNI_ABORT` (powoduje zwolnienie bufora bez aktualizowania tablicy).
- `void GetXxxArrayRegion(JNIEnv *env, jarray array, jint start, jint length, Xxx elems[])`  
kopiuje elementy tablicy platformy Java do tablicy języka C. `Xxx` może być `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` lub `Double`.
- `void SetXxxArrayRegion(JNIEnv *env, jarray array, jint start, jint length, Xxx elems[])`  
kopiuje elementy tablicy języka C do tablicy platformy Java. `Xxx` może być `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` lub `Double`.

## 12.8. Obsługa błędów

Metody macierzyste stanowią zagrożenie z punktu widzenia bezpieczeństwa programów w języku Java. Język C nie posiada żadnych zabezpieczeń przed przekroczeniem zakresu tablicy czy wykorzystaniem źle zainicjowanych wskaźników. Dlatego też niezwykle istotne jest, aby, tworząc metody macierzyste, właściwie obsługiwać powstałe błędy i zachować integralność platformy Java. Szczególnie jeśli metoda macierzysta natrafi na problem, którego nie potrafi obsługiwać, to powinna poinformować o tym maszynę wirtualną Java.

W takiej sytuacji kod wyrzuca zwykle wyjątek. Jednak język C nie posiada wyjątków. Rozwiązaniem jest utworzenie obiektu wyjątku za pomocą funkcji Throw lub ThrowNew, które zostaną wyrzucone przez maszynę wirtualną Java po zakończeniu wykonywania metody macierzystej.

Aby skorzystać z funkcji Throw, musimy najpierw wywołać funkcję NewObject, aby utworzyć obiekt jednej z klas implementujących interfejs Throwable. Poniżej prezentujemy fragment kodu, który tworzy obiekt wyjątku EOFException i wyrzuca go.

```
jclass class_EOFException = (*env)->FindClass(env, "java/io/EOFException");
jmethodID id_EOFException = (*env)->GetMethodID(env, class_EOFException,
    "<init>", "()V"); /* pobiera identyfikator domyślnego konstruktora */
jthrowable obj_exc = (*env)->NewObject(env, class_EOFException, id_EOFException);
(*env)->Throw(env, obj_exc);
```

Zwykle jednak wygodniej skorzystać z funkcji ThrowNew, która sama tworzy obiekt wyjątku na podstawie przekazanej jej klasy i łańcucha w kodzie „zmodyfikowany UTF-8”.

```
(*env)->ThrowNew(env, (*env)->FindClass(env,
    "java/io/EOFException"),
    "Unexpected end of file");
```

Funkcje Throw i ThrowNew *przygotowują* jedynie obiekt wyjątku do wyrzucenia, ale nie przerwają wykonania metody macierzystej. Dopiero po zakończeniu jej wykonywania wyjątek zostaje wyrzucony przez maszynę wirtualną Java. Dlatego też po każdym wywołaniu funkcji Throw lub ThrowNew należy umieścić w kodzie instrukcję return.



Choć można sobie wyobrazić przekład wyjątków C++ na wyjątki Java, to obecnie wyrzucanie wyjątków przez metody macierzyste tworzone w języku C++ nie jest zaimplementowane. Dlatego w kodzie macierzystym w języku C++ należy nadal korzystać z metod Throw i ThrowNew oraz upewnić się, że nie wyrzuca on wyjątków języka C++.

Zwykle kod macierzysty nie powinien być zainteresowany obsługą wyjątków platformy Java. Jeśli jednak wywołuje on metody języka Java, to może zdarzyć się, że wyrzucią one wyjątek. Co więcej wiele funkcji interfejsu JNI także może wyrzucić wyjątek. Na przykład funkcja SetObjectArrayElement wyrzuca wyjątek ArrayIndexOutOfBoundsException, jeśli indeks elementu tablicy jest spoza zakresu lub wyjątek ArrayStoreException, jeśli klasa umieszczonego w tablicy obiektu nie jest klasą pochodną klasy elementów tablicy. W takiej sytuacji metoda macierzysta może wywołać funkcję ExceptionOccured, aby ustalić, czy został wyrzucony wyjątek. Wywołanie

```
jthrowable obj_exc = (*env)->ExceptionOccured(env);
```

zwróci wartość `NULL`, jeśli wyjątek nie został wyrzucony lub wskaźnik do obiektu bieżącego wyjątku. Jeśli interesuje nas wyłącznie fakt wyrzucenia wyjątku, to wystarczy wywołać jedyne funkcję `ExceptionCheck`:

```
jboolean occurred = (*env)->ExceptionCheck(env);
```

Zwykle po wystąpieniu wyjątku metoda macierzysta powinna zakończyć swoje wykonywanie, aby umożliwić maszynie wirtualnej propagację wyjątku w kodzie Java. Jednak metoda macierzysta *może* też przeanalizować wyjątek, aby ustalić, czy potrafi go obsłużyć. Jeśli zdecyduje się go obsłużyć, to powinna skorzystać z poniższego wywołania, aby zapobiec propagacji wyjątku.

```
(*env)->ExceptionClear(env);
```

W naszym kolejnym przykładzie wyposażymy metodę macierzystą `fprint` w obsługę błędów. Metoda ta będzie teraz wyrzucać następujące wyjątki:

- `NullPointerException`, jeśli nie został podany łańcuch formatujący,
- `IllegalArgumentException`, jeśli łańcuch formatujący nie określa formatu liczby zmiennoprzecinkowej typu `double`,
- `OutOfMemoryError`, jeśli wywołanie metody `malloc` nie powiedzie się.

Aby zaprezentować też sposób sprawdzania wystąpienia wyjątku podczas wywołania metody języka Java przez kod macierzysty, będziemy wysyłać łańcuch do strumienia znak po znaku, za każdym razem wywołując funkcję `ExceptionOccured`. Listing 12.17 prezentuje implementację metody macierzystej, a listing 12.18 — kod źródłowy klasy zawierającej tę metodę. Zwróćmy uwagę, że metoda macierzysta nie kończy swojego wykonania natychmiast po stwierdzeniu wyrzucenia wyjątku przez metodę `PrintWriter.print`, ale zwalnia najpierw bufor `cstr`. Wyjątek zostaje wyrzucony przez maszynę wirtualną Java po zakończeniu wykonywania metody macierzystej. Program testowy z listingu 12.19 demonstruje wyrzucenie wyjątku przez metodę macierzystą, dostarczając jej niedozwolony łańcuch formatujący.

**Listing 12.17.** `printf4/Printf4.c`

```
/***
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */

#include "Printf4.h"
#include <string.h>
#include <stdlib.h>
#include <float.h>

/**
 * @param format łańcuch określający sposób formatowania
 * (na przykład "%6.2f"). Podłańcuchy "%%" są pomijane.
 * @return wskaźnik specyfikatora formatu (z pominięciem znaku '%')
 * lub wartość NULL, jeśli nie został odnaleziony.
 */
char* find_format(const char format[])
{
    char* p;
    char* q;
```

```

p = strchr(format, '%');
while (p != NULL && *(p + 1) == '%') /*pomija %%*/
{
    p = strchr(p + 2, '%');
    if (p == NULL) return NULL;
    /* sprawdza, czy % jest unikalny */
    p++;
    q = strchr(p, '%');
    while (q != NULL && *(q + 1) == '%') /*pomija %%*/
    {
        q = strchr(q + 2, '%');
        if (q != NULL) return NULL; /*% nie jest unikalny */
        q = p + strspn(p, " -0#"); /*pomija znaczniki */
        q += strspn(q, "0123456789"); /*i specyfikację szerokości pola */
        if (*q == '.') { q++; q += strspn(q, "0123456789"); }
        /*pomija specyfikację precyzji */
        if (strchr("eEffFgG", *q) == NULL) return NULL;
        /*to nie jest format zmiennoprzecinkowy*/
    }
    return p;
}

JNIEXPORT void JNICALL Java_Printf4_fprint(JNIEnv* env, jclass cl,
 jobject out, jstring format, jdouble x)
{
    const char* cformat;
    char* fmt;
    jclass class_PrintWriter;
    jmethodID id_print;
    char* cstr;
    int width;
    int i;

    if (format == NULL)
    {
        (*env)->ThrowNew(env,
            (*env)->FindClass(env,
                "java/lang/NullPointerException"),
            "Printf4.fprint: format is null");
        return;
    }

    cformat = (*env)->GetStringUTFChars(env, format, NULL);
    fmt = find_format(cformat);

    if (fmt == NULL)
    {
        (*env)->ThrowNew(env,
            (*env)->FindClass(env,
                "java/lang/IllegalArgumentException"),
            "Printf4.fprint: format is invalid");
        return;
    }

    width = atoi(fmt);
    if (width == 0) width = DBL_DIG + 10;
    cstr = (char*)malloc(strlen(cformat) + width);

    if (cstr == NULL)
    {
        (*env)->ThrowNew(env,

```

```
        (*env)->FindClass(env, "java/lang/OutOfMemoryError"),
        "Printf4.fprint: malloc failed");
    return;
}

sprintf(cstr, cformat, x);

(*env)->ReleaseStringUTFChars(env, format, cformat);

/* wywołuje metodę ps.print(str) */

/* ustala klasę */
class_PrintWriter = (*env)->GetObjectClass(env, out);

/* pobiera identyfikator metody */
id_print = (*env)->GetMethodID(env, class_PrintWriter, "print", "(C)V");

/* wywołuje metodę */
for (i = 0; cstr[i] != 0 && !(*env)->ExceptionOccurred(env); i++)
    (*env)->CallVoidMethod(env, out, id_print, cstr[i]);

free(cstr);
}
```

---

**Listing 12.18.** printf4/Printf4.java

```
import java.io.*;

/**
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */
class Printf4
{
    public static native void fprintf(PrintWriter ps, String format, double x);

    static
    {
        System.loadLibrary("Printf4");
    }
}
```

---

**Listing 12.19.** printf4/Printf4Test.java

```
import java.io.*;

/**
 * @version 1.10 1997-07-01
 * @author Cay Horstmann
 */
class Printf4Test
{
    public static void main(String[] args)
    {
        double price = 44.95;
        double tax = 7.75;
        double amountDue = price * (1 + tax / 100);
    }
}
```

```

    PrintWriter out = new PrintWriter(System.out);
    /* Poniższe wywołanie spowoduje wyrzucenie wyjątku ze względu na kombinację %% */
    Printf4.fprint(out, "Amount due = %%8.2f\n", amountDue);
    out.flush();
}
}

```

 Obsługa błędów w języku C

■ `jint Throw(JNIEnv *env, jthrowable obj)`

przygotowuje wyjątek do wyrzucenia, które nastąpi po zakończeniu wykonywania kodu macierzystego. Zwraca wartość 0, jeśli przygotowanie przebiegło pomyślnie, wartość ujemną — w przeciwnym razie.

■ `jint ThrowNew(JNIEnv *env, jclass cl, const char msg[])`

przygotowuje wyjątek typu `cl` do wyrzucenia, które nastąpi po zakończeniu wykonywania kodu macierzystego. Zwraca wartość 0, jeśli przygotowanie przebiegło pomyślnie, wartość ujemną — w przeciwnym razie. Parametr `msg` jest łańcuchem w kodzie „zmodyfikowany UTF-8” zawierającym komunikat wyjątku.

■ `jthrowable ExceptionOccurred(JNIEnv *env)`

zwraca obiekt wyjątku, jeśli wystąpił lub wartość `NULL` — w przeciwnym razie.

■ `jboolean ExceptionCheck(JNIEnv *env)`

zwraca wartość `true`, jeśli wystąpił wyjątek.

■ `void ExceptionClear(JNIEnv *env)`

kasuje oczekujące wyjątki.

## 12.9. Interfejs programowy wywołań języka Java

Dotychczas rozpatrywaliśmy programy tworzone w języku Java, które wywoływały kilka metod macierzystych napisanych w języku C, ze względu na ich większą efektywność lub możliwość dostępu do usług lub urządzeń nieosiągalnych na platformie Java. Rozważmy teraz sytuację odwrotną, czyli program w języku C lub C++, który chce wykorzystać kod utworzony w języku Java. Wiemy już, w jaki sposób kod taki może wywoływać metody języka Java. Jednak to nie wystarcza, ponieważ aby wykonać kod w języku Java, musimy uruchomić dodatkowo maszynę wirtualną. Interfejs programowy wywołań języka Java umożliwia wykorzystanie maszyny wirtualnej Java przez programy napisane w językach C lub C++. Poniżej prezentujemy minimum kodu wymaganego do prawidłowejinicjalizacji maszyny wirtualnej.

```

JavaVMOption options[1];
JavaVMnitsArgs vm_args;
JavaVM *jvm;
JNIEnv *env;

options[0].optionString = "-Djava.class.path=.";

```

```

memset(&vm_args, 0, sizeof(vm_args));
vm_args.version = JNI_VERSION_1_2;
vm_args.nOptions = 1;
vm_args.options = options;

JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);

```

Wywołanie metody `JNI_CreateJavaVM` uruchamia maszynę wirtualną oraz nadaje wartość wskaźnikowi maszyny wirtualnej `jvm` i wskaźnikowi środowiska wykonania `env`.

Maszynie wirtualnej możemy przekazać dowolną liczbę opcji. W tym celu należy odpowiednio zwiększyć rozmiar tablicy `options` i wartość `vm_args.nOptions`. Poniższy przykład pokazuje sposób przekazania opcji wyłączającej kompilator JIT.

```
options[i].optionString = "-Djava.compiler=NONE";
```



Jeśli uruchamiany program zawiesza się, odmawiainicjalizacji maszyny wirtualnej lub nie ładuje klas, to pomocne może okazać się włączenie trybu uruchomieniowego interfejsu JNI. W tym celu należy przekazać maszynie wirtualnej poniższą opcję.

```
options[i].optionString = "-verbose:jni";
```

Spowoduje ona pojawienie się szeregu komunikatów o postępieinicjalizacji maszyny wirtualnej. Jeśli nie odnajdziemy wśród nich informacji o załadowaniu naszych klas, powinniśmy sprawdzić ścieżkę dostępu do klas.

Po uruchomieniu maszyny wirtualnej możemy wywoływać metody języka Java w sposob zaprezentowany w poprzednich podrozdziałach, korzystając jak zwykle ze wskaźnika `env`.

Wskaźnik `jvm` przydatny jest jedynie w przypadku wywoływania innych funkcji interfejsu wywołań metod języka Java. Obecnie dostępne są tylko cztery takie funkcje. Najważniejszą z nich umożliwia zakończenie pracy maszyny wirtualnej:

```
(*jvm)->DestroyJavaVM(jvm);
```

Niestety, w systemie Windows dynamiczne dołączenie funkcji `JNI_CreateJavaVM` z biblioteki `jre/bin/client/jvm.dll` stało się problematyczne ze względu na zmienione zasady działania bibliotek dynamicznych w systemie Vista, podczas gdy implementacja dostarczana przez firmę Oracle bazuje na starszej wersji bibliotek języka C. W naszym przykładowym programie obeszliśmy ten problem, „ręcznie” ładując wspomnianą bibliotekę. Takie samo rozwiązanie stosuje zresztą program `java` — co można sprawdzić w pliku `launcher/java_md.c` znajdującym się w archiwum `src.jar` należącym do pakietu JDK.

Program w języku C, którego kod źródłowy zawiera listing 12.20, uruchamia maszynę wirtualną i wywołuje metodę `main` klasy `Welcome` omówionej w 2. rozdziale książki *Java 2. Podstawy*. (Pamiętajmy o skompilowaniu pliku `Welcome.java` przed uruchomieniem programu wywołującego).

#### **Listing 12.20. invocation/InvocationTest.c**

```

/**
 * @version 1.20 2007-10-26
 * @author Cay Horstmann
 */

```

```

#include <jni.h>
#include <stdlib.h>

#ifndef _WINDOWS

#include <windows.h>
static HINSTANCE loadJVMLibrary(void);
typedef jint (JNICALL *CreateJavaVM_t)(JavaVM **, void **, JavaVMInitArgs *);

#endif

int main()
{
    JavaVMOption options[2];
    JavaVMInitArgs vm_args;
    JavaVM *jvm;
    JNIEnv *env;
    long status;

    jclass class_Welcome;
    jclass class_String;
    jobjectArray args;
    jmethodID id_main;

#ifndef _WINDOWS
    HINSTANCE hjvmlib;
    CreateJavaVM_t createJavaVM;
#endif

    options[0].optionString = "-Djava.class.path=.";

    memset(&vm_args, 0, sizeof(vm_args));
    vm_args.version = JNI_VERSION_1_2;
    vm_args.nOptions = 1;
    vm_args.options = options;

#ifndef _WINDOWS
    hjvmlib = loadJVMLibrary();
    createJavaVM = (CreateJavaVM_t) GetProcAddress(hjvmlib, "JNI_CreateJavaVM");
    status = (*createJavaVM)(&jvm, (void **) &env, &vm_args);
#else
    status = JNI_CreateJavaVM(&jvm, (void **) &env, &vm_args);
#endif

    if (status == JNI_ERR)
    {
        fprintf(stderr, "Error creating VM\n");
        return 1;
    }

    class_Welcome = (*env)->FindClass(env, "Welcome");
    id_main = (*env)->GetStaticMethodID(env, class_Welcome, "main",
                                         "[Ljava/lang/String;)V");

    class_String = (*env)->FindClass(env, "java/lang/String");
    args = (*env)->NewObjectArray(env, 0, class_String, NULL);
}

```

```
(*env)->CallStaticVoidMethod(env, class_Welcome, id_main, args);

(*jvm)->DestroyJavaVM(jvm);

return 0;
}

#ifndef _WINDOWS

static int GetStringFromRegistry(HKEY key, const char *name, char *buf, jint
→bufsize)
{
    DWORD type, size;

    return RegQueryValueEx(key, name, 0, &type, 0, &size) == 0
        && type == REG_SZ
        && size < (unsigned int) bufsize
        && RegQueryValueEx(key, name, 0, 0, buf, &size) == 0;
}

static void GetPublicJREHome(char *buf, jint bufsize)
{
    HKEY key, subkey;
    char version[MAX_PATH];

    /* Odnajduje zainstalowaną wersję JRE */
    char *JRE_KEY = "Software\\JavaSoft\\Java Runtime Environment";
    if (RegOpenKeyEx(HKEY_LOCAL_MACHINE, JRE_KEY, 0, KEY_READ, &key) != 0)
    {
        fprintf(stderr, "Error opening registry key '%s'\n", JRE_KEY);
        exit(1);
    }

    if (!GetStringFromRegistry(key, "CurrentVersion", version, sizeof(version)))
    {
        fprintf(stderr, "Failed reading value of registry
→key:\n\t%s\\CurrentVersion\n", JRE_KEY);
        RegCloseKey(key);
        exit(1);
    }

    /* Odnajduje katalog, w którym zainstalowano JRE. */
    if (RegOpenKeyEx(key, version, 0, KEY_READ, &subkey) != 0)
    {
        fprintf(stderr, "Error opening registry key '%s\\%s'\n", JRE_KEY, version);
        RegCloseKey(key);
        exit(1);
    }

    if (!GetStringFromRegistry(subkey, "JavaHome", buf, bufsize))
    {
        fprintf(stderr, "Failed reading value of registry key:\n\t%s\\%s\\JavaHome\n",
            JRE_KEY, version);
        RegCloseKey(key);
        RegCloseKey(subkey);
        exit(1);
    }
}
```

```

    RegCloseKey(key);
    RegCloseKey(subkey);
}

static HINSTANCE loadJVMLibrary(void)
{
    HINSTANCE h1, h2;
    char msrvcdll[MAX_PATH];
    char javadll[MAX_PATH];
    GetPublicJREHome(msrvcdll, MAX_PATH);
    strcpy(javadll, msrvcdll);
    strncat(msrvcdll, "\\bin\\msvcr71.dll", MAX_PATH - strlen(msrvcdll));
    msrvcdll[MAX_PATH - 1] = '\0';
    strncat(javadll, "\\bin\\client\\jvm.dll", MAX_PATH - strlen(javadll));
    javadll[MAX_PATH - 1] = '\0';

    h1 = LoadLibrary(msrvcdll);
    if (h1 == NULL)
    {
        fprintf(stderr, "Can't load library msvcr71.dll\n");
        exit(1);
    }

    h2 = LoadLibrary(javadll);
    if (h2 == NULL)
    {
        fprintf(stderr, "Can't load library jvm.dll\n");
        exit(1);
    }
    return h2;
}

#endif

```

Aby skompilować program w systemie Linux, skorzystamy z następującego polecenia:

```

gcc -I jdk/include -I jdk/include/linux
-o InvocationTest
-L jdk/jre/lib/i386/client
-ljvm
InvocationTest.c

```

Natomiast w systemie Solaris będzie miało ono następującą postać.

```

cc -I jdk/include -I jdk/include/solaris
-o InvocationTest
-L jdk/jre/lib/sparc
-ljvm
InvocationTest.c

```

Kompilując program w systemie Windows, kompilator Microsoft C wywołamy w następujący sposób:

```

cl -D_WINDOWS -I jdk/include\ -I jdk/include\win32
InvocationTest.c c jdk\lib\jvm.lib advapi32.lib

```

Upewnij się, że zmienne środowiskowe INCLUDE i LIB zawierają ścieżki do plików nagłówkowych Windows API i bibliotek.

Ta sama operacja przy użyciu kompilatora z zestawu Cygwin będzie wyglądać następująco:

```
gcc -D_WINDOWS -mno-cygwin -I jdk\include -I jdk\include\win32 -D_int64="long long"
-I c:\cygwin\usr\include\w32api -o InvocationTest
```

Zanim uruchomimy program w systemie Linux/Unix, należy upewnić się, czy zmienna LD\_LIBRARY\_PATH zawiera katalogi bibliotek współdzielonych. Na przykład korzystając z powłoki bash systemu Linux, wydamy w tym celu polecenie:

```
export LD_LIBRARY_PATH=jdk/jre/lib/i386/client:$LD_LIBRARY_PATH
```

#### Funkcje interfejsu programowego wywołań języka Java

- `jint JNI_CreateJavaVM(JavaVM** p_jvm, void** p_env, JavaVMInitArgs* vm_args)`

uruchamia maszynę wirtualną Java. Zwraca wartość 0, jeśli uruchomienie powiodło się i wartość JNI\_ERR — w przypadku błędu.

*Parametry:*    `p_jvm`                zostaje wypełniony wskaźnikiem tabeli funkcji interfejsu wywołań,

`p_env`                zostaje wypełniony wskaźnikiem tabeli funkcji JNI,

`vm_args`                parametry maszyny wirtualnej.

- `jint DestroyJavaVM(JavaVM* jvm)`

kończy pracę maszyny wirtualnej. Zwraca wartość 0, jeśli operacja przebiegła pomyślnie, wartość ujemną — w przeciwnym razie. Funkcja ta musi zostać wywołana za pomocą wskaźnika maszyny wirtualnej, czyli na przykład (`*jvm`)->DestroyJavaVM(`jvm`).

