Spis Treści

SPIS TREŚCI **1**

PODZIĘKOWANIA **11**

O AUTORZE 11

WPROWADZENIE **12**

PRAWDZIWY KOD DLA PRAWDZIWYCH PROGRAMISTÓW 13
W JAKI SPOSÓB ZORGANIZOWANA JEST TA KSIĄŻKA 13
Część 1.: Serwlety *14*
Część 2.: JavaServer Pages *15*
Część 3.: Technologie pomocnicze *16*
ZASTOSOWANE KONWENCJE 16
Podstawowa metoda 16
O WITRYNIE WWW 17

ROZDZIAŁ 1. PODSTAWOWE INFORMACJE O SERWLETACH I JAVA SERVER PAGES **18**

1.1 SERWLETY 18
1.2 ZALETY SERWLETÓW W PORÓWNANIU Z „TRADYCYJNYMI" PROGRAMAMI CGI 19
Efektywność *19*
Wygoda *20*
Duże możliwości *20*
Przenośność *20*
Bezpieczeństwo *20*
Niewielkie koszty *21*
1.3 JAVA SERVER PAGES 21
1.4 ZALETY JSP 22
W porównaniu z Active Server Pages (ASP) *22*
W porównaniu z PHP *22*
W porównaniu z serwletami *22*
W porównaniu z Server-Side Includes (SSI) *22*
W porównaniu z językiem JavaScript *23*
W porównaniu ze statycznym kodem HTML *23*
1.5 INSTALACJA I KONFIGURACJA 23
Zdobywanie oprogramowania do obsługi serwletów i dokumentów JSP *23*
Zapamiętaj adres lub zainstaluj dokumentację Java Servlet oraz JSP API *25*
Wskaż klasy używane przez kompilator Javy *25*
Unix (C Shell) 26
Windows 26
Umieść klasy w pakietach *26*

Skonfiguruj serwer ... *26*
Numer portu ... 27
Zmienna środowiskowa JAVA_HOME ... 27
Ustawienia pamięci systemu DOS ... 27
Ustawienie CR/LF w serwerze Tomcat 3.0 ... 27
Uruchomienie serwera ... *28*
Kompilacja i instalacja własnych serwletów ... *28*
Tomcat ... 28
Tomcat 3.1 ... 28
JSWDK ... 29
Java Web Server 2.0 ... 29

ROZDZIAŁ 2. PIERWSZE SERWLETY ... 30

2.1 PODSTAWOWA STRUKTURA SERWLETÓW ... 30
2.2 PROSTY SERWLET GENERUJĄCY ZWYCZAJNY TEKST ... 31
Kompilacja i instalacja serwletów ... *32*
Wywoływanie serwletów ... *33*
2.3 SERWLETY GENERUJĄCE KOD HTML ... 34
2.4 UMIESZCZANIE SERWLETÓW W PAKIETACH ... 35
Tworzenie serwletów należących do konkretnego pakietu ... *36*
Kompilacja serwletów należących do pakietów ... 36
Wywoływanie serwletów należących do pakietów ... *38*
2.5 PROSTE NARZĘDZIA POMOCNE PRZY TWORZENIU DOKUMENTÓW HTML ... 38
2.6 CYKL ŻYCIOWY SERWLETÓW ... 40
Metoda init ... *40*
Metoda service ... *41*
Metody doGet, doPost oraz doXxx ... *42*
Interfejs SingleThreadModel ... *43*
Metoda destroy ... *43*
2.7 PRZYKŁAD UŻYCIA PARAMETRÓW INICJALIZACYJNYCH ... 44
2.8 PRZYKŁAD WYKORZYSTANIA INICJALIZACJI SERWLETU I DATY MODYFIKACJI STRONY ... 47
2.9 TESTOWANIE SERWLETÓW ... 50
2.10 WEBCLIENT: INTERAKTYWNA WYMIANA INFORMACJI Z SERWEREM WWW ... 52
WebClient ... *52*
HttpClient ... *55*
NetworkClient ... *56*
SocketUtil ... *57*
CloseableFrame ... *57*
LabeledTextField ... *58*
Interruptible ... *59*

ROZDZIAŁ 3. OBSŁUGA ŻĄDAŃ: DANE PRZESYŁANE Z FORMULARZY ... 60

3.1 ZNACZENIE INFORMACJI PRZESYŁANYCH Z FORMULARZY ... 60
3.2 ODCZYTYWANIE DANYCH FORMULARZY W SERWLETACH ... 61
3.3 PRZYKŁAD: ODCZYT TRZECH KONKRETNYCH PARAMETRÓW ... 61
3.4 PRZYKŁAD: ODCZYT WSZYSTKICH PARAMETRÓW ... 64
3.5 SERWIS REJESTRACJI ŻYCIORYSÓW ... 67
3.6 FILTROWANIE ŁAŃCUCHÓW W POSZUKIWANIU ZNAKÓW SPECJALNYCH HTML ... 76
Implementacja filtrowania ... *76*
Przykład ... *77*

ROZDZIAŁ 4. OBSŁUGA ŻĄDAŃ: NAGŁÓWKI ŻĄDAŃ HTTP 80

4.1 Odczytywanie wartości nagłówków żądania w serwletach 80
4.2 Wyświetlanie wszystkich nagłówków 82
4.3 Nagłówki żądań protokołu HTTP 1.1 84
4.4 Przesyłanie skompresowanych stron WWW 88
4.5 Ograniczanie dostępu do stron WWW 90

ROZDZIAŁ 5. DOSTĘP DO STANDARDOWYCH ZMIENNYCH CGI 95

5.1 Odpowiedniki zmiennych CGI dostępne w serwletach 95
5.2 Serwlet wyświetlający wartości zmiennych CGI 98

ROZDZIAŁ 6. GENERACJA ODPOWIEDZI: KODY STATUSU 100

6.1 Określanie kodów statusu 100
6.2 Kody statusu protokołu HTTP 1.1 oraz ich przeznaczenie 102
6.3 Interfejs użytkownika obsługujący różne serwisy wyszukiwawcze 109

ROZDZIAŁ 7. GENERACJA ODPOWIEDZI: NAGŁÓWKI ODPOWIEDZI HTTP 114

7.1 Określanie nagłówków odpowiedzi z poziomu serwletów 114
7.2 Nagłówki odpowiedzi protokołu HTTP 1.1 oraz ich znaczenie 116
Accept-Ranges 116
Age 116
Allow 116
Cache-Control 116
Connection 117
Content-Encoding 117
Content-Language 118
Content-Length 118
Content-Location 118
Content-MD5 118
Content-Range 118
Content-Type 118
Date 120
ETag 120
Expires 120
Last-Modified 120
Location 120
Pragma 121
Refresh 121
Retry-After 121
Server 122
Set-Cookie 122
Trailer 122
Transfer-Encoding 122
Upgrade 122
Vary 122
Via 122
Warning 122
WWW-Authenticate 123
7.3 Trwałe przechowywanie stanu serwletu i automatyczne odświeżanie stron 123
7.4 Stosowanie trwałych połączeń HTTP 130
7.5 Wykorzystanie servletów do generacji obrazów GIF 133

ROZDZIAŁ 8. OBSŁUGA COOKIES 141

8.1 KORZYŚCI STOSOWANIA COOKIES 141
Identyfikacja użytkowników podczas trwania sesji na witrynach komercyjnych 141
Unikanie konieczności podawania nazwy użytkownika i hasła 142
Dostosowywanie witryny 142
Dobór reklam 142
8.2 NIEKTÓRE PROBLEMY ZWIĄZANE ZE STOSOWANIEM COOKIES 142
8.3 Narzędzia obsługi cookies dostępne w servletach 143
Tworzenie cookies 144
Atrybuty cookies 144
Umieszczanie cookies w nagłówkach odpowiedzi 146
Odczytywanie cookies nadesłanych przez przeglądarkę 146
8.4 PRZYKŁADY GENERACJI I ODCZYTYWANA COOKIES 146
8.5 PROSTE NARZĘDZIA DO OBSŁUGI COOKIES 149
Odnajdywanie cookie o określonej nazwie 150
Tworzenie cookies o długim czasie istnienia 150
8.6 INTERFEJS WYSZUKIWAWCZY Z MOŻLIWOŚCIĄ ZAPAMIĘTYWANIA USTAWIEŃ 151

ROZDZIAŁ 9. ŚLEDZENIE SESJI 156

9.1 POTRZEBA ŚLEDZENIA SESJI 156
Cookies 156
Przepisywanie adresów URL 157
Ukryte pola formularzy 157
Śledzenie sesji w serwletach 157
9.2 NARZĘDZIA PROGRAMISTYCZNE DO ŚLEDZENIA SESJI 158
Pobieranie obiektu HttpSession skojarzonego z bieżącym żądaniem 158
Pobieranie informacji skojarzonych z sesją 158
Kojarzenie informacji z sesją 160
Zakańczanie sesji 161
Kodowanie adresów URL przesyłanych do przeglądarki 161
9.3 SERVLET GENERUJĄCY INDYWIDUALNY LICZNIK ODWIEDZIN DLA KAŻDEGO UŻYTKOWNIKA . 162
9.4 INTERNETOWY SKLEP WYKORZYSTUJĄCY KOSZYKI I ŚLEDZENIE SESJI 164
Tworzenie interfejsu użytkownika 165
Obsługa zamówień 169
To czego nie widać: Implementacja koszyka i katalogu towarów 172

ROZDZIAŁ 10. ELEMENTY SKRYPTOWE JSP 178

10.1 ELEMENTY SKRYPTOWE 180
Tekst szablonu 180
10.2 WYRAŻENIA JSP 180
Predefiniowane zmienne 180
Składnia XML stosowana w wyrażeniach 181
Zastosowanie wyrażeń jako wartości atrybutów 181
Przykład 182
10.3 SKRYPTLETY JSP 183
Wykorzystanie skryptletów do warunkowego wykonania fragmentu strony JSP 185
Specjalna składnia skryptletów 186
10.4 DEKLARACJE JSP 186
Specjalna składnia zapisu deklaracji 187
10.5 PREDEFINIOWANE ZMIENNE 187

ROZDZIAŁ 11. DYREKTYWA PAGE: STRUKTURALIZACJA GENEROWANYCH SERWLETÓW 190

11.1 ATRYBUT IMPORT 190
Katalogi służące do przechowywania własnych klas 191
Przykład 192
11.2 ATRYBUT CONTENTTYPE 193
Generacja zwyczajnych dokumentów tekstowych 194
Generacja arkuszy kalkulacyjnych programu Microsoft Excel 195
11.3 ATRYBUT ISTHREADSAFE 198
11.4 ATRYBUT SESSION 200
11.5 ATRYBUT BUFFER 200
11.6 ATRYBUT AUTOFLUSH 200
11.7 ATRYBUT EXTENDS 200
11.8 ATRYBUT INFO 201
11.9 ATRYBUT ERRORPAGE 201
11.10 ATRYBUT ISERRORPAGE 201
11.11 ATRYBUT LANGUAGE 203
11.12 SKŁADNIA XML ZAPISU DYREKTYW 204

ROZDZIAŁ 12. DOŁĄCZANIE PLIKÓW I APLETÓW DO DOKUMENTÓW JSP 205

12.1 DOŁĄCZANIE PLIKÓW W CZASIE PRZEKSZTAŁCANIA STRONY 205
12.2 DOŁĄCZANIE PLIKÓW PODCZAS OBSŁUGI ŻĄDAŃ 207
DOŁĄCZANIE APLETÓW KORZYSTAJĄCYCH Z JAVA PLUG-IN 209
Znacznik akcji jsp:plugin 210
Znaczniki akcji jsp:param oraz jsp:params 212
Znacznik akcji jsp:fallback 213
Przykład: Generacja tekstu z cieniem 213

ROZDZIAŁ 13. WYKORZYSTANIE KOMPONENTÓW JAVABEANS W DOKUMENTACH JSP 219

13.1 PODSTAWOWE SPOSOBY UŻYCIA KOMPONENTÓW 220
Dostęp do właściwości komponentów 221
Określanie właściwości komponentów — prosty przypadek 222
Instalacja klas komponentów 222
13.2 PRZYKŁAD: STRINGBEAN 223
13.3 OKREŚLANIE WARTOŚCI WŁAŚCIWOŚCI KOMPONENTÓW 224
Kojarzenie właściwości z parametrami wejściowymi 227
Automatyczna konwersja typów 228
Kojarzenie wszystkich właściwości z parametrami wejściowymi 229
13.4 WSPÓLNE WYKORZYSTYWANIE KOMPONENTÓW 230
Warunkowe tworzenie komponentów 231

ROZDZIAŁ 14. TWORZENIE BIBLIOTEK ZNACZNIKÓW 234

14.1 ELEMENTY TWORZĄCE BIBLIOTEKĘ ZNACZNIKÓW 235
Klasa obsługi znacznika 235
Plik deskryptora biblioteki znaczników 236
Plik JSP 237
14.2 DEFINIOWANIE PROSTYCH ZNACZNIKÓW 238
Klasa obsługi znacznika 238
Plik deskryptora biblioteki znaczników 239
Plik JSP 240

14.3 Przypisywanie atrybutów znacznikom ... 241
Klasa obsługi znacznika ... *242*
Plik deskryptora biblioteki znaczników ... *243*
Plik JSP ... *244*
14.4 Dołączanie zawartości znacznika ... 245
Klasa obsługi znacznika ... *245*
Plik deskryptora biblioteki znaczników ... *247*
Plik JSP ... *248*
14.5 Opcjonalne dołączanie zawartości znacznika ... 249
Klasa obsługi znacznika ... *249*
Plik deskryptora biblioteki znaczników ... *250*
Plik JSP ... *251*
14.6 Manipulowanie zawartością znacznika ... 253
Klasa obsługi znacznika ... *253*
Plik deskryptora biblioteki znaczników ... *254*
Plik JSP ... *255*
14.7 Wielokrotne dołączanie lub obsługa zawartością znacznika ... 256
Klasa obsługi znacznika ... *257*
Plik deskryptora biblioteki znaczników ... *257*
Plik JSP ... *258*
14.8 Stosowanie znaczników zagnieżdżonych ... 259
Klasy obsługi znaczników ... *260*
Plik deskryptora biblioteki znaczników ... *263*
Plik JSP ... *265*

ROZDZIAŁ 15. INTEGRACJA SERWLETÓW I DOKUMENTÓW JSP ... 267

15.1 Przekazywanie żądań ... 267
Użycie zasobów statycznych ... *268*
Przekazywanie informacji do strony docelowej ... *269*
Interpretacja względnych adresów URL przez stronę docelową ... *270*
Inne sposoby pobierania obiektu RequestDispatcher ... *271*
15.2 Przykład: Internetowe biuro podróży ... 271
15.3 Dołączanie danych statycznych bądź dynamicznych ... 282
15.4 Przykład: Prezentacja nieprzetworzonych wyników zwracanych przez serwlety lub strony JSP ... 284
15.5 Przekazywanie żądań ze stron JSP ... 287

ROZDZIAŁ 16. FORMULARZE HTML ... 289

16.1 Jak przesyłane są dane z formularzy HTML ... 289
16.2 Element FORM ... 293
16.3 Tekstowe elementy kontrolne ... 297
Pola tekstowe ... *297*
VALUE ... 298
Pola hasła ... *299*
Wielowierszowe pola tekstowe ... *299*
16.4 Przyciski ... 301
Przycisk SUBMIT ... *301*
Przyciski RESET ... *303*
Przyciski JavaScript ... *304*
16.5 Pola wyboru i przyciski opcji ... 305

Pola wyboru ... *305*
Przyciski opcji ... *306*
16.6 Listy i listy rozwijane ... 307
16.7 Element kontrolny służący do przesyłania plików ... 310
16.8 Mapy odnośników obsługiwane na serwerze ... 311
IMAGE — standardowe mapy odnośników obsługiwane po stronie serwera ... *312*
ISMAP — alternatywny sposób tworzenia map odnośników obsługiwanych po stronie serwera ... *313*
16.9 Pola ukryte ... 315
16.10 Grupowanie elementów kontrolnych ... 316
16.11 Określanie kolejności poruszania się pomiędzy elementami formularzy ... 318
16.12 Testowy serwer WWW ... 319
EchoServer ... *319*
ThreadedEchoServer ... *321*
NetworkServer ... *322*

ROZDZIAŁ 17. UŻYCIE APLETÓW JAKO INTERFEJSU UŻYTKOWNIKA DLA SERWLETÓW ... 325

17.1 Przesyłanie danych metodą GET i wyświetlanie wynikowej strony WWW ... 326
17.2 Narzędzie korzystające z wielu serwisów wyszukiwawczych ... 326
17.3 Przesyłanie danych metodą GET i bezpośrednie przetwarzanie wyników (tunelowanie HTTP) ... 329
Odczyt danych binarnych lub danych ASCII ... *330*
Odczyt serializowanych struktur danych ... 331
Po stronie klienta ... 331
Po stronie serwera ... 332
17.4 Przeglądarka zapytań wykorzystująca serializację obiektów i tunelowanie ... 333
17.5 Przesyłanie danych metodą POST i bezpośrednie przetwarzanie danych (tunelowanie HTTP) ... 338
17.6 Aplet przesyłający dane metodą POST ... 340
17.7 Pomijanie serwera HTTP ... 344

ROZDZIAŁ 18. JDBC ORAZ ZARZĄDZANIE PULAMI POŁĄCZEŃ ... 346

18.1 Podstawowe etapy wykorzystania JDBC ... 346
Załadowanie sterownika ... *347*
Określenie adresu URL połączenia ... *347*
Nawiązanie połączenia ... *348*
Stworzenie polecenia ... *349*
Wykonanie zapytania ... *349*
Przetworzenie wyników ... *349*
Zamknięcie połączenia ... *350*
18.2 Prosty przykład wykorzystania JDBC ... 350
18.3 Narzędzia ułatwiające korzystanie z JDBC ... 354
18.4 Wykorzystanie narzędzi ułatwiających obsługę JDBC ... 360
18.5 Interaktywna przeglądarka zapytań ... 363
Kod przeglądarki zapytań ... *366*
18.6 Przygotowane polecenia (prekompilowane zapytania) ... 370
18.7 Zarządzanie pulami połączeń ... 373
18.8 Zarządzanie pulami połączeń: Studium zagadnienia ... 377
18.9 Współużytkowanie pul połączeń ... 382
Współużytkowanie pul połączeń przy wykorzystaniu kontekstu serwletu ... *382*

Współużytkowanie pul połączeń przy wykorzystaniu klas „singleton” *383*

DODATEK A. KRÓTKI PRZEWODNIK PO SERWLETACH I JSP **384**

A.1 PREZENTACJA SERWLETÓW I JSP 384
Zalety serwletów *384*
Zalety JSP *384*
Bezpłatnie dostępne oprogramowanie do obsługi serwletów i JSP *384*
Dokumentacja *385*
Kompilacja serwletów: informacje podawane w zmiennej środowiskowej CLASSPATH *385*
Standardowe katalogi serwera Tomcat 3.0 ..... *385*
Standardowe katalogi serwera Tomcat 3.1 ..... *385*
Standardowe katalogi serwera JSWDK 1.0.1 ..... *385*
Standardowe katalogi serwera Java Web Server 2.0 ..... *385*
A.2 PIERWSZE SERWLETY 386
Prosty serwlet *386*
Instalacja serwletów *386*
Uruchamianie serwletów *386*
Cykl życiowy serwletów *386*
A.3 OBSŁUGA ŻĄDAŃ: DANE PRZESYŁANE Z FORMULARZY 387
Odczyt parametrów *387*
Przykład serwletu *387*
Przykład formularza *388*
Filtrowanie znaków specjalnych HTML *388*
A.4 OBSŁUGA ŻĄDAŃ: NAGŁÓWKI ŻĄDAŃ HTTP 388
Metody odczytujące nagłówki żądania *388*
Inne informacje o żądaniu *389*
Najczęściej używane nagłówki żądań protokołu HTTP 1.1 ..... *389*
A.5 DOSTĘP DO STANDARDOWYCH ZMIENNYCH CGI 390
Możliwości, które nie zostały opisane gdzie indziej *390*
Odpowiedniki zmiennych CGI dostępne w serwletach *390*
A.6 GENERACJA ODPOWIEDZI: KODY STATUSU HTTP 391
Format odpowiedzi HTTP *391*
Metody określające kod statusu *391*
Kategorie kodów statusu *391*
Najczęściej wykorzystywane kody statusu protokołu HTTP 1.1 ..... *391*
A.7 GENERACJA ODPOWIEDZI: NAGŁÓWKI ODPOWIEDZI PROTOKOŁU HTTP 392
Generacja dowolnych nagłówków *392*
Generacja najczęściej używanych nagłówków *392*
Najczęściej używane nagłówki odpowiedzi protokołu HTTP 1.1 ..... *392*
Generacja obrazów GIF przez serwlety *393*
A.8 OBSŁUGA COOKIES 394
Typowe zastosowania cookies *394*
Problemy napotykane przy stosowaniu cookies *394*
Ogólny sposób użycia cookies *394*
Metod do obsługi cookies *394*
A.9 ŚLEDZENIE SESJI 395
Pobieranie informacji o sesji — getValue *395*
Kojarzenie informacji z sesją — putValue *395*
Metody interfejsu HttpSession *396*
Kodowanie adresów URL *396*

A.10 Elementy skryptowe JSP 397
Typy elementów skryptowych 397
Tekst szablonu 397
Predefiniowane zmienne 397
A.11 Dyrektywa page: określanie postaci generowanych serwletów 398
Atrybut import 398
Atrybut contentType 398
Przykład użycia atrybutu contentType 398
Przykład wykorzystania metody setContentType 398
Atrybut isThreadSafe 399
Atrybut session 399
Atrybut buffer 399
Atrybut autoflush 399
Atrybut extends 400
Atrybut info 400
Atrybut errorPage 400
Atrybut isErrorPage 400
Atrybut language 400
Zapis XML-owy 400
A.12 Dołączanie plików i apletów do dokumentów JSP 400
Dołączanie plików w czasie przekształcania strony 400
Dołączanie plików w czasie obsługi żądania 401
Aplety obsługiwane przy użyciu Java Plug-In: Prosty przypadek 401
Atrybuty znacznika jsp:plugin 401
Parametry określane w kodzie HTML: jsp:param 401
Tekst alternatywny 402
A.13 Wykorzystanie komponentów JavaBeans w dokumentach JSP 402
Podstawowe wymagania jakie należy spełnić by klasa była komponentem 402
Podstawowe sposoby użycia komponentów 402
Kojarzenie właściwości z parametrami przesłanymi w żądaniu 402
Wspólne wykorzystywanie komponentów: Atrybut scope znacznika akcji jsp:useBean 403
Warunkowe tworzenie komponentów 403
A.14 Tworzenie bibliotek znaczników 403
Klasa obsługi znacznika 403
Plik deskryptora biblioteki znaczników 404
Plik JSP 404
Przypisywanie atrybutów znacznikom 404
Dołączanie zawartości znacznika 404
Opcjonalne dołączanie zawartości znacznika 404
Przetwarzanie zawartości znacznika 405
Wielokrotne dołączanie lub przetwarzanie zawartości znacznika 405
Stosowanie zagnieżdżonych znaczników 405
A.15 Integracja serwletów i dokumentów JSP 405
Opis ogólny 405
Składnia służąca do przekazania żądania 405
Przekazywanie żądań do zwyczajnych dokumentów HTML 405
Tworzenie globalnie dostępnych komponentów JavaBeans 406
Tworzenie komponentów JavaBeans dostępnych w sesji 406
Interpretacja względnych adresów URL na stronie docelowej 406
Alternatywne sposoby pobierania obiektu RequestDispatcher (wyłącznie Java Servlet 2.2) . 406
Dołączenie danych statycznych lub dynamicznych 406

Przekazywanie żądań ze stron JSP *406*
A.16 STOSOWANIE FORMULARZY HTML 407
Element FORM *407*
Pola tekstowe *407*
Pola hasła *407*
Obszary tekstowe *407*
Przyciski SUBMIT *407*
Alternatywna postać przycisków SUBMIT *408*
Przyciski RESET *408*
Alternatywna postać przycisków RESET *408*
Przyciski JavaScript *408*
Alternatywna postać przycisków JavaScript *408*
Pola wyboru *408*
Przyciski opcji *409*
Listy rozwijane *409*
Elementy kontrolne umożliwiające przesyłanie plików na serwer *409*
Mapy odnośników obsługiwane na serwerze *409*
Pola ukryte *409*
Możliwości dostępne w Internet Explorerze *409*
A.17 WYKORZYSTANIE APLETÓW JAKO INTERFEJSU UŻYTKOWNIKA DLA SERWLETÓW 410
Przesyłanie danych metodą GET i wyświetlanie strony wynikowej *410*
Przesyłanie danych metodą GET i bezpośrednie przetwarzanie wyników (tunelowanie HTTP) *410*
Przesyłanie serializowanych danych: Kod apletu *411*
Przesyłanie serializowanych danych: Kod serwletu *411*
Przesyłanie danych metodą POST i bezpośrednie przetwarzanie wyników (tunelowanie HTTP) *412*
Pomijanie serwera HTTP *413*
A.18 JDBC I ZARZĄDZANIE PULAMI POŁĄCZEŃ Z BAZAMI DANYCH 413
Podstawowe etapy wykorzystania JDBC *413*
Narzędzia obsługi baz danych *414*
Przygotowane polecenia (prekompilowane zapytania) *415*
Etapy implementacji puli połączeń *415*

Podziękowania

Wiele osób pomagało mi podczas tworzenia tej książki. Bez ich wsparcia wciąż pisałbym jej trzeci rozdział. John Guthrie, Ammy Karlson, Rich Slywczak oraz Kim Topley dostarczyli mi cennych technicznych informacji, z których korzystałem niemal we wszystkich rozdziałach. Innymi osobami, które wskazywały popełnione błędy oraz udzielały cennych sugestii są: Don Aldridge, Camille Bell, Ben Benokraitis, Carl Burnham, Adrew Burton, Rick Cannon, Kevin Cropper, Chip Downs, Frank Erickson, Payam Fard, Daniel Goldman, Rob Gordon, Andy Gravatt, Jeff Hall, Russell Holley, David Hopkins, Lis Immer, Herman Ip, Troung Le, Frank Lewis, Tanner Lovelace, Margaret Lyell, Paul McNamee, Mike Oliver, Barb Ridenour, Himanso Sahni, Bob Samson, Ron Tosh, Tsung-Wen Tsai, Peggy Sue Vickers oraz Maureen Knox Yencha. Mam nadzieję, że dobrze wykorzystałem ich rady. Mary Lou „Eagle Eye”[1] Nohr odszukała błędnie umieszczone przecinki, dziwne wyrażenia, błędy typograficzne oraz niespójności gramatycznie. Jej praca sprawiła, że książka ta stała się znacznie lepsza. Jonnne Anzalone stworzyła końcową wersję niniejszej książki. Joanne wykonała wspaniałą pracę, niezależnie od zmian wprowadzanych przeze mnie w ostatniej chwili. Ralph Semmel dostarczył pomocnego środowiska pracy i elastycznego harmonogramu, oraz interesujących projektów serwletów i stron JSP. Greg Doench z wydawnictwa Prentice Hall od samego początku wierzył w tę książkę i zachęcał mnie do jej napisania. Rachel Borden przekonała do niej także wydawnictwo Sun Microsystems Press. Dziękuje im wszystkim.

Przede wszystkim chciałbym podziękować B.J., Lindsay oraz Nathanowi za cierpliwość dla mojego śmiesznego terminarza oraz ciągłego wykorzystania komputera gdy chcieli na nim popracować lub pograć. Bóg pobłogosławił mnie, dając mi wspaniałą rodzinę.

O autorze

Marty Hall jest starszym specjalistą komputerowym w Research and Technology Development Center w Laboratorium Fizyki Stosowanej na Uniwersytecie Johna Hopkinsa. Specjalizuje się w wykorzystaniu języka Java oraz technologiach związanych w WWW. Marty uczy także języka Java oraz programowania aplikacji WWW na Uniwersytecie Johna Hopkinsa w ramach programu kursów dokształcających, gdzie zajmuje się zagadnieniami przetwarzania rozproszonego oraz technologii internetowych. Jeśli tylko ma okazję, prowadzi także krótkie kursy poświęcone serwletom, JSP oraz innym technologiom związanym z językiem Java. Marty jest także autorem książki *Core Web Programming*, wydanej przez Wydawnictwo Prentice Hall w 1998 roku. Można się z nim skontaktować pisząc na następujący adres:

Reseach and Technology Development Center
The Johns Hopkins University Applied Phisics Laboratory
11100 Johns Hopkins Road
Laurel, MD 20723
hall@coreservlets.com

[1] „Eagle Eye” — „Orle oko”

Wprowadzenie

Na początku 1996 roku zacząłem używać języka Java w większości moich projektów programistycznych. Napisałem kilka programów CGI i w niewielkim stopniu zajmowałem się także wczesnymi wersjami serwletów, jednak w przeważającej mierze tworzyłem aplikacje działające po stronie klienta. Jednak w ciągu ostatnich kilku lat coraz większy nacisk kładziono na programy działające po stronie serwera i z tego względu poważniej zająłem się serwletami oraz technologią JavaServer Pages. Zeszłego roku zainteresowanie tymi technologiami programistów, firm programistycznych oraz twórców specyfikacji platformy języka Java, wzrosło w ogromnym stopniu. Wzrost zainteresowania tymi technologiami jest tak duży, iż w szybkim tempie stają się one standardowym narzędziem do tworzenia dynamicznych aplikacji WWW oraz internetowych programów umożliwiających korzystania z baz danych i aplikacji działających na serwerze.

Niestety, bardzo trudno było jednak znaleźć dobre, praktyczne informacje dotyczące tworzenia serwletów oraz JSP. Znalazłem trochę książek poświęconych serwletom, jednak tylko kilka z nich zawierało informacje o najnowszych wersjach specyfikacji, zaawansowanych technikach i odzwierciedlało doświadczenia nabyte podczas realizacji realnie wykorzystywanych projektów. Spośród tych kilku książek, jeśli któraś z nich w ogóle opisywała zagadnienia związane z JSP, to dotyczyły one specyfikacji JSP 1.0, nigdy JSP 1.1. Jednak w wielu sytuacjach JSP znacznie lepiej nadaje się do rozwiązania problemu niż serwlety; a zatem, cóż była by warta książka o servletach, która nie opisywałaby także JSP? W ciągu kilku ostatnich miesięcy na rynku pojawiło się nieco więcej dobrych książek poświęconych JSP. Jednak znaczna większość z nich w ogóle nie omawia serwletów. Ale czy to ma sens? Przecież integralną częścią technologii JavaServer Pages jest wykorzystanie elementów skryptowych do stworzenia kodu serwletu. A zatem, bez dogłębnej znajomości zasad działania i tworzenia serwletów, nie można efektywnie wykorzystywać JSP. Poza tym, większość działających na Internecie witryn nigdy nie wykorzystuje tylko jednej z tych technologii, lecz obie jednocześnie. I w końcu ostatnia sprawa. Podczas prowadzanie kursów w ramach dokształcających na Uniwersytecie Johna Hopkinsa zauważyłem, że bardzo niewielu spośród moich słuchaczy (których większość stanowili profesjonalni programiści) znała zagadnienia związane z protokołem HTTP 1.1, działaniem formularzy HTML oraz obsługą JDBC — czyli trzema kluczowymi technologiami pomocniczymi. Zmuszanie tych osób od kupowania książek poświęconych każdemu z tych zagadnień było bezsensowne, gdyż w tym przypadku ilość książek, którą programista musiałby kupić i przeczytać w celu tworzenia poważnych aplikacji wykorzystujących serwlety i JSP, wzrosłaby do *pięciu*.

A zatem, w połowie 1999 roku, stworzyłem krótki kurs tworzenia serwletów oraz stron JSP poparty kilkunastoma przykładami, opublikowałem go na WWW i spróbowałem przedstawić ten sam materiał na kilku spośród prowadzonych przeze mnie kursach. Reakcja była oszałamiająca. Już po kilku miesiącach opublikowany przez mnie kurs odwiedzało kilkaset osób dziennie, nie wspominając w ogóle o setkach próśb o poszerzenie zamieszczonych informacji. W końcu pogodziłem się z nieuchronnym losem i zacząłem pisać. Niniejsza książka jest efektem mej pracy. Mam nadzieję, że okaże się przydatna.

Prawdziwy kod dla prawdziwych programistów

Ta książka przeznaczona jest dla poważnych programistów. Ta książka nie wychwala potencjału internetowego handlu ani sposobów w jaki internetowe aplikacje mogą zrewolucjonizować Twoją firmę. Zamiast tego jest to praktyczna książka przeznaczona dla programistów, którzy już doskonale rozumieją konieczność tworzenia dynamicznych witryn WWW, a jej zadaniem jest pokazanie jak należy to robić w poprawny sposób. Prezentując sposoby tworzenia dynamicznych witryn WWW starałem się zilustrować najważniejsze używane techniki i ostrzec Cię przed najczęściej napotykanymi problemami. Jednocześnie wykorzystałem bardzo dużo praktycznych przykładów, na przykład — ponad sto klas Javy. Starałem się podać szczegółowe przykłady dla wszystkich najważniejszych i najczęściej wykorzystywanych możliwości, zamieścić podsumowania opisujące możliwości rzadziej wykorzystywane i wskazać (dostępne na WWW) źródła informacji o narzędziach programistycznych (API) umożliwiających zaimplementowanie możliwości najrzadziej stosowanych.

Nie jest to także książka, która pobieżnie, na wysokim poziomie, omawia wiele różnych technologii. Choć nie roszczę sobie pretensji, aby książka ta była ostatecznym źródłem informacji na wszystkie omawiane tematy (istnieje przykładowo kilka, podobnej wielkości, książek poświęconych wyłącznie JDBC), to jednak, jeśli już opisuję w niej jakieś zagadnienie, to robię to na tyle szczegółowo byś mógł rozpocząć tworzenie programów nadających się do praktycznego zastosowania. Jedynym wyjątkiem od tej reguły jest sam język Java. Oczekuję bowiem, że będziesz znał podstawy jego wykorzystania. Jeśli nie znasz Javy, to będziesz musiał sięgnąć po jakąś dobrą książkę, która nauczy Cię zasad programowania w tym języku, taką jak *„Java 1.1”* wydaną przez Wydawnictwo Helion.

Muszę Cię jednak ostrzec. Nikt nie staje się wspaniałym programistą czytając książki. Prócz lektury konieczne jest także pisanie programów. Im więcej ich stworzysz tym lepiej. W każdym rozdziale radzę, abyś zaczął od napisania krótkiego programu lub zmodyfikowania jednego z przykładów podanych wcześniej, a następnie spróbował własnych sił tworząc bardziej skomplikowany projekt. Pomiń fragmenty książki omawiające zagadnienia, których na razie nie planujesz używać i wróć do nich później, gdy będziesz gotów je wypróbować.

Jeśli będziesz postępował w ten sposób, szybko powinieneś wyrobić sobie umiejętność rozwiązywania praktycznych problemów, które były głównym powodem sięgnięcia po tę książkę. Powinieneś być w stanie określić gdzie należy użyć serwletów, gdzie lepszym rozwiązaniem będzie zastosowanie JSP, lub kiedy należy użyć kombinacji obu tych technologii. Powinieneś nie tylko być w stanie generować dokumenty HTML, lecz rozumieć jak można przekazywać informacje innych typów, na przykład obrazy GIF lub arkusze kalkulacyjne programu Excel. Powinieneś także, na tyle dobrze rozumieć protokół HTTP 1.1, aby móc wykorzystywać jego możliwości do zwiększenia efektywności działania tworzonych stron. Nie powinieneś także obawiać się tworzenia aplikacji WWW — czy to w formie formularzy HTML czy też apletów — stanowiących interfejs pozwalający użytkownikom na korzystanie z korporacyjnych baz danych. Powinieneś także być w stanie implementować skomplikowane zachowania w formie komponentów JavaBeans lub bibliotek własnych znaczników JSP i zdecydować kiedy należy użyć tych komponentów bezpośrednio, a kiedy rozpoczynać przetwarzanie żądań za pośrednictwem serwletów, które następnie wygenerują stronę prezentującą wyniki. Czytając tę książkę powinieneś także mieć sporo zabawy. A potem zasłużysz na podwyżkę.

W jaki sposób zorganizowana jest ta książka

Książka została podzielona na trzy części: Serwlety, Java Server Pages oraz Technologie pomocnicze.

Część 1.: Serwlety

Część 1. obejmuje tworzenie serwletów według specyfikacji 2.1 oraz 2.2. Choć specyfikacja 2.2 (oraz specyfikacja JSP 1.1) jest elementem Java 2 Platform, Enterprise Edition, to jednak wiele komercyjnych produktów korzysta jeszcze z wcześniejszych specyfikacji. A zatem ważne jest, aby rozumieć różnice występujące pomiędzy nimi. Poza tym, choć kod serwletów może być przenoszony i wykorzystywany na wielu różnych serwerach i systemach operacyjnych, to jednak proces instalacji i konfiguracji serwerów nie jest standaryzowany. Z tego względu podałem szczegółowe informacje dotyczące Apache Tomcata, JavaServer Web Development Kit (JSWDK) firmy Sun oraz Java Web Servera. Poniżej podałem listę omawianych zagadnień dotyczących serwletów:

- kiedy i dlaczego należy stosować serwlety,
- zdobycie i instalacja potrzebnego oprogramowania,
- podstawowa struktura serwletów,
- proces kompilacji, instalacji oraz wywoływania serwletów,
- generacja kodu HTML z poziomu serwletu,
- cykl życiowy serwletu,
- daty modyfikacji stron oraz pamięć podręczna przeglądarek,
- strategie testowania serwletów,
- obsługa żądań GET oraz POST przez jeden serwlet,
- internetowa usługa przesyłania życiorysów,
- odczytywanie nagłówków żądań HTTP w serwletach,
- przeznaczenie każdego z nagłówków żądań HTTP 1.1,
- redukcja czasu pobierania stron poprzez ich kompresję,
- ograniczanie dostępu za pomocą serwletów chronionych hasłem,
- odpowiedniki każdej ze standardowych zmiennych środowiskowych CGI,
- wykorzystanie kodów statusu HTTP,
- znaczenie każdej z wartości kodów statusu HTTP 1.1,
- interfejs użytkownika obsługujący przeszukiwanie WWW,
- określanie kodów odpowiedzi w serwletach,
- znaczenie każdego z nagłówków odpowiedzi HTTP 1.1,
- najczęściej używane typy MIME,
- serwlet wykorzystujący nagłówek Refresh w celu cyklicznego dostępu do długotrwałych obliczeń,
- serwlety wykorzystujące trwałe połączenia HTTP,
- generacja obrazów GIF z poziomu serwletów,
- przeznaczenie i problemy wiążące się z wykorzystaniem cookies,
- API do obsługi cookies,
- narzędzia ułatwiające obsługę cookies,
- konfigurowalny interfejs użytkownika do przeszukiwania WWW,
- zastosowanie śledzenia sesji,
- API do śledzenia sesji dostępne w serwletach,
- wykorzystanie sesji do stworzenia liczników odwiedzin dla poszczególnych użytkowników,
- internetowy sklep wykorzystujący śledzenie sesji, koszyki oraz dynamiczne generowanie stron na podstawie katalogu.