## 12.10. Kompletny przykład: dostęp do rejestru systemu Windows

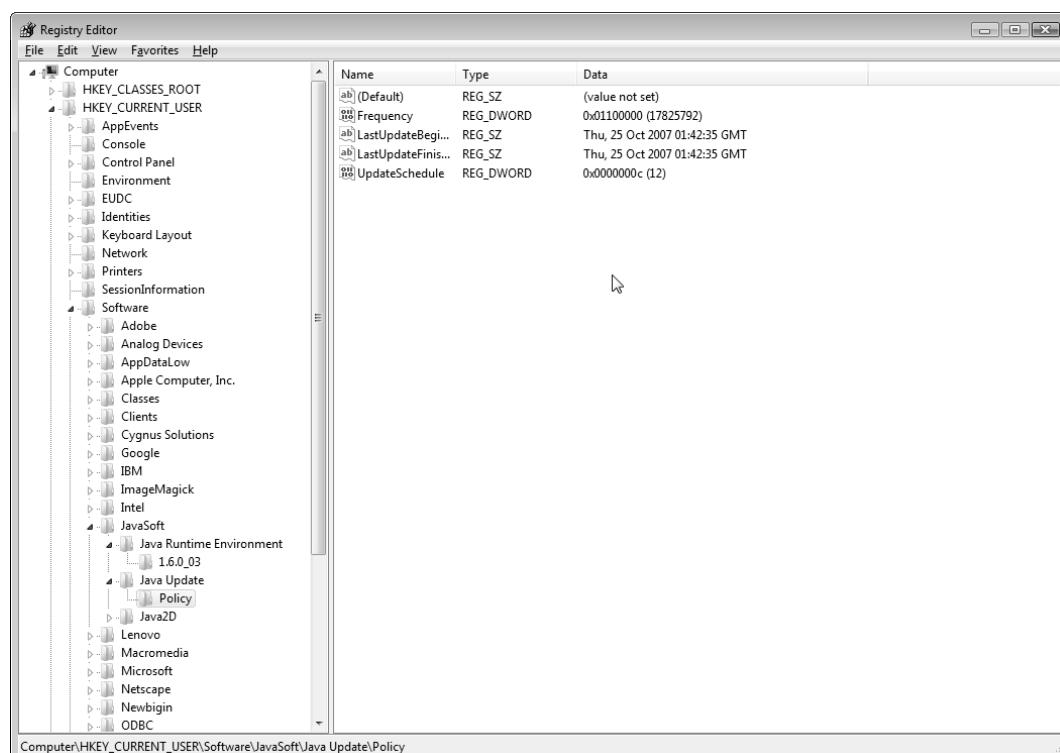
Przedstawimy teraz kompletny, działający przykład programu wykorzystujący w praktyce informacje przedstawione dotąd w bieżącym rozdziale i dotyczące implementacji metod macierzystych, operujących na łańcuchach znaków, tablicach i obiektach, oraz konstruktorów i obsługi błędów. Przykład ten polegać będzie na obudowaniu programem w języku Java interfejsu programowego w języku C pozwalającego na dostęp do rejestru systemu Windows. Oczywiście korzystając ze specyficznego dla systemu Windows rozwiązania, jakim jest rejestr systemowy, nie będziemy mogli przenieść programu na inne platformy. Z tego właśnie powodu platforma Java nie umożliwia dostępu do rejestru systemu Windows i celowe okazuje się zastosowanie metod macierzystych.

## 12.10.1. Rejestr systemu Windows

Rejestr systemu Windows stanowi repozytorium danych przechowujące informacje o konfiguracji systemu operacyjnego i aplikacji. Umieszczenie tych informacji we wspólnym rejestrze ułatwia administrację systemem i tworzenie kopii zapasowych. Należy jednak zauważyć, że równocześnie rejestr stał się newralgicznym elementem systemu — jeśli go uszkodzimy, to możemy spowodować nieprawidłowe działanie systemu, a nawet uniemożliwić jego uruchomienie!

Nie polecamy korzystania z rejestru w celu przechowywania parametrów konfiguracji programów w języku Java. Lepiej użyć w tym celu interfejsu programowego preferencji, który omawiamy w rozdziale 10. książki *Java 2. Podstawy*. Nasz program korzysta z rejestru jedynie w celu zademonstrowania sposobu obudowania macierzystych interfejsów programowych kodem w języku Java.

Podstawowym narzędziem służącym do przeglądania zawartości rejestru jest *edytor rejestru*. Ze względu na niebezpieczeństwo jego użycia przez niedoświadczonych użytkowników nie udostępniono w systemie ikony bądź skrótu umożliwiającego jego szybkie uruchomienie. W tym celu musimy otworzyć okno polecen (lub okno dialogowe *Start/Uruchom*) i wpisać nazwę pliku wykonywalnego *regedit*. Rysunek 12.4 pokazuje edytor rejestru w działaniu.



Rysunek 12.4. Edytor rejestru

W lewej części okna programu pokazane są klucze rejestru tworzące strukturę drzewiastą. Zwróćmy uwagę, że każdy z kluczów należy do poddrzewa jednego z węzłów HKEY:

```
HKEY_CLASSES_ROOT  
HKEY_CURRENT_USER  
HKEY_LOCAL_MACHINE
```

```
.
```

Natomiast w prawej części okna prezentowane są pary nazwa-wartość związane z wybranym kluczem. Na przykład jeśli zainstalujemy Java SE 7, to klucz

```
HKEY_LOCAL_MACHINE\Software\JavaSoft\Java Runtime Environment
```

zawiera parę postaci

```
CurrentVersion="1.7.0_10"
```

W przykładzie tym wartość jest łańcuchem znaków, ale w ogóle wartości mogą być też liczbami całkowitymi bądź tablicami bajtów.

## 12.10.2. Interfejs dostępu do rejestru na platformie Java

Zaprojektujemy teraz prosty interfejs dostępu do rejestru systemu Windows z programów w języku Java i zaimplementujemy go za pomocą kodu macierzystego. Interfejs ten będzie umożliwiał jedynie kilka podstawowych operacji. Aby ograniczyć rozmiary kodu, pominiemy implementację kilku istotnych operacji, takich jak dodawanie, usuwanie i przeglądanie kluczów. (Implementacja tych operacji nie powinna przysporzyć Czytelnikowi kłopotu).

Nawet gdy dostarczymy ograniczonego podzbioru zaimplementowanych operacji, program umożliwiać będzie:

- przeglądanie nazw związanych z kluczem,
- odczyt wartości dla wybranej nazwy,
- nadanie wartości dla wybranej nazwy.

Poniżej prezentujemy klasę języka Java hermetyzującą klucz rejestru.

```
public class Win32RegKey  
{  
    public Win32RegKey(int theRoot, String thePath) { . . . }  
    public Enumeration names() { . . . }  
    public native Object getValue(String name);  
    public native void setValue(String name, Object value);  
  
    public static final int HKEY_CLASSES_ROOT = 0x80000000;  
    public static final int HKEY_CURRENT_USER = 0x80000001;  
    public static final int HKEY_LOCAL_MACHINE = 0x80000002;  
    . . .  
}
```

Metoda names zwraca wyliczenie wszystkich nazw dla danego klucza. Możemy przeglądać je, korzystając ze znanych już metod hasMoreElements/nextElement. Metoda getValue zwraca obiekt, który może być łańcuchem znaków, obiektem klasy Integer bądź tablicą bajtów. Podobnie parametr metody setValue musi być jednym z wymienionych obiektów.

## 12.10.3. Implementacja dostępu do rejestru za pomocą metod macierzystych

W programie naszym zaimplementujemy trzy operacje:

- pobrania wartości klucza,
- nadania wartości klucza,
- przeglądania nazw klucza.

Poznaliśmy już praktycznie wszystkie potrzebne w tym celu techniki, takie jak konwersje pomiędzy łańcuchami znakowymi języka Java i tablicami języka C czy wyrzucanie wyjątków Java z kodu macierzystego.

Jednak dwa fakty spowodują, że implementacja metod macierzystych będzie tym razem nieco bardziej skomplikowana niż w dotychczasowych przykładach. Metody `getValue` i `setValue` będą operować na obiektach typu `Object`, który w rzeczywistości będą obiektami typu `String`, `Integer` lub `byte[]`. Natomiast obiekt wyliczenia będzie musiał zachowywać swój stan między kolejnymi wywołaniami metod `hasMoreElements`/`nextElement`.

Przyjrzyjmy się najpierw implementacji metody `getValue`. Jej kod (zaprezentowany w listingu 12.22) wykonuje następujące działania.

- 1.** Otwiera klucz rejestru. Klucze wymagają operacji otwarcia poprzedzającej operacje odczytu związanych z nimi wartości.
- 2.** Pobiera typ i rozmiar wartości związanego z nazwą.
- 3.** Wczytuje dane do bufora.
- 4.** Jeśli typem wartości jest `REG_SZ` (czyli jest to łańcuch znaków), to wywołuje funkcję `NewStringUTF` w celu utworzenia nowego łańcucha dla wczytanych danych.
- 5.** Jeśli typem wartości jest `REG_DWORD` (liczba całkowita o 32-bitowej reprezentacji), to wywołuje konstruktor klasy `Integer`.
- 6.** Jeśli typem wartości jest `REG_BINARY`, to wywołuje funkcję `NewByteArray` w celu utworzenia nowej tablicy bajtów oraz umieszcza w niej dane za pomocą funkcji `SetByteArrayRegion`.
- 7.** Jeśli typ wartości nie jest jednym z wymienionych wyżej lub podczas wywołania funkcji JNI wystąpił błąd, to wyrzuca wyjątek i zwalnia wszystkie użycie dotąd zasoby.
- 8.** Zamyka klucz i zwraca utworzony obiekt (typu `String`, `Integer` lub `byte[]`).

Przykład ten doskonale ilustruje sposób tworzenia różnego typu obiektów platformy Java przez kod macierzysty.

W przypadku powyższej metody zastosowanie ogólnego typu zwracanej wartości jest oczywiste. Referencja obiektu `jstring`, `jobject` lub `jarray` zwracana jest zawsze jako referencja obiektu typu `Object`. Jednak metoda `setValue` otrzymuje jako parametr referencję obiektu `Object` i sama musi ustalić, czy reprezentuje on łańcuch znaków, całkowitą wartość liczbową

lub tablicę bajtów. W tym celu pobiera ona klasę parametru `value` i za pomocą funkcji `IsAssignableFrom` porównuje ją z referencjami klas `java.lang.String`, `java.lang.Integer` i `byte[]`.

Jeśli `class1` i `class2` reprezentują referencje klas, to wywołanie

```
(*env)->IsAssignableFrom(env, class1, class2)
```

zwraca wartość `JNI_TRUE`, jeśli `class1` i `class2` reprezentują tę samą klasę lub `class1` reprezentuje klasę pochodną klasy `class2`. W obu przypadkach referencje obiektów klasy `class1` mogą być rzutowane do klasy `class2`. Na przykład jeśli wywołanie

```
(*env)-> IsAssignableFrom(env,  
    (*env)->GetObjectClass(env, value),  
    (*env)->FindClass(env, "[B"))
```

zwróci wartość `true`, to wiemy, że `value` jest tablicą bajtów.

Metoda `setValue` wykonuje kolejno opisane poniżej operacje.

1. Otwiera klucz rejestru.
2. Określa typ zapisywanej wartości.
3. Jeśli typem tym jest `String`, to wywołuje metodę `GetStringUTFChars` w celu uzyskania wskaźnika do znaków łańcucha.
4. Jeśli typem wartości jest `Integer`, to wywołuje metodę `intValue`, aby uzyskać liczbę całkowitą obudowaną dotąd obiektem.
5. Jeśli wartość jest typu `byte[]`, to wywołuje funkcję `GetByteArrayElements` w celu uzyskania wskaźnika do bajtów umieszczonej w tablicy.
6. Przekazuje dane i ich rozmiar do rejestru.
7. Zamiera klucz.
8. Jeśli typem wartości był `String` lub `byte[]`, to zwalnia także odpowiedni wskaźnik.

Przejdźmy teraz do omówienia metod macierzystych związanych z przeglądaniem nazw kluczy i umieszczonych w klasie `Win32RegKeyNameEnumeration` (patrz listing 12.21). Gdy rozpoczyna się proces przeglądania, musimy najpierw otworzyć odpowiedni klucz. Uchwyt klucza trzeba zachować przez cały czas trwania procesu przeglądania i dlatego jego wartość umieścimy w obiekcie wyliczenia. Uchwyt klucza jest 32-bitową wartością typu `DWORD` i może być przechowany za pomocą typu `int` platformy Java. Umieścimy go w składowej `hkey` klasy wyliczenia. Po rozpoczęciu procesu przeglądania składową zainicjujemy za pomocą funkcji `Set->IntField`. Kolejne wywołania metod podczas przeglądania nazw będą pobierać wartość tej składowej przy użyciu funkcji `GetIntField`.

---

**Listing 12.21.** `win32reg/Win32RegKey.java`

```
import java.util.*;  
  
/**  
 * Klasa Win32RegKey umożliwiająca pobieranie i nadawanie  
 * wartości nazwom związanym z kluczami rejestru Windows.  
 * @version 1.00 1997-07-01
```

```

 * @author Cay Horstmann
 */
public class Win32RegKey
{
    public static final int HKEY_CLASSES_ROOT = 0x80000000;
    public static final int HKEY_CURRENT_USER = 0x80000001;
    public static final int HKEY_LOCAL_MACHINE = 0x80000002;
    public static final int HKEY_USERS = 0x80000003;
    public static final int HKEY_CURRENT_CONFIG = 0x80000005;
    public static final int HKEY_DYN_DATA = 0x80000006;

    private int root;
    private String path;

    /**
     * Pobiera wartość z rejestru.
     * @param name nazwa, której wartość pobieramy
     * @return wartość związana z nazwą
     */
    public native Object getValue(String name);

    /**
     * Wpisuje wartość do rejestru.
     * @param name nazwa, dla której wpisujemy wartość
     * @param value nowa wartość
     */
    public native void setValue(String name, Object value);

    /**
     * Tworzy obiekt reprezentujący klucz rejestru.
     * @param theRoot jedna z wartości HKEY_CLASSES_ROOT,
     * HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE, HKEY_USERS,
     * HKEY_CURRENT_CONFIG, HKEY_DYN_DATA
     * @param thePath ścieżka klucza w rejestrze
     */
    public Win32RegKey(int theRoot, String thePath)
    {
        root = theRoot;
        path = thePath;
    }

    /**
     * Tworzy wyliczenie nazw umieszczonych w rejestrze
     * dla danego klucza.
     * @return wyliczenie nazw
     */
    public Enumeration<String> names()
    {
        return new Win32RegKeyNameEnumeration(root, path);
    }

    static
    {
        System.loadLibrary("Win32RegKey");
    }
}

class Win32RegKeyNameEnumeration implements Enumeration<String>
{

```

```
public native String nextElement();
public native boolean hasMoreElements();
private int root;
private String path;
private int index = -1;
private int hkey = 0;
private int maxsize;
private int count;

Win32RegKeyNameEnumeration(int theRoot, String thePath)
{
    root = theRoot;
    path = thePath;
}
}

class Win32RegKeyException extends RuntimeException
{
    public Win32RegKeyException()
    {
    }

    public Win32RegKeyException(String why)
    {
        super(why);
    }
}
```

---

**Listing 12.22.** win32reg/Win32RegKey.c

---

```
/***
 * @version 1.00 1997-07-01
 * @author Cay Horstmann
 */

#include "Win32RegKey.h"
#include "Win32RegKeyNameEnumeration.h"
#include <string.h>
#include <stdlib.h>
#include <windows.h>

JNIEXPORT jobject JNICALL Java_Win32RegKey_getValue(JNIEnv* env, jobject this_obj,
                                                       jstring name)
{
    const char* cname;
    jstring path;
    const char* cpath;
    HKEY hkey;
    DWORD type;
    DWORD size;
    jclass this_class;
    jfieldID id_root;
    jfieldID id_path;
    HKEY root;
    jobject ret;
    char* cret;
```

```

/* ustala klasę */
this_class = (*env)->GetObjectClass(env, this_obj);

/* pobiera identyfikatory pól */
id_root = (*env)->GetFieldID(env, this_class, "root", "I");
id_path = (*env)->GetFieldID(env, this_class, "path", "Ljava/lang/String;");

/* pobiera wartości pól */
root = (HKEY) (*env)->GetIntField(env, this_obj, id_root);
path = (jstring)(*env)->GetObjectField(env, this_obj, id_path);
cpath = (*env)->GetStringUTFChars(env, path, NULL);

/* otwiera klucz rejestru */
if (RegOpenKeyEx(root, cpath, 0, KEY_READ, &hkey) != ERROR_SUCCESS)
{
    (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
        "Open key failed");
    (*env)->ReleaseStringUTFChars(env, path, cpath);
    return NULL;
}

(*env)->ReleaseStringUTFChars(env, path, cpath);
cname = (*env)->GetStringUTFChars(env, name, NULL);

/* pobiera typ i rozmiar wartości */
if (RegQueryValueEx(hkey, cname, NULL, &type, NULL, &size) != ERROR_SUCCESS)
{
    (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
        "Query value key failed");
    RegCloseKey(hkey);
    (*env)->ReleaseStringUTFChars(env, name, cname);
    return NULL;
}

/* przydziela pamięć potrzebną do przechowania wartości */
cret = (char*)malloc(size);

/* pobiera wartość */
if (RegQueryValueEx(hkey, cname, NULL, &type, cret, &size) != ERROR_SUCCESS)
{
    (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
        "Query value key failed");
    free(cret);
    RegCloseKey(hkey);
    (*env)->ReleaseStringUTFChars(env, name, cname);
    return NULL;
}

/* w zależności od typu wartości przechowuje ją jako
    łańcuch znaków, liczbę całkowitą lub tablicę bajtów */
if (type == REG_SZ)
{
    ret = (*env)->NewStringUTF(env, cret);
}
else if (type == REG_DWORD)
{
    jclass class_Integer = (*env)->FindClass(env, "java/lang/Integer");
    /* pobiera identyfikator konstruktora */
}

```

```

jmethodID id_Integer = (*env)->GetMethodID(env, class_Integer, "<init>",
    ↳"(I)V");
int value = *(int*) cret;
/* wywołuje konstruktor */
ret = (*env)->NewObject(env, class_Integer, id_Integer, value);
}
else if (type == REG_BINARY)
{
    ret = (*env)->NewByteArray(env, size);
    (*env)->SetByteArrayRegion(env, (jarray) ret, 0, size, cret);
}
else
{
    (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
        "Unsupported value type");
    ret = NULL;
}

free(cret);
RegCloseKey(hkey);
(*env)->ReleaseStringUTFChars(env, name, cname);

return ret;
}

JNIEXPORT void JNICALL Java_Win32RegKey_setValue(JNIEnv* env, jobject this_obj,
    jstring name, jobject value)
{
    const char* cname;
    jstring path;
    const char* cpath;
    HKEY hkey;
    DWORD type;
    DWORD size;
    jclass this_class;
    jclass class_value;
    jclass class_Integer;
    jfieldID id_root;
    jfieldID id_path;
    HKEY root;
    const char* cvalue;
    int ivalue;

    /* ustala klasę */
    this_class = (*env)->GetObjectClass(env, this_obj);

    /* pobiera identyfikatory pól */
    id_root = (*env)->GetFieldID(env, this_class, "root", "I");
    id_path = (*env)->GetFieldID(env, this_class, "path", "Ljava/lang/String;");

    /* pobiera wartości pól */
    root = (HKEY)(*env)->GetIntField(env, this_obj, id_root);
    path = (jstring)(*env)->GetObjectField(env, this_obj, id_path);
    cpath = (*env)->GetStringUTFChars(env, path, NULL);

    /* otwiera klucz rejestru */
    if (RegOpenKeyEx(root, cpath, 0, KEY_WRITE, &hkey) != ERROR_SUCCESS)
    {

```

```

(*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
    "Open key failed");
(*env)->ReleaseStringUTFChars(env, path, cpath);
return;
}

(*env)->ReleaseStringUTFChars(env, path, cpath);
cname = (*env)->GetStringUTFChars(env, name, NULL);

class_value = (*env)->GetObjectClass(env, value);
class_Integer = (*env)->FindClass(env, "java/lang/Integer");
/* ustala typ obiektu value */
if ((*env)->IsAssignableFrom(env, class_value, (*env)->FindClass(env,
    "java/lang/String")))
{
    /* jeśli łańcuch - pobiera wskaźnik do jego znaków */
    cvalue = (*env)->GetStringUTFChars(env, (jstring) value, NULL);
    type = REG_SZ;
    size = (*env)->GetStringLength(env, (jstring) value) + 1;
}
else if ((*env)->IsAssignableFrom(env, class_value, class_Integer))
{
    /* jeśli liczba całkowita — wywołuje intValue, aby pobrać wartość */
    jmethodID id_intValue = (*env)->GetMethodID(env, class_Integer, "intValue",
        "()I");
    ivalue = (*env)->CallIntMethod(env, value, id_intValue);
    type = REG_DWORD;
    cvalue = (char*)&ivalue;
    size = 4;
}
else if ((*env)->IsAssignableFrom(env, class_value, (*env)->FindClass(env,
    "[B")))
{
    /* jeśli tablica bajtów — pobiera wskaźnik bajtów */
    type = REG_BINARY;
    cvalue = (char*)(*env)->GetByteArrayElements(env, (jarray) value, NULL);
    size = (*env)->GetArrayLength(env, (jarray) value);
}
else
{
    /* jeśli jeszcze inny typ, to nie potrafi go obsłużyć */
    (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
        "Unsupported value type");
    RegCloseKey(hkey);
    (*env)->ReleaseStringUTFChars(env, name, cname);
    return;
}

/* Wpisuje wartość do rejestru */
if (RegSetValueEx(hkey, cname, 0, type, cvalue, size) != ERROR_SUCCESS)
{
    (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
        "Set value failed");
}

RegCloseKey(hkey);
(*env)->ReleaseStringUTFChars(env, name, cname);