Część 2.: JavaServer Pages

JSP stanowi bardzo wygodną alternatywę dla serwletów, w szczególności w przypadku generacji stron, których zawartość w znacznej części nie ulega zmianie. W drugiej części książki omówię technologię JavaServer Pages w wersji 1.0 oraz 1.1. Oto lista omawianych zagadnień dotyczących JSP:

- kiedy i dlaczego należy używać JavaServer Pages,
- w jaki sposób wywoływane są strony JSP,
- stosowanie rozszerzeń JSP, skryptletów oraz deklaracji,
- predefiniowane zmienne, których można używać w wyrażeniach i skryptletach,
- dyrektywa `page`,
- określanie importowanych klas,
- określanie typu MIME dla strony,
- generacja arkuszy kalkulacyjnych programu Excel,
- kontrola modelu wątkowego,
- wykorzystanie sesji,
- określanie wielkości i działania bufora wyjściowego,
- określanie stron służących do obsługi błędów JSP,
- składnia dyrektyw zgodna z XML,
- dołączanie plików JSP w czasie gdy strona główna jest przekształcana do postaci serwletu,
- dołączanie plików HTML lub plików tekstowych w momencie przesyłania żądania,
- dołączanie apletów wykorzystujących Java Plug-In,
- wykorzystanie JavaBeans w stronach JSP,
- tworzenie i dostęp do komponentów JavaBeans,
- jawne określanie właściwości komponentów,
- kojarzenie właściwości komponentów z parametrami wejściowymi,
- automatyczna konwersja typów właściwości komponentu,
- współużytkowanie komponentów przez wiele stron JSP i serwletów,
- tworzenie bibliotek znaczników JSP,
- klasy obsługi znaczników,
- pliki opisu biblioteki znaczników,
- dyrektywa `taglib` JSP,
- proste znaczniki,
- znaczniki posiadające atrybuty,
- znaczniki posiadające zawartość pomiędzy znacznikiem otwierającym i zamykającym,
- znaczniki modyfikujące swą zawartość,
- znaczniki pętli,
- znaczniki zagnieżdżone,
- integracja serwletów oraz JSP,
- przekazywanie żądań z serwletów do zasobów statycznych i dynamicznych,
- wykorzystanie serwletów do konfiguracji komponentów JavaBeans wykorzystywanych na stronach JSP,
- internetowe biuro podróży wykorzystujące serwlety oraz JSP,
- wykorzystanie wyników wykonania stron JSP w serwletach,
- przekazywanie żądań ze stron JSP.

Część 3.: Technologie pomocnicze

W trzeciej części książki opisuję trzy zagadnienia bardzo często wykorzystywane wraz z serwletami oraz JSP — formularze HTML, aplety komunikujące się z serwletami oraz JDBC. Poniżej przedstawiłem listę zagadnień omawianych w tej części książki:

- przesyłanie danych z formularzy,
- tekstowe elementy formularzy,
- przyciski,
- pola wyboru oraz przyciski opcji,
- listy rozwijane oraz listy,
- element sterujący umożliwiający przesyłanie plików na serwer,
- mapy odnośników obsługiwane po stronie serwera,
- pola ukryte,
- grupowanie elementów formularzy,
- kolejność elementów,
- serwer WWW służący do testowania formularzy,
- przesyłanie danych z apletu żądaniem GET i wyświetlanie ich w przeglądarce,
- wysyłanie danych żądaniem GET i przetwarzanie ich przez ten sam aplet (tunelowanie HTTP),
- wykorzystanie serializacji obiektów w celu przekazywania złożonych struktur danych pomiędzy apletami i serwletami,
- wysyłanie danych żądaniem typu POST i przetwarzanie ich przez ten sam aplet,
- aplety, które nie wykorzystują serwerów WWW.

Zastosowane konwencje

W tekście niniejszej książki kody programów oraz generowane przez nie wyniki są przedstawiane czcionką o stałej szerokości. Na przykład, abstrakcyjnie omawiając programy działające po stronie serwera i wykorzystujące protokół HTTP, mogę używać takich wyrażeń jak „serwlety HTTP" lub, po prostu, „serwlety". Gdy jednak piszę `HTTPServlet`, to mam na myśli konkretną klasę Javy.

Informacje wprowadzane przez użytkownika w wierszu poleceń prezentowane są pogrubioną czcionką, natomiast generowane komunikaty mogą być bądź to ogólne (oznaczane jako `Prompt>`) bądź też wskazywać rodzaj używanego systemu operacyjnego (na przykład `DOS>`). Przykładowo, poniższy fragment tekstu oznacza, iż wykonanie polecenia „`java PewienProgram`" na dowolnej platformie systemowej spowoduje wygenerowanie wyników o postaci „`Odpowiednie wyniki`":

```
Prompt> java PewienProgram

Odpowiednie wyniki
```

Ważne, standardowe techniki są w tekście książki oznaczane w specjalny, przedstawiony poniżej, sposób:

Podstawowa metoda

Zwróć szczególną uwagę na fragmenty oznaczone jako „Podstawowa metoda". Zawierają one informacje o technikach, które powinne być stosowane zawsze lub prawie zawsze.

Notatki oraz ostrzeżenia są oznaczane w podobny sposób.

O witrynie WWW

Istnieje specjalna witryna WWW poświęcona niniejszej książce, jej adres to *http://www.coreservlet.com/*. Korzystanie z tej witryny jest bezpłatne, a można na niej znaleźć:

- kody źródłowe wszystkich przykładów podanych w niniejszej książce (można ich używać w nieograniczony sposób bez konieczności jakiejkolwiek rejestracji),
- internetową dokumentację API (w formacie Javadoc) wszystkich klas stworzonych w książce,
- aktualne adresy witryny umożliwiających pobranie oprogramowania, które można wykorzystać przy tworzeniu serwletów i stron JSP,
- informacje o zniżkach przy zakupie niniejszej książki,
- doniesienia dotyczące kursów tworzenia serwletów i stron JSP,
- informacje dotyczące nowych wydań i aktualizacji niniejszej książki.

Rozdział 1.
Podstawowe informacje o serwletach i Java Server Pages

Ten rozdział zawiera krótką prezentację serwletów oraz JavaServer Pages (JSP) i przedstawia zalety każdej z tych technologii. Podałem w nim także informacje o tym gdzie zdobyć i jak skonfigurować oprogramowanie konieczne do tworzenia serwletów oraz dokumentów JSP.

1.1 Serwlety

Serwlety to odpowiedź technologii związanych z językiem Java na programy CGI (Common Gateway Interface). Serwlety są programami wykonywanymi na serwerze WWW. Stanowią one warstwę pośrednią pomiędzy żądaniami przesyłanymi przez przeglądarkę WWW lub inny program używający protokołu HTTP oraz bazami danych bądź aplikacjami wykonywanymi na serwerze HTTP. Ich zadaniem jest:

1) **Odczytywanie wszelkich danych przesyłanych przez użytkownika.**
   Dane te są zazwyczaj wprowadzane w formularzu umieszczonym na stronie WWW, jednak mogą także pochodzić z apletu Javy lub innego programu używającego protokołu HTTP.
2) **Odszukanie wszelkich innych informacji dotyczących żądania, umieszczonych w żądaniu HTTP.**
   Informacje te zawierają szczegółowe dane dotyczące możliwości przeglądarki, cookies, nazwy komputera na którym działa program, i tak dalej.
3) **Generacja wyników.**
   Ten proces może wymagać wymiany informacji z bazą danych, wykonania wywołania RMI lub CORBA, uruchomienia aplikacji bądź bezpośredniego obliczenia wyników.
4) **Sformatowanie wyników wewnątrz dokumentu.**
   W większości przypadków sformatowanie wyników polega na umieszczeniu ich wewnątrz dokumentu HTML.
5) **Podanie odpowiednich parametrów odpowiedzi HTTP.**
   Oznacza to przekazanie do przeglądarki informacji określających typ przesyłanego dokumentu (na przykład: HTML), podanie cookies, określenia parametrów zarządzających przechowywaniem strony w pamięci podręcznej przeglądarki, itp.
6) **Wysłanie dokumentu z powrotem do użytkownika.**
   Dokument może zostać przesłany w formacie tekstowym (HTML), binarnym (obrazy GIF), a nawet w postaci skompresowanej (na przykład gzip) modyfikującej dokument zapisany w innym formacie.

Wiele nadsyłanych żądań można obsłużyć przesyłając gotowe dokumenty. Żądania tego typu są obsługiwane przez serwer bez wykonywania serwletów. Jednak w wielu innych sytuacjach statyczne wyniki nie są wystarczające, a wynikowa strona musi zostać wygenerowana osobno dla każdego żądania. Istnieje wiele przyczyn, dla których strony WWW muszą być generowane dynamicznie. Oto niektóre z nich:

- **Strona WWW generowana jest na podstawie informacji przesłanych przez użytkownika.**

  Na przykład, strony generowane przez mechanizmy wyszukiwawcze oraz strony zawierające potwierdzenia zamówień składanych w sklepach internetowych budowane są na podstawie konkretnego żądania.
- **Strona WWW jest tworzona na podstawie informacji, które często ulegają zmianie.**

  Na przykład witryny zawierające prognozy pogody lub serwisy informacyjne mogą generować strony dynamicznie lub zwracać poprzednio stworzoną stronę jeśli jeszcze jest aktualna.
- **Przy tworzeniu strony WWW wykorzystywane są informacje pochodzące z korporacyjnej bazy danych lub innych zasobów dostępnych na serwerze.**

  Na przykład, witryna służąca do handlu internetowego może wykorzystywać serwlety w celu stworzenia strony prezentującej listę dostępnych artykułów wraz z informacjami o ich cenach i możliwości zakupu.

W zasadzie wykorzystanie serwletów nie ogranicza się w cale do WWW lub serwerów aplikacji obsługujących żądania HTTP. Można ich także używać w serwerach wszelkich innych typów. Przykładowo, serwlety można umieścić w serwerze pocztowym lub serwerze FTP, rozszerzając w ten sposób ich możliwości funkcjonalne. Jednak w praktyce ten sposób wykorzystania serwletów nie zyskał popularności; dlatego też, w niniejszej książce, ograniczę się do omówienia serwletów HTTP.

1.2 Zalety serwletów w porównaniu z „tradycyjnymi” programami CGI

Serwlety są bardziej efektywne, łatwiejsze w użyciu, bezpieczniejsze i tańsze od tradycyjnych programów CGI oraz technologii zbliżonych do CGI. Poza tym, mają większe możliwości oraz zapewniają lepszą przenaszalność.

Efektywność

W przypadku wykorzystania tradycyjnej technologii CGI, dla każdego żądania HTTP tworzony jest nowy proces. Jeśli sam program CGI jest stosunkowo krótki, to przeważającą część wykonania programu może stanowić uruchomienie procesu. W przypadku serwletów, wirtualna maszyna Javy działa bez przerwy i obsługuje wszystkie nadsyłane żądania wykorzystując do tego niewielkie wątki Javy, a nie procesy systemu operacyjnego, które wykorzystują wiele zasobów systemowych. Co więcej, w przypadku korzystania z tradycyjnych programów CGI, jeśli jednocześnie zostanie nadesłanych *N* żądań skierowanych do tego samego programu, jego kod zostanie *N* razy załadowany do pamięci. W przypadku serwletów, w takiej sytuacji zostanie utworzonych *N* wątków, lecz wykorzystywana będzie wyłącznie jedna kopia klasy serwletu. I ostatnia sprawa. Gdy program CGI skończy obsługiwać żądanie, zostanie on zakończony. Utrudnia to przechowywanie wyników obliczeń w pamięci podręcznej, utrzymywanie otwartych połączeń z bazami danych oraz wykonywanie wszelkich innych optymalizacji bazujących na trwałych informacjach. Serwlety natomiast pozostają w pamięci nawet po zakończeniu obsługi żądania,

dzięki czemu przechowanie dowolnie złożonych danych pomiędzy poszczególnymi żądaniami staje się bardzo proste.

Wygoda

Serwlety dysponują rozbudowanymi narzędziami służącymi do automatycznego przetwarzania i dekodowania danych przesyłanych z formularzy HTML, odczytywania i określania nagłówków HTTP, obsługi cookies, śledzenia sesji oraz wieloma innymi narzędziami wysokiego poziomu. Poza tym znasz już język programowania Java, po co zatem miałbyś się uczyć Perla? Nie trzeba Cię także przekonywać, że dzięki użyciu języka Java oraz jego technologii można tworzyć bezpieczniejszy kod zapewniający większe możliwości wielokrotnego użycia, w porównaniu z analogicznym kodem napisanym w języku C++. Po co zatem miałbyś wracać do tworzenia programów działających na serwerze pisanych w C++?

Duże możliwości

Serwlety udostępniają kilka możliwości, których implementacja w tradycyjnych programach CGI jest wyjątkowo trudna, lub wręcz niemożliwa. Serwlety mogą bezpośrednio wymieniać informacje z serwerami WWW. Programy CGI nie dysponują taką możliwością, a przynajmniej nie bez wykorzystania specjalnego API (interfejsu programistycznego) serwera. Przykładowo, możliwość bezpośredniej komunikacji z serwerem ułatwia translację względnych adresów URL na ścieżki dostępu do konkretnych plików. Kilka serwletów może także korzystać ze wspólnych danych, co znacznie ułatwia implementację wielokrotnego wykorzystywania połączeń z bazami danych oraz innych, podobnych rozwiązań optymalizujących wykorzystanie zasobów. Serwlety mogą także przechowywać informacje pomiędzy kolejnymi żądaniami, dzięki czemu ułatwiają wykorzystanie takich technik jak śledzenie sesji oraz przechowywanie wyników poprzednich obliczeń.

Przenośność

Serwlety są pisane w języku Java i wykorzystują standardowy interfejs programistyczny. W rezultacie, serwlety napisane z myślą o, dajmy na to, I-Planet Enterprise Serverze, mogą działać, w zasadzie, w niezmienionej postaci także na serwerze Apache, Microsoft Internet Information Serverze (IIS), IBM WebSphere, czy też serwerze WebStar firmy StarNine. Na przykład, niemal wszystkie przykładowe serwlety oraz dokumenty JSP przedstawione w niniejszej książce były wykonywane na Java Web Serverze firmy Sun, serwerze Tomcat opracowanym przez fundację Apache Software oraz na JavaServer Web Development Kit firmy Sun, i to bez wprowadzania jakichkolwiek zmian w kodzie. Wiele z przykładów zostało także przetestowanych na serwerach BEA WebLogic oraz IBM WebSphere. De facto, serwlety są obsługiwane bezpośrednio lub za pośrednictwem odpowiednich plug-inów przez niemal *wszystkie* najpopularniejsze serwery WWW. Aktualnie serwlety stanowią część Java 2 Platform, Enterprise Edition (J2EE; patrz *http://java.sun.com/j2ee/*), dzięki czemu przemysłowe wsparcie serwletów stanie się jeszcze większe.

Bezpieczeństwo

Jedno z podstawowych zagrożeń istniejących w tradycyjnych programach CGI wynikało z faktu, iż programy te często były wykonywane przez powłoki systemowe ogólnego przeznaczenia. Z tego względu programiści tworzący programy CGI musieli zwracać szczególną uwagę na odnajdywanie i eliminację znaków traktowanych przez powłokę systemową w sposób specjalny,

takich znaków jak odwrotne apostrofy (`) oraz średniki (;). Zadanie to jest trudniejsze niż można przypuszczać, a problemy wynikające z tego powodu wciąż są odnajdywane w powszechnie wykorzystywanych bibliotekach CGI. Kolejnym źródłem problemów jest fakt, iż do tworzenia programów CGI używane są języki, które nie sprawdzają granic tablic i łańcuchów znaków. Na przykład w języku C lub C++ dopuszczalne jest przydzielenie pamięci dla 100-elementowej tablicy, a następnie określenie wartości jej 999-go elementu, który w rzeczywistości zostanie zapisany w bliżej nie określonym miejscu pamięci programu. A zatem, programiści, którzy zapomną o samodzielny sprawdzaniu zakresów tablic i łańcuchów znaków, narażają swój system na celowe bądź przypadkowe ataki typu przepełnienie buforu. W przypadku stosowania serwletów zagrożenia tego typu nie występują. Nawet jeśli serwlet wykona zdalne wywołanie systemowe w celu wykonania jakiegoś programu w lokalnym systemie operacyjnym, to i tak do tego celu używana powłoka systemowa. Jeśli zaś chodzi o sprawdzanie zakresów i inne mechanizmy ochrony pamięci, to stanowią one integralną część języka Java.

Niewielkie koszty

Dostępnych jest wiele darmowych lub bardzo tanich serwerów WWW doskonale nadających się do użytku „osobistego" lub do obsługi witryn o niewielkim natężeniu ruchu. Jednak za wyjątkiem serwera Apache, który jest dostępny bezpłatnie, większość innych serwerów WWW o komercyjnej jakości jest dosyć droga. Niemniej jednak, gdy już zdobędziesz serwer WWW to niezależnie od jego ceny dodanie do niego narzędzi obsługi serwletów (jeśli sam serwer nie jest w nie wyposażony) będzie stanowiło znikomy wydatek. Znacznie odróżnia to technologie wykorzystujące serwlety od alternatywnych rozwiązań CGI, w przypadku których konieczny jest zakup drogich bibliotek lub pakietów programistycznych.

1.3 Java Server Pages

Technologia Java Server Pages (w skrócie JSP) pozwala na mieszanie zwykłego, statycznego kodu HTML z informacjami generowanymi dynamicznie przez serwlety. Wiele stron tworzonych przez programistów WWW to statyczne dokumenty, w których zawartość generowana dynamicznie ogranicza się do zaledwie kilku niewielkich miejsc. Na przykład, początkowa strona większości sklepów internetowych jest identyczna dla wszystkich użytkowników, za wyjątkiem niewielkiej wiadomości powitalnej zawierające imię danego użytkownika (jeśli system je zna). Z kolei wiele wariacji technologii CGI, w tym także serwlety, zmusza do generacji całej strony WWW z poziomu programu, nawet jeśli znaczna jej część w ogóle nie jest zmieniana. JSP pozwala na rozdzielenie obu fragmentów stron. Przykład takiego rozdzielenia przedstawiłem na listingu 1.1. Znaczna część przedstawionej na nim strony to zwyczajny kod HTML, który jest przekazywany do przeglądarki użytkownika bez jakichkolwiek modyfikacji. Natomiast dynamicznie generowane fragmenty zostały oznaczone przy użyciu specjalnych znaczników przypominających znaczniki HTML i umieszczonych bezpośrednio w kodzie strony.

Listing 1.1 Przykładowa strona JSP

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<HTML>
<HEAD><TITLE>Witamy w naszym sklepie</TITLE></HEAD>
<BODY>
<H1>Witamy w naszym sklepie</H1>
<SMALL>Witamy,
<!-- W przypadku użytkowników odwiedzających sklep po raz
pierwszy nazwa użytkownika ma postać "Nowy użytkownik" -->
<%= Utils.getUserNameFromCookie(request) %>.
Dostęp do <A HREF="Account-Settings.html">ustawień Twojego konta</A>
</SMALL>
```

```
Cała reszta strony to zwyczajny kod HTML.
</BODY>
</HTML>
```

1.4 Zalety JSP

JSP ma wiele zalet w porównaniu z rozwiązaniami alternatywnymi. Poniżej przedstawiłem kilka z nich.

W porównaniu z Active Server Pages (ASP)

ASP jest konkurencyjną technologią opracowaną przez firmę Microsoft. Zalety JSP w porównaniu z ASP są dwojakiego rodzaju. Po pierwsze, dynamiczne fragmenty dokumentów JSP są pisane w języku Java, a nie w języku VBScript bądź innym języku, który można wykorzystywać w technologii ASP. A zatem dokumenty JSP mają większe możliwości i lepiej nadają się do tworzenia skomplikowanych aplikacji wymagających wielokrotnego wykorzystywania kodu. Po drugie, dokumenty JSP można przenosić na inne systemy operacyjne i serwery WWW, dzięki czemu nie trzeba wykonywać ich wyłącznie w środowisku Windows NT/2000 oraz na serwerach IIS. Dokładnie te same zalety można podać porównując JSP z ColdFusion — tworząc dokumenty JSP używamy języka Java i nie musimy korzystać z żadnego konkretnego serwera.

W porównaniu z PHP

PHP jest darmową technologią o ogólnie dostępnym kodzie źródłowym. PHP to język skryptowy przypominający nieco ASP i JSP, a pisane w nim programy umieszczane są bezpośrednio w kodzie dokumentów HTML. Zaletą JSP w porównaniu z PHP jest tworzenie dynamicznych części dokumentu przy użyciu języka Java, który już prawdopodobnie znasz i który dysponuje rozbudowanymi narzędziami programistycznymi do komunikacji sieciowej, wykorzystania baz danych, obsługi obiektów rozproszonych, itp. Natomiast wykorzystanie PHP wiąże się z koniecznością nauki nowego języka programowania.

W porównaniu z serwletami

W zasadzie absolutnie wszystko co można zrobić przy użyciu JSP można także zaimplementować za pomocą serwletów. W rzeczywistości, w sposób całkowicie niewidoczny, dokumenty JSP są automatycznie tłumaczone do postaci serwletów. Jednak znacznie wygodniej jest tworzyć (i modyfikować) kod HTML niż tysiące wywołań metody `println` generujących ten sam kod. Co więcej, oddzielając zagadnienia prezentacji od treści, można wykorzystać różne osoby do realizacji różnych zadań — projektanci stron WWW mogą tworzyć dokumenty HTML przy użyciu swoich ulubionych narzędzi, zostawiając w nich miejsca w których programiści serwletów umieszczą dynamicznie wygenerowane informacje.

W porównaniu z Server-Side Includes (SSI)

SSI jest powszechnie wykorzystywaną technologią, służącą do wstawiania zewnętrznych elementów do statycznych stron WWW. Technologia JSP jest znacznie lepsza od SSI, gdyż udostępnia znacznie szerszy zbiór narzędzi do tworzenia tych zewnętrznych elementów oraz większą ilość możliwości określania na jakim etapie przetwarzania żądania HTTP elementy te mają być wstawione. Poza tym SSI było tworzone z myślą o prostym wstawianiu treści jednego

dokumentu do innego, a nie do tworzenia „prawdziwych" programów wykorzystujących dane podawane w formularzach, połączenia z bazami danych, itd.

W porównaniu z językiem JavaScript

Język JavaScript, który ma bardzo niewiele wspólnego z Javą, jest zazwyczaj używany do dynamicznego generowania kodu HTML po stronie *klienta* — czyli tworzenia elementów stron WWW w momencie gdy przeglądarka ładuje stronę. To bardzo przydatna możliwość, jednak można z niej korzystać wyłącznie w sytuacjach, gdy dynamicznie generowane informacje bazują na środowisku klienta. Za wyjątkiem cookies, informacje przekazywane w nagłówkach HTTP nie są dostępne dla procedur JavaScriptu implementowanych w programach „klienckich". Język JavaScript nie został także wyposażony w procedury komunikacji sieciowej, a zatem pisane w nim skrypty nie są w stanie korzystać z zasobów dostępnych na serwerze, takich jak bazy danych, katalogi, informacje o cenach, itp. Skrypty pisane w języku JavaScript mogą być także wykonywane na serwerze — przede wszystkim chodzi tu o serwery firmy Netscape oraz o język skryptowy serwera IIS. Język Java ma jednak znacznie większe możliwości, jest bardziej elastyczny, bezpieczny i zapewnia większe możliwości przenoszenia kodu.

W porównaniu ze statycznym kodem HTML

Zwyczajne kod HTML nie może zawierać informacji dynamicznych, a zatem dokumenty HTML nie mogą bazować na informacjach podawanych przez użytkowników ani na informacjach pobieranych z zasobów dostępnych na serwerze. Technologia JSP jest tak prosta i wygodna, iż sensownym rozwiązaniem jest wzbogacenie o elementy dynamiczne wszelkich dokumentów HTML, nawet tych, które niewiele na tym zyskają. Uprzednio stopień złożoności dostępu do danych dynamicznych przekreślał możliwości ich wykorzystania we wszystkich przypadkach, za wyjątkiem tych najistotniejszych.

1.5 Instalacja i konfiguracja

Zanim zaczniesz, będziesz musiał skopiować konieczne oprogramowanie i zainstalować je na swoim komputerze. Poniżej przedstawiłem opis czynności jakie należy w tym celu wykonać. Zwróć jednak uwagę, iż choć kody serwletów, które będziesz pisał, będą zgodne ze standardowym API, to jednak nie istnieje żaden standard określający zasady pobierania i konfiguracji serwerów WWW oraz serwerów aplikacji. Z tego względu, w odróżnieniu od większości pozostałych części książki, zamieszczone tu informacje będą się zmieniać w zależności od instalowanego serwera, a przedstawione przykłady powinne być traktowane wyłącznie jako ogólne demonstracje. Szczegółowych instrukcji należy szukać w dokumentacji instalowanego serwera.

Zdobywanie oprogramowania do obsługi serwletów i dokumentów JSP

Pierwszym krokiem powinno być skopiowanie oprogramowania implementującego specyfikacje Java Servlet 2.1 lub 2.2 oraz Java Server Pages 1.0 lub 1.1. Jeśli korzystasz z nowoczesnego serwera aplikacji lub serwera WWW to istnieje duże prawdopodobieństwo, że dysponujesz już wszelkimi koniecznymi narzędziami. Przejrzyj dokumentację używanego serwera lub aktualną listę serwerów obsługujących serwlety (można ją znaleźć pod adresem *http://java.sun.com/products/servlet/industry.html*). Choć docelowo niewątpliwie będziesz chciał wdrażać swoje aplikacje na serwerze o jakości komercyjnej, to jednak początkowo, do nauki, warto

wykorzystać darmowy system, który można zainstalować na własnym komputerze do celów tworzenia i testowania aplikacji. Oto kilka najpopularniejszych rozwiązań:

- **Tomcat fundacji Apache Software Foundation**.

  Tomcat jest oficjalną implementacją wzorcową specyfikacji Java Servlet 2.2 oraz JSP 1.1. Można go używać jako niewielkiego samodzielnego serwera wystarczającego do testowania serwletów i dokumentów JSP; bądź zintegrować z serwerem WWW Apache[2]. Wiele firm tworzących inne serwery zapowiedziało, że w przyszłości ich produkty będą obsługiwać te specyfikacje, więc właśnie je omówię szczegółowo w niniejszej książce. Tomcat, podobnie jak serwer WWW Apache, jest dostępny bezpłatnie. Instalacja i konfiguracja Tomcata wymaga znacznie więcej wysiłku niż wykorzystanie komercyjnych narzędzi do obsługi serwletów, to samo z resztą dotyczy serwera Apache (który jest bardzo szybki i niezawodny, lecz nieco trudny do zainstalowania i konfiguracji). Wszelkie informacje na temat serwera Tomcat można znaleźć pod adresem *http://jakarta.apache.org/*.

- **JavaServer Web Development Kit (JSWDK)**.

  JSWDK jest oficjalną implementacją wzorcową specyfikacji Java Servlet 2.1 oraz JavaServer Pages 1.0. Jest on używany jako niewielki, niezależny serwer przeznaczony do testowania serwletów i dokumentów JSP zanim zostaną opublikowane i uruchomione na docelowym serwerze WWW obsługującym obie te technologie. JSWDK jest bezpłatny i godny zaufania, lecz jego instalacja i konfiguracja może nastręczyć nieco problemów. Więcej informacji na temat tego produktu można znaleźć pod adresem *http://java.sun.com/products/servlet/archive.html*.

- **JRun firmy Allaire**.

  JRun jest mechanizmem obsługi serwletów oraz dokumentów JSP, który można zintegrować z serwerami Netscape Enterprise oraz Netscape Fast Track, IIS, Microsoft PWS, starszymi wersjami serwera Apache oraz serwerami WebSite firmy O'Reilly i WebSTAR firmy StarNine. Bezpłatnie dostępna jest ograniczona wersja produktu, będąca w stanie obsługiwać do pięciu równoczesnych połączeń. Ograniczeń tych nie ma wersja komercyjna, wyposażona w dodatkowe narzędzia takie jak konsola do zdalnego administrowania. Więcej informacji na temat tego produktu można znaleźć pod adresem *http://www.allaire.com/products/jrun/*.

- **ServletExec firmy New Atlanta Communications**.

  ServletExec to mechanizm obsługujący serwlety oraz dokumenty JSP, który można zintegrować z większością najbardziej popularnych serwerów WWW działających w systemach Solaris, Windows, MacOS, HP-UX oraz Linux. Można go skopiować i używać bezpłatnie, jednak wiele bardziej zaawansowanych możliwości zostanie uaktywnionych dopiero po zakupieniu licencji. Więcej informacji na temat tego produktu można znaleźć pod adresem *http://newatlanta.com/*.

- **LiteWebServer (LWS) firmy Gefion Software**.

  LWS jest niewielkim, bezpłatnie dostępnym serwerem WWW bazującym na Tomcat-cie, który obsługuje specyfikacje Java Servlet 2.2 i JSP 1.1. Firma Gefion dysponuje także bezpłatnym plug-inem o nazwie WAICoolRunner przeznaczonym dla serwerów Netscape FastTrack oraz Enterprise, implementującym specyfikacje Java Servlet 2.2 oraz JSP 1.1. Więcej informacji na temat tych produktów można znaleźć pod adresem *http://www.gefionsoftware.com/*.

- **Java Web Server firmy Sun**[3].

---

[2] Można także zintegrować Tomcata z Internet Information Serverem firmy Microsoft oraz z serwerami firmy Netscape.

[3] Od maj 2001 roku firm Sun przestała udostępniać Java Web Server. (przyp. red.)

Ten serwer został w całości napisany w języku Java i był jednym z pierwszych serwerów WWW, które w pełni obsługiwały specyfikacje Java Servlet 2.1 oraz JSP 1.0. Choć serwer ten nie jest już aktywnie modernizowany, gdyż firma Sun koncentruje swe wysiłki na serwerze Netscape/I-Planet, to wciąż jednak jest on często wykorzystywany do nauki tworzenia serwletów oraz stron JSP. Bezpłatną, testową wersję serwera (z ograniczeniem czasowym) można znaleźć pod adresem *http://www.sun.com/software/jwebserver/try/*. Informacje na temat bezpłatnej wersji serwera, której działanie nie jest ograniczone czasowo, dostępnej dla instytucji oświatowych i przeznaczonej do celów dydaktycznych można znaleźć pod adresem *http://freeware.thesphere.com/*.

Zapamiętaj adres lub zainstaluj dokumentację Java Servlet oraz JSP API

Żaden poważny programista nie powinien zabierać się za tworzenie ogólnych aplikacji w języku Java bez dostępu do dokumentacji JDK 1.1 lub 1.2 API. Podobnie żaden poważny programista nie powinien zabierać się za tworzenie serwletów i dokumentów JSP bez dostępu do dokumentacji klas dostępnych w pakiecie `javax.servlet`. Poniżej przedstawiłem informacje o tym gdzie można znaleźć tę dokumentację:

- *http://java.sun.com/products/jsp/download.html*

  Z tej strony możesz skopiować na swój lokalny komputer bądź to dokumentację API 2.1/1.0 bądź API 2.2/1.1. Być może będziesz musiał skopiować całą implementację wzorcową i z niej pobrać dokumentację.
- *http://java.sun.com/products/servlet/2.2/javadoc/*

  Ta strona pozwala na przeglądanie dokumentacji Java Servlet API 2.2 dostępnej na WWW.
- *http://java.sun.com/j2ee/j2sdkee/techdocs/api/*

  Powyższa strona pozwala na przeglądanie pełnej dokumentacji API dla Java 2 Platform, Enterprise Edition (J2EE), która zawiera także dokumentację pakietów Java Servlet 2.2 oraz JSP 1.1.

Jeśli firma Sun lub fundacja Apache udostępni na WWW jakiekolwiek inne materiały (na przykład dokumentację API 2.1/1.0) to adresy odpowiednich stron zostaną podane na stronie „Chapter 1: Overviewe of Servlets and JavaServer Pages", w dziale „Source Code Archive" witryny *http://www.coreservlets.com/*.

Wskaż klasy używane przez kompilator Javy

Gdy już zdobędziesz potrzebne oprogramowanie, będziesz musiał poinformować kompilator Javy (program *javac*) gdzie może znaleźć pliki klasowe serwletów oraz JSP, których będzie potrzebował podczas kompilacji serwletów. Wszelkie konieczne informacje powinieneś znaleźć w dokumentacji używanego pakietu. Jednak zazwyczaj pliki klasowe, które będą Ci potrzebne są umieszczane w katalogu *lib* wewnątrz instalacyjnego folderu serwera. Pliki klasowe serwletów są zazwyczaj umieszczane w pliku *servlet.jar*, a pliki klasowe JSP w pliku *jsp.jar*, *jspengine.jar* bądź *jasper.jar*. Istnieje kilka różnych sposobów poinformowania kompilatora Javy o położeniu tych plików; przy czym najprostszą z nich jest skopiowanie odpowiednich plików JAR do jednego z folderów podanych w zmiennej środowiskowej `CLASSPATH`. Jeśli do tej pory nie słyszałeś nic o tej zmiennej środowiskowej, to powinieneś wiedzieć, iż jest to zmienna określająca foldery w których program *javac* będzie szukał klas podczas kompilacji programów napisanych w języku Java. Jeśli zmienna ta nie została określona, program będzie szukał klas w aktualnym folderze oraz w

standardowych folderach systemowych. Jeśli samemu określasz wartość tej zmiennej systemowej, to nie zapomnij dołączyć do niej znaku kropki (.), oznaczającego aktualny folder.

Poniżej przedstawiłem krótkie informacje o tym, jak należy określać zmienne środowiskowe w dwóch platformach systemowych. Przyjmij, że *dir* to nazwa katalogu w którym zostały umieszczone pliki klasowe serwletów oraz JSP.

Unix (C Shell)

```
setenv CLASSPATH .:dir/servlet.jar:dir/jspengine.jar
```

Jeśli wartość zmiennej środowiskowej CLASSPATH została już wcześniej określona, a teraz chcesz dodać do niej nowe katalogi (nie usuwają przy tym jej aktualnej zawartości), to na końcu polecenia setenv dopisz wyrażenie :$CLASSPATH. Zwróć uwagę, iż w systemach Unix nazwy poszczególnych katalogów w ścieżce są od siebie oddzielone znakami ukośnika, natomiast poszczególne ścieżki — znakami dwukropka. W systemach Windows do tych samych celów używane są znaki odwrotnego ukośnika oraz średnika. Aby zmiany w wartości zmiennej środowiskowej CLASSPATH były trwałe, powinieneś umieścić powyższe polecenie w pliku *.cshrc*.

Windows

```
set CLASSPATH=.;dir\servlet.jar;dir\jspengine.jar
```

Jeśli wartość zmiennej środowiskowej CLASSPATH została już wcześniej określona, a teraz chcesz dodać do niej nowe katalogi, to na końcu polecenia dopisz wyrażenie ;%CLASSPATH%. Zwróć uwagę, iż w systemach Windows nazwy poszczególnych katalogów w ścieżce są od siebie oddzielone znakami odwrotnego ukośnika, natomiast poszczególne ścieżki — znakami średnika. W systemach Unix do tych samych celów używane są znaki ukośnika oraz dwukropka. Aby powyższe zmiany zostały wprowadzone na stałe, w systemach Windows 95/98 należy dopisać powyższe polecenie do pliku *autoexec.bat*. W systemie Windows NT należy wybrać z menu opcję *Start⇨Settings⇨Control Panel*, dwukrotnie kliknąć ikonę *System*, przejść na zakładkę *Environment* i dodać nazwę zmiennej środowiskowej oraz jej wartość. W systemie Windows 2000 należy wybrać opcje *Start⇨Ustawienia⇨Panel sterowania*, dwukrotnie kliknąć ikonę *System*, następnie przejść na zakładkę *Zaawansowane*, kliknąć przycisk *Zmienne środowiskowe* i dodać nazwę zmiennej oraz jej wartość.

Umieść klasy w pakietach

W następnym rozdziale przekonasz się, że tworząc serwlety będziesz zazwyczaj chciał umieszczać je w pakietach. W ten sposób będziesz mógł uniknąć konfliktów nazw pomiędzy Twoimi serwletami a serwletami pisanymi przez inne osoby i wykorzystywanymi na tym samym serwerze WWW bądź serwerze aplikacji. W takim przypadku wygodnie jest dodać ścieżkę dostępu do głównego katalogu hierarchii katalogów reprezentujących pakiety, do zmiennej środowiskowej CLASSPATH. Więcej szczegółowych informacji na ten temat znajdziesz w podrozdziale 2.4, pt.: „Umieszczanie serwletów w pakietach”.

Skonfiguruj serwer

Zanim uruchomisz serwer, będziesz zapewne chciał określić wartości pewnych parametrów, takich jak numer portu na którym serwer będzie oczekiwał na żądania, nazwy katalogów w których będzie szukał plików HTML, itp. Proces ten jest zależy wyłącznie od używanego serwera i w przypadku serwerów komercyjnych powinien być dokładnie opisany w dokumentacji, w części poświęconej instalacji serwera. Niemniej jednak w przypadku niewielkich serwerów fundacji Apache lub firmy Sun dostarczanych jako przykładowe implementacje specyfikacji Java Servlet 2.2 i JSP 1.1 (Tomcat) lub JavaSevlet 2.1 i JSP 1.0 (Sun JSWDK), dostępnych jest kilka ważnych lecz słabo udokumentowanych opcji konfiguracyjnych, które opiszę w kolejnych częściach rozdziału.

Numer portu

W celu uniknięcia konfliktów z istniejącymi serwerami WWW, zarówno Tomcat jak i JSWDK używają niestandardowych portów. Jeśli używasz jednego z tych produktów do wstępnego tworzenia i testowania serwletów i stron JSP, a nie masz innego serwera WWW, to zapewne dojdziesz do wniosku, że wygodniej będzie używać standardowego portu HTTP o numerze 80. W przypadku Tomcata 3.0 numer portu można zmienić edytując plik *katalog_instalacyjny/server.xml*[4]. W pliku tym należy odszukać poniższy wiersz i zamienić w nim liczbę 8080 na 80:

```
<ContextManager port="8080" hostName="" inet="">
```

W przypadku JSWDK 1.0.1 należy zmodyfikować plik *katalog_instalacyjny/webserver.xml*. W pliku tym należy odszukać poniższy wiersz i zmienić w nim liczbę 8080 na 80:

```
port NMTOKEN "8080"
```

Także Java Web Server 2.0 używa portu o niestandardowym numerze. Aby go zmienić należy skorzystać z interfejsu do zdalnej administracji serwerem. W tym celu w przeglądarce należy wyświetlić stronę o adresie *http://adres_komputera:9090/*, gdzie *adres_komputera* to prawdziwa nazwa komputera na jaki działa serwer, lub „localhost" w przypadku gdy serwer działa na lokalnym komputerze.

Zmienna środowiskowa JAVA_HOME

Jeśli wraz z Tomcatem lub JSWDK używasz JDK 1.2 lub 1.3, to będziesz musiał określić wartość zmiennej środowiskowej `JAVA_HOME` i podać w niej nazwę katalogu instalacyjnego JDK. W przypadku JDK 1.1 określanie tej zmiennej środowiskowej nie jest konieczne. Najprostszym sposobem podania wartości zmiennej środowiskowej `JAVA_HOME` jest dodanie odpowiedniego polecenia do skryptu *startup* (w przypadku Tomcata) lub *startserver* (w przypadku JSWDK). Poniżej przedstawiłem przykładową postać pierwszych dwóch wierszy używanych przeze mnie plików *startup.bat* oraz *startserver.bat*:

```
rem Marty Hall: dodana zmienna JAVA_HOME
set JAVA_HOME=C:\jdk1.2.2
```

Ustawienia pamięci systemu DOS

Jeśli uruchamiasz Tomcata lub JSWDK w systemach Windows 95 lub Windows 98 to zapewne będziesz musiał zmodyfikować wielkość pamięci przydzielanej dla zmiennych środowiskowych programów MS-DOS. W tym celu należy uruchomić nowe okno trybu MS-DOS, kliknąć ikonę znajdującą się w jego lewym, górnym wierzchołku i z wyświetlonego menu wybrać opcję *Właściwości*. Następnie należy przejść na zakładkę *Pamięć* i z listy rozwijanej *Środowisko pierwotne* wybrać wartość *2816*. Tę modyfikację wystarczy wykonać tylko raz.

Ustawienie CR/LF w serwerze Tomcat 3.0

W pierwszych wersjach Tomcata występował pewien poważny problem — pliki tekstowe były zapisane w formacie uniksowym (końce wierszy były oznaczane znakami przewinięcia wiersza) a nie w formacie systemu Windows (w którym końce wiersza oznaczane są znakami powrotu karetki/przewinięcia wiersza). W efekcie skrypty uruchamiające i zamykające serwer nie działały poprawnie. Bardzo łatwo możesz sprawdzić czy posiadane przez Ciebie wersja Tomcata będzie stwarzać podobne problemy — wystarczy otworzyć plik *katalog_instalacyjny/startup.bat* w Notatniku. Jeśli zawartość pliku zostanie wyświetlona jako jeden, niezrozumiały, długi wiersz to bezzwłocznie zamknij Notatnik, a następnie otwórz poniższe pliku w programie WordPad (nie Notatnik) i od razu je zapisz:

- *katalog_instalacyjny/startup.bat*,

[4] Poczynając do serwera Tomcat 3.1, wszystkie pliki konfiguracyjne znajdują się w katalogu *katalog_instalacyjny/conf/*. (przyp. tłum.)

- *katalog_instalacyjny/tomcat.bat*,
- *katalog_instalacyjny/shutdown.bat*,
- *katalog_instalacyjny/tomcatEnv.bat*,
- *katalog_instalacyjny/webpages/WEB-INF/web.xml*,
- *katalog_instalacyjny/examples/WEB-INF/web.xml*.

Uruchomienie serwera

Aby uruchomić jeden z „prawdziwych" serwerów WWW, będziesz musiał przejrzeć jego dokumentację i dowiedzieć się jak należy to zrobić. W wielu przypadkach wymaga to wykonania programu *httpd* bądź to z poziomu wiersza poleceń, bądź też poprzez poinformowanie systemu operacyjnego, że należy ten program wykonać automatycznie podczas uruchamiania systemu.

W przypadku Tomcata 3.0 serwer uruchamia się poprzez wykonanie skryptu *startup* umieszczonego w głównym katalogu instalacyjnym. W przypadku JSWDK 1.0.1 należy uruchomić podobny skrypt o nazwie *startserver*.

Kompilacja i instalacja własnych serwletów

Kiedy już poprawnie określisz wartość zmiennej środowiskowej `CLASSPATH`, zgodnie z informacjami podanymi we wcześniejszej części tego rozdziału, to aby skompilować serwlet wystarczy wydać polecenie `javac NazwaServletu.java`. Wynikowy plik klasowy należy umieścić w odpowiednim miejscu, w którym serwer będzie szukał serwletu podczas próby jego wykonania. Zgodnie z tym czego się mogłeś spodziewać, miejsce w którym należy umieszczać pliki klasowe serwletów zależy od używanego serwera. Poniżej podane zostały katalogi wykorzystywane do przechowywania plików klasowych serwletów w najnowszych wersjach Tomcata, JSWDK oraz Java Web Servera. We wszystkich przypadkach *katalog_instalacyjny*, to główny katalog, w którym serwer został zainstalowany.

Tomcat

- *katalog_instalacyjny/webpages/WEB-INF/classes*
  Standardowe miejsce służące do przechowywania plików klasowych serwletów.
- *katalog_instalacyjny/classes*
  Alternatywne miejsce w którym można umieszczać pliki klasowe serwletów.
- *katalog_instalacyjny/lib*
  Katalog służący do przechowywania plików JAR zawierających pliki klasowe serwletów.

Tomcat 3.1

Tuż przed opublikowaniem niniejszej książki fundacja Apache udostępniła wersję beta serwera Tomcat 3.1[5]. Jeśli w momencie gdy będziesz kopiował Tomcata będzie dostępna końcowa wersja tego serwera, to właśnie jej powinieneś użyć. Poniżej przedstawiłem nową strukturę katalogów używaną w serwerze Tomcat 3.1:

- *katalog_instalacyjny/webapps/ROOT/WEB-INF/classes*
  Standardowe miejsce służące do przechowywania plików klasowych serwletów.
- *katalog_instalacyjny/classes*
  Alternatywne miejsce w którym można umieszczać pliki klasowe serwletów.

[5] W momencie oddawania do druku tłumaczenia niniejszej książki najnowsza wersja serwera Tomcat miała numer 4.0; w porównaniu z wersją 3.1 struktura podstawowych katalogów serwera nie uległa zmianie. (przyp. tłum.)

- *katalog_instalacyjny/lib*
  Katalog służący do przechowywania plików JAR zawierających pliki klasowe serwletów.

JSWDK

- *katalog_instalacyjny/webpages/WEB-INF/servlets*
  Standardowe miejsce służące do przechowywania plików klasowych serwletów.
- *katalog_instalacyjny/classes*
  Alternatywne miejsce w którym można umieszczać pliki klasowe serwletów.
- *katalog_instalacyjny/lib*
  Katalog służący do przechowywania plików JAR zawierających pliki klasowe serwletów.

Java Web Server 2.0

- *katalog_instalacyjny/servlets*
  Katalog w którym należy umieszczać pliki klasowe serwletów, które często ulegają zmianom. Serwer automatycznie wykrywa kiedy serwlety umieszczone w tym katalogu zostaną zmienione i w razie konieczności ponownie je załaduje do pamięci. Odróżnia to Java Web Server od Tomcata oraz JSWDK, gdyż w ich przypadku, w razie zmiany serwletu znajdującego się w pamięci serwera, należy ponownie uruchomić serwer.
- *katalog_instalacyjny/classes*
  Katalog w którym należy umieszczać pliki klasowe serwletów, które nie zmieniają się zbyt często.
- *katalog_instalacyjny/lib*
  Katalog służący do przechowywania plików JAR zawierających pliki klasowe serwletów.

Zdaję sobie sprawę, iż wszystkie te informacje mogą Cię przerażać. Ale nie martw się, w następnym rozdziale, gdy przedstawię kody kilku serwletów, zademonstruję także proces ich uruchamiania na kilku różnych serwerach.

Rozdział 2.
Pierwsze serwlety

W poprzednim rozdziale pokazałem jak zainstalować oprogramowanie, którego będziesz potrzebował oraz jak skonfigurować środowisko programistyczne. Teraz zapewne, chcesz już napisać kilka pierwszych serwletów. W porządku. W tym rozdziale dowiesz się jak można to zrobić, poznasz także strukturę niemal wszystkich serwletów, czynności jakie należy wykonać aby skompilować i wykonać serwlet oraz znajdziesz szczegółowe informacje na temat sposobu inicjalizacji serwletów oraz momentów w jakich są wywoływane ich poszczególne metody. W rozdziale tym przedstawię także kilka ogólnych narzędzi, które mogą Ci się przydać przy pisaniu własnych serwletów.

2.1 Podstawowa struktura serwletów

Na listingu 2.1 przedstawiony został prosty serwlet obsługujący żądania `GET`. Osoby, które nie znają protokołu HTTP powinne wiedzieć, iż są to standardowe żądania używane przez przeglądarki w celu pobierania stron WWW. Przeglądarki generują te żądania gdy użytkownik poda adres strony WWW na pasku adresu, kliknie połączenie umieszczone na oglądanej stronie lub prześle formularz, w którym nie określono atrybutu `METHOD` znacznika `<FORM>`. Serwlety równie łatwo mogą obsługiwać żądania `POST`, generowane w przypadkach gdy użytkownik prześle formularz HTML, w którym atrybutowi `METHOD` znacznika `<FORM>` przypisano wartość `"POST"` (`METHOD="POST"`). Szczegółowe informacje dotyczące formularzy HTML znajdziesz w rozdziale 16.

Aby klasa była serwletem musi ona być klasą potomną klasy `HttpServlet` i przesłaniać metodę `doGet` lub `doPost`, w zależności od tego czy dane są przesyłane do serwletu przy użyciu żądań `GET` czy też `POST`. Jeśli chcesz, aby ten sam serwlet obsługiwał zarówno żądania `GET` jak i `POST`, i w obu przypadkach wykonywał te same czynności, to wystarczy że zaimplementujesz w nim zarówno metodę `doGet` jak i `doPost`, przy czym jedna z nich będzie wywoływać drugą.

Listing 2.1 ServletTemplate.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletTemplate extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
              throws ServletException, IOException {

    // zmiennej "request" (żądanie) należy używać do odczytywania
    // nagłówków HTTP przekazanych przez przeglądarkę (na przykład
    // cookies) oraz danych podawanych w formularzach HTML (czyli
    // danych wpisanych i przesłanych przez użytkownika).

    // zmiennej "response" (odpowiedź) należy używać do określania
    // kodu statusu odpowiedzi HTTP oraz nagłówków odpowiedzi (na przykład
    // typu danych oraz cookies).

    PrintWriter out = response.getWriter();
    // zmiennej "out" możesz używać do przesyłania kodu HTML
```

```
    // do przeglądarki
  }
}
```

Obie te metody pobierają dwa argumenty — `HttpServletRequest` oraz `HttpSevletResponse`. Interfejs `HttpServletRequest` udostępnia metody przy użyciu których można zdobyć informacje o danych przekazanych z przeglądarki, takich jak informacje podane przez użytkownika w formularzu, nagłówki żądania HTTP, czy też nazwa komputera użytkownika. Interfejs `HttpServletResponse` pozwala na określenie informacji które zostaną przesłane do przeglądarki, takich jak kod statusu HTTP (200, 404, itd.), nagłówki odpowiedzi (`Content-type`, `Set-Cookie`, itp.), a co ważniejsze, pozwala na pobranie obiektu `PrintWriter` używanego do generowania zawartości dokumentu, która zostanie następnie przesłana do przeglądarki użytkownika. W przypadku tworzenia prostych serwletów przeważająca część pracy sprowadza się do pisania wywołań metody `println` generujących treść wynikowej strony. Obsługa danych przesyłanych z formularzy, nagłówków żądań HTTP, odpowiedzi HTTP oraz cookies omówię w kolejnych rozdziałach.

Zarówno metoda `doGet` jak i `doPost` zgłaszają wyjątki, a zatem ich nazwy należy umieścić w deklaracjach obu metod. Konieczne także będzie zaimportowanie klas z pakietów `java.io` (klasa `PrintWriter`, itp.), `javax.servlet` (klasa `HttpServlet`, itp.) oraz `javax.servlet.http` (`HttpServletRequest` oraz `HttpServletResponse`).

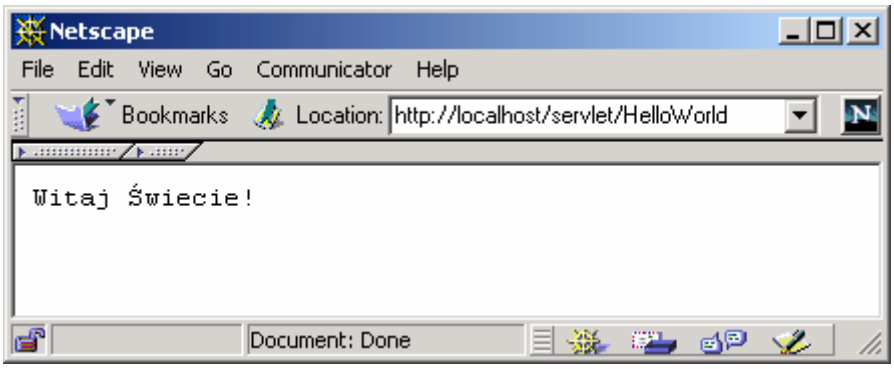
Chcąc być precyzyjnym należy zaznaczyć, iż klasa `HttpServlet` nie jest jedyną klasą którą się można posłużyć przy tworzeniu serwletów, gdyż teoretycznie rzecz biorąc serwlety mogą rozszerzać możliwości funkcjonalne serwerów poczty elektronicznej, FTP oraz innych. Serwlety działające w takich środowiskach będą klasami potomnymi jakiejś klasy, która z kolei jest klasą potomną `GenericServlet`. Klasa `GenericServlet` jest klasą bazową klasy `HttpServlet`. Jednak w praktyce serwlety są używane niemal wyłącznie na serwerach wykorzystujących protokół HTTP (czyli serwerach WWW oraz serwerach aplikacji); z tego względu w niniejszej książce ograniczę się do opisu serwletów właśnie tego typu.

2.2 Prosty serwlet generujący zwyczajny tekst

Na listingu 2.2 przedstawiłem prosty serwlet generujący zwyczajny tekst. Wyniki wykonania tego serwletu zostały przedstawione na rysunku 2.1. W podrozdziale 2.3 pt.: „Serwlet generujący kod HTML”, przedstawiłem nieco bardziej standardowy przykład generacji kodu HTML. Zanim jednak zajmiemy się kolejnymi zagadnieniami, warto poświęcić nieco czasu na przedstawienie procesu instalacji, kompilacji oraz uruchamiania serwletów. Za pierwszym razem bez wątpienia dojdziesz do wniosku, iż proces ten jest dosyć męczący. Okaż jednak nieco cierpliwości — proces ten jest niezmienny i szybko się do niego przyzwyczaisz, zwłaszcza jeśli go częściowo zautomatyzujesz przy wykorzystaniu skryptów (przykład takiego skryptu przedstawiłem w kolejnym podrozdziale).

Listing 2.2 HelloWorld.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
              throws ServletException, IOException {

    response.setContentType( "text/plain; charset=ISO-8859-2" );
    PrintWriter out = response.getWriter();
    out.println( "Witaj Świecie!" );  }
}
```



Rysunek 2.1 Wyniki wykonania serwletu z listingu 2.1 (HelloWorld.java).

Kompilacja i instalacja serwletów

W pierwszej kolejności powinieneś sprawdzić czy serwer został poprawnie skonfigurowany oraz czy zmienna środowiskowa `CLASSPATH` odwołuje się do pliku JAR zawierającego standardowe pliki klasowe serwletów. Wszelkie informacje na temat czynności jakie należy w tym celu wykonać znajdziesz w podrozdziale 1.5, pt.: „Instalacja i konfiguracja".

Kolejnym etapem jest podjęcie decyzji gdzie należy umieścić plik klasowy serwletu. Miejsca, w jakich można umieszczać pliki klasowe serwletów zależą od używanego serwera, a zatem konkretnych nazw katalogów powinieneś szukać w dokumentacji. Istnieją jednak pewne, w miarę ogólnie stosowane konwencje. Większość serwerów udostępnia trzy miejsca w których można umieszczać pliki klasowe serwletów; oto one:

1. **Katalog przeznaczony do umieszczania plików klasowych serwletów, które są często modyfikowane.**

   Serwlety umieszczone w tym katalogu są automatycznie ładowane do pamięci serwera, za każdym razem gdy ich plik klasowy zostanie zmodyfikowany. A zatem, podczas tworzenia serwletów powinieneś używać właśnie tego katalogu. Na przykład, w serwerach Java Web Server firmy Sun oraz WebSphere firmy IBM, katalog ten nosi standardowo nazwę *katalog_instalacyjny/servlets*, a w przypadku serwera BEA WebLogic nazwę *katalog_instalacyjny/myserver/servlet-classes*. Większość serwerów pozwala administratorom na podanie innych nazw katalogów. Ani Tomcat ani JSWDK nie dysponują możliwością automatycznego przeładowywania serwletów[6]. Niemniej jednak serwery te udostępniają podobny katalog przeznaczony do przechowywania plików klasowych serwletów. W przypadku gdy będziesz chciał zmienić istniejący serwlet konieczne będzie zatrzymanie i ponowne uruchomienie mini-serwera. W przypadku Tomcata 3.0 katalog przeznaczony do przechowywania plików klasowych serwletów nosi nazwę *katalog_instalacyjny/webpages/WEB-INF/classes*; serwer JSWDK 1.0.1 w tym samym celu używa katalogu *katalog_instalacyjny/webpages/WEB-INF/servlets*.
2. **Katalog przeznaczony do umieszczania plików klasowych serwletów, które nie są często modyfikowane.**

   Serwlety umieszczone w tym katalogu działają nieco bardziej efektywnie, gdyż serwer nie musi bezustannie sprawdzać dat ich modyfikacji. Niemniej jednak modyfikacje umieszczonych w nim serwletów wymagają zatrzymania i ponownego uruchomienia serwera. Serwlety wdrażane na „produkcyjnych" serwerach lub serwerach o wysokim natężeniu ruchu należy umieszczać właśnie w tym katalogu, bądź też w katalogu opisanym

[6] Nowsze wersje serwera Tomcat — 3.2 i kolejne — dysponują już możliwością automatycznego przeładowywania serwletów. (przyp. tłum.)

w kolejnym (3) punkcie. Katalog ten nosi zazwyczaj nazwę *katalog_instalacyjny/classes*, przynajmniej tak jest w przypadku serwerów Tomcat, JSWDK oraz Java Web Server. Ani Tomcat ani JSWDK nie dysponują możliwością automatycznego przeładowywania serwletów, a zatem w ich przypadku katalog ten pełni tę samą funkcję co katalog opisany w poprzednim punkcie. Dlatego też większość programistów korzysta z katalogu opisanego punkcie 1.

3. **Katalog przeznaczony do umieszczania plików klasowych serwletów, które nie są często modyfikowane i zostały zapisane w pliku JAR.**

   W przypadku katalogu opisanego w poprzednim punkcie, pliki klasowe serwletów są umieszczane bezpośrednio w katalogu *classes*, bądź też w jego podkatalogach, których nazwy odpowiadają nazwom pakietów do których należą serwlety. Kolejną możliwością jest umieszczenie plików klasowych serwletów w pliku JAR i umieszczenie tego pliku w odpowiednim katalogu. W przypadku serwerów Tomcat, JSWDK, Java Web Server oraz większości innych serwerów, katalog ten nosi nazwę *katalog_instalacyjny/lib*. Za każdym razem gdy zmienisz pliki umieszczone w tym katalogu, aby zmiany zostały uwzględnione, konieczne jest zatrzymanie i powtórne uruchomienie serwera.

Kiedy już skonfigurowałeś serwer, określiłeś wartość zmiennej środowiskowej `CLASSPATH` i umieściłeś plik klasowy serwletu w odpowiednim katalogu, będziesz mógł go skompilować. W tym celu wystarczy wydać polecenie, na przykład: `javac HelloWorld.java`. Jednak w środowiskach produkcyjnych, serwlety są często umieszczane w pakietach, co ma na celu uniknięcie konfliktów nazw, które mogą się pojawić pomiędzy serwletami pisanymi przez różne osoby. Wykorzystanie pakietów wymaga wykonania kilku dodatkowych czynności, pisanych w podrozdziale 2.4, pt.: „Umieszczanie serwletów w pakietach". Poza tym, bardzo często stosuje się formularze HTML jako „interfejsy użytkownika" pośredniczące w wykorzystaniu serwletów (patrz rozdział 16). Aby skorzystać z tej techniki, będziesz musiał wiedzieć gdzie należy umieszczać pliki HTML, tak aby były one dostępne dla serwera. Położenie to zależy wyłącznie od serwera; w przypadku serwerów JSWDK oraz Tomcat pliki HTML należy umieszczać w katalogu *katalog_instalacyjny/webpages/ścieżka/*. Aby odwołać się do dokumentu o nazwie *plik.html* (zapisanego jako *katalog_instalacyjny/webpages/ścieżka/plik.html*) należy podać w przeglądarce adres *http://localhost/ścieżka/plik.html* (przy czym, w przypadku gdy serwer nie działa na lokalnym komputerze, wyrażenie „*localhost*" należy zastąpić jego poprawną nazwą). Strony JSP mogą być zapisywane wszędzie tam gdzie normalne dokumenty HTML.

Wywoływanie serwletów

Różne serwery pozwalają na umieszczanie plików klasowych serwletów w różnych katalogach — pod tym względem standaryzacja pomiędzy poszczególnymi serwerami jest niewielka. Niemniej jednak, w przypadku wywoływania serwletów obowiązuje jedna, wspólna konwencja — należy używać adresu URL o postaci *http://komputer/sevlet/NazwaServletu*. Zwróć uwagę, iż adres ten odwołuje się do katalogu *servlet*; pomimo faktu, iż w rzeczywistości serwlety zostały umieszczone w katalogu *servlets* lub katalogu o zupełnie innej nazwie (np.: *classes* lub *lib*).

Rysunek 2.1 przedstawiony we wcześniejszej części rozdziału prezentuje sposób odwołania się do serwletu w przypadku gdy serwer został uruchomiony na lokalnym komputerze („localhost" oznacza „lokalny komputer").

Większość serwerów pozwala także rejestrować nazwy serwletów, dzięki czemu można się do nich odwoływać za pomocą adresu URL o postaci *http://komputer/dowolna_ścieżka/dowolny_plik*. Czynności jakie należy w tym celu wykonać zależą od używanego serwera, a informacji na ich temat należy szukać w jego dokumentacji.

2.3 Serwlety generujące kod HTML

Serwlet przedstawiony w poprzednim przykładzie generował zwyczajny tekst. Większość serwletów generuje jednak nie tekst, lecz kod HTML. Aby stworzyć taki serwlet będziesz musiał wykonać dwie dodatkowe czynności:

1. poinformować przeglądarkę, iż wyniki które otrzyma są kodem HTML,
2. zmodyfikować wywołania metody `println` tak, aby generowały poprawny kod HTML.

Aby wykonać pierwszą czynność należy określić wartość nagłówka odpowiedzi HTTP o nazwie `Content-Type`. Ogólnie rzecz biorąc, nagłówki są określane przy użyciu metody `setHeader` interfejsu `HttpServletResponse`; jednak określanie typu zawartości jest czynnością wykonywaną tak często, iż została stworzona specjalna metoda służąca właśnie do tego celu — `setContentType`. Aby określić, że informacje generowane przez serwlet są kodem HTML należy użyć typu MIME „`text/html`”[7]; można to zrobić przy użyciu następującego wywołania:

```
response.setContentType("text/html");
```

Choć serwlety są najczęściej używane właśnie do tworzenia kodu HTML, to jednak wykorzystanie ich do generacji dokumentów innych typów nie jest rozwiązaniem niezwykłym. Na przykład w podrozdziale 7.5 pt.: „Wykorzystanie serwletów do generacji obrazów GIF” pokażę w jaki sposób można wykorzystać serwlety do generacji własnych obrazów, posługując się przy tym typem MIME `image/gif`. Kolejny przykład, zamieszczony w podrozdziale 11.2 pt.: „Atrybut contentType”, przedstawia w jaki sposób można wygenerować i zwrócić arkusz kalkulacyjny programu Microsoft Excel, posługując się przy tym typem MIME `application/vnd.ms-excel`.

Nie przejmuj się, jeśli jeszcze nie wiesz niczego o nagłówkach odpowiedzi HTTP — omówię je szczegółowo w rozdziale 7. Zwróć uwagę, iż nagłówki odpowiedzi należy podać zanim wygenerujesz jakąkolwiek treść dokumentu, posługując się metodami klasy `PrintWriter`. Taki sposób postępowania wynika z faktu, iż odpowiedzi HTTP składają się z wiersza statusu, jednego lub kilku nagłówków, pustego wiersza oraz treści dokumentu; przy czym wszystkie te elementy muszą występować dokładnie w wymienionej kolejności. Same nagłówki mogą być zapisane w dowolnej kolejności, serwlety bowiem buforują je i wysyłają jednocześnie. Z tego względu dozwolone jest określenie kodu statusu (elementu umieszczanego w pierwszym wierszu), nawet po podaniu innych nagłówków odpowiedzi. Jednak serwlety nie muszą wcale buforować samej treści dokumentu; wynika to z faktu, iż użytkownicy mogą chcieć, aby nawet częściowe wyniki były wyświetlane w przeglądarce. W wersji 2.1 specyfikacji serwletów, informacje generowane przy użyciu klasy `PrintWriter` w ogóle nie są buforowane. Oznacza to, że jeśli choćby raz użyjesz jakiejś metody tej klasy, to nie będziesz już mógł określać nagłówków odpowiedzi. Według specyfikacji Java Servlet 2.2, mechanizmy obsługi serwletów mogą częściowo buforować generowane informacje wyjściowe, jednak wielkość stosowanego bufora nie została precyzyjnie określona. Interfejs `HttpServletResponse` udostępnia metodę `getBufferSize`, która pozwala poznać wielkość bufora oraz metodę `setBufferSize`, która umożliwia podanie jego wielkości. W przypadku mechanizmów obsługi serwletów działających zgodnych ze specyfikacją Java Servlet 2.2 możliwe jest ustawianie nagłówków do momentu zapełnienia buforu i przesłania jego zawartości do klienta. Jeśli nie jesteś pewny czy zawartość buforu została już przesłana czy nie, możesz to sprawdzić przy użyciu metody `isCommitted`.

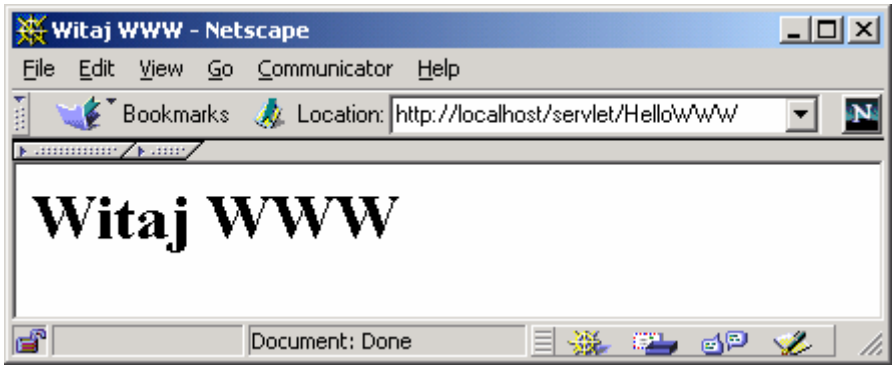
Metoda
Zawsze określaj typ zawartości *zanim* zaczniesz generować faktyczną treść dokumentu.

[7] Aby w wynikowej stronie WWW były dostępne polskie znaki diakrytyczne, należy oprócz typu MIME określić także sposób kodowania strony. W tym celu, po typie MIME `"text/html"` wystarczy dopisać `"; charset=ISO-8859-2"` (lub Windows-1250). A zatem całe wywołanie metody `setContentType` będzie miało postać `response.setContentType("text/html; charset=ISO-8859-2")`.

Kolejnym etapem tworzenia serwletu generującego dokument HTML, jest napisanie wywołań metody `println`, które wygenerują poprawny kod HTML. Struktura dokumentów HTML została bardziej szczegółowo omówiona w podrozdziale 2.5 pt.: „Proste narzędzia pomocne przy tworzeniu dokumentów HTML", niemniej jednak większość czytelników powinna ją już znać. Na listingu 2.3 przedstawiłem przykładowy serwlet generujący prosty dokument HTML, a na rysunku 2.2 wyniki jego wykonania.

Listing 2.3 HelloWWW.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWWW extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>Witaj WWW</TITLE></HEAD>\n" +
                "<BODY>\n" +
                "<H1>Witaj WWW</H1>\n" +
                "</BODY></HTML>");
  }
}
```



Rysunek 2.2 Wyniki wykonania serwletu przedstawionego na listingu 2.3 (HelloWWW.java).

2.4 Umieszczanie serwletów w pakietach

W środowiskach produkcyjnych wielu programistów może tworzyć serwlety wykorzystywane na tym samym serwerze. W takim przypadku, umieszczanie wszystkich serwletów na głównym poziomie jednego katalogu może sprawić, że zarządzanie jego zawartością będzie bardzo trudne; a co gorsze, może spowodować powstanie konfliktów nazw (jeśli się zdarzy, że dwaj programiści przypadkowo nadadzą swym serwletom identyczne nazwy). Naturalnym rozwiązaniem tego problemu jest zastosowanie pakietów. W przypadku wykorzystania pakietów zmienia się sposób tworzenia serwletów, sposób ich kompilacji oraz ich wywoływania. Przyjrzyjmy się dokładniej tym trzem zagadnieniom — omówię je szczegółowo w trzech kolejnych podrozdziałach. Tworzenie i kompilacja serwletów należących do konkretnego pakietu nie różni się niczym od tworzenia jakichkolwiek innych klas Javy należących do jakiegoś pakietu; pod tym względem serwlety nie są żadnym wyjątkiem.

Tworzenie serwletów należących do konkretnego pakietu

Aby umieścić serwlet w pakiecie, należy wykonać dwie czynności:

1. **Przenieść plik klasowy serwletu do katalogu którego nazwa odpowiada nazwie pakietu.**
   Na przykład, przeważająca większość przykładowych serwletów przedstawionych w niniejszej książce została umieszczona w pakiecie `coreservlets`. A zatem, pliki klasowe tych serwletów muszą zostać umieszczone w podkatalogu o nazwie *coreservlets*.
2. **Dodać instrukcję `package` do kodu serwletu.**
   Na przykład, aby umieścić klasę w pakiecie o nazwie JakisPakiet, *pierwszy* wiersz pliku zawierającego kod źródłowy klasy musi mieć następującą postać:
   ```
   package JakisPakiet;
   ```

Listing 2.4 przedstawia zmodyfikowaną wersję serwletu `HelloWWW`, o nazwie `HelloWWW2`. Serwlet ten różni się od poprzedniej wersji wyłącznie tym, iż został umieszczony w pakiecie `coreservlets`. W przypadku używania serwera Tomcat 3.0 plik klasowy tego serwletu należy umieścić w katalogu *katalog_instalacyjny/webpages/WEB-INF/classes/coreservlets*, w przypadku korzystania z serwera JSWDK 1.0.1 w katalogu *katalog_instalacyjny/webpages/WEB-INF/servlets/coreservlets*, i w końcu, w przypadku serwera Java Web Server w katalogu *katalog_instalacyjny/servlets/coreservlets*.

Listing 2.4 HelloWWW2.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWWW2 extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>Witaj WWW</TITLE></HEAD>\n" +
                "<BODY>\n" +
                "<H1>Witaj WWW</H1>\n" +
                "</BODY></HTML>");
  }
}
```

Kompilacja serwletów należących do pakietów

Istnieją dwa podstawowe sposoby kompilacji klas należących do pakietów. Pierwszy z nich polega na umieszczeniu podkatalogu pakietu, bezpośrednio w katalogu przeznaczonym do przechowywania plików klasowych serwletów (nazwa tego katalogu zależy od używanego serwera). W takim przypadku trzeba dodać do zmiennej systemowej `CLASSPATH` nazwę katalogu nadrzędnego względem katalogu zawierającego tworzone serwlety — czyli do głównego katalogu przeznaczonego do przechowywania serwletów na danym serwerze. Po wykonaniu tych czynności można przejść do katalogu zawierającego kody źródłowe serwletów i skompilować je w standardowy sposób. Na przykład, jeśli główny folder przeznaczony do przechowywania kodów źródłowych serwletów nosi nazwę *C:\JavaWebServer2.0\servlets*, a pakiet w którym chcesz umieścić serwlet (a zatem także i odpowiedni podkatalog) ma nazwę `coreservlets`, i jeśli używasz systemu Windows, to aby skompilować swój serwlet powinieneś wykonać następujące polecenia:

```
DOS> set CLASSPATH=C:\JavaWebServer2.0\servlets;%CLASSPATH%
DOS> cd C:\JavaWebServer2.0\servlets\coreservlets
DOS> javac HelloWWW2.java
```

Sądzę, że pierwszą czynność — określenie wartości zmiennej środowiskowej CLASSPATH — chciałbyś wykonać tylko raz; a nie powtarzać jej za każdym razem gdy otworzysz nowe okno trybu MS-DOS. W tym celu, w systemach Windows 95/98, powinieneś umieścić to polecenie w pliku *autoexec.bat*, gdzieś poniżej polecenia zapisującego w zmiennej środowiskowej CLASSPATH położenie pliku *server.jar* oraz plików JAR zawierających pliki klasowe JSP. W systemie Windows NT należy wybrać z menu opcję *Start⇨Settings⇨Control Panel*, dwukrotnie kliknąć ikonę *System*, przejść na zakładkę *Environment* i podać wartość zmiennej środowiskowej CLASSPATH . W systemie Windows 2000 należy wybrać opcje *Start⇨Ustawienia⇨Panel sterowania*, dwukrotnie kliknąć ikonę *System*, następnie przejść na zakładkę *Zaawansowane*, kliknąć przycisk *Zmienne środowiskowe* i podać wartość zmiennej środowiskowej. W systemach Unix (C Shell) wartość zmiennej środowiskowej CLASSPATH można określić w następujący sposób:

```
setenv CLASSPATH /katalog_instalacyjny/servlets:$CLASSPATH
```

Aby polecenie to miało trwałe skutki, należy umieścić je w pliku *.cshrc*.

Jeśli nazwa pakietu byłaby bardziej złożona (na przykład: pakiet1.pakiet2.pakiet3), a nie tak prosta jak w naszym przypadku (pakiet1), to zmienna CLASSPATH i tak powinna wskazywać główny katalog przeznaczony do przechowywania plików klasowych serwletów (czyli katalog zawierający katalog pakiet1).

Drugi sposób kompilacji klas należących do pakietów polega na przechowywaniu kodów źródłowych oraz plików klasowych w odrębnych katalogach. W takim przypadku, w pierwszej kolejności, powinieneś umieścić katalog z kodami źródłowymi w dowolnie wybranym miejscu. Nazwę tego katalogu powinieneś następnie dodać do zmiennej środowiskowej CLASSPATH. Następnie, podczas kompilacji klas, będziesz musiał użyć opcji -d programu javac, aby umieścić pliki klasowe serwletów w katalogu, w którym będzie ich poszukiwał serwer. Także w tym przypadku warto w trwały sposób zmodyfikować wartość zmiennej środowiskowej CLASSPATH.

```
DOS>  cd c:\MojeServlety\coreservlets
DOS>  set CLASSPATH=C:\MojeServlety;%CLASSPATH%
DOS>  javac -d C:\tomcat\webpages\WEB-INF\classes HelloWWW2.java
```

Osobiście, w swej pracy wykorzystuję właśnie to rozwiązanie, polegające na przechowywaniu kodu źródłowego oraz plików klasowych w odrębnych katalogach. Aby dodatkowo skomplikować sobie życie, używam kilku różnych ustawień zmiennej środowiskowej CLASSPATH, wykorzystywanych w zależności od projektu nad którym aktualnie pracuję, poza tym, zazwyczaj używam JDK 1.2 a nie JDK 1.1 którego oczekuje Java Web Server. W powyższych względów, doszedłem do wniosku, iż w systemie Windows warto zautomatyzować cały proces kompilacji servletów przy wykorzystaniu pliku wsadowego. Przykład takiego pliku, o nazwie *servletc.bat*, przedstawiłem na listingu 2.5 (zwróć uwagę, iż polecenie set CLASSPATH=... powinno być zapisane w jednym wierszu, na listingu podzieliłem je, aby poprawić czytelność przykładu). Plik ten umieściłem w katalogu *C:\Windows\Command* lub w innym katalogu podanym w zmiennej środowiskowej PATH systemu Windows. W ten sposób, aby skompilować servlet HelloWWW2 i zainstalować go w katalogu, w jakim będzie go poszukiwał Java Web Server, wystarczy przejść do katalogu *C:\MojeServlety\coreservlets* i wydać polecenie servletc HelloWWW2.java. Kody źródłowe dostępne na serwerze FTP Wydawnictwa HELION (*ftp://ftp.helion.pl/przyklady/jsjsp.zip*) zawierają zmodyfikowane wersje plików wsadowych *servletc.bat* przeznaczone do kompilacji servletów, które będą uruchamiane na serwerach JSWDK oraz Tomcat.

Listing 2.5 servletc.bat

```
@echo off

rem To jest wersja dla Java Web Server
rem Inne wersje znajdziesz pod adresem
rem ftp://ftp.helion.pl/przyklady/jsjsp.zip

set CLASSPATH=C:\JavaWebServer2.0\lib\servlet.jar;
    C:\JavaWebServer2.0\lib\jsp.jar;C:\MojeServlety
C:\JDK1.1.8\bin\javac -d C:\JavaWebServer2.0\servlets %1%
```

Wywoływanie serwletów należących do pakietów

Aby wywołać serwlet umieszczony w pakiecie, należy podać adres URL o następującej, ogólnej postaci:

```
http://komputer/servlet/nazwa_pakietu.nazwa_servletu
```

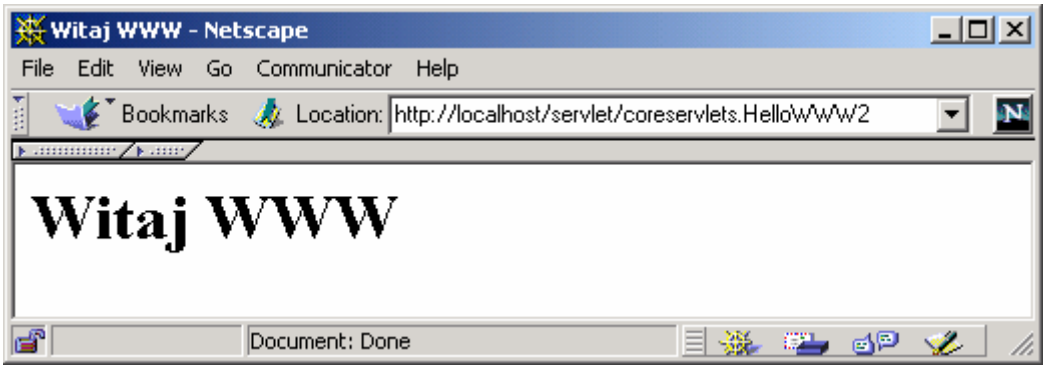
W takim przypadku nie należy posługiwać się adresem URL:

```
http://komputer/servlet/nazwa_servletu
```

A zatem, zakładając, że serwer został uruchomiony na lokalnym komputerze, adres

```
http://localhost/servlet/coreservlets.HelloWWW2
```

spowoduje wykonanie serwletu HelloWWW2 i wyświetlenie wyników przedstawionych na rysunku 2.3.



Rysunek 2.3 Wywołanie serwletu przy użyciu adresu *http://localhost/servlet/coreservlet.HelloWWW2*

2.5 Proste narzędzia pomocne przy tworzeniu dokumentów HTML

Poniżej przedstawiłem podstawową strukturę dokumentów HTML:

```
<!DOCTYPE ...>
<HTML>
<HEAD><TITLE>...</TITLE>...</HEAD>
<BODY ...>
...
</BODY>
</HTML>
```

Być może będzie Cię kusiło by pominąć któryś z elementów tej struktury, a w szczególności deklarację `DOCTYPE`, gdyż wszystkie główne przeglądarki ignorują ją, niezależnie od tego, iż w specyfikacjach HTML 3.2 oraz 4.0 jest ona wymagana. Odradzam jednak stosowanie takich rozwiązań. Deklaracja ta ma tę zaletę, iż informuje narzędzia sprawdzające poprawność kodu HTML, jakiej specyfikacji należy użyć przy weryfikacji dokumentu. Narzędzia te są niezwykle przydatne podczas testowania stron WWW, gdyż wykrywają błędy składniowe w kodzie HTML, które w nowych przeglądarkach zostałyby poprawnie zinterpretowane, lecz w starych doprowadziły do błędnego wyświetlenia strony. Dwoma najpopularniejszymi narzędziami sprawdzającymi poprawność kodu HTML dostępnymi na WWW są serwisy World Wide Web Consortium (*http://validator.w3.org/*) oraz Web Digest Group (*http://www.htmlhelp.com/tools/validator/*). Serwisy te pozwalają na podanie adresu URL, a następnie pobierają wskazaną stronę, sprawdzają jej poprawność syntaktyczną i zgodność z formalną specyfikacją języka HTML i zwracają raport z informacjami o błędach. Z punktu widzenia użytkownika serwlety generujące kod HTML niczym się nie różnią od zwyczajnych stron WWW; z tego względu wyniki ich działania można sprawdzać przy użyciu wspomnianych wcześnie serwisów. Jedynym wyjątkiem są serwlety, które wymagają przekazania informacji metodą `POST`. Pamiętaj, że dane przesyłane metodą `GET` są dołączane do

adresu URL, a zatem możliwe jest przekazanie do serwisu sprawdzającego adresu URL zawierającego dane, jakie mają zostać przesłane do serwletu metodą GET.

Metoda
Używaj serwisów sprawdzających aby skontrolować poprawność dokumentów HTML generowanych przez servlety.

Bez wątpienia generacja kodu HTML przy użyciu wywołań metody println jest nieco niewygodnym rozwiązaniem, zwłaszcza w przypadku generacji długich wierszy kodu, takich jak deklaracje DOCTYPE. Niektórzy rozwiązują ten problem tworząc w języku Java narzędzia służące do generacji kodu HTML i wykorzystując je w serwletach. Ja jednak sceptycznie podchodzę do tworzenia rozbudowanej biblioteki narzędzi tego typu. Przede wszystkim, trudności jakich nastręcza generacja kodu HTML z poziomu programu jest jednym z głównych problemów rozwiązywanych przez Java Server Pages (technologię, której poświęcona jest druga część niniejszej książki). JSP jest znacznie lepszym rozwiązaniem, a zatem, nie warto tracić czasu na tworzenie rozbudowanego pakietu narzędzi służących do generacji kodu HTML z poziomu serwletów. Po drugie, procedury generujące kod HTML mogą być niewygodne w użyciu i zazwyczaj nie udostępniają wszystkich istniejących atrybutów znaczników HTML (takich jak CLASS i ID używanych wraz z arkuszami stylów, procedur obsługi zdarzeń języka JavaScript, atrybutu określającego kolor tła komórek tabel, i tak dalej). Pomimo tych wątpliwych zalet dużych bibliotek służących do generacji kodu HTML, jeśli zauważysz, że bardzo często generujesz te same fragmenty kodu, to możesz stworzyć proste narzędzie, które w przyszłości ułatwi Ci pracę. W przypadku prostych serwletów istnieją dwa elementy dokumentów HTML (DOCTYPE oraz HEAD) których postać nie powinna ulegać zmianie i których generację można sobie ułatwić tworząc proste narzędzia. Przykładową postać takich narzędzi przedstawiłem na listingu 2.6. Listing 2.7 zawiera zmodyfikowaną wersję serwletu HelloWWW2, wykorzystującą narzędzia z listingu 2.6. W dalszej części książki narzędzie te zostaną bardziej rozbudowane.

Listing 2.6 ServletUtilities.java

```
package coreservlets;

import javax.servlet.*;
import javax.servlet.http.*;

public class ServletUtilities {
  public static final String DOCTYPE =
    "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
    "Transitional//EN\">";

  public static String headWithTitle(String title) {
    return(DOCTYPE + "\n" +
           "<HTML>\n" +
           "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n");
  }

 /* ... dalsza część klasy używana w dalszej części książki ... */

}
```

Listing 2.7 HelloWWW3.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWWW3 extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println(ServletUtilities.headWithTitle("Witaj WWW") +
                "<BODY>\n" +
```

```
                "<H1>Witaj WWW</H1>\n" +
                "</BODY></HTML>");
  }
}
```

2.6 Cykl życiowy serwletów

We wcześniejszej części książki podałem pobieżną informację, że tworzona jest tylko jedna kopia serwletu, a podczas obsługi każdego żądania tworzony jest nowy wątek, przekazywany odpowiednio do metody `doGet` lub `doPost`. W tej części rozdziału dokładniej omówię sposób tworzenia i niszczenia serwletów, podam również informacje dotyczące sposobów oraz sytuacji w jakich wywoływane są różne metody. Na razie podam krótkie podsumowanie informacji dotyczących tych zagadnień, a dokładniej omówię je w kolejnych podrozdziałach.