```

```

/*jeśli wartość była lańcuchem lub tablicą, zwalnia odpowiedni wskaźnik */
if (type == REG_SZ)
{
    (*env)->ReleaseStringUTFChars(env, (jstring) value, cvalue);
}
else if (type == REG_BINARY)
{
    (*env)->ReleaseByteArrayElements(env, (jarray) value, (jbyte*) cvalue, 0);
}

/*funkcja pomocnicza wyliczenia nazw */
static int startNameEnumeration(JNIEnv* env, jobject this_obj, jclass this_class)
{
    jfieldID id_index;
    jfieldID id_count;
    jfieldID id_root;
    jfieldID id_path;
    jfieldID id_hkey;
    jfieldID id_maxsize;

    HKEY root;
    jstring path;
    const char* cpath;
    HKEY hkey;
    DWORD maxsize = 0;
    DWORD count = 0;

    /*pobiera identyfikatory pól */
    id_root = (*env)->GetFieldID(env, this_class, "root", "I");
    id_path = (*env)->GetFieldID(env, this_class, "path", "Ljava/lang/String;");
    id_hkey = (*env)->GetFieldID(env, this_class, "hkey", "I");
    id_maxsize = (*env)->GetFieldID(env, this_class, "maxsize", "I");
    id_index = (*env)->GetFieldID(env, this_class, "index", "I");
    id_count = (*env)->GetFieldID(env, this_class, "count", "I");

    /*pobiera wartości pól */
    root = (HKEY)(*env)->GetIntField(env, this_obj, id_root);
    path = (jstring)(*env)->GetObjectField(env, this_obj, id_path);
    cpath = (*env)->GetStringUTFChars(env, path, NULL);

    /*otwiera klucz rejestru */
    if (RegOpenKeyEx(root, cpath, 0, KEY_READ, &hkey) != ERROR_SUCCESS)
    {
        (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
                           "Open key failed");
        (*env)->ReleaseStringUTFChars(env, path, cpath);
        return -1;
    }
    (*env)->ReleaseStringUTFChars(env, path, cpath);

    /*pobiera liczbę nazw i rozmiar najdłuższej z nazw */
    if (RegQueryInfoKey(hkey, NULL, NULL, NULL, NULL, NULL, NULL, &count, &maxsize,
                        NULL, NULL, NULL) != ERROR_SUCCESS)
    {
        (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
                           "Query info key failed");
        RegCloseKey(hkey);
        return -1;
    }
}

```

```

    }

/* nadaje wartości polom */
(*env)->SetIntField(env, this_obj, id_hkey, (DWORD) hkey);
(*env)->SetIntField(env, this_obj, id_maxsize, maxsize + 1);
(*env)->SetIntField(env, this_obj, id_index, 0);
(*env)->SetIntField(env, this_obj, id_count, count);
return count;
}

JNIEEXPORT jboolean JNICALL Java_Win32RegKeyNameEnumeration_hasMoreElements(JNIEnv* env,
 jobject this_obj)
{
    jclass this_class;
    jfieldID id_index;
    jfieldID id_count;
    int index;
    int count;
/* ustala klasę */
this_class = (*env)->GetObjectClass(env, this_obj);

/* pobiera identyfikatory pól */
id_index = (*env)->GetFieldID(env, this_class, "index", "I");
id_count = (*env)->GetFieldID(env, this_class, "count", "I");

index = (*env)->GetIntField(env, this_obj, id_index);
if (index == -1) /* pierwszy raz? */
{
    count = startNameEnumeration(env, this_obj, this_class);
    index = 0;
}
else
    count = (*env)->GetIntField(env, this_obj, id_count);
return index < count;
}

JNIEEXPORT jobject JNICALL Java_Win32RegKeyNameEnumeration.nextElement(JNIEnv* env,
 jobject this_obj)
{
    jclass this_class;
    jfieldID id_index;
    jfieldID id_hkey;
    jfieldID id_count;
    jfieldID id_maxsize;

HKEY hkey;
int index;
int count;
DWORD maxsize;

char* cret;
jstring ret;

/* ustala klasę */
this_class = (*env)->GetObjectClass(env, this_obj);

/* pobiera identyfikatory pól */

```

```
id_index = (*env)->GetFieldID(env, this_class, "index", "I");
id_count = (*env)->GetFieldID(env, this_class, "count", "I");
id_hkey = (*env)->GetFieldID(env, this_class, "hkey", "I");
id_maxsize = (*env)->GetFieldID(env, this_class, "maxsize", "I");

index = (*env)->GetIntField(env, this_obj, id_index);
if (index == -1) /*pierwszy raz?*/
{
    count = startNameEnumeration(env, this_obj, this_class);
    index = 0;
}
else
    count = (*env)->GetIntField(env, this_obj, id_count);

if (index >= count) /*koniec wyliczenia*/
{
    (*env)->ThrowNew(env, (*env)->FindClass(env,
        "java/util/NoSuchElementException"),
        "past end of enumeration");
    return NULL;
}

maxsize = (*env)->GetIntField(env, this_obj, id_maxsize);
hkey = (HKEY)(*env)->GetIntField(env, this_obj, id_hkey);
cret = (char*)malloc(maxsize);

/*wyszukuje następną nazwę*/
if (RegEnumValue(hkey, index, cret, &maxsize, NULL, NULL, NULL, NULL) !=
    ERROR_SUCCESS)
{
    (*env)->ThrowNew(env, (*env)->FindClass(env, "Win32RegKeyException"),
        "Enum value failed");
    free(cret);
    RegCloseKey(hkey);
    (*env)->SetIntField(env, this_obj, id_index, count);
    return NULL;
}

ret = (*env)->NewStringUTF(env, cret);
free(cret);

/*zwiększa indeks*/
index++;
(*env)->SetIntField(env, this_obj, id_index, index);

if (index == count) /*koniec*/
{
    RegCloseKey(hkey);
}

return ret;
}
```

---

W naszym przykładzie w obiekcie wyliczenia umieścimy jeszcze trzy inne dane. Po rozpoczęciu procesu przeglądania pobierzemy z rejestru liczbę par nazwa-wartość oraz rozmiar najdłuższej nazwy. Informacje te wykorzystamy do utworzenia tablic znakowych języka C przechowujących nazwy i przechowamy za pomocą składowych count i maxsize obiektu

wyliczenia. Natomiast składową `index` zainicjujemy najpierw wartością `-1`, aby rozpoznać początek procesu przeglądania, następnie zmienimy ją na `0` po zainicjowaniu pozostałych składowych i będziemy zwiększać o `1` po każdym kolejnym kroku przeglądania.

Przyjrzyjmy się zatem metodom macierzystym związanym z przeglądaniem nazw. Sposób działania metody `hasMoreElements` jest prosty.

- 1.** Pobiera ona wartość składowych `index` i `count`.
- 2.** Jeśli wartością składowej `index` jest `-1`, to wywołuje funkcję `startNamesEnumeration`, która otwiera klucz, pobiera liczbę nazw i rozmiar najdłuższej z nich i inicjalizuje składowe `hkey`, `count`, `maxsize` i `index`.
- 3.** Zwraca wartość `JNI_TRUE`, jeśli wartość składowej `index` jest mniejsza od wartości `count` lub wartość `JNI_FALSE` — w przeciwnym przypadku.

Metoda `nextElement` jest trochę bardziej skomplikowana.

- 1.** Pobiera wartość składowych `index` i `count`.
- 2.** Jeśli wartością składowej `index` jest `-1`, to wywołuje funkcję `startNamesEnumeration`, która otwiera klucz, pobiera liczbę nazw i rozmiar najdłuższej z nich i inicjalizuje składowe `hkey`, `count`, `maxsize` i `index`.
- 3.** Jeśli wartość składowych `index` i `count` jest taka sama, to wyrzuca wyjątek `NoSuchElementException`.
- 4.** Odczytuje następną nazwę z rejestru.
- 5.** Zwiększa wartość składowej `index` o `1`.
- 6.** Jeśli wartość składowych `index` i `count` jest taka sama, to zamknuje klucz.

Przed komilacją należy pamiętać o uruchomieniu programu `javah` dla plików `Win32RegKey` i `Win32RegKeyNameEnumeration`. Polecenie komilacji w przypadku Microsoft C++ wyglądać będzie następująco:

```
c1 -I jdk\include\ -I jdk\include\win32 -LD
Win32RegKey.c advapi32.lib -FeWin32RegKey.dll
```

Natomiast kompilator pakietu Cygwin wywołamy w poniższy sposób.

```
gcc -mno-cygwin -D __int64="long long" -I jdk\include
-I jdk\include\win32 -I c:\cygwin\usr\include\w32api
-shared -Wl,--add-stdcall-alias -o
Win32regKey.dll Win32Regkey.c
```

Ponieważ rejestr jest cechą charakterystyczną wyłącznie dla systemu Windows, nasz program nie będzie działać w innych systemach.

Listing 12.23 prezentuje program testujący zaimplementowane przez nas metody dostępu do rejestru. Dodaje on do klucza

```
HKEY_CURRENT_USER\Software\JavaSoft\Java Runtime Environment
```

trzy pary nazwa-wartość (łańcuch znaków, całkowitą wartość liczbową i tablicę bajtów).

**Listing 12.23.** win32reg/Win32RegKeyTest.java

```

import java.util.*;

/**
 * @version 1.02 2007-10-26
 * @author Cay Horstmann
 */
public class Win32RegKeyTest
{
    public static void main(String[] args)
    {
        Win32RegKey key = new Win32RegKey(
            Win32RegKey.HKEY_CURRENT_USER, "Software\\JavaSoft\\Java Runtime
            Environment");

        key.setValue("Default user", "Harry Hacker");
        key.setValue("Lucky number", new Integer(13));
        key.setValue("Small primes", new byte[] { 2, 3, 5, 7, 11 });

        Enumeration<String> e = key.names();

        while (e.hasMoreElements())
        {
            String name = e.nextElement();
            System.out.print(name + "=");

            Object value = key.getValue(name);

            if (value instanceof byte[])
                for (byte b : (byte[]) value) System.out.print((b & 0xFF) + " ");
            else
                System.out.print(value);

            System.out.println();
        }
    }
}

```

Następnie program przegląda wszystkie nazwy dla tego klucza i prezentuje ich wartość. Rezultatem jego działania będzie wyświetlenie poniższych napisów

```

Default user=Harry Hacker
Lucky number=13
Small primes=2 3 5 7 11 13

```

Chociaż dodanie tych par do klucza nie powinno zaszkodzić prawidłowemu działaniu rejestru, to jednak lepiej będzie, jeśli usuniemy je po zakończeniu testowania za pomocą edytora rejestru.

#### Funkcje kontroli typów

- `jboolean IsAssignableFrom(JNIEnv *env, jclass c11, jclass c12)`  
zwraca wartość `JNI_TRUE`, jeśli obiekty klasy `c11` mogą być podstawiane za obiekty klasy `c12` lub wartość `JNI_FALSE` — w przeciwnym wypadku. Pierwszy z przypadków

zachodzi, gdy `c11` i `c12` reprezentują tę samą klasę, gdy klasa `c11` jest klasą pochodną klasy `c12` lub `c12` reprezentuje interfejs implementowany przez klasę `c11` lub jedną z jej klas bazowych.

■ `jclass GetSuperClass(JNIEnv *env, jclass c1)`

zwraca klasę bazową danej klasy. Jeśli `c1` reprezentuje klasę `Object` lub interfejs, to funkcja zwraca wartość `NULL`.

W ten sposób dotarłeś do końca książki *Java 2. Techniki zaawansowane*. Długa lektura pozwoliła Ci poznać wiele zaawansowanych interfejsów programowych. Rozpoczęłeś ją od poznania zagadnień, które powinien opanować każdy programista języka Java: strumieni, XML, programowania aplikacji sieciowych, obsługi baz danych i internacjonalizacji. Kolejne trzy obszerne rozdziały poświęcone były tworzeniu grafiki i programowaniu interfejsu użytkownika. Na koniec kilka wyjątkowo „technicznych” rozdziałów poświęciliśmy problematyce bezpieczeństwa, metodom zdalnym, przetwarzaniu adnotacji i metodom macierzystym. Mamy nadzieję, że podróż, którą odbyłeś po krainie interfejsów programowych platformy Java, pozwoli Ci spożytkować nabycią wiedzę w wielu nowych, ciekawych projektach.



# Skorowidz

## A

- abort(), 776
- absolute(), 268
- AbstractButton, 673
- AbstractCellEditor, 391, 392
- AbstractListModel, 350
- AbstractTableModel, 363, 385
- accept(), 191, 193
- acceptChanges(), 272, 273
- ActionListener, 835
- ActionListenerFor, 835, 838
- ActionListenerInstaller, 835
  - processAnnotations(), 835
- actionPerformed(), 835
- Activatable, 885, 889
  - exportObject(), 886, 889
  - register(), 886, 889
- ActivationDesc, 890
- ActivationGroup, 890
  - getSystem(), 890
- ActivationGroupDesc, 890
- ActivationSystem, 890
  - registerGroup(), 890
- add(), 403, 610
- addActionListener(), 833, 836
- addAttribute(), 181
- addBatch(), 286
- addCellEditorListener(), 394
- addChangeListener(), 479, 483
- addClass(), 420
- addColumn(), 374, 380
- addElement(), 354, 355
- addFlavorListener(), 617
- addHyperlinkListener(), 462
- addListSelectionListener(), 349
- addPropertyChangeListener(), 680, 682, 699
- addSelectionListener(), 415
- addTab(), 478, 482
- addTreeModelListener(), 426, 428
- addVetoableChangeListener(), 495, 500, 676, 681, 682
- adnotacje
  - @ActionListenerFor, 832, 843
  - @BugReport, 844
  - @Deprecated, 841, 842
  - @Documented, 841, 844
  - @Inherited, 841, 844
  - @interface, 832
  - @LogEntry, 852
  - @Override, 841, 842
  - @Persistent, 845
  - @Property, 845
  - @Retention, 832, 841, 843
  - @Serializable, 844
  - @SuppressWarnings, 841, 842
  - @Target, 832, 841, 843
  - @TestCase, 831
- ActionListenerFor, 835, 838
- cykliczne zależności, 839
- deklaracja elementu, 837
- elementy, 831
- format, 837
- interfejs, 831, 837
- kod bajtowy, 851
- kolejność elementów, 837
- metaadnotacje, 843
- metody, 831
- modyfikacja kodu bajtowego podczas ładowania, 857
- obsługa zdarzeń, 832
- pojedyncze wartości, 838
- przetwarzanie, 845
- regularne, 842
- składnia, 837
- skróty, 838
- standardowe, 841

- adnotacje
  - typy elementów, 839, 843
  - zmienne lokalne, 840
  - znacznikowe, 838
- adres internetowy, 189
- adres localhost, 193
- adres URI, 205
- adres URL, 148, 204, 205, 458, 750
- AES, 796, 797, 803
- AffineTransform, 536, 537
  - getRotateInstance(), 537, 538
  - getScaleInstance(), 538
  - getShearInstance(), 538
  - getTranslateInstance(), 538
  - setToRotation(), 538
  - setToScale(), 538
  - setToShear(), 538
  - setToTranslation(), 538
- AffineTransformOp, 572, 578
  - TYPE\_BILINEAR, 572
  - TYPE\_NEAREST\_NEIGHBOR, 572
- afterLast(), 269
- aktualizacje wsadowe, 284
- aktualizowalne zbiory wyników zapytań, 263, 265
- aktywacja obiektów serwera, 884
- alfa, 542
- algorytmy
  - AES, 796, 797, 803
  - DES, 795
  - DSA, 780
  - kryptograficzne, 728
  - MD5, 777, 778
  - RSA, 780, 803
  - SHA1, 777
  - szyfrowania, 795
    - z kluczem symetrycznym, 803
- AllPermission, 752
- AllPermissions, 746, 755
- AlphaComposite, 544
  - getInstance(), 549
- animacje GIF, 562
- AnnotatedElement, 835, 836
  - getAnnotation(), 836
  - getAnnotations(), 836
  - getDeclaredAnnotations(), 836
  - isAnnotationPresent(), 836
- Annotation, 838, 840
  - annotationType(), 840
  - equals(), 840
  - hashCode(), 841
  - toString(), 841
- annotationType(), 840
- antialiasing, 549
- Apache, 288
- aplety, 230, 663, 776
  - lokalizacja, 328
- aplikacje interaktywne, 198
- aplikacje klient-serwer, 230
- aplikacje rozproszone, 861
- append(), 520, 523, 599
- appendChild(), 160, 164
- applets.policy, 792
- appletviewer, 743, 748, 790
- applyPattern(), 319
- apt, 845
- Arc2D, 509, 511
- Arc2D.Double, 522
- architektura JDBC, 228
- architektura klient-serwer, 230
- architektura n-warstwowa, 230
- architektura trójwarstwowa, 230
- ArcMaker, 521
- Area, 523, 524
- ARGB, 567, 571
- arkusz właściwości, 658
- Array, 287
- ARRAY, 287
- ArrayIndexOutOfBoundsException, 923
- ArrayList, 427, 885
- ArrayStoreException, 923
- ASCII, 291
- ASP, 216
- atrybuty, 105, 122
- atrybuty drukowania, 604, 607
  - dokumenty, 604
  - hierarchia, 605
  - klasy, 607
  - usługi drukowania, 604
  - wydruk, 604
  - zbiór, 606
  - żądanie wydruku, 604
- AttributeSet, 604
- ATTLIST, 122, 128
- Attribute, 604, 606, 609
  - getCategory(), 609
  - getName(), 609
- Attributes, 155
  - getLength(), 155
  - getLocalName(), 155
  - getQName(), 156
  - getURI(), 156
  - getValue(), 156
- AttributeSet, 610
  - add(), 610
  - get(), 610
  - remove(), 610
  - toArray(), 610
- AttributesImpl, 181
  - addAttribute(), 181

- AudioPermission, 752  
AuthPermission, 752  
AuthTest.policy, 766  
AWT  
  drukowanie, 580  
  figury, 508  
  filtrowanie obrazów, 571  
  Graphics, 505, 508  
  obszar przycięcia, 507  
  operacje na obrazach, 565  
  pliki graficzne, 555  
  pola, 523  
  potokowe tworzenie grafiki, 506  
  przeciągnij i upuść, 625  
  przekształcenia układu współrzędnych, 507, 534  
  przezroczystość, 541  
  przycinanie, 539  
  rysowanie figur, 506  
  schowek, 610  
  składanie obrazów, 541  
  ślad pędzla, 507, 524  
  wskazówki operacji graficznych, 549  
  wypełnianie obszaru, 507, 532  
  zasady składania obrazów, 507  
AWTPermission, 751
- B**
- Banner, 590  
Base64, 210  
base64Encode(), 209  
Base64Encoder, 214  
BasicPermission, 752  
BasicStroke, 524, 525, 530  
baza danych, 227  
  adres URL, 237  
  aktualizowalne zbiory wyników zapytań, 265  
JNDI, 288  
kolumny, 231  
kursory, 264  
łączenie tabel, 232, 233  
metadane, 274  
model dostępu, 228  
modyfikacja danych, 235  
ODBC, 228  
przewijanie zbioru rekordów, 264  
rekordy, 231  
rekordy wstawiania, 266  
spójność, 283  
SQL, 227  
tabele, 231  
transakcje, 283  
wstawianie danych, 235  
wypełnianie, 248  
zapytania, 232
- zarządzanie połączaniami, 288  
zbiory rekordów, 270  
BCEL, 852  
BEA WebLogic, 288  
Bean Builder, 666  
BeanClass, 846  
BeanDescriptor, 699  
BeanInfo, 683, 684, 685, 698  
  getBeanDescriptor(), 698  
  getIcon(), 685  
  getPropertyDescriptors(), 685, 690  
BeanInfoAnnotationProcessor, 850  
beforeFirst(), 268  
bezpieczeństwo  
  hierarchia klas pozwoleń, 745  
  JAAS, 762  
  Java 2, 744  
  java.policy, 748  
  Kerberos, 806  
  klasy pozwoleń, 755  
  kryptografia klucza publicznego, 803  
  ładowanie klas, 728  
  menedżer bezpieczeństwa, 728, 742  
  piaskownica, 744  
  pliki polityki, 744, 747  
  podpis cyfrowy, 776  
  podpisywanie kodu, 788  
  polityka bezpieczeństwa, 745  
  pozwolenia, 742, 745  
  przydzielanie praw, 744  
  skróty wiadomości, 777  
  SSL, 806  
  szifrowanie, 795  
  uwierzytelnianie użytkowników, 762  
  uwierzytelnianie wiadomości, 784  
  weryfikacja kodu maszyny wirtualnej, 738  
źródło kodu, 744  
biblioteki  
  BCEL, 852  
  DLL, 895  
  JCE, 801  
BigDecimal, 287  
bind(), 870, 871  
BitSet, 713  
BLOB, 236, 287  
Book, 598, 599  
  append(), 599  
  getPrintable(), 599  
boolean, 236, 287  
breadthFirstEnumeration(), 410, 414  
Buffer, 198  
BufferedImage, 533, 545, 565, 570  
  getColorModel(), 570  
  getRaster(), 570