W momencie tworzenia serwletu, wywoływana jest jego metoda `init`. Stanowi ona zatem miejsce, w którym należy umieścić jednokrotnie wykonywany kod inicjalizujący serwletu. Obsługa każdego żądania powoduje stworzenie wątku i wywołanie metody `service` utworzonej wcześniej kopii serwletu. Wiele obsługiwanych jednocześnie żądań powoduje zazwyczaj utworzenie wielu wątków, które jednocześnie wywołują metodę `service`. Serwlet może jednak implementować specjalny interfejs, który sprawia, iż w danej chwili będzie mógł być wykonywany tylko jeden wątek. Metoda `service` wywołuje następnie metodę `doGet`, `doPost` lub dowolną inną metodę `do`*`Xxx`*, przy czym to, która z nich zostanie wykonana zależy od otrzymanych nagłówków żądania HTTP. I w końcu, gdy serwer zdecyduje się usunąć serwlet z pamięci, nim to się stanie zostanie wywołana metoda `destroy`.

Metoda init

Metoda `init` wywoływana jest tylko raz, bezpośrednio po utworzeniu serwletu; nie jest ona wywoływana podczas obsługi poszczególnych żądań. Metoda ta może zatem służyć do przeprowadzenia jednokrotnej inicjalizacji serwletu, podobnie jak metoda o identycznej nazwie stosowana w apletach. W zależności od sposobu zarejestrowania serwletu na serwerze, może on zostać utworzony w momencie gdy użytkownik po raz pierwszy poda adres URL odpowiadający danemu serwletowi, lub też, podczas uruchamiania serwera. Jeśli serwlet nie został zarejestrowany, lecz umieszczono go w jednym ze standardowych katalogów serwera, to zostanie on utworzony podczas obsługi pierwszego żądania. Szczegółowe informacje dotyczące katalogów, służących do przechowywania serwletów znajdziesz w podrozdziale 2.2 pt.: „Prosty serwlet generujący zwyczajny tekst".

Dostępne są dwie wersje metody `init` — pierwsza z nich nie pobiera żadnych argumentów, natomiast druga pobiera argument będący obiektem typu `ServletConfig`. Pierwsza z wersji tej metody używana jest w przypadkach gdy serwlet nie musi odczytywać żadnych ustawień zależnych od serwera na którym jest uruchamiany. Definicja tej wersji metody `init` ma następującą postać:

```
public void init() throws ServletException {
  // kod inicjalizujący servletu
}
```

Przykład wykorzystania tej formy metody `init` możesz znaleźć w podrozdziale 2.8, pt.: „Przykład wykorzystania inicjalizacji serwletu i daty modyfikacji strony". W podrozdziale 18.8, pt.: „Zarządzanie pulami połączeń: Studium zagadnienia", w rozdziale poświęconym zagadnieniom wykorzystania JDBC, znajdziesz bardziej zaawansowany przykład użycia metody `init` w celu utworzenia wielu połączeń z bazą danych.

Druga wersja metody `init` stosowana jest w sytuacjach gdy przed zakończeniem inicjalizacji serwlet musi odczytać ustawienia zależne od serwera na jakim jest uruchamiany. Na

przykład, może się zdarzyć, że serwlet będzie musiał dysponować informacjami dotyczącymi bazy danych, plików z hasłami, charakterystycznych dla serwera parametrów związanych z efektywnością działania, nazw plików liczników lub trwale zachowanych danych przekazanych za pośrednictwem cookies podczas trwania wcześniejszych sesji. Ta wersja metody `init` ma następującą postać:

```
public void init(ServletConfig config)
    throws ServletException {
  super.init(config);
  // kod inicjalizujący servletu
}
```

W powyższym fragmencie kodu należy zwrócić uwagę na dwie sprawy. Po pierwsze, metoda `init` pobiera argument typu `ServerConfig`. Interfejs ten udostępnia metodę `getInitParameter`, której można użyć do odszukania parametrów inicjalizacyjnych skojarzonych z serwletem. Podobnie jak w przypadku metody `getParameter` stosowanych w metodzie `init` apletów, także metoda `getInitParameter` oczekuje podania argumentu będącego łańcuchem znaków (zawierającego nazwę parametru) i zwraca łańcuch znaków (zawierający wartość określonego parametru). Prosty przykład wykorzystania parametrów inicjalizacyjnych znajdziesz w podrozdziale 2.7, pt.: „Przykład użycia parametrów inicjalizacyjnych”; w podrozdziale 4.5 (pt.: „Ograniczanie dostępu do stron WWW”) przedstawiłem bardziej zaawansowany przykład, w którym metoda `getInitParameter` używana jest do pobrania nazwy pliku z hasłami. Zwróć uwagę, że wartości parametrów odczytuje się zawsze w taki sam sposób, natomiast określanie ich wartości zależy od używanego serwera. Na przykład, w przypadku użycia serwera Tomcat, wartości parametrów podawane są w specjalnym pliku o nazwie *web.xml*, w przypadku serwera JSWDK do tego samego celu służy plik *servlets.properties*, w przypadku serwera aplikacji WebLogic — plik *weblogic.properties*, a jeśli korzystasz z Java Web Servera, parametry są ustawiane interaktywnie przy użyciu konsoli administracyjnej. Przykłady prezentujące sposób określania wartości parametrów inicjalizacyjnych serwletów znajdziesz w podrozdziale 2.7, pt.: „Przykład użycia parametrów inicjalizacyjnych”.

W drugiej wersji metody `init` należy także zwrócić uwagę na pierwszy wiersz kodu, w którym zostało umieszczone wywołanie `super.init`. Jest ono niezwykle ważne! Obiekt `ServletConfig` jest bowiem używany także w innych miejscach serwletu, a metoda `init` klasy bazowej rejestruje go w taki sposób, aby można go było później odnaleźć i wykorzystać. A zatem, jeśli pominiesz wywołanie `super.init` możesz przysporzyć sobie wielu kłopotów.

Metoda
Jeśli używasz metody `init` pobierającej argument typu `ServletConfig`, to w pierwszym wierszu tej metody koniecznie musisz umieścić wywołanie `super.init`.

Metoda service

Za każdym razem gdy serwer otrzymuje żądanie dotyczące serwletu, uruchamiany jest nowy wątek i wywoływana metoda `service`. Metoda ta sprawdza typ żądania HTTP (`GET`, `POST`, `PUT`, `DELETE`, itd.) i w zależności od niego wywołuje odpowiednią metodę — `doGet`, `doPost`, `doPut`, `doDelete`, itd. Jeśli tworzysz serwlet, który ma w identyczny sposób obsługiwać żądania `GET` oraz `POST`, to możesz zastanawiać się nad zastosowaniem rozwiązania polegającego na bezpośrednim przesłonięciu metody `service` (w sposób przedstawiony poniżej), zamiast implementacji obu metod `doGet` oraz `doPost`.

```
public void service(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
  // kod servletu
}
```

Nie stosuj jednak takiego rozwiązania. Zamiast niego, wywołaj metodę `doPost` z metody `doGet` (lub na odwrót), jak pokazałem na poniższym przykładzie:

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
  // kod servletu
}

public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException {
  doGet(request, response);
}
```

Choć metoda ta wymaga wpisania kilku dodatkowych wierszy kodu, to jednak z pięciu powodów jest lepsza od bezpośredniego przesłaniania metody `service`. Poniżej podałem te powody:

1. Istnieje możliwość dodania obsługi żądań innych typów; wystarczy w tym celu (na przykład w klasie potomnej) zaimplementować metody `doPut`, `doTrace`, itd. W przypadku bezpośredniego przesłonięcia metody `service` nie można tego zrobić.
2. Istnieje możliwość dodania obsługi dat modyfikacji; w tym celu wystarczy zaimplementować metodę `getLastModified`. W przypadku wykorzystania metody `doGet`, standardowa implementacja metody `service` wywołuje metodę `getLastModified` w celu podania wartości nagłówków `Last-Modified`. Podanie wartości tych nagłówków jest konieczne dla prawidłowej obsługi warunkowych żądań `GET` (czyli żądań zawierających nagłówek `If-Modified-Since`). Stosowny przykład znajdziesz w podrozdziale 2.8, pt.: „Przykład wykorzystania inicjalizacji serwletu i daty modyfikacji strony”.
3. Uzyskuje się automatyczną obsługę żądań `HEAD`. W takim przypadku serwer zwraca wyłącznie nagłówki i kod statusu wygenerowany przez metodę `doGet`, pomijając wszelką zawartość wygenerowanego dokumentu. Żądania `HEAD` są bardzo przydatne w przypadku tworzenia programów korzystających z protokołu HTTP. Na przykład, narzędzia sprawdzające poprawność hiperpołączeń umieszczonych na stronach WWW, aby zredukować obciążenie serwera, bardzo często posługują się właśnie żądaniami `HEAD` a nie `GET`.
4. Uzyskuje się automatyczną obsługę żądań `OPTIONS`. Jeśli metoda `doGet` została zaimplementowana, to standardowa metoda `service` odpowiada na żądania `OPTIONS` zwracając nagłówek `Allow` informujący, że obsługiwane są żądania `GET`, `HEAD`, `OPTIONS` oraz `TRACE`.
5. Uzyskuje się automatyczną obsługę żądań `TRACE`. Żądania `TRACE` stosowane są podczas testowania programów korzystających z protokołu HTTP — w odpowiedzi na nie serwer zwraca wyłącznie nagłówki HTTP.

Podpowiedź
Jeśli tworzony serwlet ma w identyczny sposób obsługiwać zarówno żądania `GET` jak i `POST`, to wywołuj metodę `doPost` z metody `doGet`, lub na odwrót. Nie stosuj natomiast rozwiązania polegającego na bezpośrednim przesłonięciu metody `service`.

Metody doGet, doPost oraz do*Xxx*

Te metody zawierają najistotniejsze czynności wykonywane przez serwlet. W 99 procentach przypadków będą Cię interesowały wyłącznie żądania `GET` lub `POST`, a zatem będziesz implementował metody `doPost` lub `doGet`. Jeśli jednak będziesz chciał, to nic nie stoi na przeszkodzie, aby zaimplementować także metodę `doDelete` służącą do obsługi żądań `DELETE`, `doPut` obsługującą żądania `PUT`, `doTrace` obsługującą żądania `TRACE` oraz `doOptions` służącą do obsługi żądań `OPTIONS`. Pamiętaj jednak, że możesz skorzystać z automatycznej obsługi żądań `OPTIONS` oraz

TRACE, jaką dysponuje metoda service, opisana w poprzednim podrozdziale. Zwróć uwagę, iż nie ma metody doHead, gdyż system automatycznie wykorzystuje wiersz statusu oraz nagłówki generowane przez metodę doGet, także do obsługi żądań HEAD.

Interfejs SingleThreadModel

Standardowo system tworzy jedną kopię serwletu, a następnie używa nowych wątków do obsługi nadsyłanych żądań; przy czym, w przypadku gdy nowe żądanie nadejdzie zanim wykonywanie poprzedniego żądania zostanie zakończone, uruchamiane są kolejne wątki wykonywane jednocześnie. Oznacza to, że metody doGet oraz doPost muszą bardzo uważnie synchronizować dostęp do pól oraz innych, wspólnych informacji. Jest to konieczne, gdyż wiele wątków może jednocześnie próbować korzystać z tych danych. Więcej informacji na ten temat znajdziesz w podrozdziale 7.3, pt.: „Trwałe przechowywanie stanu serwletu i automatyczne odświeżanie stron". Jeśli nie chcesz, aby serwlet działał w ten standardowy — „wielowątkowy" sposób, wystarczy zaimplementować w nim interfejs SingleThreadModel:

```
public class MojServlet extends HttpServlet
                        implements SingleThreadModel {
  // ... kod servletu ... //
}
```

Jeśli zaimplementujesz ten interfejs, system zagwarantuje, że w dowolnej chwili z pojedynczej kopii serwletu będzie korzystał co najwyżej jeden wątek obsługujący żądania. W tym celu serwer bądź to umieszcza wszystkie żądania w kolejce i po kolei przekazuje je do pojedynczej kopii serwletu, bądź też tworzy pulę kopii serwletów, z których każda w danej chwili będzie obsługiwać tylko jedno żądanie. Oznacza to, że nie musisz się przejmować równoczesnym dostępem do zwyczajnych pól (zmiennych instancyjnych) serwletu. Niemniej jednak wciąż konieczna jest synchronizacja dostępu do zmiennych klasowych (pól oznaczonych jako static) oraz danych przechowywanych poza serwletem.

Synchroniczny dostęp do serwletów może w znaczącym stopniu ograniczyć efektywność działania serwera (czyli czas oczekiwania na wyniki) w przypadkach gdy serwlet wykorzystywany jest bardzo często. A zatem, musisz dobrze przemyśleć czy należy korzystać z interfejsu SingleThreadModel.

Metoda destroy

Serwer może podjąć decyzję o usunięciu z pamięci załadowanej do niej kopii serwletu. Decyzja taka może zostać podjęta w wyniku jawnego żądania administratora lub ze względu na fakt, iż serwlet nie był wykorzystywany przez długi okres czasu. Niemniej jednak, nim serwlet zostanie usunięty z pamięci, serwer wywoła jego metodę destroy. Metoda ta, daje serwletowi możliwość zamknięcia połączeń z bazami danych, zatrzymania wątków wykonywanych w tle, zapisania listy cookies lub wartości licznika odwiedzin w pliku na dysku, lub wykonania jakiekolwiek innych czynności porządkowych. Musisz jednak wiedzieć, że metoda ta może także spowodować awarię serwera WWW. W końcu nie wszystkie serwery WWW zostały stworzone w bezpiecznych językach programowania, takich jak Java (chodzi mi tu języki, których nazwy pochodzą od liter alfabetu), w których bez przeszkód można odczytywać i zapisywać informacje poza granicami tablic, wykonywać niedozwolone rzutowania typów lub korzystać z nieprawidłowych wskaźników powstałych na skutek nieudanych prób zwalniania pamięci. Co więcej, nawet technologia języka Java nie jest w stanie uchronić nas przed wyrwaniem z gniazdka przewodu zasilającego komputer. A zatem nie należy polegać na metodzie destroy, jako na jedynym sposobie zapisywania stanu serwletu na dysku. W przypadku liczenia odwiedzin, gromadzenia list wartości cookies oznaczających specjalne odwołania do stron, oraz wykonywania innych czynności tego typu, warto co jakiś czas zapisać na dysku aktualny stan.

2.7 Przykład użycia parametrów inicjalizacyjnych

Na listingu 2.8 przedstawiłem serwlet o nazwie `ShowMessage`, który w momencie inicjalizacji odczytuje wartości parametrów `message` oraz `repeats`. Wyniki wykonania tego serwletu, w przypadku gdy parametrowi `message` przypisana została wartość `Sinobrody`, a parametrowi `repeat` wartość `5`, przedstawiłem na rysunku 2.5. Pamiętaj, że choć serwlety odczytują parametry inicjalizacyjne w standardowy sposób, to jednak programiści muszą określać ich wartości w sposób zależy od używanego serwera. Wszelkie informacje dotyczące sposobu określania parametrów inicjalizacyjnych serwletów powinieneś znaleźć w dokumentacji serwera. Na listingu 2.9 przedstawiłem plik konfiguracyjny serwera Tomcat wykorzystany przy tworzeniu rysunku 2.5, listing 2.10 prezentuje analogiczny plik używany przez serwer JSWDK, natomiast rysunki 2.6 oraz 2.7 pokazują sposób w jaki można określać parametry inicjalizacyjne serwletów na Java Web Serverze. Wyniki wykonania serwletu są identyczne niezależnie od użytego serwera; przedstawiłem je na rysunku 2.5.

Listing 2.8 ShowMessage.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Przykład wykorzystania inicjalizacji servletu. W tym przypadku
 *  wyświetlany komunikat oraz liczba określająca ilość jego
 *  powtórzeń pobierana jest z parametrów inicjalizacyjnych.
 */

public class ShowMessage extends HttpServlet {
  private String message;
  private String defaultMessage = "Brak komunikatu.";
  private int repeats = 1;

  public void init(ServletConfig config)
      throws ServletException {
    // Zawsze wywołuj metodę super.init
    super.init(config);
    message = config.getInitParameter("message");
    if (message == null) {
      message = defaultMessage;
    }
    try {
      String repeatString = config.getInitParameter("repeats");
      repeats = Integer.parseInt(repeatString);
    } catch(NumberFormatException nfe) {
      // Wyjątek NumberFormatException obsługuje przypadki
      // gdy parametr repeatString jest równy null *i* gdy jego
      // wartość została zapisana w niewłaściwym formacie.
      // W oby tych przypadkach w razie przechwycenia wyjątku
      // nie należy niczego robić, gdyż poprzednia wartość
      // parametru repeatString (1) wciąż będzie używana. Wynika
      // to z faktu iż metoda Integer.parseInt zgłasza wyjątek
      // *przed* przypisaniem wartości do zmiennej.
    }
  }

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
      response.setContentType("text/html; charset=ISO-8859-2");
      PrintWriter out = response.getWriter();
      String title = "Serwlet ShowMessage";
      out.println(ServletUtilities.headWithTitle(title) +
                  "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                  "<H1 ALIGN=CENTER>" + title + "</H1>");
      for(int i=0; i<repeats; i++) {
        out.println(message + "<BR>");
      }
      out.println("</BODY></HTML>");
    }
}
```

Listing 2.9 web.xml (plik konfiguracyjny serwera Tomcat)

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
  <servlet>
    <servlet-name>
      ShowMsg
    </servlet-name>

    <servlet-class>
      coreservlets.ShowMessage
    </servlet-class>

    <init-param>
      <param-name>
        message
      </param-name>
      <param-value>
        Sinobrody
      </param-value>
    </init-param>

    <init-param>
      <param-name>
        repeats
      </param-name>
      <param-value>
        5
      </param-value>
    </init-param>
  </servlet>
</web-app>
```

Listing 2.10

```
# servlets.properties used in JSWDK

# Register servlet via servletName.code=servletClassFile
# You access it via http://host/examples/servlet/servletName
ShowMsg.code=coreservlets.ShowMessage

# Set init params via
#   servletName.initparams=param1=val1,param2=val2,...
ShowMsg.initparams=message=Sinobrody,repeats=5

# Standard setting
jsp.code=com.sun.jsp.runtime.JspServlet

# Set this to keep servlet source code built from JSP
jsp.initparams=keepgenerated=true
```

Rysunek 2.5 Serwlet ShowMessage wykorzystujące parametry inicjalizujące określane na serwerze.

Ze względu na fakt, iż proces definiowania parametrów inicjalizacyjnych jest zależy od serwera, warto zminimalizować ilość używanych parametrów. Dzięki temu zminimalizujesz także pracę, którą będziesz musiał wykonać podczas przenoszenia serwletów wykorzystujących parametry inicjalizacyjne z serwera na serwer. Jeśli serwlet musi pobrać wiele informacji, to zalecałbym zastosowanie rozwiązania polegającego na zapisaniu w parametrze inicjalizacyjnym wyłącznie nazwy pliku, w którym będą przechowywane informacje konieczne do inicjalizacji serwletu. Przykład zastosowania takiego rozwiązania przedstawię w podrozdziale 4.5, pt.: „Ograniczanie dostępu do stron WWW", gdzie parametr inicjalizacyjny określa wyłącznie położenie pliku z hasłami.

Metoda
W przypadkach, gdy proces inicjalizacji servletu jest złożony, staraj się zapisywać dane w pliku i używać parametrów inicjalizacyjnych wyłącznie do określenia jego położenia.

Na listingu 2.9 przedstawiłem plik konfiguracyjny używany do określania parametrów inicjalizacyjnych serwletów uruchamianych na serwerze Tomcat 3.0. W tym przypadku, wykorzystana metoda polega na skojarzeniu nazwy z plikiem klasowym serwletu, a następnie, na skojarzeniu parametrów inicjalizacyjnych z tą nazwą (a nie z plikiem klasowym). Plik konfiguracyjny przechowywany jest w katalogu *katalog_instalacyjny/webpages/WEB-INF*. Jeśli nie chce Ci się własnoręcznie tworzyć tego pliku, to pobierz go z serwera FTP Wydawnictwa HELION (*ftp://ftp.helion.pl/przyklady/jsjsp.zip*), zmodyfikuj i skopiuj do katalogu *katalog_instalacyjny/webpages/WEB-INF* na swoim komputerze.

Listing 2.10 przedstawia plik używany do określania parametrów inicjalizacyjnych serwletów, wykorzystywany przez serwer JSWDK. Podobnie jak w przypadku serwera Tomcat, także i tutaj w pierwszej kolejności z nazwą pliku klasowego serwletu kojarzona jest nazwa, a następnie, z tą nazwą są kojarzone parametry inicjalizacyjne. Plik ten należy umieścić w katalogu *katalog_instalacyjny/webpages/WEB-INF*.

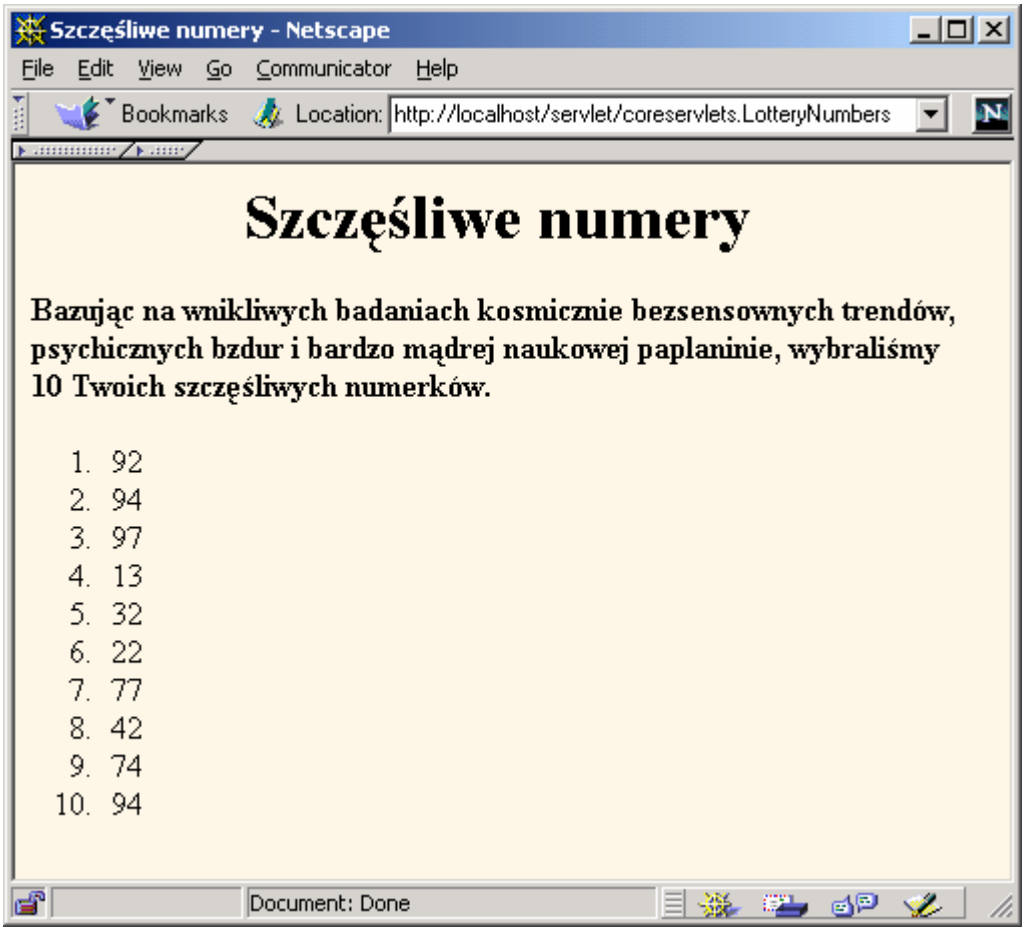
2.8 Przykład wykorzystania inicjalizacji serwletu i daty modyfikacji strony

Na listingu 2.11 przedstawiłem serwlet, którego metoda `init` wykonuje dwie czynności. Pierwszą z nich jest utworzenie dziesięcioelementowej tablicy liczb całkowitych. Liczby uzyskiwane są w wyniku bardzo skomplikowanych obliczeń i dlatego nie chcę, aby obliczenia te były powtarzane podczas obsługiwania każdego otrzymanego żądania. Z tego względu metoda `doGet` pobiera wartości obliczone w trakcie wykonywania metody `init` i przechowywane w tablicy, a nie generuje ich sama. Wyniki wykonania serwletu wykorzystującego tę metodę przedstawiłem na rysunku 2.8.

Listing 2.11 LotteryNumbers.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Przykład wykorzystujący inicjalizację apletów
 *  oraz metodę getLastModified
 */

public class LotteryNumbers extends HttpServlet {
  private long modTime;
  private int[] numbers = new int[10];

  /** Metoda init jest wywoływana wyłącznie raz,
   *  bezpośrednio po załadowaniu servletu do pamięci
   *  i przed obsłużeniem pierwszego żądania
   */

  public void init() throws ServletException {
    // Zaokrąglamy do pełnych sekund (1000 milisekund)
    modTime = System.currentTimeMillis()/1000*1000;
    for(int i=0; i<numbers.length; i++) {
      numbers[i] = randomNum();
    }
  }

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html; charset=ISO-8859-2");
    PrintWriter out = response.getWriter();
    String title = "Szczęśliwe numery";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<B>Bazując na wnikliwych badaniach" +
                "kosmicznie bezsensownych trendów, psychicznych bzdur " +
                "i bardzo mądrej naukowej paplaninie , " +
                "wybraliśmy " + numbers.length +
                " Twoich szczęśliwych numerków.</B>" +
                "<OL>");
    for(int i=0; i<numbers.length; i++) {
      out.println("  <LI>" + numbers[i]);
    }
    out.println("</OL>" +
                "</BODY></HTML>");
  }

  /** Standardowa metoda service porównuje tę datę
   *  z datą określoną w nagłówku żądania If-Modified-Since
   *  Jeśli data zwrócona przez metodę getLastModified jest
   *  późniejsza, lub jeśli nie ma nagłówka If-Modified-Since
   *  to metoda doGet jest wywoływana w zwyczajny sposób.
   *  Jeśli jednak data zwrócona przez metodę getLastModified
   *  jest taka sama lub wcześniejsza to metoda service wysyła
   *  do klienta odpowiedź 304 (Not Modified) - (nie zmieniony)
   *  i <B>nie</B> wywołuje metody doGet. W takiej sytuacji
```

```
   *  przeglądarka powinna użyć strony przechowywanej w pamięci
   *  podręcznej.
   */

  public long getLastModified(HttpServletRequest request) {
    return(modTime);
  }

  // Liczba pseudolosowa z zakresu od 0 do 99.

  private int randomNum() {
    return((int)(Math.random() * 100));
  }
}
```

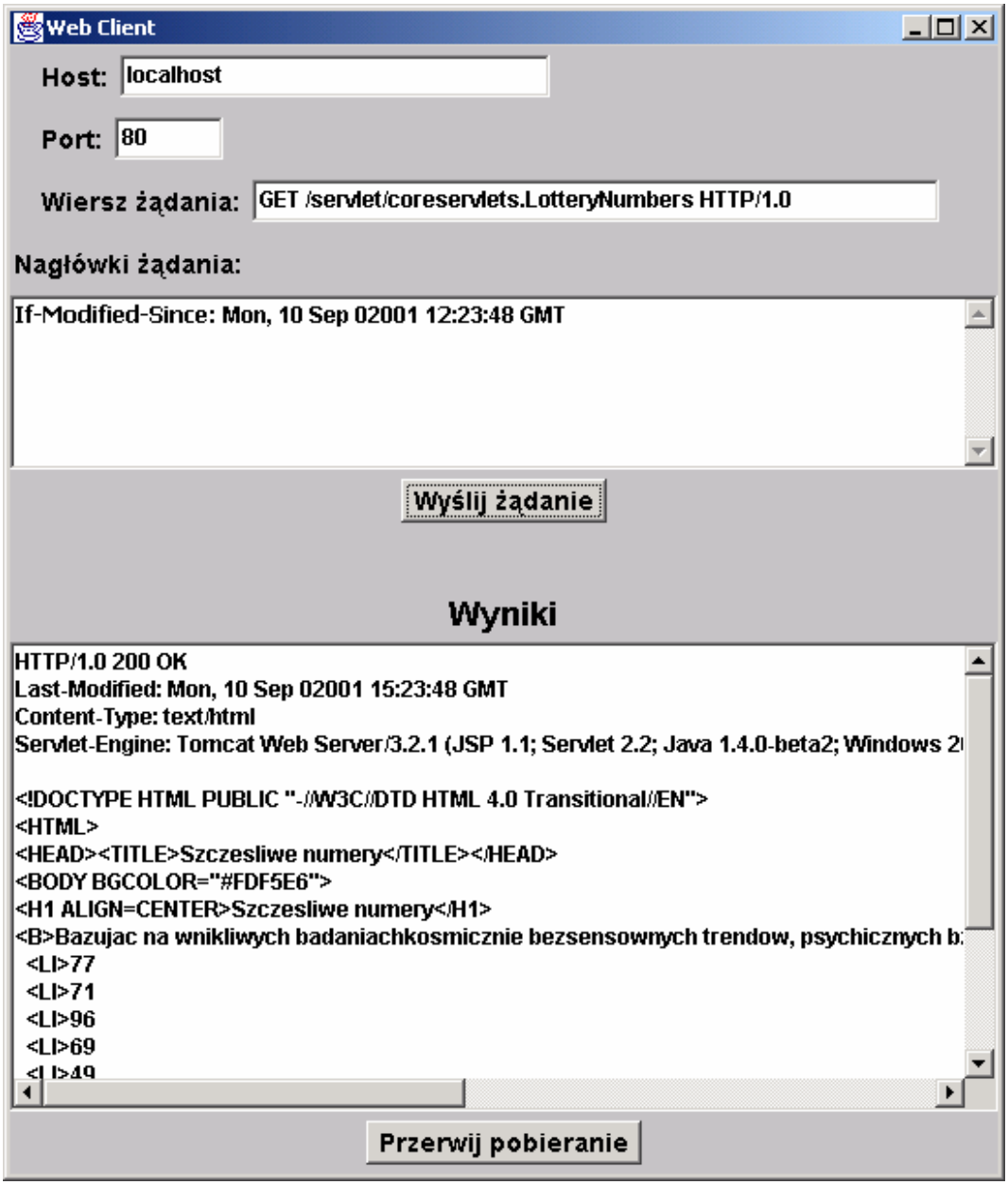


Rysunek 2.8 Wyniki wykonania serwletu LotteryNumbers.

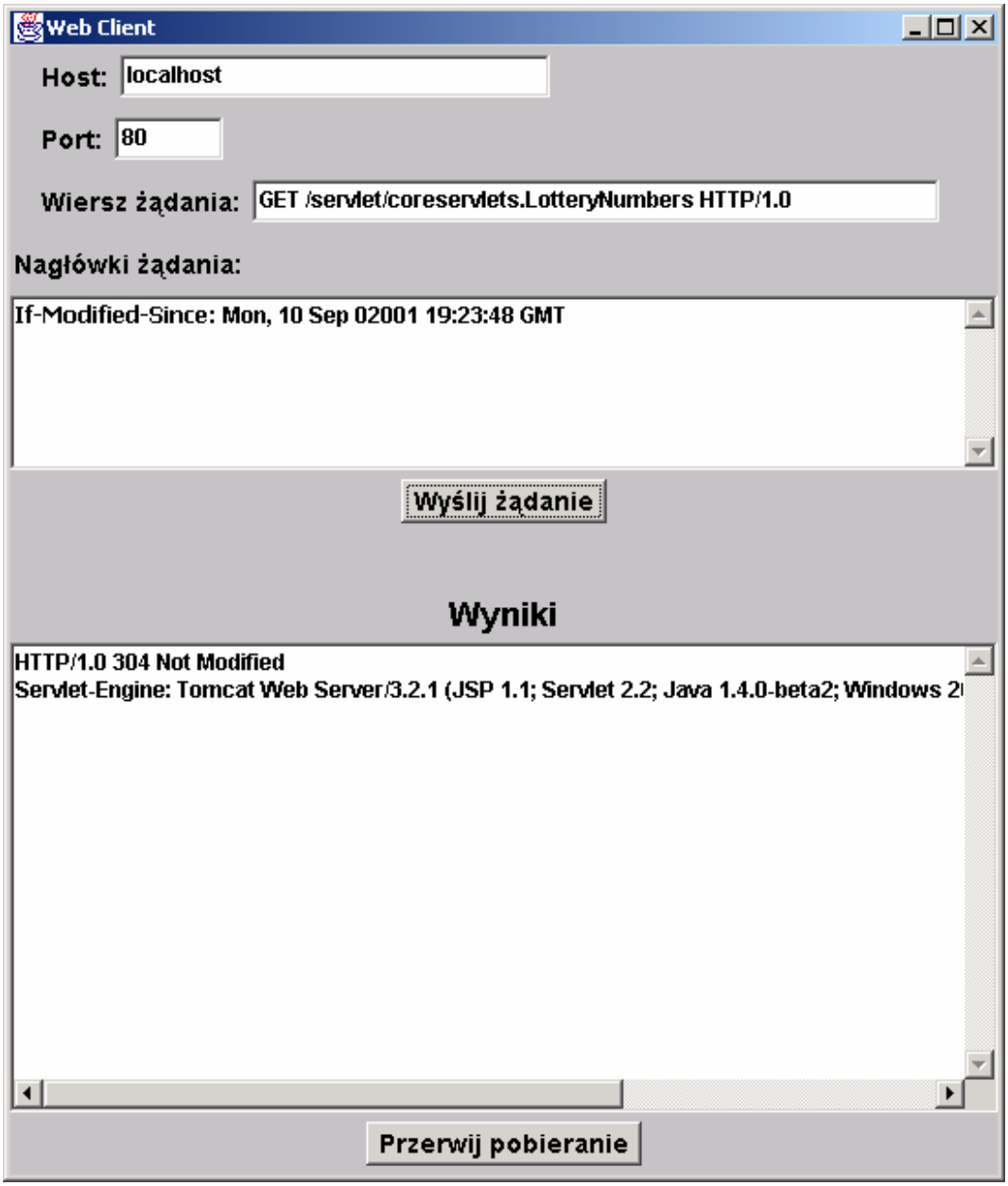
Niemniej jednak, w razie wykorzystania tej metody, wszyscy użytkownicy uzyskują te same wyniki i dlatego metoda `init` przechowuje także datę modyfikacji strony, która jest używana w metodzie `getLastModified`. Metoda ta powinna zwracać czas modyfikacji strony wyrażony jako ilość milisekund, które upłynęły od 1970 (to standardowy sposób określania dat w języku Java). Czas ten jest automatycznie konwertowany do postaci daty zapisanej w formacie GMT, odpowiadającej sposobowi zapisu wartości nagłówka `Last-Modified`. Ważniejsze jest jednak to, że jeśli serwer otrzyma warunkowe żądanie `GET` (zawierające nagłówek `If-Modified-Since` i informujące serwer, że klient poszukuje stron zmodyfikowanych po określonej dacie), to porówna je z datą zwróconą przez metodę `getLastModified` i prześle stronę wynikową wyłącznie w przypadku gdy została ona zmodyfikowana po dacie podanej w żądaniu. Przeglądarki często wykorzystują takie warunkowe żądania przy pobieraniu stron z pamięci podręcznej, a zatem obsługa żądań warunkowych nie tylko pomaga użytkownikom, lecz także ogranicza obciążenie

serwera. Nagłówki `Last-Modified` oraz `If-Modified-Since` wykorzystują wyłącznie pełne sekundy, dlatego też metoda `getLastModified` powinna zaokrąglać zwracany czas w dół, do pełnych sekund.

Rysunki 2.9 oraz 2.10 przedstawiają wyniki wykonania dwóch żądań dotyczących tego samego serwletu i zawierających nagłówki `If-Modified-Since` z dwoma różnymi datami. Aby określić nagłówki żądania i przeanalizować nagłówki odpowiedzi napisałem w języku Java specjalną aplikację o nazwie *WebClient*. Jej kod przedstawiony został w podrozdziale 2.10, pt.: „WebClient: Interaktywna wymiana informacji z serwerem WWW". Aplikacja ta pozwala własnoręcznie podawać nagłówki żądania HTTP, przesyłać je i przeanalizować otrzymane wyniki.



Rysunek 2.9 Uruchomienie serwletu LotteryNumbers przy wykorzystaniu bezwarunkowego żądania GET lub żądania warunkowego w którym podano datę wcześniejszą od momentu inicjalizacji serwletu, powoduje wygenerowanie normalnej strony WWW.



Rysunek 2.10 Uruchomienie serwletu LotteryNumbers przy wykorzystaniu żądania warunkowego w którym podano datę późniejszą od momentu inicjalizacji serwletu, powoduje wygenerowanie odpowiedzi 304 (Not Modified).

2.9 Testowanie serwletów

Oczywiście, Ty pisząc serwlety nigdy nie popełniasz błędów. Jednak, być może, któremuś z Twoich znajomych zdarza się popełniać przypadkowe pomyłki. W takim razie, możesz przekazać mu poniższe rady. Ale żarty na bok, testowanie serwletów jest w rzeczywistości trudnym zadaniem, gdyż nigdy nie są one wywoływane bezpośrednio. Wykonanie serwletów wyzwalane jest poprzez otrzymanie żądania HTTP, a same serwlety są wykonywane przez serwer WWW. Ze względu na ten zdalny sposób wykonywania serwletów, trudno jest umieszczać w ich kodzie punkty kontrolne oraz odczytywać komunikaty kontrolne lub stan stosu. Z tych powodów sposoby testowania serwletów różnią się nieco od sposobów testowania programów innych typów. Poniżej podałem kilka ogólnych strategii, które powinny ułatwić Ci życie:

1. **Przeanalizuj kod HTML.**

Jeśli wyniki wyświetlone w przeglądarce wyglądają śmiesznie, spróbuj wyświetlić kod źródłowy strony (wybierając w tym celu odpowiednią opcję[8]). Zdarza się, że drobny błąd w kodzie HTML — na przykład znacznik `<TABLE>` zamiast `</TABLE>` — sprawia, że znaczna część strony WWW staje się niewidoczna. Jeszcze lepszym rozwiązaniem jest skontrolowanie wyników generowanych przez serwlet przy użyciu jednego z serwisów sprawdzających poprawność kodu HTML. Więcej informacji na ten temat znajdziesz w podrozdziale 2.5, pt.: „Proste narzędzia pomocne przy tworzeniu dokumentów HTML".

2. **Przekazuj do klienta strony błędów.**

   Czasami tworząc serwlet można przewidzieć pojawianie się błędów pewnych określonych klas. W takim przypadkach serwlet powinien zwrócić szczegółowe informacje dotyczące zaistniałego problemu, bądź to w postaci zwyczajnej strony WWW, bądź przy wykorzystaniu metody `sendError` interfejsu `HttpServletResponse`. Szczegółowe informacje na temat metody `sendError` znajdziesz w rozdziale 6, pt.: „Generacja odpowiedzi: Kody statusu". Na przykład, powinieneś przewidzieć, że użytkownik może zapomnieć podać w formularzu jakiś koniecznych danych i zwracać stronę ze szczegółowymi informacjami jakich danych brakuje. Jednak generacja stron z informacjami o błędach nie zawsze jest możliwa. Czasami może pojawić się nieprzewidziany błąd, który doprowadzi do przerwania wykonywania serwletu. Informacje podane w pozostałych punktach pomogą Ci poradzić sobie w takich sytuacjach.

3. **Uruchamiaj serwer z poziomu wiersza poleceń.**

   Większość serwerów WWW wykonywanych jest jako procesy działające w tle, które bardzo często są uruchamiane w momencie uruchamiania systemu operacyjnego. Jeśli masz jakieś problemy z tworzonymi serwletami, możesz spróbować zamknąć serwer WWW i uruchomić go ponownie z poziomu wiersza poleceń. Dzięki temu, wywołania metod `System.out.println` oraz `System.err.printl` będzie można bez trudu odczytać w oknie, w którym został uruchomiony serwer. Jeśli serwlet nie będzie działał poprawnie, to w pierwszej kolejności powinieneś określić do jakiego miejsca serwlet jest wykonywany i zebrać informacje od kluczowych strukturach danych tuż przed momentem awarii. Wykorzystanie wywołań metody `println` do tego celu daje zaskakująco dobre rezultaty. Jeśli wykonujesz swoje serwlety na serwerze, którego nie da się w prosty sposób zatrzymać i ponownie uruchomić, to przetestuj je na własnym komputerze wykorzystując jeden z serwerów Tomcat, JSWDK lub Java Web Server.

4. **Korzystaj z pliku dziennika.**

   Klasa `HttpServlet` posiada metodę o nazwie `log` służącą do zapisywania informacji w pliku dziennika na serwerze. Odczytywanie komunikatów z pliku dziennika jest nieco mniej wygodne niż bezpośrednie obserwowanie ich na ekranie (opisane w poprzednim punkcie), jednak w tym przypadku nie jest konieczne zatrzymywanie i powtórne uruchamianie serwera. Dostępne są dwie wersje metody `log`. Pierwsza z nich wymaga podania argumentu typu `String` — czyli łańcucha znaków. Natomiast druga wymaga podania dwóch argumentów — pierwszy z nich jest typu `String`, a drugi typu `Throwable` (jest to klasa bazowa klasy `Exception`). Położenie pliku dziennika zależy od używanego serwera, zazwyczaj jednak informacje na ten temat powinieneś znaleźć w dokumentacji, a plik jest przeważnie przechowywany w jednym z podkatalogów katalogu instalacyjnego serwera.

5. **Sprawdź dane żądania HTTP.**

   Servlety odczytują dane z żądań HTTP, generują odpowiedź i przesyłają ją z powrotem do klienta. Jeśli coś w tym procesie działa nie tak jak należy, to będziesz się zapewne chciał dowiedzieć czy jest to spowodowane wysyłaniem nieprawidłowych informacji przez klienta, czy też błędnym przetwarzaniem ich

[8] W przypadku polskojęzycznej wersji Internet Explorera, należy wybrać opcję Widok⇨Źródło; w Netscape Navigatorze służy do tego opcja View⇨Page Source.

przez servlet. Klasa `EchoServer`, przedstawiona w podrozdziale 16.12, pt.: „Testowy serwer WWW", pozwala na przesłanie na serwer formularza HTML i otrzymanie wyników prezentujących informacje dokładnie w taki sposób, w jaki dotarły one na serwer.

6. **Sprawdź dane odpowiedzi HTTP.**

   Gdy już sprawdzisz poprawność żądania HTTP, warto, w podobny sposób, sprawdzić dane odpowiedzi. Klasa `WebClient`, przedstawiona w podrozdziale 2.10, pt.: „WebClient: Interaktywna wymiana informacji z serwerem WWW", pozwala na interaktywne nawiązanie połączenia z serwerem WWW, przesyłanie do niego własnoręcznie podanych nagłówków żądania HTTP i analizę wszelkich informacji otrzymanych w odpowiedzi (zarówno nagłówków odpowiedzi HTTP, jak i danych).

7. **Zatrzymaj i ponownie uruchom serwer WWW.**

   Większość najlepszych serwerów WWW wyposażonych w możliwość obsługi serwletów dysponuje specjalnym katalogiem przeznaczonym do przechowywania aktualnie tworzonych serwletów. Serwlety umieszczone w tym katalogu (w przypadku Java Web Servera jest to katalog *servlets*) są automatycznie przeładowywane w momencie gdy ich plik klasowy zostanie zmodyfikowany. Niemniej jednak, może się zdarzyć, że serwer WWW nie wykryje modyfikacji serwletu; dotyczy to przede wszystkim sytuacji gdy zostanie zmodyfikowana jedna z klas pomocniczych, a nie główna klasa serwletu. A zatem, jeśli okaże się, że zmiany wprowadzone w serwlecie nie odpowiadają jego zachowaniu, to będziesz musiał zatrzymać i powtórnie uruchomić serwer. W przypadku serwerów JSWDK oraz starszych wersji Tomcat, będziesz to musiał robić *za każdym* razem gdy zmodyfikujesz serwlet, gdyż te mini-serwery nie dysponują możliwością automatycznego przeładowywania serwletów.

2.10 WebClient: Interaktywna wymiana informacji z serwerem WWW

W tej części rozdziału przedstawię kod źródłowy programu *WebClient*, z którego korzystałem w podrozdziale 2.8, pt.: „Przykład wykorzystania inicjalizacji serwletu i daty modyfikacji strony" oraz o którym wspominałem w podrozdziale 2.9, pt.: „Testowanie serwletów". Z programu tego będę także bardzo często korzystał w rozdziale 16, pt.: „Formularze HTML". Kod tego programu, podobnie jak i kody wszystkich pozostałych przykładów przedstawionych w niniejszej książce, można skopiować z serwera FTP Wydawnictwa HELION (*ftp://ftp.helion.pl/przyklady/jsjsp.zip*) i używać bez żadnych ograniczeń.

WebClient

`WebClient` to główna klasa programu, z której będziesz korzystał. Program należy uruchamiać z poziomu wiersza poleceń, następnie, po pojawieniu się okna, można zmodyfikować wiersz żądania i nagłówki żądania HTTP i przesłać żądanie na serwer klikając przycisk *Wyślij żądanie*.

Listing 2.12 WebClient.java

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

/** Program graficzny pozwalający na interaktywne nawiązanie
 *  połączenia z serwerem WWW i przekazanie własnoręcznie podanych
 *  nagłówków żądań i wiersza żądania.
```

```
 */


public class WebClient extends CloseableFrame
    implements Runnable, Interruptible, ActionListener {
  public static void main(String[] args) {
    new WebClient("Web Client");
  }

  private LabeledTextField hostField, portField,
          requestLineField;
  private TextArea requestHeadersArea, resultArea;
  private String host, requestLine;
  private int port;
  private String[] requestHeaders = new String[30];
  private Button submitButton, interruptButton;
  private boolean isInterrupted = false;

  public WebClient(String title) {
    super(title);
    setBackground(Color.lightGray);
    setLayout(new BorderLayout(5, 30));
    int fontSize = 14;
    Font labelFont =
      new Font("Serif", Font.BOLD, fontSize);
    Font headingFont =
      new Font("SansSerif", Font.BOLD, fontSize+4);
    Font textFont =
      new Font("Monospaced", Font.BOLD, fontSize-2);
    Panel inputPanel = new Panel();
    inputPanel.setLayout(new BorderLayout());
    Panel labelPanel = new Panel();
    labelPanel.setLayout(new GridLayout(4,1));
    hostField = new LabeledTextField("Host:", labelFont,
                                     30, textFont);
    portField = new LabeledTextField("Port:", labelFont,
                                     "80", 5, textFont);
    // Dla zachowania zgodność z przeważającą większością
    // przeglądarek wykorzystamy protokół HTTP 1.0.
    // Jeśli będziesz chciał korzystać z protokołu HTTP 1.1
    // pamiętaj o konieczności generowania nagłówka odpowiedzi
    // Host:
    requestLineField =
      new LabeledTextField("Wiersz żądania:", labelFont,
                           "GET / HTTP/1.0", 50, textFont);
    labelPanel.add(hostField);
    labelPanel.add(portField);
    labelPanel.add(requestLineField);
    Label requestHeadersLabel =
      new Label("Nagłówki żądania:");
    requestHeadersLabel.setFont(labelFont);
    labelPanel.add(requestHeadersLabel);
    inputPanel.add(labelPanel, BorderLayout.NORTH);
    requestHeadersArea = new TextArea(5, 80);
    requestHeadersArea.setFont(textFont);
    inputPanel.add(requestHeadersArea, BorderLayout.CENTER);
    Panel buttonPanel = new Panel();
    submitButton = new Button("Wyślij żądanie");
    submitButton.addActionListener(this);
    submitButton.setFont(labelFont);
    buttonPanel.add(submitButton);
    inputPanel.add(buttonPanel, BorderLayout.SOUTH);
    add(inputPanel, BorderLayout.NORTH);
    Panel resultPanel = new Panel();
    resultPanel.setLayout(new BorderLayout());
    Label resultLabel =
      new Label("Wyniki", Label.CENTER);
    resultLabel.setFont(headingFont);
    resultPanel.add(resultLabel, BorderLayout.NORTH);
    resultArea = new TextArea();
    resultArea.setFont(textFont);
    resultPanel.add(resultArea, BorderLayout.CENTER);
    Panel interruptPanel = new Panel();
    interruptButton = new Button("Przerwij pobieranie");
    interruptButton.addActionListener(this);
    interruptButton.setFont(labelFont);
    interruptPanel.add(interruptButton);
```

```
    resultPanel.add(interruptPanel, BorderLayout.SOUTH);
    add(resultPanel, BorderLayout.CENTER);
    setSize(600, 700);
    setVisible(true);
  }

  public void actionPerformed(ActionEvent event) {
    if (event.getSource() == submitButton) {
      Thread downloader = new Thread(this);
      downloader.start();
    } else if (event.getSource() == interruptButton) {
      isInterrupted = true;
    }
  }

  public void run() {
    isInterrupted = false;
    if (hasLegalArgs())
      new HttpClient(host, port, requestLine,
              requestHeaders, resultArea, this);
  }

  public boolean isInterrupted() {
    return(isInterrupted);
  }

  private boolean hasLegalArgs() {
    host = hostField.getTextField().getText();
    if (host.length() == 0) {
      report("Brak nazwy hosta");
      return(false);
    }
    String portString =
      portField.getTextField().getText();
    if (portString.length() == 0) {
      report("Brak numeru portu");
      return(false);
    }
    try {
      port = Integer.parseInt(portString);
    } catch(NumberFormatException nfe) {
      report("Niedozwolony numer portu: " + portString);
      return(false);
    }
    requestLine =
      requestLineField.getTextField().getText();
    if (requestLine.length() == 0) {
      report("Brak wiersza żądania");
      return(false);
    }
    getRequestHeaders();
    return(true);
  }

  private void report(String s) {
    resultArea.setText(s);
  }

  private void getRequestHeaders() {
    for(int i=0; i<requestHeaders.length; i++)
      requestHeaders[i] = null;
    int headerNum = 0;
    String header =
      requestHeadersArea.getText();
    StringTokenizer tok =
      new StringTokenizer(header, "\r\n");
    while (tok.hasMoreTokens())
      requestHeaders[headerNum++] = tok.nextToken();
  }
}
```

HttpClient

Klasa `HttpClient` odpowiada za komunikację sieciową. Klasa ta przesyła na serwer podany wiersz żądania i nagłówki żądania, a następnie odczytuje po kolei wszystkie wiersze nadesłane z serwera i wyświetla je w polu `TextArea`, aż do momentu gdy serwer zamknie połączenie lub gdy działanie klasy `HttpClient` zostanie przerwane w wyniku ustawienia flagi `isInterrupted`.

Listing 2.13 HttpClient.java

```
import java.awt.*;
import java.net.*;
import java.io.*;

/** Bazowa klasa implementująca klienta sieciowego,
 *  używana w aplikacji WebClient.
 */

public class HttpClient extends NetworkClient {
  private String requestLine;
  private String[] requestHeaders;
  private TextArea outputArea;
  private Interruptible app;

  public HttpClient(String host, int port,
                    String requestLine, String[] requestHeaders,
                    TextArea outputArea, Interruptible app) {
    super(host, port);
    this.requestLine = requestLine;
    this.requestHeaders = requestHeaders;
    this.outputArea = outputArea;
    this.app = app;
    if (checkHost(host))
      connect();
  }

  protected void handleConnection(Socket uriSocket)
      throws IOException {
    try {
      PrintWriter out = SocketUtil.getWriter(uriSocket);
      BufferedReader in = SocketUtil.getReader(uriSocket);
      outputArea.setText("");
      out.println(requestLine);
      for(int i=0; i<requestHeaders.length; i++) {
        if (requestHeaders[i] == null)
          break;
        else
          out.println(requestHeaders[i]);
      }
      out.println();
      String line;
      while ((line = in.readLine()) != null &&
             !app.isInterrupted())
        outputArea.append(line + "\n");
      if (app.isInterrupted())
        outputArea.append("---- Pobieranie zostało przerwane ----");
    } catch(Exception e) {
      outputArea.setText("Błąd: " + e);
    }
  }

  private boolean checkHost(String host) {
    try {
      InetAddress.getByName(host);
      return(true);
    } catch(UnknownHostException uhe) {
      outputArea.setText("Zła nazwa hosta: " + host);
      return(false);
    }
  }
}
```

NetworkClient

`NetworkClient` jest klasą pomocniczą, którą można wykorzystywać przy tworzeniu wszystkich programów korzystających z połączeń sieciowych. Stanowi ona klasę bazową klasy `HttpClient`.

Listing 2.14 NetworkClient.java

```
import java.net.*;
import java.io.*;

/** Klasa wyjściowa do tworzenia klientów sieciowych (programów
 *  korzystających z komunikacji sieciowej). Będziesz
 *  musiał przesłonić metodę handleConnection, jednak
 *  w wielu przypadkach metoda connect może pozostać
 *  w niezmienionej postaci. Klasa wykorzystuje klasę
 *  SocketUtil aby uprościć proces tworzenia obiektów klas
 *  PrintWriter oraz BufferedReader.
 */

public class NetworkClient {
  protected String host;
  protected int port;

  /** Zarejestruj host i port. Połączenie sieciowe
   *  nie zostanie jednak nawiązane aż do momentu
   *  wywołania metody connect.
   */

  public NetworkClient(String host, int port) {
    this.host = host;
    this.port = port;
  }

  /** Nawiązuje połączenie, a następnie przekazuje gniazdo
   *  (socket) do metody handleConnection
   */

  public void connect() {
    try {
      Socket client = new Socket(host, port);
      handleConnection(client);
    } catch(UnknownHostException uhe) {
      System.out.println("Nieznany host: " + host);
      uhe.printStackTrace();
    } catch(IOException ioe) {
      System.out.println("IOException: " + ioe);
      ioe.printStackTrace();
    }
  }

  /** Tą metodę będziesz musiał przesłonić pisząc
   *  własny program korzystający z połączeń sieciowych.
   *  Domyślna wersja metody przesyła na serwer
   *  pojedynczy wiersz tekstu *Ogólny klient sieciowy*,
   *  odczytuje jeden wiersz wyników, wyświetla go
   *  i kończy działanie.
   */

  protected void handleConnection(Socket client)
      throws IOException {
    PrintWriter out =
      SocketUtil.getWriter(client);
    BufferedReader in =
      SocketUtil.getReader(client);
    out.println("Ogólny klient sieciowy");
    System.out.println
      ("Ogólny klient sieciowy:\n" +
       "Nawiązano połączenie z " + host +
       " i odebrano odpowiedź o postaci: '" + in.readLine() + "'.");
    client.close();
  }

  /** Nazwa komputera (hosta) dla serwera z którym nawiązujesz
```

```
   *  połączenie.
   */

  public String getHost() {
    return(host);
  }

  /** Numer portu na którym zostanie nawiązane połączenie. */

  public int getPort() {
    return(port);
  }
}
```

SocketUtil

`SocketUtil` to prosta klasa pomocnicza, ułatwiająca tworzenie niektórych typów strumieni wykorzystywanych w programach sieciowych. Jest ona wykorzystywana przez klasy `NetworkClient` oraz `HttpClient`.

Listing 2.15 SocketUtil.java

```
import java.net.*;
import java.io.*;

/** Uproszczony sposób tworzenia obiektów klas
 *  BufferedReader oraz PrintWriter skojarzonych z
 *  obiektem klasy Socket.
 */

public class SocketUtil {
  /** Buffreader ma odczytywać nadsyłane dane. */

  public static BufferedReader getReader(Socket s)
      throws IOException {
    return(new BufferedReader(
       new InputStreamReader(s.getInputStream())));
  }

  /** PrintWriter ma wysyłać informacje wyjściowe.
   *  W tym obiekcie klasy PrintWriter bufor wyjściowy
   *  będzie automatycznie opróżniany po wywołaniu metody
   *  println.
   */

  public static PrintWriter getWriter(Socket s)
      throws IOException {
    // drugi argument o wartości true oznacza, że należy
    // automatycznie opróżniać bufor wyjściowy
    return(new PrintWriter(s.getOutputStream(), true));
  }
}
```

CloseableFrame

`ClosableFrame` to rozszerzenie standardowej klasy `Frame`. Klasa ta została wyposażona w narzędzia pozwalające na zamknięcie okna, w momencie gdy użytkownik wyda takie polecenie. Główne okno programu `WebClient` jest obiektem właśnie tej klasy.

Listing 2.16 CloseableFrame.java

```
import java.awt.*;
import java.awt.event.*;

/** Klasa Frame, użytkownik może zamykać okna
 *  będące obiektami tej klasy. Punkt wyjściowy do
 *  tworzenia większości graficznych aplikacji w
 *  środowisku Java 1.1.
```

```
 */

public class CloseableFrame extends Frame {
  public CloseableFrame(String title) {
    super(title);
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
  }

  /** Dokonujemy trwałych modyfikacji, więc musimy
   *  w <B>pierwszej</B> kolejności wywołać metodę
   *  super.processWindowEvent.
   */

  public void processWindowEvent(WindowEvent event) {
    super.processWindowEvent(event);
    if (event.getID() == WindowEvent.WINDOW_CLOSING)
      System.exit(0);
  }
}
```

LabeledTextField

`LabeledTextField` to proste połączenie klas `TextField` oraz `Label`, którego używam w programie `WebClient`.

Listing 2.17 LabeledTextField.java

```
import java.awt.*;

/** Pole tekstowe (TextField) wraz z odpwiadającą mu etykietą (Label).
 */

public class LabeledTextField extends Panel {
  private Label label;
  private TextField textField;

  public LabeledTextField(String labelString,
                          Font labelFont,
                          int textFieldSize,
                          Font textFont) {
    setLayout(new FlowLayout(FlowLayout.LEFT));
    label = new Label(labelString, Label.RIGHT);
    if (labelFont != null)
      label.setFont(labelFont);
    add(label);
    textField = new TextField(textFieldSize);
    if (textFont != null)
      textField.setFont(textFont);
    add(textField);
  }

  public LabeledTextField(String labelString,
                          String textFieldString) {
    this(labelString, null, textFieldString,
         textFieldString.length(), null);
  }

  public LabeledTextField(String labelString,
                          int textFieldSize) {
    this(labelString, null, textFieldSize, null);
  }

  public LabeledTextField(String labelString,
                          Font labelFont,
                          String textFieldString,
                          int textFieldSize,
                          Font textFont) {
    this(labelString, labelFont,
         textFieldSize, textFont);
    textField.setText(textFieldString);
  }
```

```
  /** Etykieta (Label) znajdująca się po lewej stronie pola
   *  LabeledTextField.
   *  Aby obsłużyć etykietę wykorzystaj kod o następującej
   *  postaci:
   *  <PRE>
   *    LabeledTextField ltf = new LabeledTextField(...);
   *    ltf.getLabel().metodaObslugiEtykiety(...);
   *  </PRE>
   *
   *  patrz metoda getTextField
   */

  public Label getLabel() {
    return(label);
  }

  /** Pole tekstowe (TextField) znajdujące się po prawej
   *  stronie pola LabeledTextField.
   *
   *  patrz metoda getLabel
   */

  public TextField getTextField() {
    return(textField);
  }
}
```

Interruptible

`Interruptible` to bardzo prosty interfejs używany do oznaczania klas dysponujących metodą `isInterrupted`. Jest on używany w klasie `HttpClient` w celu sprawdzenia czy użytkownik nie zażądał przerwania transmisji.

Listing 2.18 Interruptible.java

```
/** Interfejs przeznaczony dla klas, które można sprawdzać
 *  czy użytkownik nie zażądał przerwania wykonywanych czynności.
 *  Używany przez klasy HttpClient oraz WebClient aby umożliwić
 *  użytkownikowi przerwanie połączenia sieciowego.
 */

public interface Interruptible {
  public boolean isInterrupted();
}
```

Rozdział 3.
Obsługa żądań: Dane przesyłane z formularzy

Jedną z głównych przyczyn tworzenia stron WWW w dynamiczny sposób jest chęć generowania ich zawartości na podstawie danych przesłanych przez użytkownika. W tym rozdziale dowiesz się w jaki sposób można korzystać z takich danych.

3.1 Znaczenie informacji przesyłanych z formularzy

Jeśli kiedykolwiek korzystałeś z mechanizmów wyszukiwawczych, odwiedziłeś internetową księgarnię, czy też próbowałeś zarezerwować bilet lotniczy za pośrednictwem Internetu, to prawdopodobnie widziałeś już te śmiesznie wyglądające adresy URL, takie jak *http://host/strona?user=Marty+Hall&origin=bwi&dest=lax*. Ścieżka podana po znaku zapytania (w naszym przypadku jest to: *user=Marty+Hall&origin=bwi&dest=lax*) określana jest jako *dane formularza*, *dane zapytania* bądź *łańcuch zapytania* i stanowi najpopularniejszy sposób przekazywania informacji ze strony WWW do programu działającego na serwerze. Dane formularza można przesyłać na serwer na dwa sposoby. Pierwszy z nich, stosowany w przypadku żądań `GET` polega na dopisaniu ich do adresu URL, po znaku zapytania (jak pokazałem na powyższym przykładzie). Drugi sposób, stosowany w przypadku żądań `POST` polega na przesłaniu tych danych w osobnym wierszu. Jeśli jeszcze nie wiesz wiele o formularzach HTML, to w rozdziale 16 (pt.: „Formularze HTML") znajdziesz szczegółowe informacje na temat ich tworzenia.

Pobieranie potrzebnych informacji spośród danych przesłanych z formularza, było zazwyczaj najbardziej mozolnym fragmentem programów CGI. Przede wszystkim, należało w inny sposób odczytywać dane przesyłane metodą `GET` (w tradycyjnych programach CGI dane przesyłane w ten sposób były zazwyczaj zapisywane w zmiennej środowiskowej `QUERY_STRING`) niż dane przesyłane metodą `POST` (w tradycyjnych programach CGI należało w tym celu odczytywać standardowy strumień wejściowy). Po drugie, pobrany łańcuch znaków należało podzielić na pary (w miejscach wystąpienia znaków „`&`"), a dla każdej z tak uzyskanych par, określić nazwę parametru (znajdującą się z lewej strony znaku równości) oraz jego wartość (umieszczoną z prawej strony znaku równości). Po trzecie, konieczne było zdekodowanie danych przesyłanych w formacie URL. Znaki alfanumeryczne są przesyłane w niezmienionej postaci, jednak odstępy są zamieniane na znaki plus („`+`"), a wszystkie pozostałe znaki są zapisywane w formacie `%XX`, gdzie `XX` to wartość znaku w kodzie ASCII (lub ISO Latin-1) zapisana w postaci dwucyfrowej liczby szesnastkowej. Program działający na serwerze musiał odwrócić ten proces. Na przykład, jeśli w formularzu HTML, w polu tekstowym o nazwie `osoby` użytkownik wpisał łańcuch znaków „`~hall, ~gates, i ~inni`", to dane te zostaną przesłane w postaci „`osoby=%7Ehall%2C+%7Egates%2C+i+%7Einni`", a program działający na serwerze musiał przekształcić dane do oryginalnej postaci. I w końcu czwartym powodem, dla którego analiza danych przesyłanych z formularzy była uciążliwa, był

fakt, iż wartości pól można pomijać (na przykład: „`param1=wart1&`**`param2=`**`&param3=wart3`"), bądź też jedno pole może mieć więcej niż jedną wartość (na przykład: „**`param1=wart11`**`&param2=wart2&`**`param1=wart12`**"). A zatem, kod analizujący dane przesyłane z formularzy musi uwzględniać i odpowiednio obsługiwać te przypadki.

3.2 Odczytywanie danych formularzy w serwletach

Jedną z miłych cech serwletów jest to, iż przetwarzanie danych formularzy odbywa się automatycznie. Aby pobrać wartość parametru, wystarczy wywołać metodę `getParameter` interfejsu `HttpServletRequest`, podając jako argument jego nazwę (przy czym wielkość znaków ma w tym przypadku znaczenie). Metody `getParameter` używa się w identyczny sposób, niezależnie od tego czy dane zostały przesłane z formularza metodą `GET` czy też `POST`. Serwlet wie jaka metoda została użyta do przesłania danych i automatycznie wykonuje konieczne czynności, w sposób niewidoczny dla programisty. Wartość zwracana przez metodę `getParameter` jest łańcuchem znaków (obiektem typu `String`) odpowiadającym zdekodowanej wartości pierwszego wystąpienia pary parametr-wartość, o podanej nazwie. Jeśli parametr o podanej nazwie istnieje lecz jest pusty, to metoda zwraca pusty łańcuch znaków; natomiast jeśli parametru nie ma, to zwracana jest wartość `null`. Jeśli istnieje prawdopodobieństwo, że parametr może posiadać więcej niż jedną wartość, to zamiast metody `getParameter` należy użyć metody `getParameterValues` (zwracającej nie pojedynczy łańcuch znaków lecz tablicę łańcuchów). Metoda ta zwraca wartość `null` jeśli parametru o podanej nazwie nie ma, lub tablicę jednoelementową gdy podany parametr ma tylko jedną wartość.

Przy określaniu nazw parametrów uwzględniana jest wielkość liter, oznacza to, że wywołań `request.getParameter("param1")` oraz `request.getParameter("Param1")` nie można używać zamiennie.

Ostrzeżenie
W argumentach metod `getParameter` oraz `getParameterValues` wielkość liter ma znaczenie.

I ostatnia sprawa. Choć większość serwletów poszukuje grupy parametrów o konkretnych nazwach, to jednak do celów testowych warto czasami pobrać pełną ich listę. Do tego celu służy metoda `getParameterNames`, zwracająca listę nazw parametrów w formie obiektu `Enumeration`. Każdy element znajdujący się na tej liście można rzutować do typu `String` i użyć w wywołaniu metody `getParameter` bądź `getParameterValues`. Warto tylko zapamiętać, że API interfejsu `HttpServletRequest` nie określa kolejności w jakiej nazwy poszczególnych parametrów zostaną zapisane o obiekcie `Enumeration`.

Ostrzeżenie
Nie możesz liczyć na to, że metoda `getParameterNames` zwróci nazwy parametrów w jakiejkolwiek, określonej kolejności.

3.3 Przykład: Odczyt trzech konkretnych parametrów

Na listingu 3.1 przedstawiłem prosty serwlet o nazwie `ThreeParams`, który odczytuje wartości trzech parametrów (o nazwach `param1`, `param2` oraz `param3`) i wyświetla ja na stronie w postaci listy wypunkowanej. Listing 3.2 przedstawia formularz HTML na którym można podać wartości tych trzech parametrów i przesłać je do serwletu. Dzięki przypisaniu atrybutowi `ACTION` formularza wartości `/servlet/coreservlets.ThreeParams` formularz ten może zostać zainstalowany w dowolnym miejscu, na serwerze, na którym uruchamiany jest serwlet. Katalog zawierający formularz, nie musi być w żaden specjalny sposób skojarzony z katalogiem, w którym przechowywany jest serwlet. Pamiętasz zapewne, że miejsce przeznaczone do umieszczania

dokumentów HTML zależy od używanego serwera WWW. W przypadku serwerów JSWDK 1.0.1 oraz Tomcat 3.0, dokumenty te można umieszczać w katalogu *katalog_instalacyjny/webpages* lub w jego podkatalogach. Aby uzyskać dostęp do tych dokumentów, należy podać adres *http://komputer/ścieżka/nazwa_pliku.html*. Na przykład, zakładając, że formularz przedstawiony na listingu 3.2 został zapisany w pliku *katalog_instalacyjny/webpages/forms/ThreeParamsForm.html* i chcemy go wyświetlić na tym samym komputerze na którym działa serwer, to adres URL, którego powinniśmy użyć będzie miał postać *http://localhost/forms/ThreeParamsForm.html*.

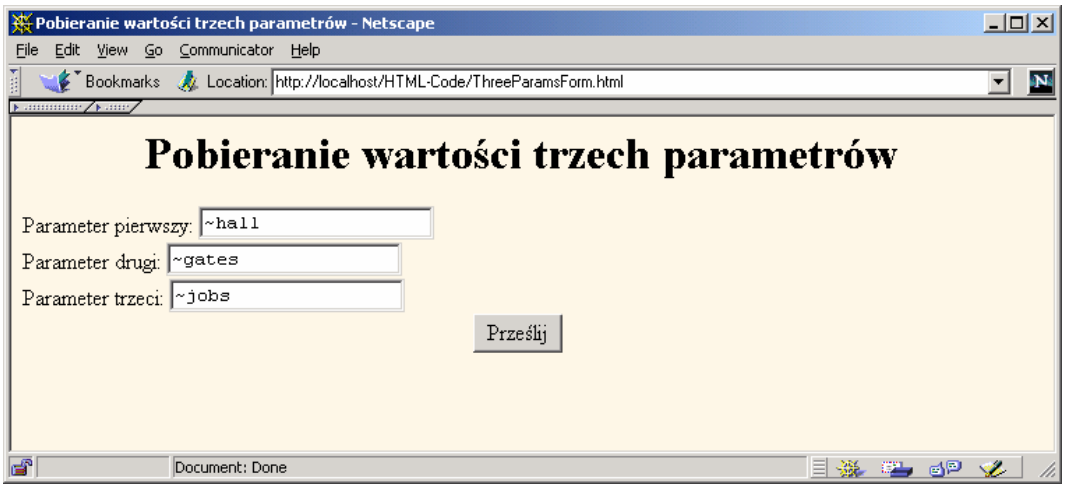
Listing 3.1 ThreeParams.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Prosty servlet odczytujący wartości trzech parametrów
 *  przesłanych z formularza.
 */

public class ThreeParams extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html; charset=ISO-8859-2");
    PrintWriter out = response.getWriter();
    String title = "Odczyt trzech parametrów";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<UL>\n" +
                "  <LI><B>param1</B>: "
                + request.getParameter("param1") + "\n" +
                "  <LI><B>param2</B>: "
                + request.getParameter("param2") + "\n" +
                "  <LI><B>param3</B>: "
                + request.getParameter("param3") + "\n" +
                "</UL>\n" +
                "</BODY></HTML>");
  }
}
```

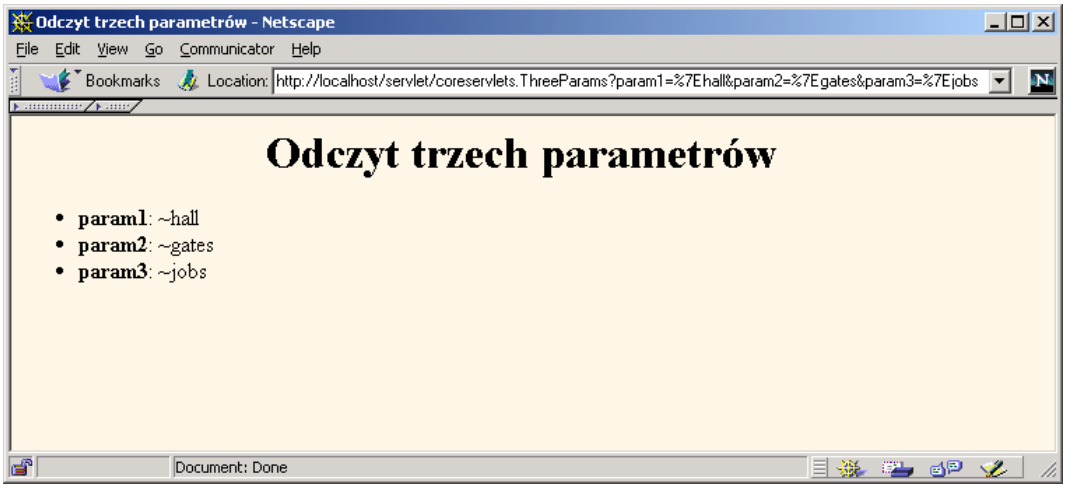
Listing 3.2 ThreeParamsForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Pobieranie wartości trzech parametrów</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">Pobieranie wartości trzech parametrów</H1>

<FORM ACTION="/servlet/coreservlets.ThreeParams">
  Parameter pierwszy:  <INPUT TYPE="TEXT" NAME="param1"><BR>
  Parameter drugi: <INPUT TYPE="TEXT" NAME="param2"><BR>
  Parameter trzeci:  <INPUT TYPE="TEXT" NAME="param3"><BR>
  <CENTER>
    <INPUT TYPE="SUBMIT" VALUE="Prześlij">
  </CENTER>
</FORM>