BufferedImageOp, 565, 571, 572, 578

filter(), 578

buforowane zbiory rekordów, 271

ButtonFrame, 835

ByteLookupTable, 576, 579

## C

C, 892

łańcuchy znakowe platformy Java, 902

obsługa błędów, 923, 927

tablice, 919

wywoływanie metod języka Java, 912, 917

wywoływanie metod statycznych, 916

C++, 895

CachedRowSet, 270, 271, 273

acceptChanges(), 273

execute(), 273

getTableName(), 273

populate(), 273

setTableName(), 273

Caesar, 736

CalendarBean, 661

Callback, 773

CallbackHandler, 775

handle(), 775

CallNonVirtualXxxMethod(), 917, 918

CallNonVirtualXxxMethodA(), 918

CallNonVirtualXxxMethodV(), 918

CallStaticObjectMethod(), 916

CallStaticXxxMethod(), 916, 918

CallStaticXxxMethodA(), 918

CallStaticXxxMethodV(), 918

CallXxxMethod(), 917

CallXxxMethodA(), 917

CallXxxMethodV(), 918

cancelCellEditing(), 391, 392, 394

cancelRowUpdates(), 269

canInsertImage(), 561, 564

catch, 246

CDATA, 106, 123

CellEditor, 394

addCellEditorListener(), 394

cancelCellEditing(), 394

getCellEditorValue(), 394

isCellEditable(), 394

removeCellEditorListener(), 394

shouldSelectCell(), 394

stopCellEditing(), 394

certyfikaty, 744

certyfikaty twórców oprogramowania, 793

certyfikaty X.509

keytool, 781

komponenty, 782

podpisywanie, 786

składnica kluczy, 782

sprawdzanie wiarygodności, 783

wydawcy certyfikatów, 783

zarządzanie, 781

CGI, 216

Channels, 204

newInputStream(), 204

newOutputStream(), 198, 204

CHAR, 236, 287

CHARACTER, 236

CharacterData, 118

getData(), 118

characters(), 151, 155

ChartBean, 846

ChartBeanBeanInfo, 688, 845

checkExit(), 743, 744, 746

checkLogin(), 769

checkPermission(), 747, 755, 756

children(), 409

ChoiceFormat, 321

Chromaticity, 607

Cipher, 795, 800

doFinal(), 801

getBlockSize(), 800

getInstance(), 800

getOutputSize(), 800

init(), 800

update(), 800

CipherInputStream, 802

read(), 802

CipherOutputStream, 802

flush(), 802

write(), 802

Class, 411, 606, 737, 747, 835

getClassLoader(), 737

getProtectionDomain(), 747

ClassLoader, 729, 733, 737

defineClass(), 738

findClass(), 738

getParent(), 737

getSystemClassLoader(), 737

ClassNameTreeCellRenderer, 413

CLASSPATH, 729

ClassTreeFrame, 414

CLEAR, 543

clearParameters(), 258

clip(), 507, 539, 541

Clipboard, 611, 614, 617, 625

addFlavorListener(), 617

getAvailableDataFlavors(), 617

getContents(), 614

getData(), 615

getTransferDataFlavors(), 617

isDataFlavorAvailable(), 615

setContents(), 615

ClipboardOwner, 612, 615  
 lostOwnership(), 615  
 Clob, 287  
 CLOB, 236, 287  
 clone(), 884  
 CloneNotSupportedException, 884  
 close(), 191, 193, 243, 244, 245, 467, 474  
 closePath(), 514, 523  
 CodeSource, 747  
   getCertificates(), 747  
   getLocation(), 747  
 CollationKey, 318  
   compareTo(), 318  
 Collator, 312, 317  
   compare(), 317  
   equals(), 317  
 getAvailableLocales(), 317  
 getCollationKey(), 318  
 getDecomposition(), 317  
 getInstance(), 317  
 getStrength(), 317  
 PRIMARY, 312  
 setDecomposition(), 317  
 setStrength(), 317  
 Collections  
   sort(), 312  
 Color, 532, 542, 567, 621  
   getRGB(), 571  
 ColorConvertOp, 577  
 ColorModel, 568, 571  
   getDataElements(), 571  
   getRGB(), 571  
 ColorSupported, 607  
 column(), 245  
 commit(), 284, 286, 776  
 Common Object Request Broker Architecture, 863  
 CommonDialog, 658  
 Comparator, 312  
 compare(), 317  
 compareTo(), 311, 312, 318  
 Compression, 607  
 CONCUR\_READ\_ONLY, 264  
 CONCUR\_UPDATABLE, 264, 266  
 connect(), 188, 207, 215  
 Connection, 245, 258, 268, 282  
   close(), 243  
   commit(), 286  
   createStatement(), 243, 268  
   getAutoCommit(), 285  
   getMetaData(), 282  
   prepareStatement(), 258, 268  
   releaseSavepoint(), 286  
   rollback(), 286  
   setAutoCommit(), 285  
     setSavepoint(), 286  
 Constructor, 835  
 containsAll(), 759  
 ContentHandler, 151, 152, 155  
   characters(), 155  
   endDocument(), 155  
   endElement(), 155  
   startDocument(), 155  
   startElement(), 155  
 Context, 870  
   bind(), 870  
   list(), 871  
   lookup(), 870  
   rebind(), 871  
   unbind(), 870  
 CONTIGUOUS\_TREE\_SELECTION, 415  
 ConvolveOp, 578, 579  
 Copies, 604, 606, 607  
 CopiesSupported, 604  
 CORBA, 863  
 COREJAVA, 236  
 CREATE TABLE, 235, 242, 249  
 createElement(), 160, 163  
 createImageInputStream(), 560, 563  
 createImageOutputStream(), 561, 563  
 createPrintJob(), 602  
 createStatement(), 242, 243, 263, 268, 284, 285  
 createTextNode(), 160, 164  
 CubicCurve2D, 509, 513, 522  
 Currency, 302, 303  
   getCurrencyCode(), 303  
   getdefaultFractionsDigits(), 303  
   getInstance(), 303  
   getSymbol(), 303  
   toString(), 303  
 curveTo(), 513, 514, 522  
 Customizer, 699  
   setObject(), 705  
 Cygwin, 895, 932  
 czas, 304  
 czerwionki, 523, 540  
   obrys, 540  
 DA, 607  
 DefaultTreeModel, 404  
 data, 292, 304  
   parsowanie, 308  
 database.properties, 288  
 DatabaseMetaData, 245, 265, 269, 274, 275, 286  
   getJDBCMajorVersion(), 282  
   getJDBCMinorVersion(), 282  
   getMaxConnections(), 282  
   getMaxStatements(), 282



getTables(), 282  
supportsBatchUpdates(), 286  
supportsResultSetConcurrency(), 270  
supportsResultSetType(), 269  
DataFlavor, 611, 615, 616  
  getHumanPresentableName(), 617  
  getMimeType(), 616  
  getRepresentationClass(), 617  
  imageFlavor, 617, 618  
  isMIMETypeEqual(), 617  
DataSource, 288  
Date, 287  
DATE, 236, 287  
DateFormat, 295, 304, 305  
  DEFAULT, 304  
  format(), 310  
  FULL, 304  
  getAvailableLocales(), 309  
  getCalendar(), 310  
  getDateInstance(), 304, 309  
  getDateTimeInstance(), 309  
  getNumberFormat(), 310  
  getTimeInstance(), 304, 309  
  getTimeZone(), 310  
  isLenient(), 310  
  LONG, 304  
  MEDIUM, 304  
  parse(), 310  
  setCalendar(), 310  
  setLenient(), 310  
  setNumberFormat(), 310  
  setTimeZone(), 310  
  SHORT, 304  
DateFormat.getDateInstance(), 304  
DateTimeAtCompleted, 607  
DateTimeAtCreation, 607  
DateTimeAtProcessing, 607  
DDL, 244  
DEC, 287  
DECIMAL, 236, 287  
decode(), 222  
DefaultCellEditor, 389, 392, 393, 406  
DefaultHandler, 152  
DefaultListModel, 354, 355  
  addElement(), 355  
  removeElement(), 355  
DefaultModelList, 354  
DefaultMutableTreeNode, 396, 401, 402, 410, 414  
  add(), 403  
  breadthFirstEnumeration(), 414  
  depthFirstEnumeration(), 414  
  postOrderEnumeration(), 414  
  preOrderEnumeration(), 414  
  setAllowsChildren(), 403  
defaultPage(), 588  
DefaultPersistenceDelegate, 711, 713, 725  
  initialize(), 725  
  instantiate(), 725  
DefaultTreeCellRenderer, 412, 413, 414  
  setClosedIcon(), 414  
  setLeafIcon(), 414  
  setOpenIcon(), 414  
DefaultTreeModel, 396, 402, 405, 409, 422  
  insertNodeInto(), 409  
  nodeChanged(), 410  
  reload(), 410  
  removeNodeFromParent(), 410  
  setAsksAllowsChildren(), 402  
defineClass(), 733, 738  
definicja typu dokumentu, 119  
deklaracja typu dokumentu, 104  
delegat trwałości, 710  
DELETE, 242  
deleteRow(), 267, 269  
depthFirstEnumeration(), 410, 414  
depthFirstTraversal(), 411  
DES, 795, 796  
deskryptory aktywacji, 884  
deskryptory wdrożeń, 830  
Destination, 607  
DestroyJavaVM(), 928, 932  
DialogCallbackHandler, 772  
digest(), 779  
DISCONTIGUOUS\_TREE\_SELECTION, 415  
DLL, 895  
doAs(), 763, 764, 767  
doAsPrivileged(), 763, 764, 767  
Doc, 601  
DocAttribute, 604, 607  
DocAttributeSet, 606  
DocFlavor, 599  
DocPrintJob, 601, 602  
  getAttributes(), 610  
  print(), 602  
DOCTYPE, 120, 161  
Document, 108, 111, 117, 163  
  createElement(), 163  
  createTextNode(), 164  
  getDocumentElement(), 117  
DocumentBuilder, 107, 111, 116, 125, 160, 163  
  parse(), 116  
  setEntityResolver(), 125  
  setErrorHandler(), 125  
DocumentBuilderFactory, 116, 124, 126, 149, 150  
  isIgnoringElementContentWhitespace(), 126  
  isNamespaceAware(), 150  
  isValidating(), 126  
  newDocumentBuilder(), 116

- newInstance(), 116  
setIgnoringElementContentWhitespace(), 126  
setNamespaceAware(), 150  
setValidating(), 126  
DocumentName, 607  
doFinal(), 796, 801  
dokumenty XML, 104  
  analiza zawartości, 108  
  atrybuty, 105  
  CDATA, 106  
  deklaracja typu dokumentu, 104  
  DOCTYPE, 120  
  DTD, 106  
  element korzenia, 104  
  elementy, 108  
  elementy podrzędne, 109, 116  
  komentarze, 106  
  kontrola poprawności, 118  
  nagłówek, 104  
  NodeList, 108  
  parsowanie, 107  
  PCDATA, 121  
  pobieranie węzłów, 110  
  przeglądanie atrybutów, 110  
  tworzenie, 160  
  tworzenie drzewa DOM, 160  
  wczytywanie, 107  
  wyszukiwanie informacji, 142  
  XML Schema, 126  
  XPath, 142  
DOM, 107, 120  
  analiza drzewa, 111  
  drzewo, 109  
  tworzenie drzewa, 160  
domena ochronna, 746  
DOMResult, 177, 181  
DOMSource, 165, 176  
DOMTreeModel, 116  
domyślny edytor komórki, 406  
doPost(), 221  
dostęp do składowych, 906  
  pola instancji, 910  
  pola statyczne, 910  
double, 236, 287  
DoubleArrayEditor, 688  
draw(), 507, 508, 509  
draw3DRect(), 508, 509  
drawArc(), 508  
drawLine(), 508  
drawOval(), 508  
drawPolygon(), 508, 509  
drawPolyline(), 508  
drawRect(), 506  
drawRectangle(), 508  
drawRoundRect(), 508  
DriverManager, 242, 288  
  getConnection(), 242  
  setLogWriter(), 242  
DROP TABLE, 242  
drukowanie, 363, 580, 610  
  algorytm rozmieszczenia materiału, 589  
  atrybuty, 604  
  format strony, 582  
  grafika, 580  
  JDK, 580  
  liczba stron, 582  
  marginesy, 583  
  podgląd wydruku, 591  
  PostScript, 603  
  Printable, 580  
  przyjęcie kontekstu graficznego, 582  
  rozmieszczenie materiału na stronach, 590  
  transparent, 590  
  usługi, 599  
  wiele stron, 589  
  wymiary strony, 583  
  zadania, 581  
drzewa, 394  
  domyślny edytor komórki, 406  
  dziedziczenie, 411  
  edycja węzłów, 406  
  ikony liści, 401  
  JTree, 394, 395  
   kolejność przeglądania węzłów, 410  
  korzeń, 395, 397  
  las, 395, 400  
  liście, 395  
  model, 396  
  modyfikacja ścieżek, 403  
  modyfikacje, 403  
  nasłuchiwanie zdarzeń, 415  
  obiekt nasłuchujący wyboru, 415  
  obiekt rysujący komórki, 412  
  obiekt użytkownika, 396  
  powiązania między węzłami, 397  
  przeglądanie od końca, 410  
  przeglądanie w głąb, 410  
  przeglądanie węzłów, 410  
  przewijalny panel, 405  
  rozwijanie ścieżek, 405  
  rysowanie węzłów, 412  
  struktura, 427  
  ścieżki, 403  
  tworzenie modeli, 421  
  węzły, 395  
  węzły nadzędne, 395  
  węzły podrzędne, 395  
  wstawianie węzłów, 406

drzewa  
wygląd, 398, 412  
zdarzenia, 415  
zmiana struktury węzła, 404  
zwinięcie, 400  
DST, 543  
DST\_ATOP, 543  
DST\_IN, 543  
DST\_OUT, 543  
DST\_OVER, 543  
DTD, 106, 119, 124  
DTDHandler, 152  
dynamiczne ładowanie klas, 878  
dziedziczenie, 411

**E**

edycja plików polityki, 754  
edytor rejestru, 933  
edytor właściwości, 667, 687  
  graficzny, 694  
  implementacja, 690  
  tworzenie, 687  
  właściwości proste, 690  
  złożone właściwości, 693  
EJB, 658  
Element, 117, 164  
  getAttribute(), 117  
  getTagName(), 117  
  setAttribute(), 164  
  setAttributeNS(), 164  
ELEMENT, 119, 121, 131  
elipsy, 521  
Ellipse2D, 509, 511  
EmployeeReader, 177  
encode(), 222  
Encoder, 724  
  getExceptionListener(), 724  
  getPersistenceDelegate(), 724  
  setExceptionListener(), 724  
  setPersistenceDelegate(), 724  
endDocument(), 151, 155  
endElement(), 151, 155  
Enterprise JavaBeans, 658  
ENTITY, 123  
EntityResolver, 107, 125, 152  
  resolveEntity(), 125  
EntryLogger, 854, 858  
Enumeration, 243  
EnumSyntax, 607  
env, 901, 928  
equals(), 317, 759, 840, 884  
error(), 125, 126  
ErrorHandler, 125, 126, 152  
  error(), 126

fatalError(), 126  
warning(), 126  
evaluate(), 143, 147  
Event, 671  
EventHandler, 709, 836  
EventListenerList, 426  
EventObject, 673  
EventSetDescriptor, 684  
ExceptionCheck(), 927  
ExceptionClear(), 927  
ExceptionListener, 724  
  exceptionThrown(), 724  
ExceptionOccured(), 923, 924, 927  
exceptionThrown(), 724  
ExecSQL, 249  
execute(), 244, 271, 273  
executeBatch(), 285, 286  
executeQuery(), 242, 244, 253, 258  
executeUpdate(), 242, 244, 253, 258, 284  
exit(), 743  
exitInternal(), 744  
exportObject(), 867, 886, 889  
Expression, 725  
extern "C", 895

**F**

fatalError(), 125, 126  
FeatureDescriptor, 685  
  getDisplayName(), 685  
  getName(), 685  
  getShortDescription(), 686  
  isExpert(), 686  
  isHidden(), 686  
  setDisplayName(), 685  
  setExpert(), 686  
  setHidden(), 686  
  setName(), 685  
  setShortDescription(), 686  
Fidelity, 607  
Field, 835, 915  
figury, 506, 508, 509, 510, 521  
  łuk, 511  
  odcinki, 513  
  prostokąt, 511  
  przycinanie, 539  
  punkty kontrolne, 521  
  rysowanie, 506  
  tworzenie, 521  
wielokąty, 514  
wypełnianie, 508, 532  
File, 397  
FileInputStream, 470, 746  
FilePermission, 745, 750  
FileReader, 746

fill(), 507, 508  
 fillOval(), 506  
 Filter, 659  
 filter(), 578  
 FilteredRowSet, 270  
 filtrowanie obrazów, 571  
     interpolacja, 572  
     negatyw, 576  
     rozmycie, 577  
     wykrywanie krawędzi, 578  
 findClass(), 733, 738  
 FindClass(), 910  
 findEditor(), 700  
 Finishings, 607  
 fireIndexedPropertyChange(), 680  
 firePropertyChange(), 675, 680, 682  
 fireVetoableChange(), 677, 682  
 first(), 268  
 FIXED, 123  
 FlavorListener, 616  
     flavorsChanged(), 617  
 flavorsChanged(), 617  
 float, 287  
 FLOAT, 236, 287  
 flush(), 802  
 focusCycleRoot, 667  
 FontType, 128  
 Format, 320  
 format liczby, 297  
 format XML, 864  
 format(), 301, 310, 319, 320  
 formatki, 658, 666  
     tworzenie, 666  
 formatowanie komunikatów, 318  
     warianty, 320  
 formatowanie liczb, 297, 899  
 formularze, 216  
     HTML, 217  
     metoda GET, 218  
     metoda POST, 218  
     nazwy pól, 219  
     odpowiedź serwera, 221  
     przetwarzanie danych, 216  
     serwlety, 216  
     skrypty CGI, 216  
     Submit, 216  
         wysyłanie informacji do serwera, 218  
 fprintf(), 924  
 fprintff(), 912  
 fraktale, 568  
 FROM, 234  
 FTP, 209  
 funkcje języka C, 892

**G**

GeneralPath, 509, 513, 520, 521, 522, 523, 530, 540  
     append(), 523  
     closePath(), 523  
     curveTo(), 522  
     lineTo(), 522  
     moveTo(), 522  
     quadTo(), 522  
 generateKey(), 797, 801  
 generowanie liczb losowych, 797  
 geometria pól, 523  
 GET, 218  
 get(), 610, 671  
 getAddress(), 189, 190  
 getAdvance(), 541  
 getAllByName(), 189, 190  
 getAllFrames(), 492, 498  
 getAllowsChildren(), 402  
 getAllowUserInteraction(), 214  
 getAnnotation(), 835, 836  
 getAnnotations(), 836  
 GetArrayLength(), 919, 922  
 getAscent(), 541  
 getAsText(), 690, 692, 694, 696  
 getAttribute(), 117, 131  
 getAttributes(), 110, 117, 610  
 getAutoCommit(), 285  
 getAvailableDataFlavors(), 616, 617  
 getAvailableIDs(), 310  
 getAvailableLocales(), 295, 298, 304, 309, 317  
 getBackground(), 356, 358  
 getBeanDescriptor(), 698  
 getBeanInfo(), 684, 685  
 getBlockSize(), 800  
 GetBooleanArrayElements(), 921  
 getBundle(), 324, 325, 326, 328  
 getByName(), 189, 190  
 GetByteArrayElements(), 936  
 getCalendar(), 310  
 getCategory(), 606, 609  
 getCellEditorValue(), 389, 392, 394  
 getCellSelectionEnabled(), 380  
 getCertificates(), 747  
 getChild(), 116, 422, 427, 428  
 getChildAt(), 409  
 getChildCount(), 404, 409, 428  
 getChildNodes(), 117  
 getClassLoader(), 729, 737  
 getClassName(), 871  
 getClip(), 541  
 getCodeSource(), 747  
 getCollationKey(), 313, 318  
 getColorModel(), 566, 570  
 getColumn(), 380

getColumnClass(), 367  
getColumnCount(), 283, 363, 364, 366  
getColumnDisplaySize(), 283  
getColumnLabel(), 283  
getColumnName(), 283, 367  
getColumnNumber(), 126  
getColumnSelectionAllowed(), 379  
getCommand(), 273  
getComponentAt(), 483  
getConcurrency(), 268  
getConnection(), 239, 242, 249, 288  
getConnectTimeout(), 215  
getContent(), 216  
getContentEncoding(), 210, 215  
getContentLength(), 210, 215  
getContentPane(), 485, 499  
getContents(), 614  
getContentType(), 210, 215  
getContextClassLoader(), 738  
getCountry(), 296  
getCurrencyCode(), 303  
getCurrencyInstance(), 297, 302  
getCustomEditor(), 694, 697  
getData(), 110, 118, 615  
getDataElements(), 567, 570, 571  
getDate(), 210, 215  
getDateInstance(), 309  
getTimeInstance(), 304, 309  
getDeclaredAnnotations(), 836  
getDecomposition(), 317  
getDefault(), 295, 296, 310  
getDefaultEditor(), 393  
getdefaultFractionsDigits(), 303  
getDefaultName(), 775  
getDefaultRenderer(), 393  
getDefaultToolkit(), 612  
getDescent(), 541  
getDesktopPane(), 499  
getDisplayCountry(), 296  
getDisplayLanguage(), 296  
getDisplayName(), 295, 296, 298, 311, 685  
getDocument(), 459  
getDocumentElement(), 108, 117  
getDoInput(), 214  
getDoOutput(), 214  
getDouble(), 243  
GetDoubleField(), 906  
getElementAt(), 350, 354  
getErrorCode(), 248  
getErrorStream(), 221, 222  
getEventType(), 459  
getExceptionListener(), 724, 725  
getExpiration(), 210, 215  
getFieldDescription(), 420  
GetFieldID(), 907, 910  
getFields(), 427  
getFileSuffixes(), 564  
getFirstChild(), 117  
getFontRenderContext(), 540  
getFontRendererContext(), 541  
getForeground(), 356, 358  
getFormatNames(), 564  
getFrameIcon(), 499  
getHeaderField(), 207, 209, 215  
getHeaderFieldKey(), 207, 209, 215  
getHeaderFields(), 207, 210, 215  
getHeight(), 564, 582, 583, 588  
getHostAddress(), 190  
getHostName(), 190  
getHumanPresentableName(), 617  
getIcon(), 685  
getIconAt(), 483  
getID(), 311  
getIfModifiedSince(), 214  
getImageableHeight(), 583, 588  
getImageableWidth(), 583, 588  
getImageableX(), 589  
getImageableY(), 589  
getImageReadersByFormatName(), 562  
getImageReadersByMIMEType(), 556, 562  
getImageReadersBySuffix(), 556, 562  
getImageWritersByFormatName(), 562  
getImageWritersByMIMEType(), 562  
getImageWritersBySuffix(), 562  
getIndex(), 681  
getIndexedReadMethod(), 687  
getIndexedWriteMethod(), 687  
getIndexOfChild(), 422, 428  
getInputStream(), 187, 191, 208, 219, 221  
getInstance(), 303, 312, 317, 544, 549, 778, 795,  
  800, 801  
GetIntField(), 906, 936  
getJavaInitializationString(), 697  
getJDBCMajorVersion(), 282  
getJDBCMinorVersion(), 282  
getKeys(), 328  
getLanguage(), 296  
getLastChild(), 110, 117  
getLastModified(), 210, 216  
getLastPathComponent(), 404, 409  
getLastSelectedPathComponent(), 404, 409  
getLayoutOrientation(), 348  
getLayoutPolicy(), 483  
getLeading(), 541  
getLength(), 108, 118, 155  
getLineNumber(), 126  
getListCellRendererComponent(), 358, 359  
getLocale(), 319

getLocalHost(), 189, 190  
 getLocalName(), 150, 155  
 getLocation(), 747  
 getLogger(), 852  
 getMaxConnections(), 282  
 getMaximum(), 473  
 getMaximumFractionDigits(), 302  
 getMaximumIntegerDigits(), 302  
 getMaxStatement(), 245  
 getMaxStatements(), 282  
 getMetaData(), 282, 283  
 GetMethodID(), 917, 919  
 getMIMEType(), 616  
 getMIMETypes(), 564  
 getMinimum(), 473  
 getMinimumFractionDigits(), 302  
 getMinimumIntegerDigits(), 302  
 getModel(), 354, 355  
 getName(), 601, 609, 685, 758, 761, 767, 768, 775, 871  
 getNameSpaceURI(), 150  
 getNewValue(), 500, 681  
 getNextException(), 247  
 getNextSibling(), 110, 117  
 getNodeName(), 110, 117, 150  
 getNodeValue(), 110, 117  
 getNumberFormat(), 310  
 getNumberInstance(), 297  
 getNumImages(), 563  
 getNumThumbnails(), 560, 563  
 getObject(), 328  
 GetObjectArrayElement(), 919, 922  
 GetObjectClass(), 907, 910  
 GetObjectField(), 906  
 getOldValue(), 681  
 getOrientation(), 589  
 getOriginatingProvider(), 556, 564, 565  
 getOutline(), 540  
 getOutputSize(), 800  
 getOutputStream(), 187, 191, 208, 218  
 getPageCount(), 590  
 getParent(), 409, 737  
 getParentNode(), 117  
 getPassword(), 273, 775  
 getPath(), 421  
 getPaths(), 421  
 getPathToRoot(), 405  
 getPercentInstance(), 297  
 getPersistenceDelegate(), 724  
 getPixel(), 566, 570, 571  
 getPixels(), 570  
 getPointCount(), 521  
 getPreferredSize(), 356, 358  
 getPreviousSibling(), 117  
 getPrincipals(), 767  
 getPrintable(), 599  
 getPrinterJob(), 581, 588  
 getPrintService(), 604  
 getPrompt(), 775  
 getProperty(), 916  
 getPropertyChangeEvent(), 683  
 getPropertyChangeListeners(), 680  
 getPropertyDescriptors(), 683, 685, 687, 688, 690  
 propertyName(), 500, 671, 681  
 getPropertyType(), 686  
 getProtectionDomain(), 747  
 getPrototypeCell(), 353  
 getQName(), 156  
 getRaster(), 565, 570  
 getReaderFormatNames(), 563  
 getReaderMIMETypes(), 563  
 getReadMethod(), 686  
 getRepresentationClass(), 617  
 getRequestProperties(), 215  
 getResultSet(), 244  
 getRGB(), 567, 571  
 getRoot(), 116, 428  
 getRotateInstance(), 536, 537, 538  
 getRow(), 268  
 getRowCount(), 363, 364, 366  
 getRowHeight(), 379  
 getRowMargin(), 379  
 getRowSelectionAllowed(), 379  
 getSavepointId(), 286  
 getSavepointName(), 286  
 getScaleInstance(), 536, 538  
 getSelectedColumns(), 374  
 getSelectedComponent(), 482  
 getSelectedIndex(), 480, 482  
 getSelectedNode(), 404  
 getSelectedValue(), 349  
 getSelectedValues(), 346, 349  
 getSelectionBackground(), 358  
 getSelectionForeground(), 358  
 getSelectionMode(), 349  
 getSelectionModel(), 370, 379  
 getSelectionPath(), 409, 416, 420, 421  
 getSelectionPaths(), 416, 421  
 getShearInstance(), 536, 538  
 getShortDescription(), 686  
 getSize(), 350, 354  
 getSQLState(), 248  
 GetStaticFieldID(), 910  
 GetStaticMethodID(), 916, 918  
 GetStaticXxxField(), 910  
 getStrength(), 317  
 getString(), 243, 274, 328, 473  
 getStringArray(), 328  
 GetStringChars(), 903