</BODY>
</HTML>
```

Rysunki 3.1 oraz 3.2 przedstawiają odpowiednio wygląd formularza oraz wyniki wygenerowane przez serwlet.



Rysunek 3.1 Wygląd formularza zapisanego w pliku ThreeParamsForm.html



Rysunek 3.2 Wyniki wykonania serwletu ThreeParams

Choć ustawienia *odpowiedzi* trzeba podać przed rozpoczęciem generowania wyników serwletu (patrz rozdziały 6 i 7), to jednak nie ma żadnych wymagań dotyczących momentu odczytywania parametrów *żądania*.

Jeśli jesteś przyzwyczajony do rozwiązań wykorzystywanych przy tworzeniu standardowych programów CGI, gdzie informacje przesyłane metodą `POST` odczytywane są ze standardowego strumienia wejściowego, powinieneś wiedzieć że dokładnie to samo można zrobić w serwlecie. W tym celu należy wywołać metodę `getReader` lub `getInputStream` interfejsu `HttpServletRequest`, a następnie pobrać dane wejściowe posługując się metodami uzyskanego strumienia. Niemniej jednak rozwiązanie takie nie jest najlepsze gdyż informacje wejściowe nie są ani przetworzone (czyli nie zostały wydzielone poszczególne pary nazwa-wartość, ani nie zostały określone nazwy i wartości poszczególnych parametrów) ani odpowiednio zdekodowane (czyli znaki „+” nie zostały zamienione na odstępy, a wyrażenia `%XX` na odpowiadające im znaki w kodzie ASCII lub ISO Latin-1). Jednak odczytywanie takich nieprzetworzonych informacji może być przydatne w przypadku obsługi plików przesyłanych na serwer lub danych przesyłanych metodą `POST` z programu, a nie z formularzy HTML. Należy jednak pamiętać, iż w przypadku wykorzystania tego sposobu odczytu danych przesyłanych metodą `POST`, wartości poszczególnych parametrów nie można pobierać przy użyciu metody `getParameter`.

3.4 Przykład: Odczyt wszystkich parametrów

W poprzednim przykładzie pobieraliśmy wartości parametrów przesyłanych z formularza na podstawie ściśle określonych i z góry znanych nazw. Dodatkowo założyliśmy, że każdy z parametrów będzie miał tylko jedną wartość. W tym podrozdziale przedstawię serwlet, który pobiera nazwy *wszystkich* parametrów przesłanych z formularza, następnie pobiera ich wartości i wyświetla je w formie tabeli. Serwlet w szczególny sposób oznacza parametry, dla których nie podano wartości oraz parametry z wieloma wartościami.

W pierwszej kolejności serwlet pobiera nazwy wszystkich parametrów wywołując w tym celu metodę `getParameterNames` interfejsu `HttpServletRequest`. Metoda ta zwraca obiekt `Enumeration` zawierający listę nazw wszystkich parametrów (przy czym ich kolejność nie jest określona). Następnie serwlet wykonuje pętlę, która w standardowy sposób pobiera wszystkie nazwy parametrów przechowywane w obiekcie `Enumeration`. Do sterowania wykonywaniem pętli wykorzystywane są dwie metody interfejsu `Enumeration` — `hasMoreElements` służąca do określenia kiedy należy przerwać pętlę oraz `nextElement` używana do pobierania następnego element listy. Metoda `nextElement` zwraca obiekt klasy `Object`, który serwlet rzutuje do obiektu `String` i przekazuje do wywołania metody `getParameterValues`. Jak wiemy, metoda ta zwraca tablicę łańcuchów znaków. Jeśli tablica ta ma tylko jeden element będący pustym łańcuchem znaków, oznacza to, że wartość parametru nie została określona; w takiej sytuacji serwlet wyświetla kursywą tekst "`Brak danych`". Jeśli tablica zawiera więcej niż jeden element, to świadczy to o tym, iż parametr miał kilka wartości; w takim przypadku serwlet wyświetla wszystkie wartości parametru, przedstawiając je w formie listy wypunktowanej. W pozostałych przypadkach, w tablicy wyświetlana jest wartość parametru. Kod źródłowy serwletu został przedstawiony na listingu 3.3. Listingu 3.4 przedstawia natomiast kod źródłowy strony WWW, której można użyć do przetestowania serwletu. Na rysunkach 3.3 oraz 3.4 zostały przedstawione odpowiednio: strona WWW służąca do testowania serwletu oraz wygenerowane przez niego wyniki.

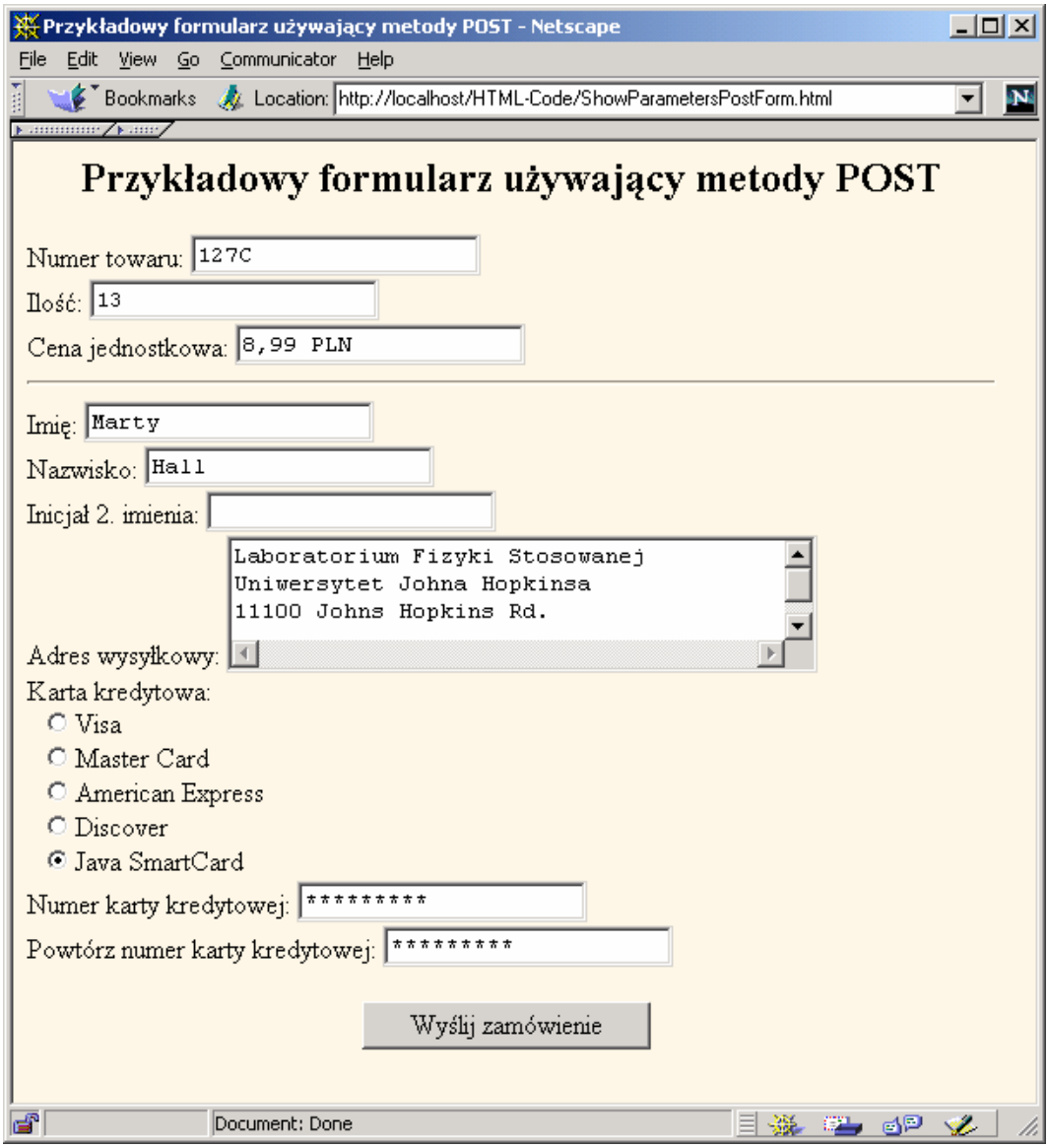
Listing 3.3 ShowParameters.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

/** Wyświetla wszystkie parametry przesłane do servletu
 *  zarówno żądaniami GET jak i POST. Parametry nie posiadające
 *  żadnej wartości lub posiadające kilka wartości są
 *  oznaczane w szczególny sposób.
 */

public class ShowParameters extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html; charset=ISO-8859-2");
    PrintWriter out = response.getWriter();
    String title = "Odczyt wszystkich parametrów";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<TABLE BORDER=1 ALIGN=CENTER>\n" +
                "<TR BGCOLOR=\"#FFAD00\">\n" +
                "<TH>Nazwa parametru<TH>Wartość/wartości parametru");
    Enumeration paramNames = request.getParameterNames();
    while(paramNames.hasMoreElements()) {
      String paramName = (String)paramNames.nextElement();
      out.print("<TR><TD>" + paramName + "\n<TD>");
      String[] paramValues =
        request.getParameterValues(paramName);
      if (paramValues.length == 1) {
        String paramValue = paramValues[0];
```

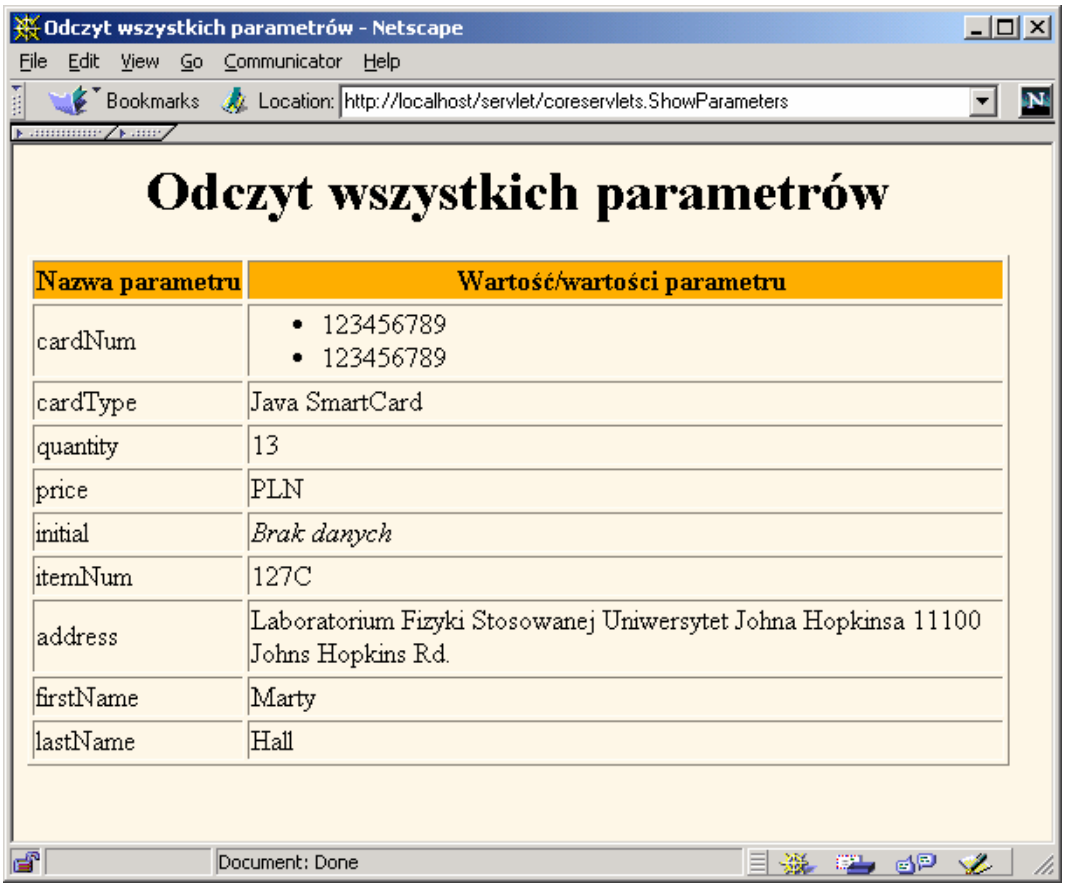
```
        if (paramValue.length() == 0)
          out.println("<I>Brak danych</I>");
        else
          out.println(paramValue);
      } else {
        out.println("<UL>");
        for(int i=0; i<paramValues.length; i++) {
          out.println("<LI>" + paramValues[i]);
        }
        out.println("</UL>");
      }
    }
    out.println("</TABLE>\n</BODY></HTML>");
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

Listing 3.4 ShowParametersPostForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Przykładowy formularz używający metody POST</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<H2 ALIGN="CENTER">Przykładowy formularz używający metody POST</H2>

<FORM ACTION="/servlet/coreservlets.ShowParameters" METHOD="POST">
  Numer towaru: <INPUT TYPE="TEXT" NAME="itemNum"><BR>
  Ilość: <INPUT TYPE="TEXT" NAME="quantity"><BR>
  Cena jednostkowa: <INPUT TYPE="TEXT" NAME="price" VALUE=" PLN"><BR>
  <HR>
  Imię: <INPUT TYPE="TEXT" NAME="firstName"><BR>
  Nazwisko: <INPUT TYPE="TEXT" NAME="lastName"><BR>
  Inicjał 2. imienia: <INPUT TYPE="TEXT" NAME="initial"><BR>
  Adres wysyłkowy:
  <TEXTAREA NAME="address" ROWS=3 COLS=40></TEXTAREA><BR>
  Karta kredytowa:<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                     VALUE="Visa">Visa<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                     VALUE="Master Card">Master Card<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                     VALUE="Amex">American Express<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                     VALUE="Discover">Discover<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
                     VALUE="Java SmartCard">Java SmartCard<BR>
  Numer karty kredytowej:
  <INPUT TYPE="PASSWORD" NAME="cardNum"><BR>
  Powtórz numer karty kredytowej:
  <INPUT TYPE="PASSWORD" NAME="cardNum"><BR><BR>
  <CENTER>
    <INPUT TYPE="SUBMIT" VALUE="Wyślij zamówienie">
  </CENTER>
</FORM>

</BODY>
</HTML>
```



Rysunek 3.3 Formularz HTML służący do podawania informacji przekazywanych do serwletu ShowParameters.



Rysunek 3.4 Wyniki wykonania serwletu ShowPrameters.

3.5 Serwis rejestracji życiorysów

Ostatnio znaczną popularność na WWW uzyskały serwisy ułatwiające znalezienie pracy. Znane i popularne witryny stanowią przydatne narzędzie dla osób poszukujących pracy, gdyż reklamują ich umiejętności; stanowią one także wygodne narzędzie dla pracodawców dając im informacje o szerokim gronie potencjalnych pracowników. W tej części rozdziału przedstawiłem serwlet obsługujący fragment takiej witryny — serwis służący do rejestracji życiorysów.

Listing 3.5 oraz rysunek 3.5 przedstawiają formularz stanowiący interfejs użytkownika naszego serwletu. Jeśli jeszcze nie znasz się na formularzach HTML, to szczegółowe informacje na ich temat znajdziesz w rozdziale 16. W przypadku tego formularza, należy zwrócić uwagę iż dane przesyłane są na serwer przy użyciu metody `POST`, a sam formularz służy do podania wartości poniższych parametrów:

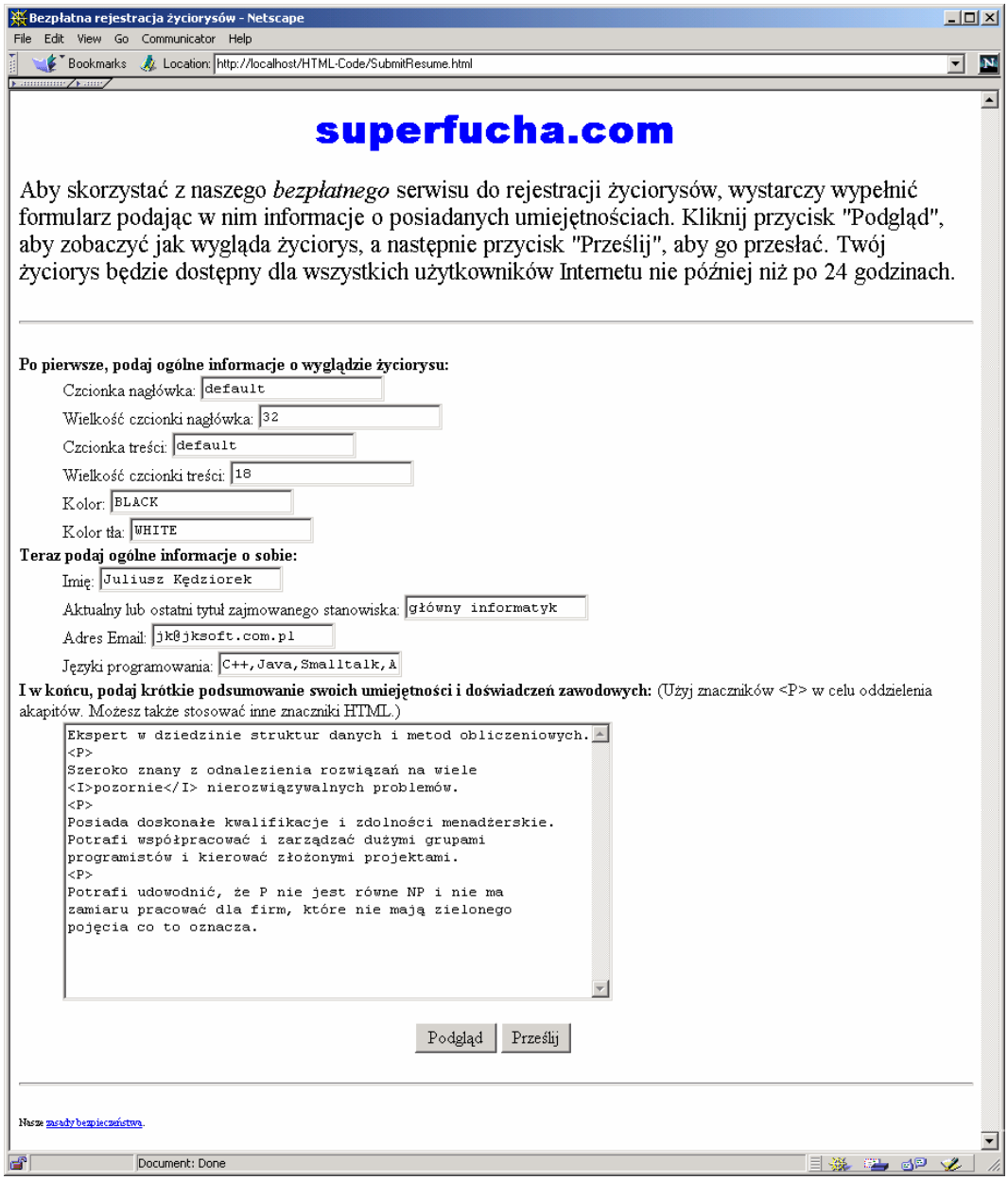
- **headingFont**
  Nagłówek strony zostanie wyświetlony tą czcionką. W przypadku podania wartości „default” zostanie użyta czcionka bezszeryfowa, taka jak Arial lub Helvetica.
- **headingSize**
  Nazwa osoby zostanie wyświetlona czcionką o tej wielkości punktowej. Nagłówki niższego stopnia będą wyświetlane nieco mniejszą czcionką.
- **bodyFont**
  Tą czcionką będzie wyświetlany zwyczajny tekst (znane języki i umiejętności).
- **bodySize**
  Zwyczajny tekst będzie wyświetlany czcionką o tej wielkości punktowej.
- **fgColor**
  Określa kolor tekstu.

- **bgColor**
  Określa kolor tła strony.
- **name**
  Ten parametr określa imię osoby podającej swój życiorys. Imię będzie wyświetlone na środku strony przy użyciu czcionki o określonym kroju i wielkości.
- **title**
  Ten parametr określa tytuł stanowiska osoby podającej swój życiorys. Zostanie on wyświetlony na środku strony, poniżej imienia osoby, przy wykorzystaniu czcionki o nieco mniejszej wielkości.
- **email**
  Adres poczty elektronicznej osoby podającej swój życiorys. Zostanie on wyświetlony poniżej tytułu stanowiska i umieszczony w hiperpołączeniu typu `mailto`.
- **language**
  Języki programowania podane w tym polu zostaną wyświetlone na stronie prezentującej życiorys, w formie listy punktowej.
- **skills**
  Tekst podany w tym wielowierszowym polu tekstowym zostanie wyświetlony na końcu strony prezentującej życiorys, poniżej nagłówka „Umiejętności i doświadczenia".

Listing 3.5 SubmitResume.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<!--
Interfejs użytkownika dla servletu obsługującego przeglądanie
i przechowywanie życiorysów nadsyłanych przez użytkowników.
-->
<HTML>
<HEAD>
  <TITLE>Bezpłatna rejestracja życiorysów</TITLE>
  <LINK REL=STYLESHEET
        HREF="jobs-site-styles.css"
        TYPE="text/css">
</HEAD>
<BODY>
<H1>superfucha.com</H1>
<P CLASS="LARGER">
Aby skorzystać z naszego <I>bezpłatnego</I> serwisu do rejestracji
życiorysów, wystarczy wypełnić formularz podając w nim informacje
o posiadanych umiejętnościach. Kliknij przycisk "Podgląd", aby
zobaczyć jak wygląda życiorys, a następnie przycisk "Prześlij", aby
go przesłać. Twój życiorys będzie dostępny dla wszystkich
użytkowników Internetu nie później niż po 24 godzinach.</P>
<HR>
<FORM ACTION="/servlet/coreservlets.SubmitResume"
      METHOD="POST">
<DL>
<DT><B>Po pierwsze, podaj ogólne informacje o wyglądzie życiorysu:</B>
<DD>Czcionka nagłówka:
    <INPUT TYPE="TEXT" NAME="headingFont" VALUE="default">
<DD>Wielkość czcionki nagłówka:
    <INPUT TYPE="TEXT" NAME="headingSize" VALUE=32>
<DD>Czcionka treści:
    <INPUT TYPE="TEXT" NAME="bodyFont" VALUE="default">
<DD>Wielkość czcionki treści:
    <INPUT TYPE="TEXT" NAME="bodySize" VALUE=18>
<DD>Kolor:
    <INPUT TYPE="TEXT" NAME="fgColor" VALUE="BLACK">
<DD>Kolor tła:
    <INPUT TYPE="TEXT" NAME="bgColor" VALUE="WHITE">

<DT><B>Teraz podaj ogólne informacje o sobie:</B>
<DD>Imię: <INPUT TYPE="TEXT" NAME="name">
<DD>Aktualny lub ostatni tytuł zajmowanego stanowiska:
    <INPUT TYPE="TEXT" NAME="title">
<DD>Adres Email: <INPUT TYPE="TEXT" NAME="email">
```

```
<DD>Języki programowania:
    <INPUT TYPE="TEXT" NAME="languages">

<DT><B>I w końcu, podaj krótkie podsumowanie swoich umiejętności
    i doświadczeń zawodowych:</B> (Użyj znaczników &lt;P&gt;
    w celu oddzielenia akapitów. Możesz także stosować inne
    znaczniki HTML.)
<DD><TEXTAREA NAME="skills"
              ROWS=15 COLS=60 WRAP="SOFT"></TEXTAREA>
</DL>
  <CENTER>
    <INPUT TYPE="SUBMIT" NAME="previewButton" Value="Podgląd">
    <INPUT TYPE="SUBMIT" NAME="submitButton" Value="Prześlij">
  </CENTER>
</FORM>
<HR>
<P CLASS="TINY">Nasze <A HREF="securitypolicy.html">
zasady bezpieczeństwa</A>.</P>
</BODY>
</HTML>
```



Rysunek 3.5 Formularz przekazujący dane do serwletu SubmitResume

Listing 3.6 przedstawia kod serwletu przetwarzającego dane przesyłane z formularza HTML. Po kliknięciu przycisku *Podgląd* serwlet odczytuje wartości parametrów określających krój i wielkość czcionek. Zanim serwlet użyje wartości tych parametrów, sprawdza czy nie są one równe `null` (co może się zdarzyć, na przykład, gdy popełniono błąd przy tworzeniu formularza HTML i dlatego poszukiwany parametr nie został podany) lub czy są one pustymi łańcuchami znaków (co może się zdarzyć gdy użytkownik usunie z pola formularza jego wartość domyślną, lecz nie poda żadnej innej). W takich przypadkach serwlet wykorzysta domyślne wartości parametrów. Wartości parametrów które mają być liczbami całkowitymi, są przekazywane do wywołania metody `Integer.parseInt`. Aby zabezpieczyć się przed przypadkami podania liczby całkowitej zapisanej w nieodpowiednim formacie, wywołanie metody `Integer.parseInt` zostało umieszczone wewnątrz bloku `try /catch`; dzięki temu, gdy liczba będzie zapisana niepoprawnie, serwlet zastosuje odpowiednią wartość domyślną. Choć na pierwszy rzut oka może się wydawać, iż obsługa takich sytuacji jest nieco uciążliwa, to jednak dzięki wykorzystaniu metod pomocniczych, takich jak `replaceIfMissing` oraz `replaceIfMissingOrDefault` przedstawionych na listingu 3.6, wykonanie tych czynności wcale nie jest aż tak pracochłonne. Niezależnie od tego obsługa sytuacji szczególnych jest pracochłonna czy też nie, użytkownicy będą od czasu do czasu zapominać o podaniu wartości pola lub błędnie zrozumieją format w jakim jego wartość ma zostać zapisana. Z tego względu, odpowiednia obsługa niepoprawnie podanych wartości parametrów jest zagadnieniem kluczowym, podobnie zresztą jak przetestowanie działania serwletu w sytuacjach gdy zostaną do niego przekazane zarówno poprawne, jak i niepoprawne informacje.

Metoda
Projektuj swoje serwlety w taki sposób, aby brakujące parametry oraz parametry o błędnie zapisanych wartościach, były odpowiednio obsługiwane. Przetestuj działanie servletów przekazując do nich poprawne, jak również niepoprawne dane.

Po określeniu poprawnych wartości wszystkich parametrów związanych z krojem i wielkością czcionek, serwlet tworzy na ich podstawie kaskadowy arkusz stylów. Jeśli jeszcze nie słyszałeś niczego arkuszach stylów, to wiedz, iż w języku HTML 4.0 stanowią one standardowy sposób określania krojów, wielkości i kolorów czcionek, odległości pomiędzy wyrazami i literami, oraz wszelkich innych informacji związanych z formatowaniem. Arkusze stylów są zazwyczaj umieszczane w odrębnych plikach; jednak w naszym przypadku wygodniej będzie umieścić go bezpośrednio w generowanej stronie WWW. W tym celu posłużymy się elementem `STYLE`. Więcej informacji na temat kaskadowych arkuszy stylów znajdziesz pod adresem *http://www.w3.org/TR/REC-CSS1*.

Po stworzeniu arkusza stylów serwlet wyświetla imię osoby podającej życiorys, tytuł zajmowanego przez nią stanowiska oraz jej adres poczty elektronicznej. Informacje te wyśrodkowane na stronie i wyświetlone jedna pod drugą. Do ich prezentacji serwlet użyje czcionki nagłówka, a adres poczty elektronicznej zostanie dodatkowo umieszczony w hiperpołączeniu typu `mailto`, dzięki czemu pracodawca będzie się mógł bezpośrednio skontaktować z daną osobą klikając ten adres. Języki programowania podane w polu *Języki programowania* są przetwarzane przy użyciu metod klasy `StringTokenizer` (zakładam przy tym, że poszczególne języki są od siebie oddzielone odstępami lub przecinkami), a następnie wyświetlane na stronie w formie listy punktowej umieszczonej poniżej nagłówka „Języki programowania”. I w końcu, tekst przekazany jak wartość parametru `skills` jest wyświetlany u dołu strony, poniżej nagłówka „Umiejętności i doświadczenia”.

Przykładowe wyniki działania serwletu zostały przedstawione na rysunkach 3.6, 3.7 oraz 3.8. Listing 3.7 prezentuje kod HTML strony z rysunku 3.6.

Listing 3.6 SubmitResume.java

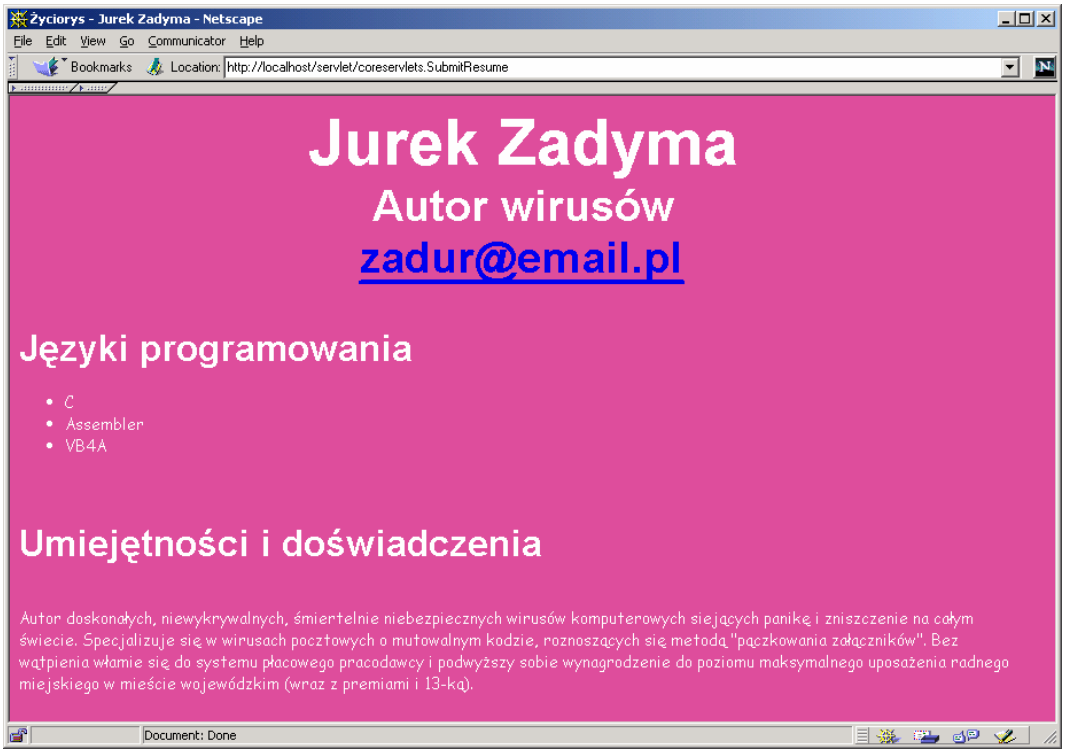
```
package coreservlets;
```

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

/** Servlet obsługujący przeglądanie i przechowywanie
 *  życiorysów przesyłanych przez osoby poszukujące pracy.
 */

public class SubmitResume extends HttpServlet {
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html; charset=ISO-8859-2");
    PrintWriter out = response.getWriter();
    if (request.getParameter("previewButton") != null) {
      showPreview(request, out);
    } else {
      storeResume(request);
      showConfirmation(request, out);
    }
  }

  /* Wyświetla podgląd nadesłanego życiorysu. Pobiera
   * informacje o czcionce i na jego podstawie tworzy
   * arkusz stylów; następnie pobiera informacje
   * życiorysu i prezentuje je na stronie. Wygląd
   * prezentowanych informacji określa arkusz stylów.
   */

  private void showPreview(HttpServletRequest request,
                           PrintWriter out) {
    String headingFont = request.getParameter("headingFont");
    headingFont = replaceIfMissingOrDefault(headingFont, "");
    int headingSize =
      getSize(request.getParameter("headingSize"), 32);
    String bodyFont = request.getParameter("bodyFont");
    bodyFont = replaceIfMissingOrDefault(bodyFont, "");
    int bodySize =
      getSize(request.getParameter("bodySize"), 18);
    String fgColor = request.getParameter("fgColor");
    fgColor = replaceIfMissing(fgColor, "BLACK");
    String bgColor = request.getParameter("bgColor");
    bgColor = replaceIfMissing(bgColor, "WHITE");
    String name = request.getParameter("name");
    name = replaceIfMissing(name, "Lou Zer");
    String title = request.getParameter("title");
    title = replaceIfMissing(title, "ob.");
    String email = request.getParameter("email");
    email =
      replaceIfMissing(email, "contact@superfucha.com");
    String languages = request.getParameter("languages");
    languages = replaceIfMissing(languages, "<I>Brak</I>");
    String languageList = makeList(languages);
    String skills = request.getParameter("skills");
    skills = replaceIfMissing(skills, "Niewiele.");
    out.println
      (ServletUtilities.DOCTYPE + "\n" +
       "<HTML>\n" +
       "<HEAD>\n" +
       "<TITLE>Życiorys - " + name + "</TITLE>\n" +
       makeStyleSheet(headingFont, headingSize,
                      bodyFont, bodySize,
                      fgColor, bgColor) + "\n" +
       "</HEAD>\n" +
       "<BODY>\n" +
       "<CENTER>\n"+
       "<SPAN CLASS=\"HEADING1\">" + name + "</SPAN><BR>\n" +
       "<SPAN CLASS=\"HEADING2\">" + title + "<BR>\n" +
       "<A HREF=\"mailto:" + email + "\">" + email +
           "</A></SPAN>\n" +
       "</CENTER><BR><BR>\n" +
       "<SPAN CLASS=\"HEADING3\">Języki programowania" +
       "</SPAN>\n" +
       makeList(languages) + "<BR><BR>\n" +
       "<SPAN CLASS=\"HEADING3\">Umiejętności i doświadczenia" +
       "</SPAN><BR><BR>\n" +
```

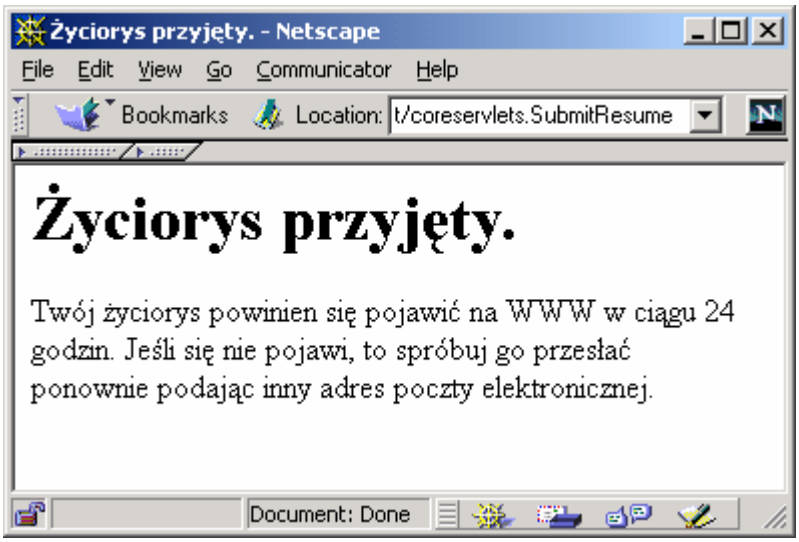
```
      skills + "\n" +
      "</BODY></HTML>");
  }

  /* Metoda tworzy kaskadowy arkusz stylów zawierający informacje
   * o trzech poziomach nagłówków oraz kolorze tła i tekstu.
   * W przypadku Internet Explorera kolor połączenia mailto
   * jest zmieniany po umieszczeniu na nim wskaźnika myszy.
   */

  private String makeStyleSheet(String headingFont,
                                int heading1Size,
                                String bodyFont,
                                int bodySize,
                                String fgColor,
                                String bgColor) {
    int heading2Size = heading1Size*7/10;
    int heading3Size = heading1Size*6/10;
    String styleSheet =
      "<STYLE TYPE=\"text/css\">\n" +
      "<!--\n" +
      ".HEADING1 { font-size: " + heading1Size + "px;\n" +
      "            font-weight: bold;\n" +
      "            font-family: " + headingFont +
                     "Arial, Helvetica, sans-serif;\n" +
      "}\n" +
      ".HEADING2 { font-size: " + heading2Size + "px;\n" +
      "            font-weight: bold;\n" +
      "            font-family: " + headingFont +
                     "Arial, Helvetica, sans-serif;\n" +
      "}\n" +
      ".HEADING3 { font-size: " + heading3Size + "px;\n" +
      "            font-weight: bold;\n" +
      "            font-family: " + headingFont +
                     "Arial, Helvetica, sans-serif;\n" +
      "}\n" +
      "BODY { color: " + fgColor + ";\n" +
      "       background-color: " + bgColor + ";\n" +
      "       font-size: " + bodySize + "px;\n" +
      "       font-family: " + bodyFont +
                   "Times New Roman, Times, serif;\n" +
      "}\n" +
      "A:hover { color: red; }\n" +
      "-->\n" +
      "</STYLE>";
    return(styleSheet);
  }

  /* Zastępuje nieistnijące (null) łańcuchy znaków
   * (gdy parametr nie został podany) oraz puste łańcuchy
   * znaków (w polu tekstowym niczego nie wpisano),
   * podaną wartością domyślną. W przeciwnym razie, zwraca
   * oryginalny łańcuch znaków.
   */

  private String replaceIfMissing(String orig,
                                  String replacement) {
    if ((orig == null) || (orig.length() == 0)) {
      return(replacement);
    } else {
      return(orig);
    }
  }

  /* Zastępuje nieistniejące (null) łańcuchy znaków, puste
   * łańcuchy oraz łańcuch o wartości "default", podanym
   * zamiennikiem. W przeciwnym razie zwraca oryginalny
   * łańcuch znaków.
   */

  private String replaceIfMissingOrDefault(String orig,
                                           String replacement) {
    if ((orig == null) ||
        (orig.length() == 0) ||
        (orig.equals("default"))) {
      return(replacement);
    } else {
```

```
      return(orig + ", ");
    }
  }

  /* Pobiera wartość całkowitą zapisaną w formie łańcucha
   * znaków i zwraca ją w formie liczby całkowitej. Jeśli
   * łańcuch znaków wynosi null lub została zapisana w
   * niewłaściwym formacie, zwracana jest wartość domyślna.
   */

  private int getSize(String sizeString, int defaultSize) {
    try {
      return(Integer.parseInt(sizeString));
    } catch(NumberFormatException nfe) {
      return(defaultSize);
    }
  }

  /* Łańcuch wejściowy "Java,C++,Lisp", "Java C++ Lisp" lub
   * "Java, C++, Lisp", powoduje wygenerowanie kodu HTML:
   * "<UL>
   *   <LI>Java
   *   <LI>C++
   *   <LI>Lisp
   *  </UL>"
   */

  private String makeList(String listItems) {
    StringTokenizer tokenizer =
      new StringTokenizer(listItems, ", ");
    String list = "<UL>\n";
    while(tokenizer.hasMoreTokens()) {
      list = list + "  <LI>" + tokenizer.nextToken() + "\n";
    }
    list = list + "</UL>";
    return(list);
  }

  /* Wyświetla stronę potwierdzenia gdy zostanie
   * kliknięty przycisk "Prześlij".
   */

  private void showConfirmation(HttpServletRequest request,
                                PrintWriter out) {
    String title = "Życiorys przyjęty.";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY>\n" +
                "<H1>" + title + "</H1>\n" +
                "Twój życiorys powinien się pojawić na WWW\n" +
                "w ciągu 24 godzin. Jeśli się nie pojawi, \n" +
                "to spróbuj go przesłać ponownie podając inny\n" +
                "adres poczty elektronicznej.\n" +
                "</BODY></HTML>");
  }

  /* Dlaczego nie należy przesyłać swojego adresu poczty
   * elektronicznej witrynom którym nie wiadomo czy można
   * zaufać.
   */

  private void storeResume(HttpServletRequest request) {
    String email = request.getParameter("email");
    putInSpamList(email);
  }

  private void putInSpamList(String emailAddress) {
    // Na wszelki wypadek usunąłem ten kod.
  }
}
```

Rysunek 3.6 Wyniki wygenerowane przez serwlet SubmitResume po kliknięciu przycisku Podgląd



Rysunek 3.7 Inna, potencjalna postać wyników wygenerowanych przez serwlet SubmitResume



Rysunek 3.8 Wyniki wygenerowane przez serwlet SubmitResume po kliknięciu przycisku Prześlij

Listing 3.7 Kod źródłowy strony wygenerowanej przez serwlet SubmitResume i przedstawionej na rysunku 3.6

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Życiorys - Juliusz K?dziorek</TITLE>
<STYLE TYPE="text/css">
<!--
.HEADING1 { font-size: 32px;
            font-weight: bold;
            font-family: Arial, Helvetica, sans-serif;
}
.HEADING2 { font-size: 22px;
            font-weight: bold;
            font-family: Arial, Helvetica, sans-serif;
}
.HEADING3 { font-size: 19px;
            font-weight: bold;
            font-family: Arial, Helvetica, sans-serif;
}
BODY { color: BLACK;
       background-color: WHITE;
       font-size: 18px;
       font-family: Times New Roman, Times, serif;
}
A:hover { color: red; }
-->
</STYLE>
</HEAD>
<BODY>
<CENTER>
<SPAN CLASS="HEADING1">Juliusz Kędziorek</SPAN><BR>
<SPAN CLASS="HEADING2">Główny informatyk<BR>
<A HREF="mailto:jk@jksoft.com.pl">jk@jksoft.com.pl</A></SPAN>
</CENTER><BR><BR>
<SPAN CLASS="HEADING3">Języki programowania</SPAN>
<UL>
  <LI>C++
  <LI>Java
  <LI>Smalltalk
  <LI>Ada
</UL><BR><BR>
<SPAN CLASS="HEADING3">Umiejętności i doświadczenia</SPAN><BR><BR>
Ekspert w dziedzinie struktur danych i metod obliczeniowych.
<P>
Szeroko znany z odnalezienia rozwiązań na wiele <I>pozornie</I> nierozwiązalnych problemów.
<P>
Posiada doskonałe kwalifikacje i zdolności menadżerskie. Potrafi współpracować i zarządzać
dużymi grupami programistów i kierować złożonymi projektami.
<P>
```

```
Potrafi udowodnić, że P nie jest równe NP i nie ma zamiaru pracować dla firm, które nie mają
zielonego pojęcia co to oznacza.

</BODY></HTML>
```

3.6 Filtrowanie łańcuchów w poszukiwaniu znaków specjalnych HTML

Zazwyczaj, gdy serwlet będzie chciał wygenerować kod HTML zawierający znaki < lub >, użyje zamiast nich standardowych symboli HTML — `<` oraz `>`. Podobnie, gdy serwlet chce umieścić cudzysłów lub znak & wewnątrz wartości atrybutu znacznika HTML, to zastosuje symbole HTML `&quote;` oraz `&`. Zastąpienie symboli HTML zwykłymi znakami może doprowadzić do powstania błędów w kodzie strony. Nawiasy < i > mogą bowiem zostać zinterpretowane jako fragmenty znaczników, cudzysłów w wartości atrybutu może zostać zrozumiany jako jej koniec, a znaki & są po prostu niedozwolonymi wartościami atrybutów. W większości przypadków łatwo jest odszukać znaki specjalne i zastąpić je symbolami HTML. Jednak w dwóch przypadkach ręczne wykonanie takiej zamiany nie jest proste.

Po pierwsze dotyczy to sytuacji gdy łańcuch znaków został otrzymany w wyniku wykonania fragmentu programu lub pochodzi z jakiegoś innego, zewnętrznego źródła i jest już zapisany w jakimś standardowym formacie. W takich przypadkach odszukanie i własnoręczne zastąpienie wszystkich znaków specjalnych może być uciążliwym i męczącym zajęciem. Jeśli jednak tego nie zrobimy to wynikowa strona WWW może zawierać błędnie sformatowane fragmenty lub jej części mogą być w ogóle niewidoczne (patrz rysunek 3.9 w dalszej części rozdziału).

Drugim przypadkiem kiedy ręczna konwersja zawodzi, są sytuacje gdy łańcuch znaków został przesłany z formularza HTML. Jest oczywiste, że w tym przypadku konwersja znaków specjalnych musi być przeprowadzona w czasie wykonywania programu, gdyż dane przesłane z formularza nie są znane podczas kompilacji serwletu. W przypadku stron WWW, które nie są ogólnie dostępne, jeśli użytkownik prześle łańcuch znaków zawierający znaki specjalne, to pominięcie ich konwersji może doprowadzić do wygenerowania strony WWW zawierającej błędne lub niewidoczne fragmenty. Pominięcie konwersji znaków specjalnych w przypadku witryny dostępnej dla ogółu użytkowników Internetu, może pozwolić na wykorzystanie strony do przeprowadzenia *ataku skryptowego przeprowadzanego pomiędzy witrynami*. Atak tego typu polega na umieszczeniu parametrów przesyłanych metodą `GET` w adresie URL odwołującym się do jednego z Twoich serwletów. Po przetworzeniu parametry te zostają zamienione na znacznik `<SCRIPT>`, który z kolei wykorzystuje znane błędy przeglądarek. Dzięki umieszczeniu kodu w adresie URL i rozpowszechnianiu nie strony WWW, lecz właśnie tego adresu, napastnik utrudnia rozpoznanie swej tożsamości, a co więcej, może wykorzystać relacje zaufania, aby przekonać użytkowników, że skrypt pochodzi z zaufanego źródła (czyli Twojego serwletu). Więcej informacji na ten temat można znaleźć na stronach *http://www.cert.org/advisories/CA-2000-02.html* oraz *http://www.microsoft.com/TechNet/itsolutions/security/topics/exsumcs.asp*.

Implementacja filtrowania

Zastąpienie znaków <, >, " oraz & w łańcuchach znaków jest prostym zadaniem, które można wykonać na wiele różnych sposobów. Koniecznie należy jednak pamiętać, że łańcuchy znaków w języku Java są niemodyfikowalne; a zatem, konkatenacja łańcuchów wiąże się z kopiowaniem i zwalnianiem z pamięci wielu krótkich łańcuchów. Przykładowo, przeanalizujmy poniższy fragment kodu:

```
String s1 = "Witam";
String s2 = s1 + " was!";
```

Łańcuch znaków `s1` nie może zostać zmodyfikowany, a zatem, podczas wykonywania drugiego wiersza kodu tworzona jest nowa kopia łańcucha s1, do której zostaje dodany łańcuch `" was!"`. Kopia ta jest następnie niszczona. Aby uniknąć strat związanych z tworzeniem tych tymczasowych obiektów (określanych jako „śmieci") należy wykorzystać strukturę danych, której wartości można modyfikować. W tym przypadku oczywistym rozwiązaniem jest zastosowanie klasy `StringBuffer`. Listing 3.8 przedstawia statyczną metodę `filter`, która wykorzystuje obiekt `StringBuffer` do efektywnego kopiowania znaków z łańcucha źródłowego do buforu wynikowego, jednocześnie odpowiednio konwertując cztery znaki specjalne HTML.

Listing 3.8 SevletUtilities.java

```
package coreservlets;

import javax.servlet.*;
import javax.servlet.http.*;

public class ServletUtilities {

  // ...
  // Inne metody klasy ServletUtilities pokazane w innych miejscach
  // ...

  /** Ta metoda, w przekazanym łańcuchu znaków zamienia wszystkie
   *  wystąpienia znaku '<' kombinacją znaków '&lt;' oraz wszystkie
   *  wystąpienia znaku '>' kombinacją znaków '&gt;' oraz (aby
   *  poprawnie obsługiwać przypadku wystąpienia tych znaków w wartościach
   *  atrybutów) wszystkie wystąpienia cudzysłowów na kombinacje znaków
   *  '&quote', wszystkie wystąpienia '&' na '&amp;'.
   *  Bez zastosowania filtrowania tego typu, żaden dowolny łańcuch
   *  znaków nie może być wstawiony do kodu strony WWW.
   */

  public static String filter(String input) {
    StringBuffer filtered = new StringBuffer(input.length());
    char c;
    for(int i=0; i<input.length(); i++) {
      c = input.charAt(i);
      if (c == '<') {
        filtered.append("&lt;");
      } else if (c == '>') {
        filtered.append("&gt;");
      } else if (c == '"') {
        filtered.append("&quot;");
      } else if (c == '&') {
        filtered.append("&amp;");
      } else {
        filtered.append(c);
      }
    }
    return(filtered.toString());
  }
}
```

Przykład

Aby zaprezentować znacznie konwersji znaków specjalnych rozważmy przykład serwletu, który ma generować stronę WWW zawierającą przedstawiony poniżej listing:

```
if (a<b) {
  zrobTo();
} else {
  zrobCosInnego();
}
```

Gdyby powyższy fragment kodu został umieszczony w kodzie strony WWW w przedstawionej postaci, to przeglądarka zinterpretowałaby `<b` jako początek znacznika HTML, a dalsza część kodu aż do pierwszego znaku `>` zostałaby uznana za niepoprawną zawartość tego znacznika. Na listingu 3.9 przedstawiłem serwlet generujący powyższy fragment kodu, a na rysunku 3.9 nienajlepsze wyniki jego wykonania. Listing 3.10 przedstawia serwlet w którym jedyna

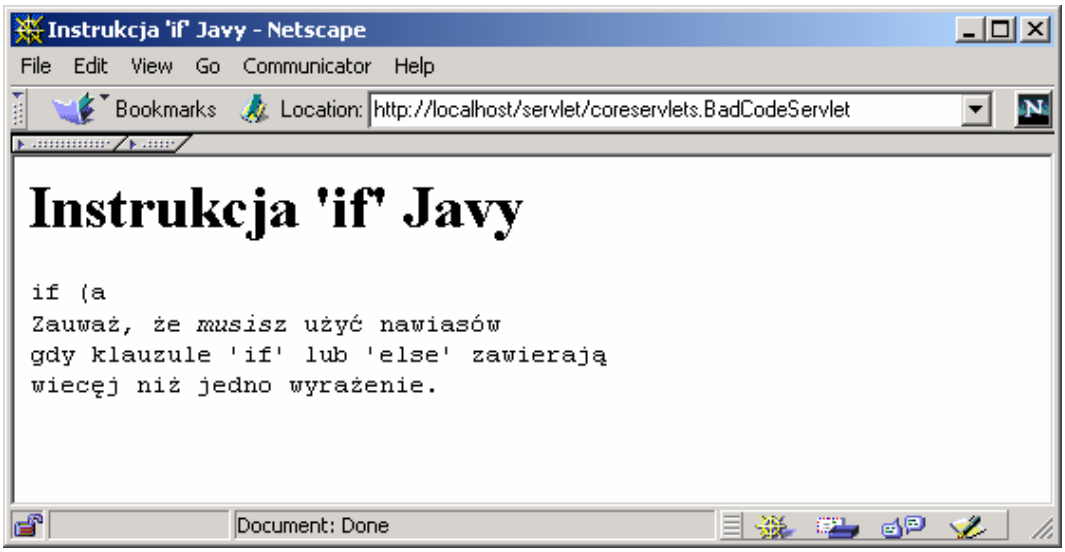
wprowadzona zmiana polega na przefiltrowaniu łańcucha znaków zawierającego fragment kodu. Wyniki jego działania przedstawiłem na rysunku 3.10; jak widać listing został przedstawiony w poprawny sposób.

Listing 3.9 BadCodeServlet.java

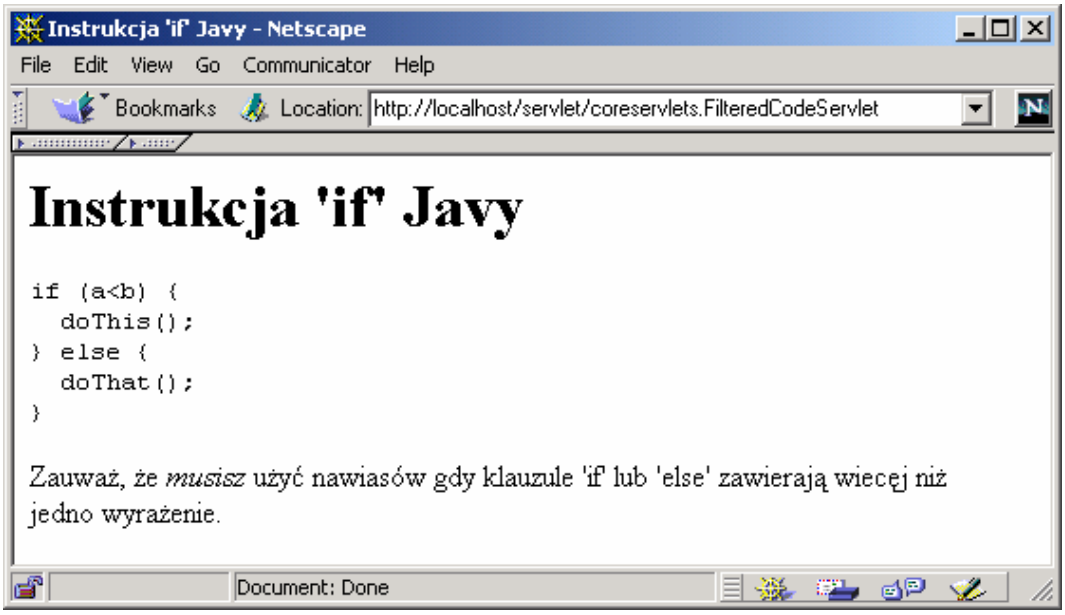
```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Servlet wyświetlający fragment listingu kodu napisanego
 *  w języku Java. Fragment ten nie został przefiltrowany
 *  w celu odszukania i zastąpienia znaków specjalnych HTML
 *  (w tym przypadku są to znaki < i >).
 */

public class BadCodeServlet extends HttpServlet {
  private String codeFragment =
    "if (a<b) {\n" +
    "  doThis();\n" +
    "} else {\n" +
    "  doThat();\n" +
    "}\n";

  public String getCodeFragment() {
    return(codeFragment);
  }

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html; charset=ISO-8859-2");
    PrintWriter out = response.getWriter();
    String title = "Instrukcja 'if' Javy";

    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY>\n" +
                "<H1>" + title + "</H1>\n" +
                "<PRE>\n" +
                getCodeFragment() +
                "</PRE>\n" +
                "Zauważ, że <I>musisz</I> użyć nawiasów\n" +
                "gdy klauzule 'if' or 'else' zawierają\n" +
                "więcej niż jedno wyrażenie.\n" +
                "</BODY></HTML>");
  }
}
```

Listing 3.10 FilteredCodeServlet.java

```
package coreservlets;

/** Klasa potomna klasy BadCodeServlet posiadająca tę samą metodę
 *  doGet; ta klasa filtruje fragment kodu w celu odszukania i
 *  zastąpienia znaków specjalnych HTML.
 *  Należy filtrować łańcuchy znaków, które mogą zawierać znaki
 *  specjalne HTML (na przykład, fragmenty kodów programów) oraz
 *  łańcuchy znaków podane przez użytkownika.
 */

public class FilteredCodeServlet extends BadCodeServlet {
  public String getCodeFragment() {
    return(ServletUtilities.filter(super.getCodeFragment()));
  }
}
```



Rysunek 3.9 Wyniki wykonania serwletu BadCodeServlet — przeważająca cześć fragmentu kodu jest niewidoczna, a tekst umieszczony pod listingiem jest nieprawidłowo wyświetlony czcionką o stałej szerokości.



Rysunek 3.10 Wyniki wykonania serwletu FilteredCodeServlet — użycie metody filter rozwiązało problem łańcuchów znaków zawierających znaki specjalne HTML.

Rozdział 4.
Obsługa żądań: Nagłówki żądań HTTP

Podstawą tworzenia efektywnych serwletów jest znajomość zasad działania protokołu HTTP (HyperText Transfer Protocol). Poznanie tego protokołu nie jest ulotnym zagadnieniem teoretycznym lecz czysto praktycznym i może wywrzeć natychmiastowy wpływ na efektywność i przydatność tworzonych serwletów. W tym rozdziale omówię informacje jakie w protokole HTTP są przesyłane z przeglądarki na serwer, czyli nagłówki żądań. Przedstawię osobno każdy z nagłówków żądań protokołu HTTP 1.1 wyjaśniając jednocześnie jak i dlaczego można by ich używać przy tworzeniu serwletów. Podam także trzy szczegółowe przykłady; pierwszy z ich będzie wyświetlał wszystkie nagłówki żądania, drugi — redukował czas pobierania strony z serwera poprzez jej kompresję, a trzeci — ochraniał dostęp do serwletu przy użyciu hasła.

Zwróć uwagę, iż nagłówki żądań HTTP to nie to samo co dane formularzy, o których pisałem w poprzednim rozdziale. Dane formularzy tworzone są na podstawie informacji podanych przez użytkownika i przesyłane bądź to jako część adresu URL (w przypadku żądań `GET`) lub w osobnym wierszu (w przypadku żądań `POST`). Nagłówki żądania są natomiast niejawnie określane przez przeglądarkę i przesyłane bezpośrednio po początkowym wierszu zawierającym żądanie `GET` lub `POST`. Na przykład, poniższy przykład przedstawia żądanie HTTP, które mogłoby zostać wygenerowane podczas poszukiwania książki w jednej z internetowych księgarni. Żądanie to jest przesyłane pod adres *http://www.jakasksiegarnia.com/szukaj*. Żądanie to zawiera nagłówki `Accept`, `Accept-Encoding`, `Connection`, `Cookie`, `Host`, `Referer` oraz `User-Agent`. Wszystkie te nagłówki mogą mieć duży wpływ na działanie serwletu, jednak ich wartości nie można określić na podstawie informacji podanych przez użytkownika ani wyznaczyć automatycznie — serwlet musi jawnie odczytać nagłówki żądania, aby skorzystać z zawartych w nich informacji.

```
GET /szukaj?keywards=servlets+jsp HTTP/1.1
Accept: image/gif, image/jpg, */*
Accept-Encoding: gzip
Connection: Keep-Alive
Cookie: userID=id34789723
Host: www.jakasksiegarnia.com
Referer: http://www.jakasksiegarnia.com/szukajksiazki.html
User-agent: Mozilla/4.7 [en] (Win98; U)
```

4.1 Odczytywanie wartości nagłówków żądania w serwletach

Odczytywanie wartości nagłówków żądania w serwletach jest bardzo proste — sprowadza się ono do wywołania metody `getHeader` interfejsu `HttpServletRequest`. Jeśli wskazany nagłówek został podany w żądaniu, to metoda ta zwraca wartość typu `String`; w innych przypadkach zwracana jest wartość `null`. Przy podawaniu nazw nagłówków wielkość liter nie ma znaczenia.

Oznacza to, że wywołania `request.getHeader("Connection")` oraz `request.getHeader("connection")` można stosować zamiennie.

Choć użycie metody `getHeader` jest ogólnym sposobem odczytywania wartości nagłówków żądania, to jednak wartości niektórych nagłówków są pobierane tak często, że interfejs `HttpServletRequest` udostępnia specjalne metody pozwalające na dostęp do tych wartości. Metody te przedstawię na poniższej liście; dodatkowe informacje na ich temat (w tym także składnię ich wywołania) możesz znaleźć w dodatku A, pt.: „Krótki przewodnik po serwletach i JSP".

- **`getCookies`**

  Metoda `getCookies` przetwarza zawartość nagłówka `Cookie` i zwraca ją w postaci tablicy obiektów `Cookie`. Metoda ta zostanie szerzej omówiona w rozdziale 8, pt.: „Obsługa cookies".

- **`getAuthType`** oraz **`getRemoteUser`**

  Metody `getAuthType` oraz `getRemoteUser` dzielą nagłówek `Authorization` na elementy składowe. Sposoby wykorzystania tego nagłówka zostały przedstawione w podrozdziale 4.5, pt.: „Ograniczanie dostępu do stron WWW".

- **`getContentLength`**

  Metoda `getContentLength` zwraca wartość nagłówka `Content-Length`, w formie liczy całkowitej (typu `int`).

- **`getContentType`**

  Metoda `getContentType` zwraca obiekt `String` zawierający wartość nagłówka `Content-Type`.

- **`getDateHeader`** oraz **`getIntHeader`**

  Metody `getDateHeader` oraz `getIntHeader` odczytują wartość nagłówka o podanej nazwie i zwracają ją odpowiednio jako wartość typu `Date`[9] oraz typu `int`.

- **`getHeaderNames`**

  Zamiast pobierać wartość konkretnego nagłówka, można stworzyć listę nazw wszystkich nagłówków umieszczonych w danym żądaniu. Służy do tego metoda `getHeaderNames`, która zwraca obiekt `Enumeration`. Sposób wykorzystania tej metody przedstawiłem w podrozdziale 4.2, pt.: „Wyświetlanie wszystkich nagłówków".

- **`getHeaders`**

  W większości przypadków, nazwa konkretnego nagłówka pojawia się w danym żądaniu tylko raz. Jednak od czasu do czasu, ten sam nagłówek może się pojawić w żądaniu kilka razy, a za każdym razem jego wartość może być różna. Przykładem takiego nagłówka może być `Accept-Language`. Jeśli w żądaniu nazwa nagłówka zostanie powtórzona, to w serwletach tworzonych według specyfikacji 2.1 nie można odczytać kolejnych wartości tego nagłówka bez odczytania strumienia wejściowego. Wynika to z faktu, iż metoda `getHeader` zwraca wyłącznie wartość pierwszego wystąpienia danego nagłówka. W przypadku serwletów tworzonych według specyfikacji 2.2, dostępna jest metoda `getHeaders`, która zwraca obiekt `Enumeration` zawierający wszystkie wartości danego nagłówka.

Poza tym, oprócz odczytywania wartości nagłówków, można także zdobyć informacje dotyczące głównego wiersza żądania. Służą do tego także metody interfejsu `HttpServletRequest`.

- **`getMethod`**

  Metoda `getMethod` zwraca metodę żądania (zazwyczaj jest to wartość `PUT` lub `POST`, jednak mogą się także pojawić wartości `HEAD`, `PUT` bądź `DELETE`).

- **`getRequestURI`**

  Metoda getRequestURI zwraca fragment adresu URL znajdujący się za nazwą komputera i numerem portu oraz przed danymi pochodzącymi z formularza. Na przykład,

[9] przyp. tłum. Precyzyjnie rzecz biorąc metoda getDateHeader zwraca liczbę typu long, która reprezentuje wartość typu Date.

dla adresu URL o postaci *http://jakiskomputer.com/servlet/szukaj.SzukajKsiazki*, metoda ta zwróci łańcuch znaków `/servlet/szukaj.SzukajKsiazki`.

- **`getProtocol`**

  Ta metoda zwraca trzecią część wiersza żądania, określającą używany protokół; zazwyczaj ma ona postać `HTTP/1.0` lub `HTTP/1.1`. Zazwyczaj, nim serwlet użyje nagłówków odpowiedzi (więcej na ich temat znajdziesz w rozdziale 7) charakterystycznych dla protokołu HTTP 1.1, powinien wywołać metodę `getProtocol` i sprawdzić jaki protokół został użyty przez klienta.

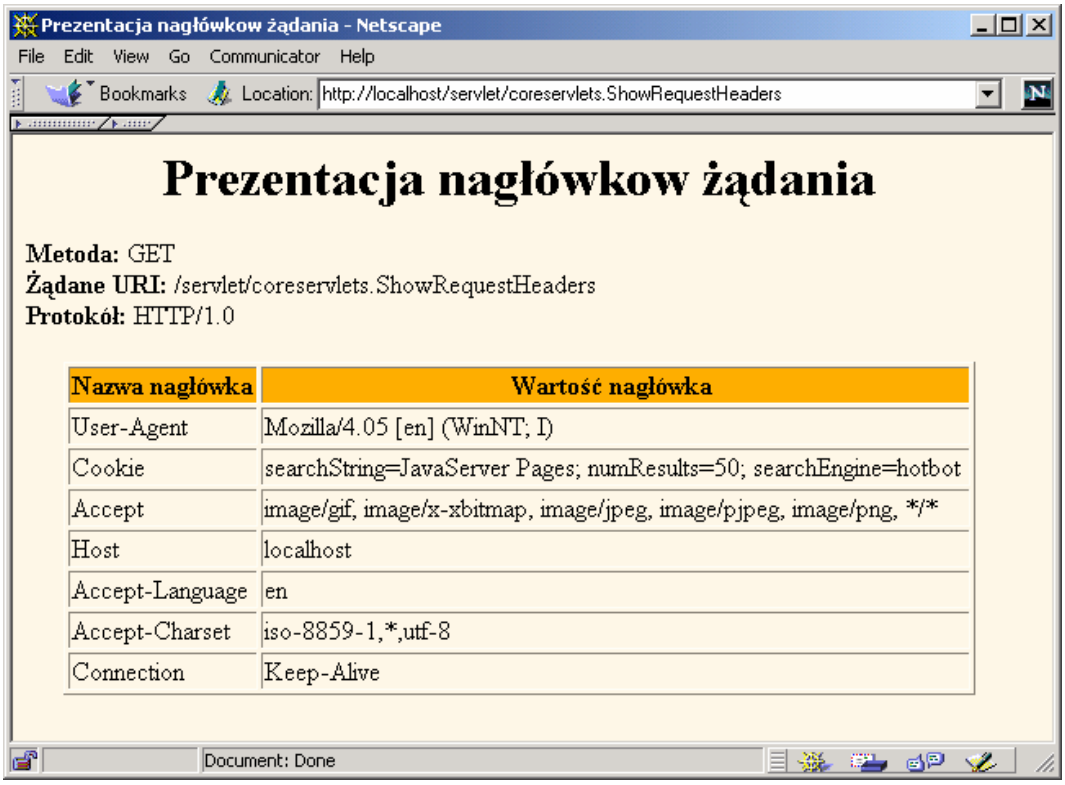
4.2 Wyświetlanie wszystkich nagłówków

Na listingu 4.1 przedstawiłem serwlet, który tworzy tabelę wszystkich nagłówków umieszczonych w żądaniu oraz ich wartości. Serwlet wyświetla także wszystkie trzy elementy głównego wiersza żądania — metodę, URI oraz protokół. Na rysunkach 4.1 oraz 4.2 zostały przedstawione typowe wyniki działania tego serwletu dla żądań przesyłanych przez przeglądarki Netscape Navigator oraz Internet Explorer.

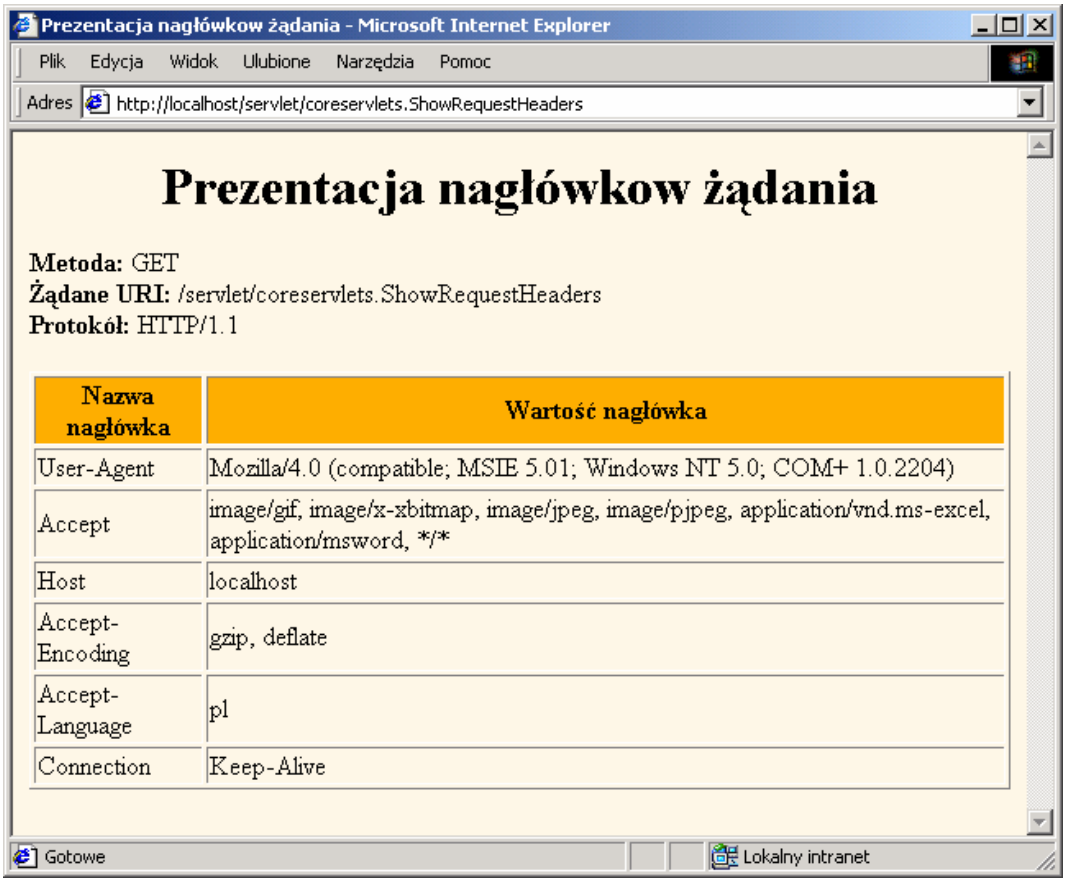
Listing 4.1 ShowRequestHeaders.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

/** Tworzy listę wszystkich nagłówków żądania
 *  przesłanych w tym żądaniu.
 */

public class ShowRequestHeaders extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html; charset=ISO-8859-2");
    PrintWriter out = response.getWriter();
    String title = "Prezentacja nagłówkow żądania";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<B>Metoda: </B>" +
                request.getMethod() + "<BR>\n" +
                "<B>Żądane URI: </B>" +
                request.getRequestURI() + "<BR>\n" +
                "<B>Protokół: </B>" +
                request.getProtocol() + "<BR><BR>\n" +
                "<TABLE BORDER=1 ALIGN=CENTER>\n" +
                "<TR BGCOLOR=\"#FFAD00\">\n" +
                "<TH>Nazwa nagłówka<TH>Wartość nagłówka");
    Enumeration headerNames = request.getHeaderNames();
    while(headerNames.hasMoreElements()) {
      String headerName = (String)headerNames.nextElement();
      out.println("<TR><TD>" + headerName);
      out.println("    <TD>" + request.getHeader(headerName));
    }
    out.println("</TABLE>\n</BODY></HTML>");
  }

  /** Niech ten sam servlet obsługuje zarówno żądania
   *  GET jak i POST
   */

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
```

```
}
```



Rysunek 4.1 Nagłówki żądania przesyłane przez przeglądarkę Netscape Navigator 4.7 działającą w systemie Windows 98.



Rysunek 4.2 Nagłówki żądania przesyłane przez przeglądarkę Internet Explorer 5 działającą w systemie Windows 98.

4.3 Nagłówki żądań protokołu HTTP 1.1

Dostęp do wartości nagłówków żądania daje możliwość optymalizacji działania serwletów oraz pozwala na implementację wielu możliwości, które bez niego nie byłyby dostępne. W tej części rozdziału przedstawiłem wszystkie nagłówki żądań jakie są stosowane w protokole HTTP 1.1, oraz opisałem w jaki sposób można je wykorzystać przy tworzeniu serwletów. W następnym podrozdziale znajdziesz przykłady zastosowania nagłów żądania.

Zwróć uwagę na to, iż protokół HTTP 1.1 pozwala na stosowanie większej liczby nagłówków niż protokół HTTP 1.0. Szczegółowe informacje na temat tych nagłówków znajdziesz specyfikacji HTTP 1.1, podanej w pliku RFC 2616. Na Internecie można znaleźć wiele miejsc, gdzie są przechowywane archiwa oficjalnych plików RFC; aktualną listę takich archiwów znajdziesz na witrynie *http://www.rfc-editor.org/*.

Accept

Ten nagłówek określa typy MIME jakie przeglądarka lub inny program jest w stanie obsługiwać. Serwlet, który jest w stanie zwracać zasoby w różnych formatach, może przeanalizować ten nagłówek, aby określić jakiego formatu użyć. Na przykład, obrazy PNG dysponują lepszymi możliwościami kompresji niż obrazy GIF, jednak format PNG obsługuje bardzo niewiele przeglądarek. Jeśli dysponujesz obrazami w obu formatach, to serwlet może wywołać metodę `request.getHeader("Accept")`, sprawdzić czy format `image/png` jest akceptowany, a jeśli tak, to we wszystkich znacznikach `<IMG>` generowanych przez serwlet będziesz mógł użyć plików *xxx.png*. W przeciwnym przypadku, konieczne będzie wykorzystanie plików *xxx.gif*.

W tabeli 7.1 znajdującej się w podrozdziale 7.2, pt.: „Nagłówki odpowiedzi protokołu HTTP 1.1 oraz ich znaczenie”, znajdziesz nazwy oraz opis znaczenia najczęściej stosowanych typów MIME.

Accept-Charset

Ten nagłówek określa zbiór znaków (na przykład: ISO-8859-2) używany przez przeglądarkę.

Accept-Encoding

Ten nagłówek podaje typy kodowania, które klient jest w stanie obsługiwać. Jeśli nagłówek ten zostanie podany, to serwer może zakodować stronę w podanym formacie (zazwyczaj po to, aby skrócić czas jej transmisji) i określić w jaki sposób została zapisana przesyłając nagłówek odpowiedzi `Content-Encoding`. Ten sposób kodowania nie ma niczego wspólnego z typem MIME generowanego dokumentu (określanym przy użyciu nagłówka odpowiedzi `Content-Type`), gdyż kodowanie jest odtwarzane zanim przeglądarka zdecyduje co zrobić z jego zawartością. Jednak z drugiej strony, wykorzystanie sposobu kodowania, którego przeglądarka nie zna, spowoduje wyświetlenie całkowicie niezrozumiałych stron WWW. Z tego względu jest niezwykle istotne, abyś sprawdził zawartość nagłówka `Accept-Encoding` zanim wykorzystasz jakiś sposób kodowania zawartości generowanych dokumentów. Standardowymi wartościami tego nagłówka są `gzip` oraz `compress`.

Kompresja stron przed ich przekazaniem do przeglądarki jest bardzo przydatną możliwością, gdyż czas konieczny do zdekodowania strony zazwyczaj jest znacznie krótszy od czasu jej transmisji. W podrozdziale 4.4, pt.: „Przesyłanie skompresowanych stron WWW”, znajdziesz przykład serwletu wykorzystującego kompresję generowanych dokumentów, dzięki której możliwe jest 10-krotne skrócenie czasu transmisji stron.

Accept-Language

Ten nagłówek określa język preferowany przez użytkownika. Można go zastosować w serwletach, zdolnych do generacji wyników w różnych językach. Wartością tego nagłówka powinien być jeden ze standardowych kodów określających język, takich jak `en`, `en-us`, `da`, i tak dalej. Więcej informacji na temat kodów języków znajdziesz w pliku RFC 1766.

Authorization

Ten nagłówek jest używany przez klienty w celu przeprowadzenia ich autoryzacji podczas żądania dostępu do stron WWW chronionych hasłem. Przykład zastosowania tego nagłówka podałem w podrozdziale 4.5, pt.: „Ograniczanie dostępu do stron WWW".

Cache-Control

Ten nagłówek może zostać wykorzystany przez klienta od podania opcji wykorzystywanych przez serwery pośredniczące w celu określenia sposobów przechowywania stron w pamięci podręcznej. Nagłówek żądania `Cache-Control` jest zazwyczaj ignorowany przez serwlety, niemniej jednak nagłówek *odpowiedzi* o tej samej nazwie może być bardzo przydatny, gdyż informuje, że strona ulega ciągłym modyfikacjom i nie powinna być przechowywana w pamięci podręcznej. Szczegółowe informacje na temat tego nagłówka odpowiedzi znajdziesz w rozdziale 7, pt.: „Generacja odpowiedzi: Nagłówki odpowiedzi HTTP".

Connection

Ten nagłówek zawiera informację czy klient jest w stanie obsługiwać trwałe połączenia HTTP. Połączenia tego typu pozwalają przeglądarkom oraz innym klientom, na pobieranie wielu plików (na przykład: strony WWW oraz kilku używanych na niej obrazów) przy wykorzystaniu jednego połączenia. W ten sposób oszczędzany jest czas, który trzeba by poświęcić na nawiązanie kilku niezależnych połączeń. W przypadku żądań HTTP 1.1, połączenia trwałe są stosowane domyślnie; a jeśli program chce wykorzystać połączenia starego typu, to musi podać w tym nagłówku wartość `close`. W przypadku wykorzystania protokołu HTTP 1.0, aby korzystać z trwałych połączeń, należy podać w tym nagłówku wartość `keep-alive`.

Każde odebrane żądanie HTTP powoduje nowe uruchomienie serwletu; wykorzystanie istniejącego lub nowego połączenia, nie ma w tym przypadku najmniejszego znaczenia. Serwer zawsze uruchamia serwlet po odczytaniu żądania HTTP; co oznacza, że aby serwlet był w stanie obsługiwać trwałe połączenia będzie musiał uzyskać pomoc ze strony serwera. A zatem, zadanie serwletu sprowadza się do umożliwienia serwerowi wykorzystania trwałych połączeń, co jest realizowane poprzez przesłanie nagłówka odpowiedzi `Content-Length`. Przykład prezentujący wykorzystanie trwałych połączeń podałem w podrozdziale 7.4, pt.: „Stosowanie trwałych połączeń HTTP".

Content-Length

Ten nagłówek jest stosowany wyłącznie w żądaniach typu `POST` i służy od określenia (w bajtach) wielkości przesyłanych danych. Aby określić wartość tego nagłówka nie musisz wywoływać metody `request.getIntHeader("Content-Length")`, lecz możesz posłużyć się metodą `request.getContentLength()`. Jednak serwlety są w stanie samodzielnie odczytać dane przesyłane z formularzy (patrz rozdział 3, pt.: „Obsługa żądań: Dane przesyłane z formularzy"), a zatem jest mało prawdopodobne, abyś jawnie korzystał z tego nagłówka.

Content-Type

Choć ten nagłówek jest zazwyczaj używany w odpowiedziach generowanych przez serwlet, to jednak może on także znaleźć się wśród nagłówków żądania. Może się to zdarzyć w sytuacji, gdy klient prześle żądanie `PUT` lub gdy dołączy jakiś dokument do danych przesyłanych żądaniem `POST`.

Wartość tego nagłówka można pobrać przy wykorzystaniu standardowej metody `getContentType` interfejsu `HttpServletRequest`.

Cookie

Nagłówek ten jest stosowany do przekazania na serwer WWW cookies, które wcześniej serwer ten przesłał do przeglądarki. Więcej szczegółowych informacji na temat cookies znajdziesz w rozdziale 8, pt.: „Obsługa cookies". Z technicznego punktu widzenia nagłówek `Cookie` nie należy do protokołu HTTP 1.1. Początkowo stanowił on rozszerzenie standardu wprowadzone przez firmę Netscape, jednak aktualnie jest powszechnie obsługiwany, w tym także przez przeglądarki firm Netscape Navigator oraz Microsoft Internet Explorer.

Expect

Ten rzadko stosowany nagłówek pozwala klientowi podać jakiego zachowania oczekuje od serwera. Jedyna standardowa wartość tego nagłówka — `100-continue` — jest przesyłana przez przeglądarkę, która ma zamiar wysłać załączony dokument i chce wiedzieć czy serwer go przyjmie. W takim przypadku serwer powinien przesłać kod statusu `100 (Continue)` (oznaczający że można kontynuować) lub `417 (Expectation Failed)` (oznaczający, że dalsze wykonywanie czynności nie jest możliwe). Więcej informacji o kodach statusu HTTP znajdziesz w rozdziale 6, pt.: „Generacja odpowiedzi: Kody statusu".

From

Ten nagłówek zawiera adres poczty elektronicznej osoby odpowiedzialnej za wygenerowanie żądania. Przeglądarki rzadko kiedy generują ten nagłówek; znacznie częściej jest on generowany przez „roboty" przeszukujące zasoby Internetu, aby ułatwić określenie sprawców dodatkowego obciążenia serwera lub powtarzających się, błędnych żądań.

Host

Przeglądarki te muszą generować ten nagłówek. Zawiera on określenie komputera (host) oraz numer portu podany w *oryginalnym* adresie URL. Ze względu na przekazywanie żądań oraz wykorzystywanie wielu nazw przez jeden komputer, jest całkiem prawdopodobne, że informacji tych można by uzyskać w inny sposób. Nagłówek ten był już dostępny w protokole HTTP 1.0, lecz jego stosowanie było opcjonalne.

If-Match

Ten rzadko stosowany nagłówek używany jest przede wszystkim w żądaniach `PUT`. Klient możne zażądać listy *znaczników elementów*, takich jak te zwracane przez nagłówek odpowiedzi `ETag`, a operacja jest przeprowadzana wyłącznie jeśli jeden z tych znaczników odpowiada znacznikowi podanemu w nagłówku.

If-Modified-Since

Ten nagłówek oznacza, że klient chce pobrać stronę wyłącznie jeśli została ona zmodyfikowana po określonej dacie. Nagłówek ten jest bardzo przydatny, gdyż pozwala przeglądarkom na przechowywanie dokumentów w pamięci podręcznej i pobieranie ich z serwera wyłącznie jeśli zostały zmodyfikowane. Niemniej jednak tworząc serwlety nie trzeba bezpośrednio operować na tym nagłówku. Zamiast tego można zaimplementować metodę `getLastModified`, dzięki której system będzie w stanie automatycznie obsłużyć daty modyfikacji. Przykład wykorzystania metody `getLastModified` przedstawiłem w podrozdziale 2.8, pt.: „Przykład wykorzystania inicjalizacji serwletu i daty modyfikacji strony".

If-None-Match

Nagłówek ten przypomina nagłówek `If-Match`, z tym że operacja zostanie wykonana jeśli nie zostaną odnalezione żadne pasujące do siebie *znaczniki elementów*.

If-Range

Ten rzadko stosowany nagłówek pozwala klientowi który dysponuje fragmentem dokumentu, zażądać jego brakujących fragmentów (jeśli dokument nie został zmodyfikowany) lub całego, nowego dokumentu (jeśli został on zmodyfikowany po określonej dacie).

If-Unmodified-Since

Nagłówek ten działa przeciwnie niż nagłówek `If-Modified-Since` wskazując, że operacja powinna zostać wykonana wyłącznie jeśli dokument nie został zmodyfikowany po podanej dacie. Zazwyczaj nagłówek `If-Modified-Since` jest stosowany w żądaniach `GET` („prześlij mi dokument wyłącznie jeśli jest on nowszy od wersji jaką mam w pamięci podręcznej"), natomiast nagłówek `If-Unmodified-Since` w żądaniach `PUT` („zaktualizuj ten dokument jeśli nikt inny nie aktualizował go od czasu gdy ja zrobiłem to po raz ostatni").

Pragma

Nagłówek `Pragma` o wartości `no-cache` informuje, że servlet działający jako pośrednik powinien przekazać żądanie dalej, nawet jeśli dysponuje kopią lokalną żądanego zasobu. Jedyną standardową wartością tego nagłówka jest `no-cache`.

Proxy-Authorization

Ten nagłówek pozwala klientom na przekazanie swej tożsamości serwerom pośredniczącym, które wymagają autoryzacji. W przypadku servletów nagłówek ten jest zazwyczaj ignorowany, a zamiast niego wykorzystuje się nagłówek `Authorization`.

Range

Ten rzadko stosowany nagłówek pozwala klientom dysponującym częściową kopią dokumentu, zażądać od serwera wyłącznie brakujących fragmentów tego dokumentu.