GetStringLength(), 903  
 GetStringRegion(), 903  
 GetStringUTFChars(), 901, 902, 904, 936  
 GetStringUTFLength(), 902  
 GetStringUTFRegion(), 903  
 getSubject(), 767  
 GetSuperClass(), 947  
 getSymbol(), 303  
 getSystem(), 890  
 getSystemClassLoader(), 737  
 getSystemClipboard(), 612, 614  
 getTabCount(), 483  
 getTableCellEditorComponent(), 391, 392, 393  
 getTableCellRendererComponent(), 391, 393  
 getTableName(), 273  
 getTables(), 282  
 getTagName(), 108, 117  
 getTags(), 692, 697  
 getTimeInstance(), 304, 309  
 getTimeZone(), 310, 311  
 getTitleAt(), 482  
 getTransferData(), 615  
 getTransferDataFlavors(), 617  
 getTranslateInstance(), 536, 538  
 getTreeCellRendererComponent(), 413, 414  
 getType(), 268  
 getUpdateCount(), 244  
 getURI(), 156  
 getUrl(), 272, 460, 463  
 getUseCaches(), 214  
 getUsername(), 272  
 getValue(), 156, 473, 606, 696, 934, 935  
 getValueAt(), 363, 364, 367, 389  
 getVendorName(), 564  
 getVersion(), 556, 564  
 getVetoableChangeListeners(), 682  
 getVisibleRowCount(), 348  
 getWidth(), 564, 582, 583, 588  
 getWriteMethod(), 686  
 getWriterFormatNames(), 559, 563  
 getWriterMIMETypes(), 563  
 GetXxxArrayElements(), 922  
 GetXxxArrayRegion(), 921, 922  
 GetXxxField(), 910  
 GIF, 555, 600, 664  
 gniazda, 187
 

- adresy internetowe, 189
- kanały, 198
- limity czasu, 187
- nawiązanie połączenia, 198
- otwieranie, 187
- połączenia częściowo zamknięte, 196
- zamykanie, 193

 gniazdka, 191

Gnu C, 895  
 GradientPaint, 532, 533  
 graficzny interfejs użytkownika, 610  
 grafika, 505
 

- antialiasing, 549
- drukowanie, 580
- filtrowanie obrazów, 571
- Java 2D, 506
- klasy obiektów graficznych, 511
- operacje na obrazach, 565
- potok rysowania, 507
- potokowe tworzenie, 506
- prostokąt ograniczający, 509
- przekształcenia układu współrzędnych, 534
- przezroczystość, 541
- przycinanie, 539
- składanie obrazów, 541
- ślad pędzla, 524
- współrzędne, 509
- wypełnianie obszaru, 532
- zbiór Mandelbrota, 567

 grant, 749  
 Graphics, 505, 506, 508, 541
 

- getClip(), 541
- setClip(), 541

 Graphics2D, 506, 508, 509, 531, 533, 534, 539, 541, 542, 549, 554
 

- clip(), 541
- draw(), 508
- fill(), 508
- getFontRendererContext(), 541
- rotate(), 539
- scale(), 539
- setComposite(), 549
- setPaint(), 533
- setRenderingHint(), 554
- setRenderingHints(), 554
- setStroke(), 531
- setTransform(), 539
- shear(), 539
- transform(), 539
- translate(), 539

 GridBagLayout, 129, 130  
 GridBagPane, 132  
 GSS, 806  
 gzip, 215

## H

handle(), 775  
 handleGetObject(), 328  
 hashCode(), 841, 884  
 HashPrintRequestAttributeSet, 581, 605  
 hasła, 769, 772  
 hasła dostępu, 209

- hasMore(), 869  
 hasMoreElements(), 934, 935, 945  
 hierarchia klas pozwoleń, 745  
 hiperłącza, 458  
 host, 184  
 HTML, 103, 185, 457  
 HTTP, 186, 230
  - nagłówki żądań, 208
  - odpowiedzi, 210
 HttpURLConnection, 222
  - getErrorStream(), 222
 HyperlinkEvent, 459, 460, 463
  - getURL(), 463
 HyperlinkListener, 459, 463
  - hyperlinkUpdate(), 463
 hyperlinkUpdate(), 459, 463
- I**
- ICC, 567  
 Icon, 355  
 iconTextGap, 667  
 ID, 123  
 identyfikator URI, 606  
 IDREF, 123  
 IIIOImage, 565  
 IIOServiceProvider, 564
  - getVendorName(), 564
  - getVersion(), 564
 ikony, 684  
 IllegalArgumentException, 924  
 IllegalStateException, 560  
 Image, 659  
 image/gif, 215  
 ImageInputStream, 560  
 ImageIO, 533, 555, 562
  - createImageInputStream(), 563
  - createImageOutputStream(), 563
  - getImageReadersByFormatName(), 562
  - getImageReadersByMIMEType(), 562
  - getImageReadersBySuffix(), 562
  - getImageWritersByFormatName(), 562
  - getImageWritersByMIMEType(), 562
  - getImageWritersBySuffix(), 562
  - getReaderFormatNames(), 563
  - getReaderMIMETypes(), 563
  - getWriterFormatNames(), 563
  - getWriterMIMETypes(), 563
  - read(), 562
  - write(), 562
 ImageOutputStream, 563  
 ImageReader, 563
  - getHeight(), 564
  - getNumImages(), 563
  - getNumThumbnails(), 563
 getOriginatingProvider(), 564  
 getWidth(), 564  
 read(), 563  
 readThumbnail(), 563  
 ImageReaderWriterSpi, 564
  - getFileSuffixes(), 564
  - getFormatNames(), 564
  - getMIMETypes(), 564
 ImageSelection, 621  
 ImageViewerBean, 660, 662, 663, 664, 665, 673, 674, 683  
 ImageViewerBeanBeanInfo, 683  
 ImageWriteParam, 561  
 ImageWriter, 561, 564
  - canInsertImage(), 564
  - getOriginatingProvider(), 565
  - setOutput(), 564
  - write(), 564
  - writeInsert(), 564
 IMAP, 806  
 IMPLIED, 123  
 implies(), 747, 755, 761  
 import, 732  
 inDaylightTime(), 311  
 IndexedPropertyChangeEvent, 681
  - getIndex(), 681
 IndexedPropertyDescriptor, 686
  - getIndexedReadMethod(), 687
  - getIndexedWriteMethod(), 687
 indexOfTab(), 483  
 IndexOutOfBoundsException, 560  
 indywidualizacja ziarnka, 697
  - Customizer, 699
  - getBeanDescriptor(), 698
  - implementacja klasy, 699
  - nazwa klasy, 698
  - PropertyChangeEvent, 700
 InetAddress, 189, 190, 712
  - getAddress(), 189, 190
  - getAllByName(), 189, 190
  - getByName(), 189, 190
  - getHostAddress(), 190
  - getHostName(), 190
  - getLocalHost(), 189, 190
 InetSocketAddress, 204
  - isUnresolved(), 204
 init(), 746, 800, 801  
 InitialContext, 288, 870  
 initialize(), 712, 725, 771, 775  
 InputSource, 120, 125  
 InputStream, 191, 204, 205  
 InputStreamReader, 470  
 INSERT, 235, 242  
 INSERT INTO, 249

- insertNodeInto(), 404, 409  
insertRow(), 267, 269  
insertTab(), 478, 482  
inspektora właściwości, 666  
instalacja JDBC, 235  
instantiate(), 711, 725  
Instrumentation, 857  
int, 236, 287  
IntegerSyntax, 606, 607  
interfejs  
    ActionListener, 835  
    adnotacje, 831  
    AnnotatedElement, 835  
    Annotation, 838  
    AttributeSet, 604  
    Attribute, 604  
    BeanInfo, 683, 684  
    BufferedImageOp, 565, 571  
    CachedRowSet, 271  
    Callback, 773  
    ClipboardOwner, 612  
    Comparator, 312  
    ContentHandler, 151, 152  
    Customizer, 699  
    DataSource, 288  
    Doc, 601  
    EntityResolver, 107  
    ErrorHandler, 125  
    HyperlinkListener, 459  
    Instrumentation, 857  
    JNDI, 288  
    JNI, 898  
    ListCellRenderer, 356  
    ListModel, 354  
    MutableTreeNode, 396  
    Node, 108  
    Pageable, 589  
    Paint, 532  
    Printable, 580, 581  
    PrintRequestAttributeSet, 581  
    PrivilegedExceptionAction, 764  
    PriviligedAction, 763  
    PropertyChangeListener, 676  
    ResultSet, 270  
    RowSet, 270  
    Shape, 520, 523  
    Source, 176  
    Stroke, 524  
    SupportedValuesAttribute, 604  
    TableCellEditor, 391  
    TableCellRenderer, 383  
    Transferable, 611, 615  
    TreeCellRenderer, 412  
    TreeModel, 116, 396, 422  
    TreeSelectionListener, 415  
    VetoableChangeListener, 495  
interfejs programowy wywołań języka Java, 927, 932  
interfejs użytkownika  
    MDI, 483  
internacjonalizacja  
    czas, 304  
    data, 292, 304  
    ISO, 293  
    języki, 293  
    komplety zasobów, 324  
    komunikaty, 318  
    liczby, 297  
    Locale, 293  
    lokalizatory, 292  
    łańcuchy znaków, 325  
    pliki tekstowe, 322  
    pliki źródłowe programów, 322  
    porządek alfabetyczny, 311  
    waluta, 297, 302  
    zbiory znaków, 322  
internalFrameClosing, 497  
InternalFrameListener, 497  
interpolacja, 572  
interrupt(), 198  
InterruptedIOException, 188  
Introspector, 684, 685  
    getBeanInfo(), 685  
intValue(), 936  
InverseEditor, 688, 694  
inżynieria kodu bajtowego, 851  
    BCEL, 852  
    modyfikacja kodu podczas ładowania, 857  
IPv6, 189  
isAdjusting(), 346  
isAfterLast(), 269  
isAnnotationPresent(), 836  
IsAssignableFrom(), 936, 946  
isBeforeFirst(), 269  
isCanceled(), 474  
isCellEditable(), 367, 384, 385, 391, 394  
isClosable(), 498  
isClosed(), 189, 499  
isConnected(), 189  
isContinuousLayout(), 477  
isDataFlavorAvailable(), 615  
isDataFlavorSupported(), 615  
isEchoOn(), 775  
isExpert(), 686  
isFirst(), 269  
isGroupingUsed(), 302  
isHidden(), 686  
isIcon(), 499  
isIconifiable(), 499

isIgnoringElementContentWhitespace(), 126  
 isIndeterminate(), 473  
 isInputShutdown(), 198  
 isLast(), 269  
 isLeaf(), 401, 402, 426, 428  
 isLenient(), 310  
 isMaximizable(), 498  
 isMaximum(), 499  
 isMIMETypeEqual(), 617  
 isNamespaceAware(), 150, 154  
 ISO, 293  
 isOneTouchExpandable(), 477  
 isOutputShutdown(), 198  
 isPaintable(), 694, 697  
 isParseIntegerOnly(), 302  
 isPropertyName(), 672  
 isResizable(), 498  
 isRunning(), 672  
 isSelected(), 499  
 isStringPainted(), 473  
 isUnresolved(), 204  
 isValidating(), 126, 154  
 isVisible(), 500  
 Item, 858  
 item(), 118  
 Iterator, 243

**J**

JAAS, 762, 767  
 moduł logowania, 768  
 moduły, 767  
 uwierzytelnianie oparte na rolach, 768  
 jaas.config, 766  
 JAR, 205, 663, 729  
 podpisywanie plików, 789  
 jarray, 919, 935  
 jarsigner, 783, 784, 790  
 java, 806  
 Java, 11  
 Java 2, 625, 744  
 Java 2 Enterprise Edition, 230  
 Java 2D, 505, 506  
 figury, 507, 508, 509  
 geometria pól, 523  
 klasy obiektów graficznych, 511  
 krzywe, 509, 512  
 krzywe Bezierra, 513  
 linie, 513  
 łuk, 511  
 operacje na polach, 524  
 pola, 523  
 prostokąt, 511  
 przezroczystość, 541  
 punkty kontrolne, 521

składanie obrazów, 541  
 ślad pędzla, 524  
 wartość alfa, 542  
 wskazówki operacji graficznych, 549  
 współrzędne, 509  
 wypełnianie obszaru, 532  
 Java Authentication and Authorization Service, 762  
 java.awt.datatransfer, 611  
 java.beans.Beans, 670  
 java.net.Socket, 187  
 java.nio, 196, 198  
 java.policy, 748  
 java.rmi, 866  
 java.security, 728, 776  
 java.security.policy, 880  
 java.text, 297  
 JavaBeans, 630  
 arkusz właściwości, 658  
 Bean Builder, 666  
 CalendarBean, 661  
 domyślny konstruktor, 660  
 edytor właściwości, 687  
 indywidualizacja ziarnka, 697  
 klasa informacyjna ziarnka, 683  
 kontrolki, 658  
 NetBeans, 663  
 plik manifestu, 663  
 pliki JAR, 663  
 projektowanie, 669  
 trwałość ziarenek, 705, 715  
 tworzenie aplikacji, 662, 666  
 tworzenie ziarenek, 660  
 typy właściwości, 673  
 ziarenka, 658  
 ziarnka, 657  
 javah, 894, 906  
 JavaMail, 210  
 javap, 912  
 Javascript, 458  
 JavaServer Faces, 216, 658  
 JavaServer Pages, 658  
 javax.imageio, 555  
 javax.mail.internet.MimeUtility, 210  
 javax.security.auth.login.LoginContext, 766  
 javax.security.auth.Subject, 767  
 javax.sql.rowset, 270  
 JAXP, 107  
 JBuilder, 657  
 JCE, 801  
 JCheckBox, 389  
 jclass, 906, 916  
 JColorChooser, 630  
 JComboBox, 389, 521

- JComponent, 403, 500, 630, 675, 682  
addPropertyChangeListener(), 682  
addVetoableChangeListener(), 500, 682  
firePropertyChange(), 682  
fireVetoableChange(), 682  
putClientProperty(), 403  
removePropertyChangeListener(), 682  
removeVetoableChangeListener(), 682  
setTransferHandle(), 630
- JDBC  
adres URL baz danych, 237  
aktualizacja danych, 227  
aktualizacja wsadowa, 284  
aktualizowalne zbiory wyników zapytań, 263, 265  
architektura, 228  
instalacja, 235  
JNDI, 288  
klient, 230  
klient-serwer, 230  
menedżer sterowników, 239  
metadane, 274  
nawiązywanie połączenia, 237  
polecenia, 245  
polecenia przygotowane, 252  
połączenia krótkotrwałe, 246  
przewijalne zbiory wyników zapytań, 263  
przewijanie zbioru rekordów, 264  
pula połączeń, 289  
rekordy wstawiania, 266  
serwer, 230  
SQL, 227, 231  
sterowniki, 228  
transakcje, 283  
warstwa pośrednia, 230  
wersje, 227  
wykonywanie zapytań, 252  
wypełnianie bazy danych, 248  
zamykanie zbioru wyników, 245  
zapytania, 227  
zarządzanie połączeniami, 245  
zastosowania, 229  
zbiory rekordów, 263, 270  
zbiory wyników, 245  
źródło danych, 237
- JDBC 3, 288  
JDBC API, 228  
jdbc.password, 249  
jdbc.property, 239  
jdbc.url, 249  
jdbc.username, 249  
JdbcRowSet, 270
- JDesktopPane, 485, 486, 487, 492, 498  
    getAllFrames(), 498  
    setDragMode(), 498
- JDialog, 496  
JEditorPane, 457, 458, 459, 460  
    addHyperlinkListener(), 462  
     setPage(), 462
- język, 293  
C, 892  
C++, 895  
DDL, 244  
HTML, 103, 185  
IDL, 863  
Java, 11  
Javascript, 458  
SGML, 103  
SQL, 227, 231  
Visual Basic, 658  
WSDL, 863  
XML, 101, 103, 864  
XML Schema, 119, 126  
XPath, 142
- jfieldID, 906  
JFileChooser, 630  
JFrame, 485, 706  
JInternalFrame, 485, 487, 495, 498, 676  
    getContentPane(), 499  
    getDesktopPane(), 499  
    getFrameIcon(), 499  
    isClosable(), 498  
    isClosed(), 499  
    isIcon(), 499  
    isIconifiable(), 499  
    isMaximizable(), 498  
    isMaximum(), 499  
    isResizable(), 498  
    isSelected(), 499  
    isVisible(), 500  
    moveToFront(), 499  
    moveToBack(), 499  
    reshape(), 499  
    setClosable(), 498  
    setClosed(), 499  
    setContentPane(), 499  
    setFrameIcon(), 499  
    setIcon(), 499  
    setIconifiable(), 499  
    setMaximizable(), 499  
    setMaximum(), 499  
    setResizable(), 498  
     setSelected(), 499  
    setVisible(), 500  
    show(), 500
- JLabel, 358, 412  
JList, 344, 348, 353, 355, 358, 630  
    addListSelectionListener(), 349  
    getBackground(), 358  
    getForeground(), 358

getLayoutOrientation(), 348  
 getModel(), 355  
 getPrototypeCell(), 353  
 getSelectedValue(), 349  
 getSelectedValues(), 346, 349  
 getSelectionBackground(), 356, 358  
 getSelectionForeground(), 356, 358  
 getSelectionMode(), 349  
 getVisibleRowCount(), 348  
 HORIZONTAL\_WRAP, 345  
 isAdjusting(), 346  
 konstruktorzy, 355  
 setCellRenderer(), 357, 359  
 setFixedCellHeight(), 353  
 setFixedCellWidth(), 353  
 setLayoutOrientation(), 348  
 setPrototypeCell(), 353  
 setSelectionMode(), 345  
 setVisibleRowCount(), 345, 348  
 valueChanged(), 346  
 VERTICAL, 345  
 VERTICAL\_WRAP, 345  
 wektor obiektów, 354  
**JNDI**, 288  
 nawiązywanie połączenia, 289  
 pula połączeń, 289  
 zarządzanie nazwami użytkowników, 289  
 źródła danych, 288  
**JndiLoginModule**, 763  
**JNI**, 898, 900  
 konwencja wywołań, 901  
 wywołania funkcji, 901  
**JNI\_CreateJavaVM()**, 928, 932  
**JNI\_TRUE**, 936  
**JNICALL**, 904  
**JNIEXPORT**, 904  
**JobAttributes**, 609  
**JobHoldUntil**, 607  
**JobImpressions**, 607  
**JobImpressionsCompleted**, 607  
**jobect**, 906, 935  
**JobKOctets**, 607  
**JobKOctetsProcessed**, 607  
**JobMediaSheets**, 607  
**JobMediaSheetsCompleted**, 607  
**JobMessageFromOperator**, 607  
**JobName**, 607  
**JobOriginatingUserName**, 608  
**JobPriority**, 608  
**JobSheets**, 608  
**JobState**, 608  
**JobStateReason**, 608  
**JobStateReasons**, 608  
**JoinRowSet**, 270  
**JOptionPane**, 496  
**JPanel**, 584  
**JPEG**, 555, 556  
**JProgressBar**, 463, 472  
 getMaximum(), 473  
 getMinimum(), 473  
 getString(), 473  
 getValue(), 473  
 isIndeterminate(), 473  
 isStringPainted(), 473  
 setIndeterminate(), 473  
 setMaximum(), 473  
 setMinimum(), 473  
 setString(), 473  
 setStringPainted(), 473  
 setValue(), 473  
**JScrollPane**, 344, 362  
**JSF**, 658  
**JSP**, 658  
**JSplitPane**, 475, 477, 497  
 HORIZONTAL\_SPLIT, 475  
 isContinuousLayout(), 477  
 isOneTouchExpandable(), 477  
 setBottomComponent(), 477  
 setContinuousLayout(), 477  
 setLeftComponent(), 477  
 setOneTouchExpandable(), 477  
 setRightComponent(), 477  
 setTopComponent(), 477  
 VERTICAL\_SPLIT, 475  
**jstring**, 900, 916, 935  
**JTabbedPane**, 478, 482  
 addChangeListener(), 483  
 addTab(), 482  
 getComponentAt(), 483  
 getIconAt(), 483  
 getLayoutPolicy(), 483  
 getSelectedComponent(), 482  
 getSelectedIndex(), 482  
 getTabCount(), 483  
 getTitleAt(), 482  
 indexOfTab(), 483  
 insertTab(), 482  
 removeTabAt(), 482  
 setComponentAt(), 483  
 setIconAt(), 483  
 setSelectedIndex(), 482  
 setTabLayoutPolicy(), 483  
 setTitleAt(), 482  
**JTable**, 359, 363, 368, 391, 393, 630  
 addColumn(), 380  
 getCellSelectionEnabled(), 380  
 getColumnSelectionAllowed(), 379  
 getDefaultEditor(), 393

JTable  
 getDefaultRenderer(), 393  
 getRowHeight(), 379  
 getRowMargin(), 379  
 getRowSelectionAllowed(), 379  
 getSelectionModel(), 379  
 moveColumn(), 380  
 removeColumn(), 380  
 setAutoResizeMode(), 379  
 setColumnSelectionAllowed(), 380  
 setRowHeight(), 379  
 setRowMargin(), 379  
 setRowSelectionAllowed(), 379  
 JTextArea, 457, 756  
 JTextComponent, 459, 630  
 JTextField, 389, 457  
 JTree, 111, 394, 395, 397, 403, 409, 421, 630  
 getLastSelectedPathComponent(), 409  
 getSelectionPath(), 409, 421  
 getSelectionPaths(), 421  
 makeVisible(), 409  
 scrollPathToVisible(), 409  
 setRootVisible(), 402  
 setShowsRootHandles(), 402  
 jvalue, 919  
 jvm, 928

**K**

kalendärz, 662  
 kalkulator emerytalny, 328  
 kanaly, 198  
 SocketChannel, 198  
 Kerberos, 806  
 Kernel, 578, 580  
 KEY\_ALPHA\_INTERPOLATION, 550  
 KEY\_ANTIALIASING, 550  
 KEY\_COLOR\_RENDERING, 550  
 KEY\_DITHERING, 550  
 KEY\_FRACTIONAL\_METRICS, 550  
 KEY\_INTERPOLATION, 550  
 KEY\_RENDERING, 550  
 KEY\_STROKE\_CONTROL, 550  
 KEY\_TEXT\_ANTIALIASING, 550  
 KeyGenerator, 797, 801  
 generateKey(), 801  
 getInstance(), 801  
 init(), 801  
 KeyPairGenerator, 803  
 KeyStoreLoginModule, 763  
 keytool, 781, 782, 787, 790  
 klasy, 732  
 AbstractCellEditor, 391  
 AbstractTableModel, 363  
 abstrakcyjne, 412

ActionListenerInstaller, 835  
 Activatable, 885, 886  
 AffineTransform, 536  
 AffineTransformOp, 572  
 AllPermissions, 746, 755  
 AlphaComposite, 544  
 Arc2D, 511  
 ArcMaker, 521  
 Area, 523  
 Banner, 590  
 Base64, 210  
 Base64Encoder, 214  
 BasicPermission, 752  
 BasicStroke, 524, 525  
 Book, 598  
 BufferedImage, 533, 565  
 ButtonFrame, 835  
 ChartBeanBeanInfo, 688, 845  
 ChoiceFormat, 321  
 Cipher, 795  
 CipherInputStream, 802  
 ClassLoader, 729, 733  
 ClassNameTreeCellRenderer, 413  
 ClassTreeFrame, 414  
 Clipboard, 611  
 Collator, 312  
 Color, 532  
 ColorConvertOp, 577  
 ColorModel, 568  
 ConvolveOp, 578  
 Copies, 604, 606  
 CopiesSupported, 604  
 CubicCurve2D, 513  
 Currency, 302  
 DatabaseMetaData, 265, 275  
 DataFlavor, 611, 615  
 DateFormat, 304, 305  
 DefaultCellEditor, 406  
 DefaultHandler, 152  
 DefaultListModel, 354  
 DefaultModelList, 354  
 DefaultMutableTreeNode, 396  
 DefaultPersistenceDelegate, 713  
 DefaultTreeModel, 396, 405, 422  
 DialogCallbackHandler, 772  
 DocFlavor, 599  
 DocPrintJob, 601  
 Document, 108  
 DocumentBuilder, 107, 160  
 domena ochronna, 746  
 DOMResult, 177  
 DOMSource, 176  
 DOMTreeModel, 116  
 DriverManager, 288

Ellipse2D, 511  
EntryLogger, 858  
EnumSyntax, 607  
File, 397  
FilePermission, 745  
GeneralPath, 513  
GradientPaint, 532  
Graphics, 505, 508  
Graphics2D, 506  
GridBagLayout, 129  
HashPrintRequestAttributeSet, 581  
HyperlinkEvent, 460  
Icon, 355  
ImageInputStream, 560  
ImageIO, 555  
ImageViewerBean, 660  
ImageWriter, 561  
InetAddress, 189  
informacyjne ziarka, 683, 830  
InputStream, 205  
IntegerSyntax, 606  
InverseEditor, 694  
JComboBox, 521  
JDesktopPane, 485, 487  
JEditorPane, 457  
JFrame, 485  
JInternalFrame, 485, 487  
JLabel, 358, 412  
JList, 349  
 JPanel, 584  
JProgressBar, 463  
JSplitPane, 497  
JTabbedPane, 478  
JTable, 368  
JTextArea, 756  
JTree, 394  
Kernel, 578  
KeyGenerator, 797  
KeyPairGenerator, 803  
komplety zasobów, 326  
Line2D, 511  
ListResourceBundle, 326  
Locale, 293, 295, 297  
LoginContext, 762  
LookupOp, 576, 577  
ładowanie, 728  
MessageDigest, 778  
MessageFormat, 318, 321  
MimeUtility, 210  
Number, 297  
NumberFormat, 297, 298, 304  
PageFormat, 582  
Permission, 755  
PersistenceDelegate, 712  
Point2D, 509  
Policy, 745  
pozwoleń, 755  
PreparedStatement, 254  
PrinterJob, 581  
PrintPreviewDialog, 591, 598  
PrintService, 601, 603  
PrintServiceLookup, 600  
PrintWriter, 218  
PrivilegedAction, 763  
ProgressMonitor, 463, 466  
ProgressMonitorInputStream, 463, 469  
Properties, 102  
PropertyChangeSupport, 675  
PropertyDescriptor, 687  
PropertyEditorSupport, 691  
Random, 797  
Raster, 567  
Reader, 176  
Rectangle2D, 511  
RenderingHints, 551  
RescaleOp, 576  
ResourceBundles, 326  
ResultSetMetaData, 275  
RetinaScanCallback, 773  
RoundRectangle2D, 511  
rozwiązywanie, 728  
Runtime, 743  
SAXSource, 177  
Scanner, 198  
SecureRandom, 797  
SecurityManager, 746  
ServerSocket, 191, 193  
ShapeMaker, 521  
ShapePanel, 521  
SimpleBeanInfo, 684  
SimpleDateFormat, 319  
SimpleLoginModule, 771  
SimplePrincipal, 769  
SimulatedActivity, 466  
Socket, 187  
SocketChannel, 198  
StreamPrintService, 603  
StreamPrintServiceFactory, 603  
StreamSource, 176  
String, 312  
StringBuffer, 472  
StringSelection, 612, 617  
sztyfrowanie plików, 737  
 TableColumn, 368  
 TableColumnModel, 368  
TextLayout, 540  
TexturePaint, 532, 533  
ThreadedEchoHandler, 194

- klasy
- TreeNode, 403
  - TreePath, 404
  - TreeSelectionModel, 415
  - UnicastRemoteObject, 867
  - UnixNumericGroupPrincipal, 763
  - UnixPrincipal, 762
  - URI, 205
  - URL, 205
  - URLClassLoader, 729
  - URLConnection, 205
  - VetoableChangeSupport, 677
  - WordCheckPermission, 756
  - WritableRaster, 567
  - XPath, 143
- klient, 185, 230, 862
- klucz prywatny, 780
- klucz publiczny, 780
- klucze, 796
- kod ASCII, 291
- kod bajtowy, 851
- kod języków, 293
- kod macierzysty, 891
- kod Unicode, 291
- kodowanie znaków, 322
- kolory, 542
- RGB, 541, 566
- kolumny, 231
- komplikator, 728
- Gnu C, 895
- komplety zasobów, 324
- implementacja klas, 326
- klasy, 326
- ładowanie, 324
- pliki właściwości, 325
- komponenty
- formatowanie tekstu, 429
  - JavaBeans, 658
  - JEditorPane, 457
  - JList, 344
  - JProgressBar, 463
  - JTextArea, 457
  - JTextField, 457
  - JTree, 394
  - Metal, 484
  - organizatory, 474
  - panele dzielone, 475
  - rozmieszczenie, 487
- komunikacja, 862
- komunikaty, 318
- formatowanie z wariantami, 320
  - indeks znacznika, 319
- konfiguracja wywołania zdalnych metod, 866
- konstruktory, 917
- kontekst graficzny, 535, 583, 584
- kontekst tworzenia czcionki, 540
- kontrola dostępu, 727
- kontrola poprawności dokumentów XML, 118
- atrybuty, 122
  - ATTLIST, 122
  - byty, 123
  - CDATA, 123
  - definicja typu dokumentu, 119
  - DOCTYPE, 120
  - DTD, 119
  - ELEMENT, 119, 121
  - parser XML, 122
  - reguły zawartości elementów, 121
  - skróty, 123
  - typy atrybutów, 122
  - wartości domyślne atrybutów, 123
  - warunki kontroli, 119
  - XML Schema, 119, 126
- kontrola pozwoleń, 747
- kontrolki, 658
- CommonDialog, 658
  - Image, 658
- kończenie pracy maszyny wirtualnej, 744
- korzeń, 395
- Krb5LoginModule, 763
- kryptografia klucza publicznego, 780, 803
- krzywe, 509, 512, 521
- Beziera, 513
  - drugiego stopnia, 513
  - punkty kontrolne, 512
  - trzeciego stopnia, 513
- kursory, 264
- L**
- las, 395, 400
- last(), 268
- layoutPages(), 590
- LD\_LIBRARY\_PATH, 932
- LDAP, 806
- liczby, 297
- formatowanie, 297, 899
  - formaty, 298
  - lokalizatory, 297
  - losowe, 797
- LIKE, 234
- Line2D, 508, 509, 511
- lineTo(), 513, 514, 522
- linie, 513
- Linux, 234, 931
- list(), 871
- ListCellRenderer, 356, 359
- getListCellRendererComponent(), 359

- ListModel, 354  
     getElementAt(), 354  
     getSize(), 354  
 ListResourceBundle, 326  
 ListSelectionListener, 349  
 ListSelectionModel, 381  
     setSelectionMode(), 381  
 listy, 343  
     JList, 344  
     kolor tła komórki, 358  
     modele, 349  
 obiekt odrysowyujący zawartość komórek, 355  
 odrysowywanie zawartości, 355  
 powiadomienia, 346  
 prezentacja elementów, 345  
 przewijanie zawartości, 344  
 rozwijsalne, 344  
 tworzenie, 344  
 usuwanie elementów, 354  
 wizualizacja danych, 350  
 wstawianie elementów, 354  
 zaznaczanie elementów, 345  
 liście, 395  
 loadClass(), 733  
 loadImage(), 668, 684, 685  
 loadLibrary(), 896, 897  
 Locale, 293, 294, 295, 296, 297  
     getCountry(), 296  
     getDefaut(), 296  
     getDisplayCountry(), 296  
     getDisplayLanguage(), 296  
     getDisplayName(), 296  
     getLanguage(), 296  
     setDefaut(), 296  
     toString(), 296  
 localFile, 750  
 localhost, 868  
 LoggingPermission, 752  
 logika biznesowa, 764  
 login(), 766, 776  
 LoginContext, 762, 766, 771  
     getSubject(), 767  
     login(), 766  
     logout(), 766  
 LoginModule, 775  
     abort(), 776  
     commit(), 776  
     initialize(), 775  
     login(), 776  
     logout(), 776  
 logout(), 766, 776  
 logowanie, 763, 774  
 lokalizacja, 293, 328  
     kod, 744  
     zasoby, 324  
 lokalizatory, 292  
     czas, 304  
     data, 304  
     domyślny, 295  
     języki, 294  
     liczby, 297  
 lookup(), 288, 870, 871  
 LookupOp, 576, 577, 579  
 lookupPrintServices(), 600, 601  
 LookupTable, 576  
 lostOwnership(), 615
- Ł**
- ładowanie klas, 728  
     ClassLoader, 733  
     implementacja procedury ładowającej, 733  
     klasy systemowe, 729  
     maszyna wirtualna, 728  
     procedura rozszerzona, 729  
     procedura systemowa, 729  
     procedury, 730  
     przestrzeń nazw, 732  
     rozszerzenia maszyny wirtualnej, 729  
     URLClassLoader, 729  
 łańcuch zaufania, 784  
 łańcuchy, 321  
 łączenie tabel, 232  
 łuk, 511, 521
- M**
- macierz przekształceń, 536  
 mailto, 205  
 main(), 728  
 makeShape(), 521  
 makeVisible(), 405, 409  
 mapy bitowe, 684  
 MarshalledObject, 885, 889  
     get(), 890  
 MD5, 777  
 MDI, 483, 484  
 mechanizm przeciągnij i upuść, 625  
 MediaName, 608  
 MediaSize, 608  
 MediaSizeName, 608  
 MediaTray, 608  
 menedżer bezpieczeństwa, 728, 742, 749, 879  
     checkExit(), 743  
     domyślny, 744  
     pliki polityki, 749  
     pozwolenia, 742  
     SecurityManager, 746  
 MessageDigest, 778, 779  
     digest(), 779  
     reset(), 779  
     update(), 779

MessageFormat, 318, 319, 321  
 applyPattern(), 319  
 format(), 318, 320  
 getLocale(), 319  
 setLocale(), 319  
 Messager, 850  
 metaadnotacje, 843  
 metadane, 274  
 DatabaseMetaData, 275  
 pozyskiwanie, 274  
 ResultSetMetaData, 275  
 Metal, 398  
 Method, 835, 915  
 MethodDescriptor, 684  
 metody  
 main(), 728  
 sygnatury, 911  
 metody macierzyste  
 alternatywne sposoby wywoływania metod, 917  
 dostęp do pól instancji, 906, 910  
 dostęp do pól statycznych, 910  
 funkcje języka C, 892  
 implementacja, 895  
 interfejs programowy wywołań języka Java, 927  
 jarray, 919  
 język C++, 895  
 JNI, 898  
 konstruktory, 917  
 łańcuchy znaków, 900  
 obsługa błędów, 923, 927  
 parametry numeryczne, 898  
 przeciążanie identyfikatorów, 893  
 rejestr systemu Windows, 932  
 składowe obiektu, 906  
 sygnatury, 911  
 tablice, 919, 922  
 wartości zwracane, 898  
 wyrzucanie wyjątków, 923  
 wywoływanie metod języka Java, 912, 917  
 wywoływanie metod obiektów, 912  
 wywoływanie metod statycznych, 916

Microsoft ASP, 216  
 MIME, 555, 556, 616  
 MimeUtility, 210  
 MissingResourceException, 325  
 model drzewa, 396  
 model kolorów, 566  
 model tabeli, 359, 363  
 model-widok-nadzorca, 349  
 moduł JAAS, 767  
 moduł logowania, 763, 768  
 hasła, 769, 772  
 logowanie, 774  
 nazwa użytkownika, 772

options, 771  
 SimpleLoginModule, 769  
 moduł uwierzytelniania, 763  
 modyfikacja kodu bajtowego podczas lądowania, 857  
 modyfikatory, 831  
 monitorowanie postępu strumieni wejścia, 469  
 monitory postępu, 466  
 mostek JDBC/ODBC, 228  
 moveColumn(), 380  
 moveTo(), 514, 522  
 moveToBack(), 499  
 moveToCurrentRow(), 267, 269  
 moveToFront(), 499  
 moveToInsertRow(), 267, 269  
 MultipleDocumentHandling, 608  
 MutableTreeNode, 396, 402  
 setUserObject(), 402

**N**

nagłówki żądań HTTP, 208  
 NameCallback, 773, 775  
 getDefaultName(), 775  
 getName(), 775  
 getPrompt(), 775  
 setName(), 775  
 NameClassPair, 869, 871  
 getClassName(), 871  
 getName(), 871  
 NamedNodeMap, 110, 118  
 getLength(), 118  
 item(), 118  
 namiastka, 863, 864  
 klienta, 862  
 serwera, 862  
 szeregowanie parametrów, 865  
 Naming, 871  
 bind(), 871  
 list(), 871  
 lookup(), 871  
 rebind(), 871  
 unbind(), 871  
 native2ascii, 323  
 negatyw obrazu, 576  
 NetBeans, 657, 663, 665  
 edytor właściwości, 667, 687  
 implementacja klasy indywidualizacji, 699  
 inspektor właściwości, 669  
 tworzenie formatki, 666  
 tworzenie projektu, 665  
 właściwości, 667  
 zdarzenia, 668  
 NetPermission, 751  
 NewByteArray(), 935  
 newDocument(), 160

- newDocumentBuilder(), 116  
 newInputStream(), 204  
 newInstance(), 116, 147, 154, 164  
 NewObject(), 919, 923  
 NewObjectA(), 919  
 NewObjectV(), 919  
 newOutputStream(), 198, 204  
 newSAXParser(), 154  
 NewString(), 903  
 NewStringUTF(), 902, 935  
 newXPath(), 147  
 NewXxxArray(), 921  
 next(), 244  
 nextElement(), 934, 935, 945  
 NMOKEN, 123  
 NMOKENS, 123  
 Node, 108, 117, 150
  - appendChild(), 164
  - getAttributes(), 117
  - getChildNodes(), 117
  - getFirstChild(), 117
  - getLastChild(), 117
  - getLocalName(), 150
  - getNameSpaceURI(), 150
  - getNextSibling(), 117
  - getNodeName(), 117
  - getNodeValue(), 117
  - getParentNode(), 117
  - getPreviousSibling(), 117
- nodeChanged(), 405, 410  
 NodeList, 108, 118
  - getLength(), 118
  - item(), 118
- NOT LIKE, 234  
 NTLoginModule, 763  
 NullPointerException, 924  
 Number, 297  
 NumberFormat, 297, 298, 301, 304
  - format(), 301
  - getAvailableLocales(), 298
  - getCurrencyInstance(), 302
  - getMaximumFractionDigits(), 302
  - getMaximumIntegerDigits(), 302
  - getMinimumFractionDigits(), 302
  - getMinimumIntegerDigits(), 302
  - isGroupingUsed(), 302
  - isParseIntegerOnly(), 302
  - parse(), 301
  - setGroupingUsed(), 302
  - setMaximumFractionDigits(), 302
  - setMaximumIntegerDigits(), 302
  - setMinimumFractionDigits(), 302
  - setMinimumIntegerDigits(), 302
  - setParseIntegerOnly(), 302
- NumberOfDocuments, 608  
 NumberOfInterveningJobs, 608  
 NumberUp, 608  
 NUMERIC, 236, 287  
 numeryczne parametry metod, 898
- ## 0
- obiekty
  - dostęp do składowych, 906
  - nasłuchujące, 674
  - odrysowujące zawartość komórek listy, 355
  - serializacja, 706
  - użytkownika, 396
- obrazy
  - dostęp do danych, 565
  - filtrowanie, 571
  - model kolorów, 566
  - modyfikacja pikseli, 565
  - negatyw, 576
  - obrót, 572
  - operacje, 565
  - próbki piksela, 566
  - przejrzystość, 541
  - przyrostowe tworzenie, 565
  - rozmycie, 577
  - sekwencje, 560
  - wczytywanie, 555
  - wykrywanie krawędzi, 578
  - zapisywanie, 555
  - zbiór Mandelbrota, 567
- obrót, 534, 572  
 obrys czcionek, 540  
 obrys figury, 506  
 obrys tekstu, 540  
 obsługa błędów, 923  
 obsługa zdarzeń
  - adnotacje, 832
- ODBC, 228  
 odcinki, 513  
 odcisk palca, 777  
 odszyfrowywanie danych, 801  
 odwołanie transakcji, 283  
 okna dialogowe, 478
  - drukowanie, 587
  - ramki wewnętrzne, 496
  - wybór plików, 659
- openConnection(), 207, 214  
 openInputStream(), 216  
 openOutputStream(), 216  
 openStream(), 205, 208, 214  
 operacje na obrazach, 565  
 ORDER BY, 243  
 organizatory komponentów, 474  
 OrientationRequested, 608

OutOfMemoryError, 924  
OutputDeviceAssigned, 608  
OutputStream, 191, 204

**P**

Package, 835  
Pageable, 589  
PageAttributes, 609  
pageDialog(), 583, 588  
PageFormat, 582, 587, 588  
  getHeight(), 588  
  getImageableHeight(), 588  
  getImageableWidth(), 588  
  getImageableX(), 589  
  getImageableY(), 589  
  getOrientation(), 589  
  getWidth(), 588  
PageRanges, 608  
PagesPerMinute, 608  
PagesPerMinuteColor, 608  
Paint, 532  
paint(), 506  
paintComponent(), 358, 506, 530, 584, 598  
paintValue(), 694, 697  
pakiety, 732  
panele dzielone, 475  
  JSplitPane, 475  
panele pulpitu, 483  
panele z zakładkami, 478  
  JTabbedPane, 478  
parse(), 116, 155, 181, 297, 301, 310  
ParseException, 298  
parser  
  SAX, 150  
parser XML, 107  
  DOM, 107  
  implementacja, 122  
  SAX, 107  
parsowanie dokumentów XML, 107, 112  
  DOM, 107, 109  
  DOMTreeModel, 116  
  SAX, 107  
  TreeModel, 116  
  XML Schema, 129  
pasek postępu, 463  
  nieokreślony, 466  
  usawianie wartości, 463  
pasek przewijania, 362  
PasswordCallback, 773, 775  
  getPassword(), 775  
  getPrompt(), 775  
  isEchoOn(), 775  
  setPassword(), 775  
pathFromAncestorEnumeration(), 411

PCDATA, 121, 124, 131  
PDLOverrideSupported, 608  
Permission, 755, 761  
  getName(), 761  
  implies(), 761  
PersistenceDelegate, 712, 725  
  initialize(), 725  
piaskownica, 744  
pixsel, 541  
  docelowy, 542  
PJA, 607  
PKCS#5, 796  
pliki, 753  
  class, 728  
  JAR, 205, 663, 664, 729  
  java.policy, 748  
  pomocnicze, 830  
  pozwoleń, 745  
  rt.jar, 730  
  tekstowe, 322  
  właściwości, 102, 325  
  XML, 102, 707  
  XML Schema, 119  
  zasoby, 324  
  ZIP, 729  
  źródłowe, 322  
pliki graficzne, 555  
  animacje GIF, 562  
  formaty, 555  
  GIF, 555  
  ImageIO, 555  
  interfejs dostawcy, 556  
  JPEG, 555, 556  
  MIME, 555, 556  
  obiekt odczytu, 556  
  obiekty, 555  
  sekwencje obrazów, 560  
  wczytywanie, 555  
  zapisywanie, 555, 559  
pliki polityki, 744, 747, 880  
  baza kodu, 750  
  edycja, 754  
  grant, 749  
  implementacja klasy pozwoleń, 756  
  java.policy, 748  
  klasy pozwoleń, 755  
  menedżer bezpieczeństwa, 749  
  niezależne od platformy systemowej, 754  
  odczyt, 754  
  operacje sieciowe, 753  
  położenie, 748  
  pozwolenia, 750, 758  
  składnica kluczy, 790  
  system plików, 752

- testowanie aplikacji, 749  
 właściwości systemowe, 753  
 zapis, 754  
 PNG, 555  
 pochylanie, 535  
 poczta elektroniczna, 183, 222  
 SMTP, 222  
 wysyłanie, 222  
 podgląd wydruku, 591  
 podpis cyfrowy, 776, 785  
 aplety, 776  
 generator liczb losowych, 797  
 MD5, 777  
 MessageDigest, 778  
 podpisywanie wiadomości, 779  
 SHA1, 777  
 skrót wiadomości, 777  
 weryfikacja, 781  
 podpisywanie certyfikatów, 786  
 podpisywanie kodu, 728, 788  
 aplety dostępne przez Internet, 789  
 aplety intranetowe, 789  
 certyfikaty twórców oprogramowania, 793  
 jarsigner, 790  
 pliki JAR, 789  
 podpisywanie wiadomości, 779  
 Point2D, 509, 521, 540  
 pola, 523  
 add, 523  
 exclusiveOr, 523  
 intersect, 523  
 operacje, 524  
 subtract, 523  
 ślad pędzla, 524  
 tworzenie, 523  
 Policy, 745  
 policytool, 754  
 polityka bezpieczeństwa, 744, 745  
 pliki, 747  
 połączenia bazodanowe, 227  
 JNDI, 288  
 połączenia sieciowe  
 częściowo zamknięte, 196  
 gniazda, 187  
 HTTP, 186  
 klient, 185  
 porty, 184  
 serwer, 183  
 strumienie, 187  
 TCP, 187  
 telnet, 183  
 UDP, 187  
 URL, 204  
 wysyłanie danych do formularzy, 216
- populate(), 273  
 porównywanie łańcuchów, 311  
 porty, 184  
 porządek alfabetyczny, 311  
 POST, 218  
 postOrderEnumeration(), 414  
 postOrderTraversal(), 411  
 PostScript, 603  
 potok rysowania, 507  
 potokowe tworzenie grafiki, 506  
 pozwolenia, 742, 745, 758  
 implementacja klasy, 756  
 klasy, 755  
 operacje sieciowe, 753  
 parametry, 750, 751, 752  
 pliki polityki, 750  
 stos wywołań metod, 746  
 system plików, 752  
 PRA, 607  
 premain(), 857  
 preOrderEnumeration(), 414  
 preOrderTraversal(), 411  
 PreparedStatement, 253, 254, 258  
 clearParameters(), 258  
 executeQuery(), 258  
 executeUpdate(), 258  
 setXxx(), 258  
 prepareStatement(), 258, 263, 268  
 PresentationDirection, 608  
 Previous(), 268  
 Principal, 767  
 getName(), 767  
 Print, 584  
 print(), 580, 583, 587, 588, 602, 913  
 Printable, 580, 581, 582, 587  
 printDialog(), 581, 588  
 PrinterException, 581  
 PrinterInfo, 608  
 PrinterIsAcceptingJobs, 608  
 PrinterJob, 581, 582, 583, 588, 589, 598  
 defaultPage(), 588  
 getPrinterJob(), 588  
 pageDialog(), 588  
 print(), 588  
 printDialog(), 588  
 setPageable(), 598  
 setPrintable(), 588  
 PrinterLocation, 609  
 PrinterMakeAndModel, 609  
 PrinterMessageFromOperator, 609  
 PrinterMoreInfo, 609  
 PrinterMoreInfoManufacturer, 609  
 PrinterName, 609  
 PrinterResolution, 609

PrinterState, 609  
PrinterStateReason, 609  
PrinterStateReasons, 609  
PrinterURI, 609  
printf(), 892, 911  
    formatowanie liczb, 899  
PrintJob, 581  
PrintJobAttribute, 604, 607  
PrintPreviewCanvas, 598  
PrintPreviewDialog, 591, 598  
PrintQuality, 606, 609  
PrintRequestAttribute, 604, 607  
PrintRequestAttributeSet, 581  
PrintService, 601, 602, 603, 610  
    createPrintJob(), 602  
    getAttributes(), 610  
PrintServiceAttribute, 604, 607  
PrintServiceLookup, 600, 602  
PrintWriter, 191, 218, 912, 913  
PrivilegedAction, 763, 767  
PrivilegedExceptionAction, 764, 767  
PrivilegedAction, 763, 764, 773  
problem uwierzytelniania, 784  
procedury ładowania klas  
    implementacja, 733  
procesor XSLT, 174  
processAnnotations(), 835  
profil ICC, 567  
ProgressMonitor, 463, 466, 474  
    close(), 474  
    isCanceled(), 474  
    setNote(), 474  
    setProgress(), 474  
ProgressMonitorInputStream, 463, 469, 474  
Properties, 102  
Property, 671  
PropertyChange, 674  
propertyChange(), 676, 680  
PropertyChangeEvent, 496, 500, 676, 681, 700  
    getNewValue(), 500, 681  
    getOldValue(), 681  
    getPropertyName(), 500, 681  
PropertyChangeListener, 676, 680  
    propertyChange(), 680  
PropertyChangeSupport, 675, 680  
    addPropertyChangeListener(), 680  
    fireIndexedPropertyChange(), 680  
    firePropertyChange(), 680  
    getPropertyChangeListeners(), 680  
    removePropertyChangeListener(), 680  
PropertyDescriptor, 683, 686, 687, 690  
    getPropertyType(), 686  
    getReadMethod(), 686  
    getWriteMethod(), 686  
    setPropertyEditorClass(), 690  
    PropertyEditor, 694  
    PropertyEditorSupport, 691, 696  
        getAsText(), 696  
        getCustomEditor(), 697  
        getJavaInitializationString(), 697  
        getTags(), 697  
        getValue(), 696  
        isPaintable(), 697  
        paintValue(), 697  
        setAsText(), 696  
        setValue(), 696  
        supportsCustomEditor(), 697  
    PropertyPermission, 750  
    PropertyVetoException, 493, 494, 495, 496, 500,  
        676, 677, 682  
        getPropertyChangeEvent(), 683  
    prostokąt, 511, 521  
        ograniczający, 509  
    ProtectionDomain, 747  
        getCodeSource(), 747  
        implies(), 747  
protokoły  
    HTTP, 208  
    Kerberos, 806  
    LDAP, 806  
    SMTP, 222  
    TCP, 187  
    UDP, 187  
prywatne metody macierzyste, 744  
przeciąganie zarysu ramki, 497  
przeciągnij i upuść, 625  
    gesty, 626  
    inicjacja operacji przeciągnięcia, 625  
    kopiowanie, 626  
    przesunięcie, 626  
    tworzenie łącza, 626  
    upuszczenie obiektu, 626  
przeglądarka klas, 415  
przekazywanie parametrów zdalnym metodom, 876  
    equals(), 884  
    hashCode(), 884  
    zdalne obiekty, 884  
przekształcenia afiniczne, 536, 572  
przekształcenia figur, 506  
przekształcenia układu współrzędnych, 507, 534  
    macierz przekształceń, 536  
    obrót, 534  
    pochylanie, 535  
    przesunięcie, 535  
    skalowanie, 534  
    składanie przekształceń, 535  
przekształcenia XSL  
    Java, 176

style, 173  
 szablon przekształcenia, 173  
 XSLT, 174  
 przenośne pliki źródłowe, 323  
 przestrzeń nazw, 148, 732  
     alias, 149  
     definiowanie, 149  
     identyfikatory, 148  
     URI, 148  
     XML, 148  
 przesunięcie, 535  
 przetwarzanie adnotacji, 845  
 przetwarzanie dokumentów XML, 107  
 przewijalne zbiorzy wyników zapytań, 263  
 przezroczystość, 541  
 przycinanie, 506, 539  
     określanie obszaru, 540  
 przyrostowe tworzenie obrazów, 565  
 PSA, 607  
 public, 831  
 PUBLIC, 120  
 pula połączeń, 289  
 punkty kontrolne transakcji, 284  
 putClientProperty(), 399, 403

**Q**

QuadCurve2D, 509, 513  
 QuadCurve2D.Double, 522  
 quadTo(), 513, 522  
 QueuedJobCount, 609

**R**

ramki wewnętrzne, 483  
 obiekt nasłuchujący weta zmiany, 495  
 odrysowywanie zawartości, 497  
 okna dialogowe, 496  
 przeciąganie zarysu ramki, 497  
 rozmieszczenie, 487  
 zamknięcie, 495  
     zgłaszanie weta zmiany właściwości, 495  
 Random, 797  
 RandomAccessFile, 563  
 Raster, 567, 570  
     getDataElements(), 570  
     getPixel(), 570  
     getPixels(), 570  
 read(), 198, 555, 560, 562, 563, 802  
 ReadableByteChannel, 198  
 Reader, 176  
 readObject(), 724  
 readThumbnail(), 561, 563  
 REAL, 236, 287  
 rebind(), 871

Rectangle2D, 508, 509, 511  
 Rectangle2D.Double, 711  
 ReferenceUriSchemesSupported, 609  
 ReflectPermission, 751  
 refleksja, 671, 843  
 REG\_BINARY, 935  
 REG\_DWORD, 935  
 REG\_SZ, 935  
 register(), 889  
 registerGroup(), 890  
 rejestr systemu Windows, 932, 933  
     edytor rejestru, 933  
     funkcje kontroli typów, 946  
     interfejs dostępu, 934  
     klucze, 934  
     metody macierzyste, 935  
     pobieranie wartości, 935  
     przeglądanie nazw kluczy, 936  
     uchwyt klucza, 936  
     węzły, 934  
     Win32RegKey, 934  
     Win32RegKeyNameEnumeration, 936  
     zapisywane wartości, 936  
 rejestracja aktywności RMI, 874  
 rejestratory RMI, 875  
 rekordy, 231  
     wstawiania, 266  
 RELATIVE, 130  
 relative(), 264, 268  
 relativize(), 207  
 Relax NG, 119  
 releaseSavepoint(), 284, 286  
 ReleaseStringChars(), 903  
 ReleaseStringUTFChars(), 902, 904  
 ReleaseXxxArrayElements(), 922  
 reload(), 405, 410  
 REMAINDER, 130  
 REMARKS, 282  
 RemoteException, 866, 867, 884  
 remove(), 610  
 removeCellEditorListener(), 394  
 removeColumn(), 374, 380  
 removeElement(), 354, 355  
 removeNodeFromParent(), 404, 410  
 removePropertyChangeListener(), 680, 682, 699  
 removeTabAt(), 478, 482  
 removeTreeModelListener(), 426, 428  
 removeVetoableChangeListener(), 676, 681, 682  
 RenderingHints, 551, 554  
 RequestingUserName, 609  
 REQUIRED, 123  
 RescaleOp, 576, 579  
 reset(), 779  
 reshape(), 486, 499

resolve(), 207  
 resolveEntity(), 125  
 ResourceBundle, 324, 327  
     getBundle(), 324, 328  
     getKeys(), 328  
     getObject(), 328  
     getString(), 328  
     getStringArray(), 328  
     handleGetObject(), 328  
 ResourceBundles, 326  
 Result, 176  
 ResultSet, 242, 244, 245, 263, 270, 283  
     absolute(), 268  
     afterLast(), 269  
     beforeFirst(), 268  
     cancelRowUpdates(), 269  
     close(), 245  
     column(), 245  
     CONCUR\_READ\_ONLY, 264  
     CONCUR\_UPDATABLE, 266  
     deleteRow(), 269  
     first(), 268  
     getConcurrency(), 268  
     getMetaData(), 283  
     getRow(), 268  
     getType(), 268  
     getXxx(), 245  
     insertRow(), 269  
     isAfterLast(), 269  
     isBeforeFirst(), 269  
     isFirst(), 269  
     isLast(), 269  
     last(), 268  
     moveToCurrentRow(), 269  
     moveToInsertRow(), 269  
     next(), 244  
     previous(), 268  
     relative(), 268  
     TYPE\_SCROLL\_INSENSITIVE, 264, 266  
     updateRow(), 269  
     updateXxx(), 269  
 ResultSetMetaData, 275, 283  
     getColumnCount(), 283  
     getColumnDisplaySize(), 283  
     getColumnName(), 283  
     getColumnName(), 283  
 RetentionPolicy, 843  
     SOURCE, 845  
 RetinaScanCallback, 773  
 return, 923  
 RGB, 541, 566  
 RMI, 863, zob. także wywoływanie zdalnych metod  
     rejestracja aktywności, 874  
     rejestratory, 875  
 RMIClassLoader, 875  
 RMI Security Manager, 880  
 role, 768  
 rollback(), 284, 286  
 rotate(), 535, 539  
 RoundRectangle2D, 508, 509, 511  
 RoundRectangle2D.Double, 522  
 RowSet, 270, 272  
     execute(), 273  
     getCommand(), 273  
     getPassword(), 273  
     getURL(), 272  
     getUsername(), 272  
     setCommand(), 273  
     setPassword(), 273  
     setURL(), 272  
     setUsername(), 273  
 rozmieszczenie  
     kaskadowe, 487  
     sąsiadujące, 487  
 rozmycie obrazu, 577  
 rozszerzenia maszyny wirtualnej, 729  
 rozwijanie klasy, 728  
 RSA, 780, 803  
     klucze, 803  
 rt.jar, 729, 730  
 RTF, 457  
 run(), 767  
 Runtime, 743  
 RuntimePermission, 751  
 rysowanie  
     antialiasing, 550  
     figur, 506  
     potok, 508  
     węzłów, 412  
 rysunki, 505

## \\$

SASL, 806  
 Savepoint, 286  
     getSavepointId(), 286  
     getSavepointName(), 286  
 SAX, 107, 150  
     wyszukiwanie elementów, 152  
 SAXParseException, 126  
     getColumnName(), 126  
     getLineNumber(), 126  
 SAXParser, 152  
     parse(), 155  
 SAXParserFactory, 152, 153, 154  
     isNamespaceAware(), 154  
     isValidating(), 154  
     newInstance(), 154

newSAXParser(), 154  
 setNamespaceAware(), 154  
 setValidating(), 154  
 SAXSource, 176, 177, 181  
 scale(), 534, 535, 539  
 Scanner, 191, 198  
 schowek, 610  
     Clipboard, 611  
     Copy, 612  
     formaty danych, 615  
     interfejsy, 611  
     klasy, 611  
     przekazywanie danych, 611  
     przekazywanie obiektów Java, 621  
     przekazywanie obrazów, 617  
     przekazywanie tekstu, 612  
     rodzaje danych, 611  
     Transferable, 611, 615  
 scrollPathToVisible(), 405, 409  
 SecretKeySpec, 801  
 SecureRandom, 797  
 SecurityException, 744, 746  
 SecurityManager, 746, 747  
     checkPermission(), 747  
 SecurityPermission, 752  
 sekwencje obrazów, 560  
 SELECT, 233  
     FROM, 234  
     LIKE, 234  
     NOT LIKE, 234  
     WHERE, 234  
 Serializable, 844  
 SerializablePermission, 751  
 serializacja, 706, 864  
 ServerSocket, 191, 193  
     accept(), 193  
     close(), 193  
 serwer, 183, 230, 862  
     aplikacji, 230, 288  
     FTP, 209  
     gniazda, 191  
     HTTP, 191  
     implementacja, 191  
     obsługa wielu klientów, 194  
     odpowiedź, 221  
     połączenia, 183  
     wątki, 194  
     Web, 183, 216  
         wysyłanie danych do formularzy, 216  
 serwlety, 216, 288  
 set(), 507, 671  
 setAllowsChildren(), 401, 403  
 setAllowUserInteraction(), 208, 214  
 setAsksAllowsChildren(), 401, 402  
     setAsText(), 690, 692, 696  
     setAsynchronousLoadPriority(), 459  
     setAttribute(), 160, 164  
     setAttributeNS(), 164  
     setAutoCommit(), 283, 285  
     setAutoResizeMode(), 379  
     setBottomComponent(), 477  
     SetByteArrayRegion(), 935  
     setCalendar(), 310  
     setCellEditor(), 393  
     setCellRenderer(), 357, 359, 393  
     setCellSelectionEnabled(), 370  
     setClip(), 539, 541, 583  
     setClosable(), 498  
     setClosed(), 495, 496, 499, 676  
     setClosedIcon(), 414  
     setColumnSelectionAllowed(), 370, 380  
     setCommand(), 271, 273  
     setComponentAt(), 483  
     setComposite(), 507, 544, 549  
     setConnectTimeout(), 215  
     setContentHandler(), 181  
     setContentPane(), 499  
     setContents(), 612, 615  
     setContextClassLoader(), 738  
     setContinuousLayout(), 475, 477  
     setDataElements(), 567, 570  
     setDecomposition(), 317  
     setDefault(), 296  
     setDefaultRenderer(), 384  
     setDisplayNames(), 685  
     setDoInput(), 208, 214  
     setDoOutput(), 208, 214, 218, 221  
     SetDoubleField(), 906  
     setDragEnabled(), 631  
     setDragMode(), 497, 498  
     setEditable(), 406, 458  
     setEntityResolver(), 125  
     setErrorhandler(), 125  
     setExceptionListener(), 724  
     setExpert(), 686  
     setFileName(), 668, 671  
     setFixedCellHeight(), 353  
     setFixedCellWidth(), 353  
     setFrameIcon(), 485, 499  
     setGroupingUsed(), 302  
     setHeaderRenderer(), 384, 393  
     setHeaderValue(), 384, 393  
     setHidden(), 686  
     setIcon(), 499  
     setIconAt(), 483  
     setIconifiable(), 499  
     setIfModifiedSince(), 214  
     setIgnoringElementContentWhitespace(), 124, 126

setIndeterminate(), 466, 473  
setInput(), 560  
SetIntField(), 906, 936  
setLayoutOrientation(), 348  
setLeafIcon(), 414  
setLeftComponent(), 477  
setLenient(), 310  
setLocale(), 295, 319  
setLogWriter(), 242  
setMaximizable(), 499  
setMaximum(), 463, 473, 493, 499  
setMaximumFractionDigits(), 302  
setMaximumIntegerDigits(), 302  
setMaxWidth(), 368, 380  
setMillisToDecidePopup(), 469  
setMillisToPopup(), 469  
setMinimum(), 463, 473  
setMinimumFractionDigits(), 302  
setMinimumIntegerDigits(), 302  
setMinWidth(), 368, 380  
setModifiedSince(), 208  
setName(), 685, 775  
setNamespaceAware(), 129, 149, 150, 153, 154  
setNote(), 474  
setNumberFormat(), 310  
setObject(), 699, 700, 705  
SetObjectArrayElement(), 919, 923  
SetObjectField(), 906  
setOneTouchExpandable(), 475, 477  
setOpenIcon(), 414  
setOutput(), 564  
setOutputProperty(), 164  
setPage(), 458, 460, 462  
setPageable(), 589, 598  
setPaint(), 507, 532, 533  
setParseIntegerOnly(), 302  
setPassword(), 271, 273, 775  
setPersistenceDelegate(), 724  
setPixel(), 565, 566, 571  
setPixels(), 566, 571  
setPreferredWidth(), 368, 380  
setPrintable(), 588  
setProgress(), 467, 474  
setProperty(), 880  
setPropertyEditorClass(), 687, 690  
setPropertyName(), 671  
setPrototypeCell(), 353  
setPrototypeCellValue(), 353  
setReadTimeout(), 215  
setRenderHints(), 551  
setRenderingHint(), 549, 550, 554  
setRenderingHints(), 506, 554  
setRequestProperty(), 208, 209, 215  
setResizable(), 368, 381, 498  
setRightComponent(), 477  
setRootVisible(), 400, 402  
setRowHeight(), 370, 379  
setRowMargin(), 370, 379  
setRowSelectionAllowed(), 370, 379  
setRunning(), 672  
setSavepoint(), 284, 286  
setSecurityManager(), 749  
setSeed(), 798  
setSelected(), 499  
setSelectedIndex(), 478, 482  
setSelectionMode(), 345, 370, 381  
setShortDescription(), 686  
setShowsRootHandles(), 400, 402  
setSoTimeout(), 188  
SetStaticXxxField(), 910  
setStrength(), 317  
setString(), 253, 464, 473  
setStringPainted(), 464, 473  
setStroke(), 507, 524, 525, 530, 531  
setTabLayoutPolicy(), 479, 483  
setTableName(), 273  
setText(), 459, 666  
setTimeZone(), 310  
setTitle(), 701  
setTitleAt(), 482  
setTopComponent(), 477  
setToRotation(), 537, 538  
setToScale(), 537, 538  
setToShear(), 537, 538  
setToTranslation(), 537, 538  
setTransferHandle(), 630  
setTransform(), 537, 539  
setURL(), 271, 272  
setUseCaches(), 208, 214  
setUsername(), 271, 273  
setUserObject(), 397, 402  
setValidating(), 126, 154  
setValue(), 473, 696, 934, 935, 936  
setValueAt(), 367, 392  
setVisible(), 486, 500  
setVisibleRowCount(), 345, 348  
setWidth(), 369, 381  
setXxx(), 258  
SetXxxArrayRegion(), 921, 922  
SetXxxField(), 910  
Severity, 609  
SGML, 103  
SHA1, 777  
Shape, 520, 523  
ShapeMaker, 521  
ShapePanel, 521  
shear(), 535, 539  
SheetCollate, 609

- short, 287  
**ShortLookupTable**, 576  
**shouldSelectCell()**, 391, 392, 394  
**show()**, 500  
**shutdownInput()**, 198  
**shutdownOutput()**, 198  
**Sides**, 609  
**sieć**, 183
  - hasła dostępu, 209
  - poczta elektroniczna, 222
  - przesyłanie danych, 191**SimpleBeanInfo**, 684, 685
  - loadImage()**, 685**SimpleDateFormat**, 319  
**SimpleDoc**, 601, 603  
**SimpleLoginModule**, 769, 771  
**SimplePrincipal**, 769  
**SimulatedActivity**, 466  
**SINGLE\_TREE\_SELECTION**, 415  
**skalowanie**, 534  
**składanie obrazów**, 541
  - CLEAR**, 543
  - DST**, 543
  - DST\_ATOP**, 543
  - DST\_IN**, 543
  - DST\_OUT**, 543
  - DST\_OVER**, 543
  - piksel docelowy, 542
  - projektowanie reguły, 542
  - reguły Portera-Duffa, 544
  - reguły składania, 542
  - SRC**, 543
  - SRC\_ATOP**, 543
  - SRC\_IN**, 543
  - SRC\_OUT**, 543
  - SRC\_OVER**, 543
  - XOR**, 543**składanie przekształceń**, 535  
**składnica kluczy**, 782, 783, 790  
**składowe obiektu**, 906  
**skrót wiadomości**, 777  
**skrypty CGI**, 216  
**SMALLINT**, 236, 287  
**SMTP**, 222  
**Socket**, 187, 188, 198
  - connect()**, 188
  - getInputStream()**, 187
  - getOutputStream()**, 187
  - isClosed()**, 189
  - isConnected()**, 189
  - isInputShutdown()**, 198
  - isOutputShutdown()**, 198
  - setSoTimeout()**, 188
  - shutdownInput()**, 198**shutdownOutput()**, 198  
**SocketChannel**, 198, 204  
**SocketPermission**, 750  
**SocketTimeoutException**, 215  
**sort()**, 312  
**sortowanie**
  - porządek alfabetyczny, 311**Source**, 176  
**splot**, 577  
**spójność bazy danych**, 283  
**SQL**, 227, 231
  - aktualizowalne zbiory wyników zapytań, 265
  - ARRAY**, 287
  - CREATE TABLE**, 235, 242
  - DDL**, 244
  - DROP TABLE**, 242
  - FROM**, 233
  - funkcje, 235
  - INSERT**, 235
  - LIKE**, 234
  - łączenie tabel, 232
  - metadane, 274
  - modyfikacja danych, 235
  - NOT LIKE**, 234
  - polecenia, 245
  - polecenia przygotowane, 252
  - SELECT**, 233
  - słowa kluczowe, 233
  - tworzenie tabel, 235
  - typy danych, 236, 287
  - WHERE**, 234
  - wstawianie danych, 235
  - wykonywanie zapytań, 252
  - wynik zapytania, 232
  - wypełnianie bazy danych, 248
  - zapytania, 233
  - zarządzanie połączonymi, 245
  - zbiór wyników, 245**SQLException**, 247, 284
  - getErrorCode()**, 248
  - getNextException()**, 247
  - getSQLState()**, 248**SQLPermission**, 752  
**SRC**, 543  
**SRC\_ATOP**, 543  
**SRC\_IN**, 543  
**SRC\_OUT**, 543  
**SRC\_OVER**, 543  
**sRGB**, 567  
**SSL**, 806  
**startDocument()**, 151, 155  
**startElement()**, 151, 152, 153, 155  
**stateChanged()**, 480

Statement, 245, 263, 286, 725  
 addBatch(), 286  
 close(), 244, 245  
 createStatement(), 242  
 execute(), 244  
 executeBatch(), 286  
 executeQuery(), 244  
 executeUpdate(), 242, 244  
 getResultSet(), 244  
 getUpdateCount(), 244  
 static, 831  
 stdarg.h, 919  
 sterowniki JDBC, 228  
     typ 1, 228  
     typ 2, 229  
     typ 3, 229  
     typ 4, 229  
 stopCellEditing(), 391, 392, 394  
 StreamPrintService, 603  
 StreamPrintServiceFactory, 603  
     getPrintService(), 604  
 StreamResult, 165  
 StreamSource, 176, 177, 180  
 strefy czasowe, 310  
 String, 287, 312, 913  
 StringBuffer, 472  
 StringSelection, 612, 617  
 Stroke, 524  
 strona WWW, 216  
 struktura dokumentu XML, 104  
 struktury danych  
     nieskończone, 428  
 strumienie, 187, 219  
     monitorowanie postępu, 469  
     tworzenie, 470  
     usługi drukowania, 603  
 strumienie szyfrujące, 801  
     CipherInputStream, 802  
 Subject, 767  
     doAs(), 767  
     doAsPrivileged(), 767  
     getPrincipals(), 767  
 sun.misc.Base64Encoder, 214  
 SupportedValuesAttribute, 604  
 supportsBatchUpdates(), 286  
 supportsCustomEditor(), 694, 697  
 supportsResultSetConcurrency(), 265, 270  
 supportsResultSetType(), 265, 269  
 SVG, 162  
 Swing  
     drzewa, 394  
     filtr strumieni, 469  
     formatowanie tekstu, 429  
     JDesktopPane, 485  
 JEditorPane, 457  
 JFrame, 485  
 JInternalFrame, 485  
 JList, 344  
 JProgressBar, 463  
 JScrollPane, 344  
 JSplitPane, 475  
 JTabbedPane, 478  
 JTextArea, 457  
 JTextField, 457  
 JTree, 394  
 listy, 343  
 monitory postępu, 466  
 organizatory komponentów, 474  
 panele dzielone, 475  
 panele pulpitu, 483  
 panele z zakładkami, 478  
 pasek przewijania, 362  
 przekazywanie danych, 627  
 ramki wewnętrzne, 483  
 rozmieszczenie komponentów, 487  
 tabele, 359  
 tekst, 457  
     wskaźnik postępu, 463  
 Swng, 675  
 sygnatury metody, 911  
 SyncProviderException, 272, 273  
 System, 897  
     loadLibrary(), 896, 897  
     setSecurityManager(), 749  
 szeregowanie parametrów, 864, 865  
 szyfr Cezara, 736  
 szyfrowanie, 736, 795  
     AES, 796, 797, 803  
     algorytmy, 795  
     Cipher, 795  
     DES, 795, 796  
     dopełnienie ostatniego bloku, 796  
     klucze, 796, 797  
     pliki klas, 737  
     RSA, 780, 803  
     strumienie, 801  
     symetryczne, 795  
     tryb pracy algorytmu, 796  
     z kluczem publicznym, 803

**§**

ścieżki drzewa, 403  
 ślad pędzla, 507, 524  
 połączenia, 525  
 przerywany wzór, 526  
 szerokość, 524  
 wartość graniczna, 525  
 zakończenia, 524

**T**

tabele, 231, 359  
 drukowanie, 363  
 edycja zawartości komórek, 367  
 implementacja edytora komórek, 390  
 JTable, 359  
 kolumny, 368  
 model, 359, 363  
 model wyboru, 370  
 obiekt rysujący, 367  
 prezentacja danych, 367  
 przesuwanie kolumny, 362  
 przewijanie zawartości, 362  
 rysowanie komórek, 367  
 szerokość kolumn, 362, 368  
 tworzenie, 235  
 tworzenie edytów, 390  
 ukrywanie kolumn, 374  
 widoki, 362  
 wybór kolumn, 370  
 wybór komórek, 370  
 wybór wierszy, 370  
 wysokość komórek, 370  
 wyświetlanie kolumn, 374  
 TABLE\_CAT, 282  
 TABLE\_NAME, 282  
 TABLE\_SCHEM, 282  
 TABLE\_TYPE, 282  
 TableCellEditor, 391, 392, 393  
   getTableCellEditorComponent(), 393  
 TableCellRenderer, 383, 393  
   getTableCellRendererComponent(), 393  
 TableColumn, 368, 380, 393  
   setMaxWidth(), 380  
   setMinWidth(), 380  
   setPreferredWidth(), 380  
   setResizable(), 381  
   setWidth(), 381  
 TableColumnModel, 368, 380  
   getColumn(), 380  
   getCellEditor(), 393  
   getCellRenderer(), 393  
   setHeaderRenderer(), 393  
   setHeaderValue(), 393  
 TableModel, 366  
   getRowCount(), 366  
   getColumnName(), 367  
   getRowCount(), 366  
   getValueAt(), 367  
   isCellEditable(), 367  
   setValueAt(), 367  
 tablice, 916, 919  
   C, 919  
   C++, 920

  jarray, 919  
   rozmiar, 919  
 TCP, 187  
 tekst, 457  
   hiperłącza, 459  
 telnet, 183  
 testowanie, 830  
 Text, 110  
 text/plain, 215  
 TextField, 297  
 TextLayout, 540, 541  
   getAdvance(), 541  
   getAscent(), 541  
   getDescent(), 541  
   getLeading(), 541  
 TexturePaint, 532, 533, 534  
 Thread, 738  
   getContextClassLoader(), 738  
   setContextClassLoader(), 738  
 ThreadedEchoHandler, 194  
 Throw(), 923, 927  
 ThrowNew(), 923, 927  
 throws, 839  
 Time, 287  
 TIME, 236, 287  
 Timestamp, 287  
 TIMESTAMP, 236, 287  
 TimeZone, 310  
   getAvailableIDs(), 310  
   getDefault(), 310  
   getDisplayName(), 311  
   getID(), 311  
   getTimeZone(), 311  
   inDaylightTime(), 311  
   useDaylightTime(), 311  
 TitlePositionEditor, 688  
 toArray(), 610  
 Tomcat, 288  
 Toolkit, 614  
   getSystemClipboard(), 614  
 toString(), 296, 303, 396, 521, 841  
 transakcje, 283  
   aktualizacje wsadowe, 284  
   automatyczne zatwierdzanie, 283  
   odwołanie, 283  
   punkty kontrolne, 284  
   tworzenie, 283  
 Transferable, 611, 615  
   getTransferData(), 615  
   isDataFlavorSupported(), 615  
 TransferHandler, 630  
 transform(), 164, 176, 507, 537, 539  
 Transformer, 164  
   setOutputProperty(), 164  
   transform(), 164

TransformerFactory, 164, 180  
 newInstance(), 164  
 translate(), 535, 539, 590  
 TreeCellRenderer, 412, 414  
 getTreeCellRendererComponent(), 414  
 TreeModel, 116, 396, 402, 422, 428  
 addTreeModelListener(), 428  
 getChild(), 428  
 getChildCount(), 428  
 getIndexofChild(), 428  
 getRoot(), 428  
 isLeaf(), 402, 428  
 removeTreeModelListener(), 428  
 valueForPathChanged(), 429  
 TreeModelEvent, 429  
 TreeModelListener, 429  
 treeNodesChanged(), 429  
 treeNodesInserted(), 429  
 treeNodesRemoved(), 429  
 treeStructureChanged(), 429  
 TreeNode, 396, 402, 403, 409  
 children(), 409  
 getAllowsChildren(), 402  
 getChildAt(), 409  
 getChildCount(), 409  
 getParent(), 409  
 isLeaf(), 402  
 treeNodesChanged(), 429  
 treeNodesInserted(), 429  
 treeNodesRemoved(), 429  
 TreePath, 404, 409, 416  
 getLastPathComponent(), 409  
 TreeSelectionEvent, 416, 421  
 getPath(), 421  
 getPaths(), 421  
 TreeSelectionListener, 415, 421  
 valueChanged(), 421  
 TreeSelectionModel, 415  
 treeStructureChanged(), 429  
 trim(), 110, 298  
 trwałość ziarnek JavaBeans, 705, 715  
 dowolne dane, 709  
 implementacja delegatu trwałości, 710  
 kodowanie obiektu, 711  
 odtworzenie stanu obiektów, 712  
 serializacja, 706  
 tworzenie obiektu na podstawie właściwości, 711  
 tworzenie obiektu przez metodę fabryki, 712  
 właściwości tymczasowe, 713  
 XMLEncoder, 709, 710  
 try, 246  
 tunelowanie HTTP, 875  
 tworzenie  
 certyfikatów, 781  
 dokumentów XML, 160

drzewa DOM, 160  
 grafiki, 506  
 klas pozwoleń, 755  
 nazw metod, 672  
 plików JAR, 663  
 ziarnek JavaBeans, 660  
 TYPE\_BILINEAR, 572  
 TYPE\_BYTE\_GRAY, 568  
 TYPE\_FORWARD\_ONLY, 264  
 TYPE\_INT\_ARGB, 566  
 TYPE\_SCROLL\_INSENSITIVE, 264, 266  
 TYPE\_SCROLL\_SENSITIVE, 264  
 typy danych C, 898  
 typy danych Java, 287, 898  
 typy danych SQL, 236, 287  
 typy MIME, 616

**U**

UDP, 187  
 układ współrzędnych, 534  
 unbind(), 870, 871  
 UnicastRemoteObject, 867  
 exportObject(), 867  
 Unicode, 291, 323  
 uniform resource identifiers, 205  
 uniform resource locator, 205  
 uniform resource name, 205  
 UnixLoginModule, 763  
 UnixNumericGroupPrincipal, 763  
 UnixPrincipal, 762  
 UnknownHostException, 187  
 UPDATE, 242, 253  
 update(), 779, 796, 800  
 updateDouble(), 266  
 updateRow(), 266, 267, 269  
 uporządkowanie łańcuchów, 312  
 uporządkowanie słownikowe, 311  
 URL, 148, 205, 206, 606  
 absolutny, 205  
 hierarchiczny, 206  
 identyfikatory, 206  
 nieprzenikalny, 205  
 relatywizacja, 206  
 rozwiązywanie, 206  
 specyfikacja, 205  
 względny, 205  
 URL, 148, 204, 205, 214, 749  
 nagłówki żądań, 208  
 openConnection(), 214  
 openStream(), 214  
 pobieranie informacji, 207  
 URLClassLoader, 729  
 URLConnection, 205, 207, 208, 214, 218, 221  
 connect(), 215

getAllowUserInteraction(), 214  
 getConnectTimeout(), 215  
 getContent(), 216  
 getContentEncoding(), 215  
 getContentLength(), 215  
 getContentType(), 215  
 getDate(), 215  
 getDoInput(), 214  
 getDoOutput(), 214  
 getExpiration(), 215  
 getHeaderField(), 215  
 getHeaderFieldKey(), 215  
 getHeaderFields(), 215  
 getIfModifiedSince(), 214  
 getLastModified(), 216  
 getRequestProperties(), 215  
 getUseCaches(), 214  
 openConnection(), 207  
 openInputStream(), 216  
 openOutputStream(), 216  
 setAllowUserInteraction(), 208, 214  
 setConnectTimeout(), 215  
 setDoInput(), 208, 214  
 setDoOutput(), 208, 214  
 setIfModifiedSince(), 214  
 setModifiedSince(), 208  
 setReadTimeout(), 215  
 setRequestProperty(), 208, 215  
 setUseCaches(), 208, 214  
 URLDecoder, 222  
 decode(), 222  
 URLEncoder, 222  
 encode(), 222  
 URN, 205  
 useDaylightTime(), 311  
 usługi drukowania, 599  
     GIF, 600  
     rodzaje dokumentów, 599, 600  
     strumienie, 603  
     źródło danych, 599, 600  
 usługi rejestru początkowego RMI, 868  
 usługi sieciowe, 183  
 usługi uwierzytelniania, 762  
 UTF-16, 323, 900, 902  
 UTF-32, 902  
 UTF-8, 323, 900  
 uwierzytelnianie  
     oparte na rolach, 768  
 uwierzytelnianie użytkowników, 762  
     grant, 763  
     JAAS, 762  
     moduł logowania, 763  
     moduły, 763  
     nadzorcy, 763

podmiot, 763  
 pozwolenia, 764  
 uwierzytelnianie wiadomości, 784  
     klucze, 784  
     łańcuch zaufania, 784  
     podpis zaufanego pośrednika, 785  
     zaufany pośrednik, 785  
**V**  
 va\_list, 919  
 valueChanged(), 346, 415, 421  
 valueForPathChanged(), 426, 429  
 VARCHAR, 236, 287  
 Variable, 427  
 Verisign, Inc., 785  
 vetoableChange(), 496, 500, 681  
 VetoableChangeListener, 495, 500, 676, 681  
     vetoableChange(), 500, 681  
 VetoableChangeSupport, 676, 677, 681  
     addVetoableChangeListener(), 681  
     fireVetoableChange(), 682  
     getVetoableChangeListeners(), 682  
     removeVetoableChangeListener(), 681

Visual Basic, 658  
 vm\_args, 928

**W**  
 W3C, 152  
 waluta, 297, 302  
     formatowanie, 302  
     identyfikatory, 303  
 warning(), 125, 126  
 warstwa pośrednia, 230  
 wartość alfa, 542  
 wątki, 194  
 wdrażanie aplikacji RMI, 871  
 Web, 183, 216  
 WebRowSet, 270  
 wektor obiektów, 354  
 weryfikacja dokumentu XML, 124  
 weryfikacja kodu maszyny wirtualnej, 738  
 weryfikacja podpisu cyfrowego, 781  
 weryfikacja podpisu plików JAR, 783  
 weryfikator kodu, 738, 741  
 węzły, 395  
     nadrzędne, 395  
     podrzędne, 395  
     przeglądanie, 410  
     rysowanie, 412  
 WHERE, 234  
 wielokąty, 509, 514, 521  
 Win32RegKey, 934  
 Win32RegKeyNameEnumeration, 936

- WindowListener, 497  
 Windows, 931, 933  
     rejestr, 933  
 wizualne projektowanie aplikacji, 671  
 właściwości, 667  
     indeksowane, 674  
     ograniczone, 676  
     powiązane, 674  
     proste, 673  
     ziarnka, 673  
 WordCheckPermission, 756  
 WritableByteChannel, 198  
 WritableRaster, 565, 567, 570  
     setDataElements(), 570  
     setPixel(), 571  
     setPixels(), 571  
 write(), 198, 555, 562, 564, 802  
 writeInsert(), 561, 564  
 writeObject(), 724  
 writeStatement(), 724  
 WSDL, 863  
 wskazówki operacji graficznych, 506, 549  
     antialiasing, 549  
     KEY\_ALPHA\_INTERPOLATION, 550  
     KEY\_ANTIALIASING, 550  
     KEY\_COLOR\_RENDERING, 550  
     KEY\_DITHERING, 550  
     KEY\_FRACTIONAL\_METRICS, 550  
     KEY\_INTERPOLATION, 550  
     KEY\_RENDERING, 550  
     KEY\_STROKE\_CONTROL, 550  
     KEY\_TEXT\_ANTIALIASING, 550  
 wskaźnik postępu, 463  
     JProgressBar, 463  
     monitory postępu, 466  
     pasek postępu, 463  
     ProgressMonitor, 466  
     strumienie wejścia, 469  
 współrzędne, 509  
     ekranowe, 534  
     użytkownika, 534  
 wyjątki  
     ArrayIndexOutOfBoundsException, 923  
     ArrayStoreException, 923  
     CloneNotSupportedException, 884  
     IllegalStateException, 560  
     IndexOutOfBoundsException, 560  
     MissingResourceException, 325  
     ParseException, 298  
     PrinterException, 581  
     PropertyVetoException, 676  
     RemoteException, 866, 867, 884  
     SecurityException, 744, 746  
     SQLException, 284  
 SyncProviderException, 272  
 UnknownHostException, 187  
 wykonywanie zapytań SQL, 252  
 wykres, 688  
 wykrywanie krawędzi, 578  
 wynik zapytania, 232  
 wypełnianie obszaru, 506, 507, 508, 532  
     gradient, 532  
     kolory, 533  
     obrazek wzorca, 533  
     prostokąt zakotwiczenia, 533  
 wyrażenia XPath, 142  
 wyrzucanie wyjątków, 923  
 wysyłanie danych do formularzy, 216  
 wysyłanie poczty elektronicznej, 222  
 wytaj i wklej, 610  
 wywoływanie funkcji języka C, 892  
     parametry, 895  
     printf(), 892  
 wywoływanie metod języka Java, 912, 917  
 wywoływanie metod obiektów, 912  
 wywoływanie metod statycznych, 916  
 wywoływanie zdalnych metod, 861, 864  
     aktywacja obiektów serwera, 884  
     deskryptory aktywacji, 884  
     dynamiczne ładowanie klas, 878  
     identyfikator grupy, 886  
     implementacje, 866  
     interfejsy, 866  
     menedżer bezpieczeństwa, 879  
     namiastka, 864  
     plik polityki, 880  
     przekazywanie parametrów, 876  
     przygotowanie wdrożenia, 871  
     rejestracja aktywności, 874  
     rejestratory, 875  
     RMISecurityManager, 880  
     szeregowanie parametrów, 864  
     UnicastRemoteObject, 867  
     usługa rejestru początkowego, 868  
     wdrażanie aplikacji, 871  
     wyszukiwanie obiektów serwera, 868  
 wzorce model-widok-nadzorca, 395  
 wzorce nazw, 671  
 wzorce projektowe, 671

**X**

- X Window, 610  
 XML, 102, 864  
     atrybuty, 104, 105  
     ATTLIST, 122  
     CDATA, 106, 123  
     deklaracja typu dokumentu, 104  
     DOM, 107

element korzenia, 104  
 instrukcje przetwarzania, 106  
 JAXP, 107  
 komentarze, 106  
 kontrola poprawności dokumentów, 118  
 mieszana zawartość, 105  
 parser, 107  
 parsowanie dokumentów, 107  
 PCDATA, 121  
 pliki, 102  
 przestrzeń nazw, 148  
 przetwarzanie dokumentów, 107  
 referencje bytów, 106  
 referencje znaków, 106  
 SAX, 107, 150  
 struktura dokumentu, 104  
 wartości atrybutów, 104  
 wartości domyślne atrybutów, 123  
 wielkość znaków, 103  
 XPath, 142  
 znaczniki, 103  
 XML Schema, 119, 126, 148  
 atrybuty, 128  
 definicje elementów, 128  
 parsowanie dokumentu XML, 129  
 powtórzenia elementów, 128  
 przestrzeń nazw, 127  
 typ elementu, 127  
 typy proste, 127  
 typy złożone, 127  
 XMLDecoder, 724  
 getExceptionListener(), 725  
 readObject(), 724  
 setExceptionListener(), 724  
 XMLEncoder, 706, 707, 709, 710, 724  
 writeObject(), 724  
 writeStatement(), 724  
 xmlns, 148  
 xmlns:alias, 149  
 XMLReader, 181  
 parse(), 181  
 setContentHandler(), 181  
 XPath, 142, 147, 173  
 evaluate(), 147  
 funkcje, 143  
 wartość wyrażenia, 143  
 wyrażenia, 142  
 wyznaczanie wyrażeń, 143  
 zbiór węzłów, 143  
 XPathConstants, 143  
 NODE, 143  
 NUMBER, 143  
 XPathFactory, 143, 147  
 newInstance(), 147

newXPath(), 147  
 xsd:attribute, 128  
 xsd:boolean, 127  
 xsd:choice, 128  
 xsd:int, 127  
 xsd:schema, 128  
 xsd:sequence, 128  
 xsd:string, 127  
 XSL Schema Definition, 127  
 xsl:apply-templates, 173  
 xsl:output, 173  
 xsl:template, 173  
 xsl:value-of, 174  
 XSLT, 161, 173, 174

## Z

zadania drukowania, 581  
 zakładki, 479  
 zapytania SQL, 233  
 przygotowane, 252  
 wykonywanie, 252  
 zasada Gödla, 739  
 zasada składania obrazów, 506, 507  
 zasoby, 324  
 lokalizacja, 324  
 zbiory atrybutów drukowania, 606  
 zbiory pozwoleń, 744  
 zbiory rekordów, 263, 266, 270  
 aktualizowalne, 265  
 buforowane, 271  
 CachedRowSet, 271  
 modyfikacja, 272  
 przewijalne, 263  
 RowSet, 270  
 sprawdzanie, 272  
 zbiory znaków, 322  
 zbiór Mandelbrota, 567  
 zdalne obiekty, 884  
 zdalne wywoływanie metod, 863  
 zdarzenia, 668  
 PropertyChange, 674  
 zestawy znaków, 322  
 ziarnka, 657, 658  
 edytor właściwości, 687  
 EJB, 658  
 formatki, 667  
 indywidualizacja, 697  
 JavaBeans, 658  
 klasa informacyjna, 671, 683  
 korzystanie, 664  
 metody, 672  
 obiekty nasłuchujące, 674, 675  
 plik manifestu, 663  
 pliki JAR, 663

- ziarnka  
projektowanie, 669  
trwałość, 705  
tworzenie, 662  
tworzenie aplikacji, 662  
typy właściwości, 673  
właściwości indeksowane, 674  
właściwości ograniczone, 676  
właściwości powiązane, 674  
właściwości proste, 673  
wzorce nazw właściwości, 669  
wzorce nazw zdarzeń, 669  
zdarzenia, 672
- ZIP, 729  
zmienne  
LD\_LIBRARY\_PATH, 932  
znaczniki XML, 103  
znaki, 322

**ż**

- źródło danych, 288  
źródło danych JDBC, 237  
źródło danych JNDI, 288  
źródło kodu, 744  
certyfikaty, 744  
lokalizacja kodu, 744



# PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

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

Zmień swoją stronę WWW  
w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

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

GRUPA WYDAWNICZA  
**Helion SA**