Referer

Ten nagłówek zawiera adres URL strony z jakiej zostało przesłane żądanie. Na przykład, jeśli aktualnie oglądasz stronę *X* i klikniesz umieszczone na niej hiperłącze do strony *Y*, to w żądaniu dotyczącym strony *Y*, w nagłówku `Referer`, zostanie zapisany adres URL strony *X*. Wszystkie najpopularniejsze przeglądarki określają wartość tego nagłówka, dzięki czemu stanowi on wygodny sposób śledzenia skąd pochodzą żądania. Możliwości te doskonale nadają się do gromadzenia informacji o witrynach prezentujących reklamy, z których użytkownicy przechodzą na Twoją witrynę, do zmiany zawartości strony w zależności od witryny z jakiej przeszedł użytkownik, lub, po prostu, do śledzenia skąd przychodzą użytkownicy Twej witryny. W tym ostatnim przypadku większość osób korzysta z dzienników serwera, gdyż nagłówek `Referer` jest w nich zazwyczaj rejestrowany. Choć nagłówek ten jest bardzo przydatny, to nie należy na nim polegać, gdyż programy z łatwością mogą fałszować jego zawartość. I w końcu, należy zwrócić uwagę na pisownię tego nagłówka — nosi on nazwę `Referer` a nie, jak można by oczekiwać, `Referrer`, gdyż jeden z twórców protokołu HTTP popełnił prosty błąd.

Upgrade

Nagłówek `Upgrade` pozwala przeglądarce lub innemu klientowi na określenie z jakiego protokołu komunikacyjnego skorzystałaby chętniej niż z protokołu HTTP 1.1. Jeśli także serwer będzie obsługiwać ten protokół, to zarówno program jak i serwer zaczną z niego korzystać. Takie negocjowanie protokołu niemal zawsze odbywa się przed wywołaniem servletu; a zatem servlety rzadko kiedy korzystają z tego nagłówka.

User-Agent

Ten nagłówek określa jaka przeglądarka lub program nadesłał żądanie. Można go wykorzystać w sytuacjach gdy zawartość generowanego dokumentu zależy od przeglądarki do jakiej zostanie on przesłany. Stosując ten nagłówek

należy jednak zachować dużą ostrożność, gdyż tworzenie serwletu w oparciu o zakodowaną na stałe listę numerów wersji przeglądarek oraz skojarzonych z nimi listy możliwości, może doprowadzić do powstania niepewnego kodu, którego modyfikacja będzie znacznie utrudniona. Jeśli to tylko możliwe, zamiast tego nagłówka należy korzystać z innych informacji dostępnych w żądaniu HTTP. Na przykład, zamiast prób zapamiętania wszystkich przeglądarek, które obsługują kompresje gzip na poszczególnych systemach operacyjnych, wystarczy sprawdzić wartość nagłówka `Accept-Encoding`. Oczywiście, jak już wspominałem, nie zawsze jest to możliwe; lecz gdy nie jest, zawsze należy odpowiedzieć sobie na pytanie czy wykorzystanie unikalnej możliwości przeglądarki, z której chcesz skorzystać jest wart ponoszonych kosztów.

Większość wersji Internet Explorera, w pierwszej kolejności, w wierszu nagłówka `User-Agent` umieszcza określenie „Mozilla" (Netscape), a dopiero potem podaje faktyczny numer wersji programu. Służy to zachowaniu zgodności z programami pisanymi w języku JavaScript, gdzie nagłówek `User-Agent` jest czasami wykorzystywany do określania dostępnych możliwości języka. Pamiętaj także, iż wartość tego nagłówka można łatwo sfałszować. Fakt ten stawia pod dużym znakiem zapytania wartość witryn, które określają rynkowy udział poszczególnych wersji przeglądarek na podstawie wartości tego nagłówka. Hmmm... miliony dolarów wydane na reklamę opartą na statystykach, które można zniekształcić przy użyciu programu napisanego w niecałą godzinę? I ja mam wierzyć, że te liczby są precyzyjne?

Via

Nagłówek ten jest generowany przez bramy oraz serwery pośredniczące, w celu wskazania miejsc przez jakie przechodziło żądanie.

Warning

Rzadko stosowany nagłówek ogólnego przeznaczenia, który pozwala klientom na przesyłanie ostrzeżeń dotyczących błędów zawiązanych z przekształcaniem zawartości lub gromadzeniem jej w pamięci podręcznej.

4.4 Przesyłanie skompresowanych stron WWW

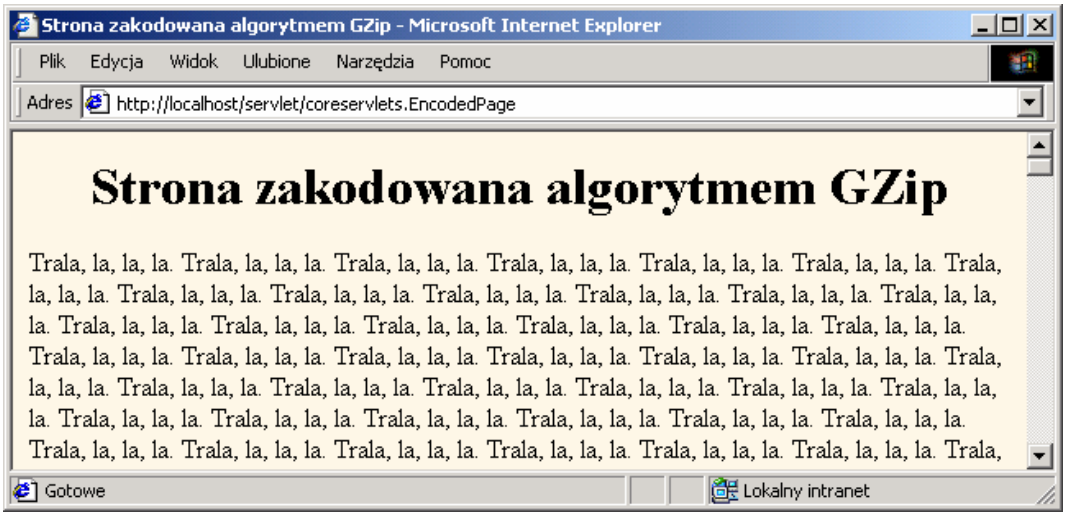
Kilka najnowszych typów przeglądarek potrafi obsługiwać skompresowane informacje, automatycznie dekompresując dokumenty oznaczone przy użyciu nagłówka `Content-Encoding` i traktując otrzymane wyniki jako oryginalny dokument. Przesyłanie skompresowanej zawartości może prowadzić do dużych oszczędności czasu, gdyż czas konieczny do skompresowania strony na serwerze oraz do jej dekompresji w przeglądarce jest zazwyczaj znacznie krótszy od czasu jej transmisji, w szczególności jeśli wykorzystywane są normalne połączenia modemowe.

Do przeglądarek obsługujących kodowanie przesyłanych stron należy większość modeli Netscape Navigatora przeznaczonych dla systemów Unix, większość wersji Internet Explorera dla systemów Windows oraz wersje 4.7 i późniejsze Netscape Navigatora dla systemów Widows. Wcześniejsze wersje Netscape Navigatora dla systemów Windows oraz Internet Explorera przeznaczone dla innych systemów operacyjnych (nie dla systemu Windows), zazwyczaj nie obsługują kodowania zawartości. Na szczęście przeglądarki dysponujące tą możliwością informują o tym, przesyłając nagłówek żądania `Accept-Encoding`. Listing 4.2 przedstawia serwlet, który sprawdza wartość tego nagłówka i przesyła skompresowaną wersję dokumentu do przeglądarek które obsługują kompresję zawartości oraz zwyczajną wersję dokumentu do wszystkich pozostałych przeglądarek. Wyniki zastosowania tego serwletu pokazują, że w przypadku wykorzystania zwyczajnych łączy modemowych, kompresja stron daje dziesięciokrotne skrócenie czasu transmisji. W wielokrotnie powtarzanych testach, w których zostały wykorzystanie przeglądarki Netscape Navigator 4.7 oraz Internet Explorer 5.0 oraz połączenie modemowe o prędkości 28,8 kbps okazało się że gdy średni czas pobierania skompresowanej strony wynosił 5 sekund, czas pobieranie jej zwyczajnej wersji nie schodził poniżej 50 sekund.

Listing 4.2 EncodedPage.java

```
package coreservlets;
```

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.zip.*;

/** Przykład pokazujący zalety przesyłania stron do przeglądarek
 *  w formie skompresowanej. (Oczywiście jeśli przeglądarka jest
 *  w stanie obsługiwać takie strony.)
 */

public class EncodedPage extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html ; charset=ISO-8859-2 ");
    String encodings = request.getHeader("Accept-Encoding");
    String encodeFlag = request.getParameter("encoding");
    PrintWriter out;
    String title;
    if ((encodings != null) &&
        (encodings.indexOf("gzip") != -1) &&
        !"none".equals(encodeFlag)) {
      title = "Strona zakodowana algorytmem GZip";
      OutputStream out1 = response.getOutputStream();
      out = new PrintWriter(new GZIPOutputStream(out1), false);
      response.setHeader("Content-Encoding", "gzip");
    } else {
      title = "Strona w postaci niezakodowanej";
      out = response.getWriter();
    }
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n");
    String line = "Trala, la, la, la. " +
                  "Trala, la, la, la.";
    for(int i=0; i<10000; i++) {
      out.println(line);
    }
    out.println("</BODY></HTML>");
    out.close();
  }
}
```


Strona zakodowana algorytmem GZip - Microsoft Internet Explorer
Plik Edycja Widok Ulubione Narzędzia Pomoc
Adres http://localhost/servlet/coreservlets.EncodedPage
Strona zakodowana algorytmem GZip
Trala, la, la, la. Trala, la, la, la. Trala, la, la, la.

Implementacja kompresji jest wyjątkowo prosta, gdyż algorytm gzip jest wbudowany w język Java i dostępny za pośrednictwem klasy `java.util.zip`. Serwlet, który chce wykorzystać kompresję danych, w pierwszej kolejności powinien sprawdzić nagłówek żądania `Accept-Encoding`, aby upewnić się, że znajdują się tam informacje o akceptowaniu kodowania gzip. Jeśli klient akceptuje to kodowanie, to serwlet może wygenerować stronę przy wykorzystaniu klasy `GZIPOutputStream` i podać wartość `gzip` w nagłówku odpowiedzi `Content-Encoding`. Przy wykorzystaniu klasy `GZIPOutputStream` należy jawnie wywołać metodę `close`. Jeśli okaże się, że klient nie obsługuje kompresji zawartości, to serwlet musi wygenerować i przesłać stronę przy użyciu klasy `PrintWriter`. Aby ułatwić przeprowadzanie testów efektywności na jednej przeglądarce, w przykładowym serwlecie zaimplementowałem możliwość wyłączenia kompresji zawartości, w momencie dodania do adresu URL parametru `?encoding=none`.

4.5 Ograniczanie dostępu do stron WWW

Wiele serwerów WWW obsługuje standardowe mechanizmy ograniczania dostępu do wybranych stron WWW. Mechanizmy te można zastosować zarówno do stron statycznych jak i do dokumentów generowanych przez serwlety. Dzięki temu, wielu autorów wykorzystuje te możliwości serwera do ograniczania dostępu do serwletów. Co więcej, znaczna większość użytkowników sklepów internetowych woli gdy autoryzacja jest przeprowadzana za pośrednictwem zwyczajnych formularzy HTML, gdyż są one lepiej znane, mogą zawierać wyjaśnienia oraz prosić o podanie dodatkowych informacji, a nie tylko nazwy użytkownika i hasła. Po przeprowadzeniu autoryzacji z wykorzystaniem formularza i udzieleniu użytkownikowi prawa dostępu do strony, serwlet może wykorzystać mechanizm śledzenia sesji, aby umożliwić użytkownikowi dostęp do pozostałych stron wymagających podobnej autoryzacji. Więcej informacji o sesjach znajdziesz w rozdziale 9, pt.: „Śledzenie sesji".

Niemniej jednak przeprowadzanie autoryzacji użytkowników przy użyciu formularzy HTML wyga więcej zachodu ze strony twórcy serwletów, a autoryzacja wykorzystująca protokół HTTP jest zupełnie wystarczająca w przypadku znacznej większości prostych aplikacji. Poniżej podałem opis czynności jakie należy wykonać w celu przeprowadzenia „prostej" autoryzacji użytkownika. Istnieje także nieco lepsza wersja autoryzacji, bazująca na skrótach wiadomości. Jednak spośród najpopularniejszych przeglądarek obsługuje ją wyłącznie Internet Explorer.

1. Sprawdź czy w nagłówkach żądania znajduje się nagłówek `Authorization`. Jeśli go nie będzie, to przejdź do punktu 2. Jeśli odnajdziesz ten nagłówek, to pomiń umieszczone po nim słowo „`basic`" i przekształć do normalnej postaci dalszą zawartość tego nagłówka, zapisaną w kodzie *base64*. W wyniku otrzymasz łańcuch znaków o postaci `nazwa_użytkownika:hasło`. Porównaj tę nazwę oraz hasła z posiadanymi informacjami o użytkownikach. Jeśli dane okażą się poprawne, to zwróć żądaną stronę WWW; w przeciwnym przypadku przejdź do punktu 2.
2. Zwróć kod odpowiedzi `401 (Unauthorized)` oraz nagłówek o następującej postaci:
   ```
   WWW-Authenticate: BASIC realm="jakaś_nazwa"
   ```

   Odpowiedź ta informuje przeglądarkę, że należy wyświetlić okienko dialogowe z prośbą podanie nazwy użytkownika i hasła, a następnie, ponownie przesłać żądanie zawierające nagłówek `Authorization` z nazwą użytkownika i hasłem zapisanymi w kodzie *base64*.

Jeśli zwracasz uwagę na szczegóły, to wiedz, że wszelkie informacje dotyczące kodowania *base64* znajdziesz pliku RFC 1521. (Pamiętaj, że aktualną listę wszystkich plików RFC znajdziesz na witrynie *http://www.rfc-edit.org/*). Istnieją jednak dwie rzeczy dotyczące tego kodowania, które prawdopodobnie powinieneś wiedzieć. Po pierwsze, nie zostało ono opracowane z myślą o zagwarantowaniu bezpieczeństwa, gdyż zakodowany tekst można łatwo przekształcić do postaci

oryginalnej. A zatem, jeśli chcesz się uchronić przed atakami osób, które mogą mieć odstęp do Twojego połączenia sieciowego (co nie jest zadaniem prostym, chyba że osoba ta ma dostęp do Twojej lokalnej podsieci), to wykorzystanie autoryzacji nie wystarczy — będziesz musiał skorzystać z protokołu SSL. SSL — Secure Socket Layer — to modyfikacja protokołu HTTP, w której cały strumień przekazywanych informacji jest szyfrowany. Protokół ten jest obsługiwany przez wiele komercyjnych serwerów, a zazwyczaj wywołuje się go poprzedzając adres URL prefiksem *https*. Serwlety mogą działać na serwerach wykorzystujących protokół SSL równie łatwo jak na standardowych serwerach WWW; szyfrowanie informacji oraz ich deszyfracja jest obsługiwana w sposób niezauważalny i wykonywana przed wywołaniem serwletu. Druga rzecz związana z kodowaniem *base64* o jakiej powinieneś wiedzieć, dotyczy klasy `sun.misc.BASE64Decoder`. Klasa ta jest dostępna w JDK 1.1 oraz 1.2 i służy do dekodowania łańcuchów znaków zapisanych w kodzie *base64*. Należy jednak pamiętać, iż klasa ta stanowi część pakietu `sun`, który nie należy do oficjalnej specyfikacji języka Java, i z tego względu nie ma żadnych gwarancji że będzie dostępny we wszystkich implementacjach Javy. Jeśli zatem będziesz używał tej klasy, to rozpowszechniając swoją aplikację nie zapomnij dołączyć do niej odpowiedniego pliku klasowego.

Listing 4.3 przedstawia serwlet, do którego dostęp chroniony jest za pomocą hasła. Serwlet ten został jawnie zarejestrowany na serwerze WWW pod nazwą `SecretServlet`. Proces rejestracji serwletów zależy od używanego serwera. W podrozdziale 2.7, pt.: „Przykład użycia parametrów inicjalizacyjnych" znajdziesz szczegółowe informacje na temat rejestracji serwletów na serwerach Tomcat, JSWDK oraz Java Web Server. Nasz przykładowy serwlet został zarejestrowany po to, aby można z nim było skojarzyć parametry inicjalizacyjne. Większość serwerów nie daje bowiem możliwości podawania parametrów inicjalizacyjnych dla wszelkich serwletów, tylko dlatego że są przechowywane w katalogu *servlets* (lub innym katalogu o analogicznym przeznaczeniu). Parametr inicjalizacyjny naszego przykładowego serwletu określa położenie pliku klasy `Properties`, zawierającego nazwy użytkowników oraz odpowiadające im hasła. Gdyby bezpieczeństwo serwletu było bardzo ważne, to zapewne chciałbyś szyfrować hasła zapisane w tym pliku, dzięki czemu przeglądnięcie jego zawartości nie pozwoliłoby na poznanie haseł.

Listing 4.3 ProtectedPage.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Properties;
import sun.misc.BASE64Decoder;

/** Przykład serwletu ochraniającego hasłem dostęp do
 *  strony WWW.
 *  <P>
 */

public class ProtectedPage extends HttpServlet {
  private Properties passwords;
  private String passwordFile;

  /** Odczytujemy plik z hasłami z dysku, określając jego
   *  nazwę na podstawie parametru inicjalizacyjnego o
   *  nazwie passwordFile.
   */

  public void init(ServletConfig config)
      throws ServletException {
    super.init(config);
    try {
      passwordFile = config.getInitParameter("passwordFile");
      passwords = new Properties();
      passwords.load(new FileInputStream(passwordFile));
    } catch(IOException ioe) {}
  }
```

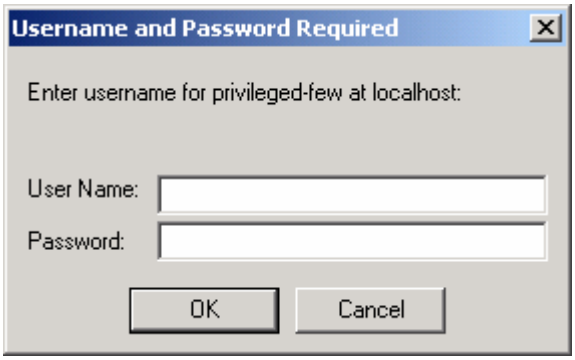
```
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html; charset=ISO-8859-2");
    PrintWriter out = response.getWriter();
    String authorization = request.getHeader("Authorization");
    if (authorization == null) {
      askForPassword(response);
    } else {
      String userInfo = authorization.substring(6).trim();
      BASE64Decoder decoder = new BASE64Decoder();
      String nameAndPassword =
        new String(decoder.decodeBuffer(userInfo));
      int index = nameAndPassword.indexOf(":");
      String user = nameAndPassword.substring(0, index);
      String password = nameAndPassword.substring(index+1);
      String realPassword = passwords.getProperty(user);
      if ((realPassword != null) &&
          (realPassword.equals(password))) {
        String title = "Witam na chronionej stronie";
        out.println(ServletUtilities.headWithTitle(title) +
                    "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                    "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                    "Gratuluję. Uzyskałeś dostęp do ściśle \n" +
                    "tajnego firmowego dokumentu.\n" +
                    "Spal lub zjedz wszystkie drukowane kopie  \n" +
                    "tego dokumentu zanim pójdziesz spać.\n" +
                    "</BODY></HTML>");
      } else {
        askForPassword(response);
      }
    }
  }

  // Jeśli w nagłówku żądania nie podano nagłówka Authorization.

  private void askForPassword(HttpServletResponse response) {
    response.setStatus(response.SC_UNAUTHORIZED); // Ie 401
    response.setHeader("WWW-Authenticate",
                       "BASIC realm=\"privileged-few\"");
  }

  /** Żądania GET i POST są obsługiwane identycznie */

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

Prócz odczytywania przesyłanych nagłówków `Authorization`, serwlet zwraca kod statusu 401 i generuje nagłówek odpowiedzi `WWW-Authenticate`. Kody statusu zostaną dokładnie omówione w rozdziale 6, pt.: „Generacja odpowiedzi: Kody statusu”, jak na razie wystarczy abyś wiedział, że kody statusu przekazują przeglądarce informacje „wysokiego poziomu” i zazwyczaj należy je generować gdy odpowiedź jest czymś innym niż żądany dokument. Najczęściej stosowanym sposobem określania kodu statusu jest wykorzystanie metody `setStatus` interfejsu `HttpServletResponse`. W wywołaniu tej metody zazwyczaj nie podaje się jawnie liczby określającej kod statusu, lecz zamiast niej używa się odpowiedniej stałej; dzięki temu łatwiej określić przeznaczenie kodu oraz zapobiec popełnieniu prostych błędów typograficznych.

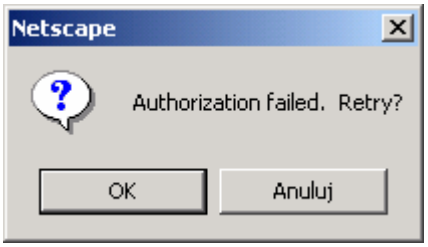
`WWW-Authenticate` oraz inne nagłówki odpowiedzi HTTP zostaną szczegółowo omówione w rozdziale 7, pt.: „Generacja odpowiedzi: Nagłówki odpowiedzi”; na razie wystarczy abyś zwrócił uwagę, iż nagłówki te zawierają dodatkowe informacje, uzupełniające odpowiedź określoną przez kod statusu. Nagłówki odpowiedzi są zazwyczaj podawane przy użyciu metody `setHeader` interfejsu `HttpServletResponse`.

Rysunek 4.4 przedstawia okno dialogowe wyświetlane przez przeglądarkę bezpośrednio po przesłaniu żądania dostępu do serwletu chronionego hasłem; kolejne dwa rysunki — 4.5 oraz 4.6

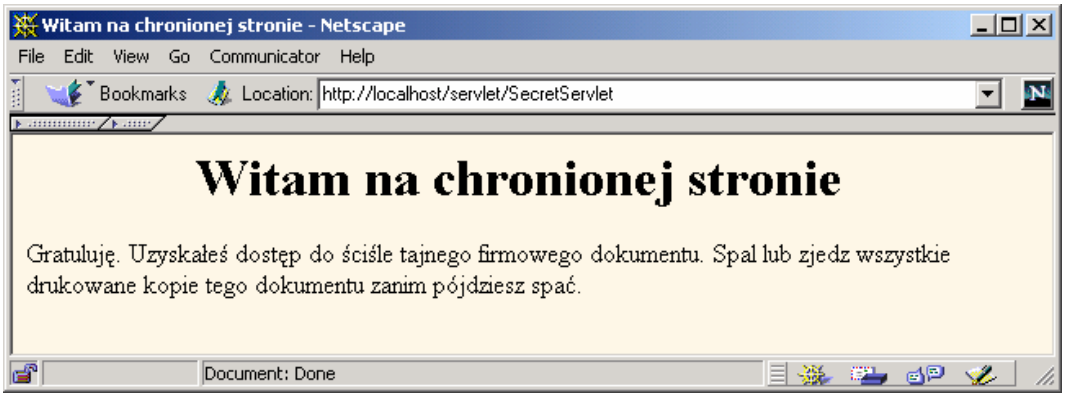
— prezentują wyniki uzyskane po podaniu nieprawidłowej nazwy użytkownika lub hasła, oraz po podaniu poprawnych informacji. Na listingu 4.4 przedstawiłem natomiast program służący do tworzenia prostego pliku z hasłami.



Rysunek 4.4 Początkowe wyniki zwrócone przez serwlet SecretServlet (pod tą nazwą został zarejestrowany serwlet ProtectedPage)



Rysunek 4.5 Wyniki wyświetlane w przypadku podania niepoprawnej nazwy użytkownika bądź hasła



Rysunek 4.6 Wyniki wyświetlane po podaniu popranej nazwy użytkownika i hasła

Listing 4.4 PasswordBuilder.java

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

/** Aplikacja zapisująca na dysku prosty plik właściwości
 *  zawierający nazwy użytkowników oraz odpowiadające im
 *  hasła.
 */

public class PasswordBuilder {
  public static void main(String[] args) throws Exception {
    Properties passwords = new Properties();
    passwords.put("marty", "martypw");
    passwords.put("bj", "bjpw");
    passwords.put("lindsay", "lindsaypw");
    passwords.put("nathan", "nathanpw");
    // Miejsce przechowywania pliku powinno być niedostępne
    // z poziomu WWW.
```

```
    String passwordFile =
      "C:\\InetPub\\Serwery\\Jakarta Tomcat 4.0\\passwords.properties";
    FileOutputStream out = new FileOutputStream(passwordFile);
    /*
    Używam JDK 1.1 aby zagwarantować przenaszalność pomiędzy
    wszystkimi mechanizmami obsługi servletów.
    W przypadku używania JDK 1.2, aby uniknąć ostrzeżeń o
    stosowaniu przestarzałych metod, należy używać metody
    "store" zamiast "save".
    */
    passwords.save(out, "Passwords");
  }
}
```

Rozdział 5.
Dostęp do standardowych zmiennych CGI

Jeśli rozpoczynasz naukę tworzenia serwletów dysponując doświadczeniami związanymi z pisaniem tradycyjnych programów CGI, to zapewne doskonale wiesz czym są i jak się stosuje „zmienne CGI". Zmienne te stanowią nieco eklektyczną kolekcję przeróżnych informacji dotyczących aktualnego żądania. Niektóre z nich bazują na wierszu żądania oraz na nagłówkach żądania HTTP (na przykład: danych przesłanych z formularza), wartości innych są określane na podstawie samego połączenia sieciowego (na przykład: nazwa oraz adres IP komputera przesyłającego żądanie), a jeszcze inne zawierają informacje o instalacyjnych parametrach serwera (takie jak skojarzenie adresów URL z fizycznymi ścieżkami).

Niezależne traktowanie wszystkich źródeł informacji (takich jak nagłówki żądania, informacje o serwerze, itp.) jest zapewne sensowne. Niemniej jednak doświadczeni programiści CGI mogą uważać, że warto by mieć w serwletach możliwość dostępu do odpowiedników wszystkich zmiennych CGI. Nie przejmuj się jeśli nie masz żadnych doświadczeń w tworzeniu tradycyjnych programów CGI — serwlety są od nich łatwiejsze w użyciu, bardziej elastyczne i efektywne. Poza tym, możesz po prostu przejrzeć ten rozdział, zwracając jedynie uwagę na te jego fragmenty, które nie są bezpośrednio związane z żądaniami HTTP. W szczególności zwróć uwagę, że można użyć metody `getServletContext().getRealPath`, aby przekształcić URI (fragmentu adresu URL podanego po nazwie komputera i adresie portu) na faktyczną ścieżkę. Poza tym pamiętaj, że nazwę oraz adres IP komputera z którego nadesłano żądanie można określić przy użyciu metod `request.getRemoteHost` oraz `request.getRemoteAddress`.

5.1 Odpowiedniki zmiennych CGI dostępne w serwletach

W tej części rozdziału podałem opisy wszystkich standardowych zmiennych CGI. Znajdziesz w nich informacje o przeznaczeniu tych zmiennych oraz o sposobach dostępu do nich z poziomu serwletów. Gdy już przeczytasz ten rozdział, będziesz mógł przypomnieć sobie zawarte w nim informacje przeglądając dodatek A, pt.: „Krótki przewodnik po serwletach i JSP". W poniższych opisach zakładam, że `request` to obiekt `HttpServletRequest` podawany w wywołaniach metod `doGet` oraz `doPost`.

AUTH_TYPE

Jeśli w żądaniu został umieszczony nagłówek `Authorization`, to ta zmienna określa typ autoryzacji (`basic` lub `digest`). Jej wartość można określić przy użyciu wywołania `request.getAuthType()`.

CONTENT_LENGTH

Zmienna dostępna wyłącznie dla żądań `POST`, zawiera wartość określającą ilość wysłanych bajtów informacji, podaną w nagłówku `Content-Length`. Zmienna CGI `CONTENT-LENGTH` jest łańcuchem znaków, a jej odpowiednikami w serwletach są wyrażenia `String.valueOf (request.getContentLength())` lub `request.getHeader("Content-Length")`. Jednak zazwyczaj będziesz chyba wolał korzystać z metody `request.getContentLength`, która zwraca wartość typu `int`.

CONTENT_TYPE

Zmienna `CONTENT_TYPE` określa typ MIME dołączonych danych, oczywiście, jeśli został on podany. W tabeli 7.1 znajdującej się w podrozdziale 7.2, pt.: „Nagłówki odpowiedzi protokołu HTTP 1.1 oraz ich znaczenie", zostały podane nazwy oraz znaczenie najczęściej stosowanych typów MIME. Wartość zmiennej `CONTENT_TYPE` można pobrać przy użyciu wywołania `request.getConetentType()`.

DOCUMENT_ROOT

Zmienna `DOCUMENT_ROOT` określa faktyczny katalog odpowiadający adresowi URL *http://host/*. Jej wartość można określić przy użyciu wywołania `getServletContext().getRealPath("/")`. We wcześniejszych specyfikacjach technologii Java Servlet wartość tej zmiennej można było pobrać przy użyciu metody `request.getRealPath("/")`, jednak aktualnie metoda ta nie jest już dostępna. Metody `getServletContext().getRealPath` można także użyć od określenia faktycznego katalogu odpowiadającemu dowolnemu URI (czyli fragmentowi adresu URL zapisanemu po nazwie komputera i numerze portu).

HTTP_XXX_YYY

Zmienne o postaci HTTP_NAGŁÓWEK_NAZWA umożliwiały programom CGI uzyskanie dostępu do dowolnych nagłówków żądania HTTP. Nagłówek `Cookie` stawał się zmienną `HTTP_COOKIE`, nagłówek `User-Agent` zmienną `HTTP_USER_AGENT`, a nagłówek `Referer` zmienną `HTTP_REFERER`, i tak dalej. W serwletach należy pobierać nagłówki przy użyciu metody `request.getHeader` lub jednej z pozostałych metod opisanych w rozdziale 4, pt.: „Obsługa żądań: Nagłówki żądań HTTP".

PATH_INFO

Ta zmienna zawiera informacje o ścieżce dołączone do adresu URL po nazwie serwletu lecz przed danymi zapytania. Na przykład, w poniższym adresie URL — `http://komputer/servlet/coreservlets.JakisServlet/costam/menu?user=Janek` — ścieżką jest `/costam/menu`. Serwlety, w odróżnieniu od standardowych programów CGI, są w stanie bezpośrednio komunikować się z serwerem, i z tego względu nie ma konieczności traktowania informacji o ścieżki w jakiś szczególny sposób. Informacje te mogą zostać przesłane jako część zwyczajnych danych pochodzących z formularza, a następnie przekształcone przy użyciu metody `getServletContext().getRealPath`. Wartość zmiennej CGI `PATH_INFO` można pobrać przy użyciu wywołania `request.getPathInfo()`.

PATH_TRANSLATED

Zmienna `PATH_TRANSLATED` zawiera informacje o ścieżce przekształcone do postaci faktycznego katalogu na serwerze. Także w tym przypadku nie ma powodu, aby zwracać szczególną uwagę na te informacje, gdyż serwlet zawsze może określić faktyczną ścieżkę na podstawie częściowych adresów URL, posługując się w tym celu metodą `getServletContext().getRealPath`. Takie odwzorowywanie adresów URL na katalogi nie było

możliwe w programach CGI, gdyż ich realizacja nie była w żaden sposób związana z serwerem. Wartość tej zmiennej można określić przy użyciu wywołania `request.getPathTranslated()`.

QUERY_STING

W przypadku żądań `GET` zmienna ta zawiera wszystkie dane przesłane z formularza w formie jednego łańcucha znaków zakodowanego w formacie URL. W serwletach, rzadko kiedy będziesz korzystał z takich nieprzetworzonych informacji; dostęp do wartości poszczególnych parametrów można łatwo uzyskać przy użyciu metody `request.getParameter` (opisanej w rozdziale 3, pt.: „Obsługa żądań: Dane przesyłane z formularzy"). Niemniej jednak, jeśli chcesz uzyskać dostęp do nieprzetworzonych informacji, możesz się posłużyć wywołaniem `request.getQueryString()`.

REMOTE_ADDR

Ta zmienna określa adres IP klienta, który przesłał żądanie. Wartość ta zwracana jest w postaci łańcucha znaków (na przykład: `"198.137.241.30"`); można ją pobrać przy użyciu wywołania `request.getRemoteAddr()`.

REMOTE_HOST

Zmienna `REMOTE_HOST` zawiera w pełni kwalifikowaną nazwę domeny (np.: `sejm.gov.pl`) klienta, który zgłosił żądanie. Jeśli nie będzie można określić nazwy domeny, metoda zwróci odpowiadający jej adres IP. Wartość tej zmiennej można określić przy użyciu wywołania `request.getRemoteHost()`.

REMOTE_USER

Jeśli nagłówek `Authorization` został podany i zdekodowany przez sam serwer WWW, to zmienna `REMOTE_USER` będzie zawierać informacje o nazwie użytkownika. W witrynach z ograniczeniami dostępu, informacje te mogą się przydać przy obsłudze sesji. Wartość tej zmiennej można pobrać przy użyciu wywołania `request.getRemoteUser()`. Więcej informacji o wykorzystaniu nagłówka Authorization w serwletach znajdziesz w podrozdziale 4.5, pt.: „Ograniczanie dostępu do stron WWW".

REQUEST_METHOD

Ta zmienna zawiera typ żądania HTTP; zazwyczaj będzie to `GET` lub `POST`, jednak może także przyjąć jedną z wartości: `HEAD`, `PUT`, `DELETE`, `OPTIONS` lub `TRACE`. W serwletach rzadko kiedy trzeba się bezpośrednio odwoływać do zmiennej `REQUEST_METHOD`, gdyż żądania poszczególnych typów są zazwyczaj obsługiwane przez odrębne metody (`doGet`, `doPost`, itp.). Wyjątkiem jest tu sposób obsługi żądań `HEAD`. Żądania tego typu są bowiem obsługiwane automatycznie przez metodę `service`, która zwraca wiersz statusu oraz wszystkie nagłówki, które zostałyby wygenerowane przez metodę `doGet`. Wartość tej zmiennej można pobrać przy użyciu wywołania `request.getMethod()`.

SCRIPT_NAME

Ta zmienna określa ścieżkę do serwletu względem głównego katalogu serwera. Jej wartość można pobrać przy użyciu wywołania `request.getServletPath()`.

SERVER_NAME

Zmienna `SERVER_NAME` zawiera nazwę komputera na którym działa serwer. Jej wartość można pobrać przy użyciu wywołania `request.getServerName()`.

SERVER_PORT

Ta zmienna zawiera numer portu na którym serwer oczekuje na żądania. W serwletach odpowiednikiem wartości tej zmiennej jest wyrażenie `String.valueOf(request.getServerPort())`, które zwraca wartość typu `String`. Zazwyczaj jednak jest używana metoda `request.getServerPort` zwracająca wartość typu `int`.

SERVER_PROTOCOL

Zmienna `SERVER_PROTOCOL` określa nazwę oraz numer protokołu podany w wierszu żądania (na przykład: `HTTP/1.0` lub `HTTP/1.1`). Wartość tej zmiennej można pobrać przy użyciu wywołania `request.getProtocol()`.

SERVER_SOFTWARE

Ta zmienna zawiera informacje określające używany serwer WWW. Informacje te można pobrać przy użyciu wywołania `getServletContext().getServerInfo()`.

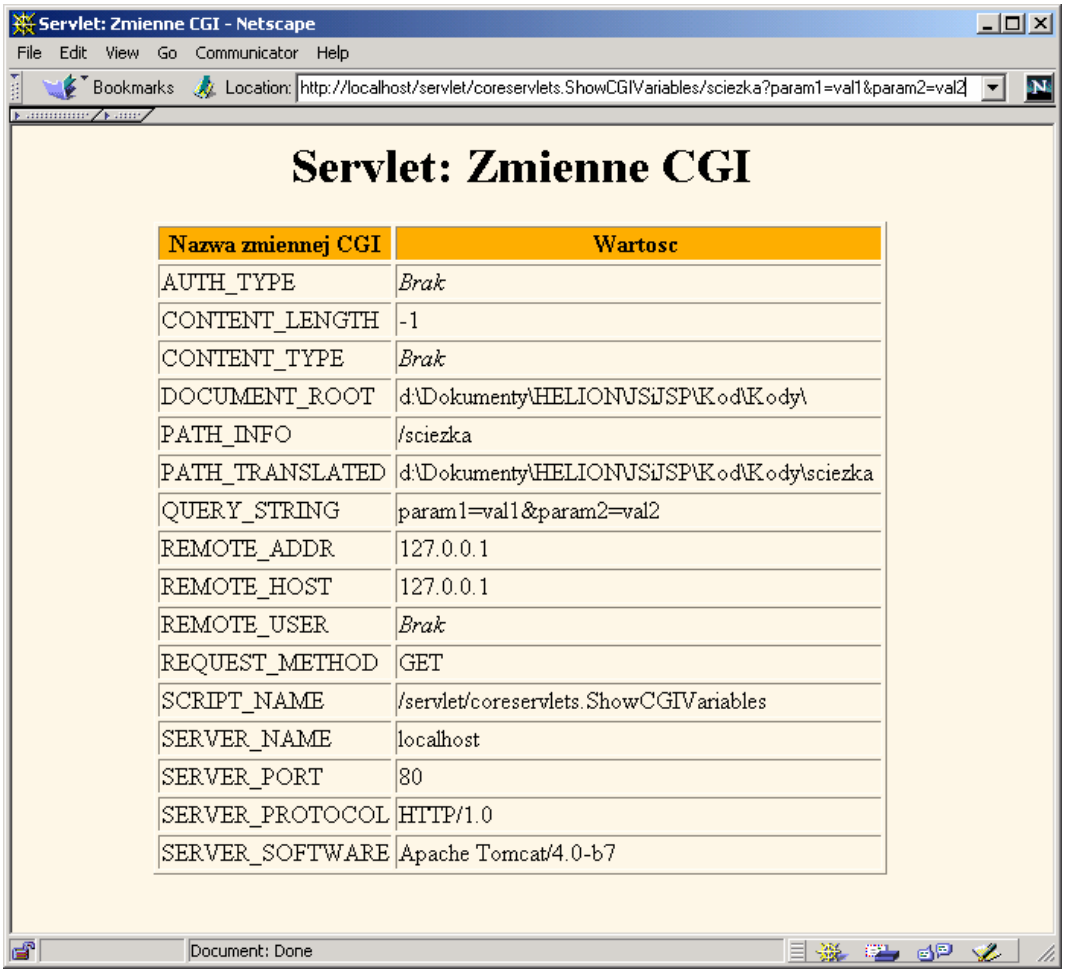
5.2 Serwlet wyświetlający wartości zmiennych CGI

Na listingu 5.1 przedstawiłem serwlet generujący tabelę zawierającą nazwy i wartości wszystkich zmiennych CGI (za wyjątkiem zmiennych `HTTP_XXX_YYY`). Zmienne te odpowiadają nagłówkom żądania HTTP opisanym w rozdziale 4. Wyniki wykonania tego serwletu w przypadku przesłania standardowego żądania przedstawiłem na rysunku 5.1.

Listing 5.1 ShowCGIVariables.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

/** Servlet tworzy tabelę zawierającą aktualne wartości
 *  standardowych zmiennych CGI.
 */

public class ShowCGIVariables extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String[][] variables =
      { { "AUTH_TYPE", request.getAuthType() },
        { "CONTENT_LENGTH",
          String.valueOf(request.getContentLength()) },
        { "CONTENT_TYPE", request.getContentType() },
        { "DOCUMENT_ROOT",
          getServletContext().getRealPath("/") },
        { "PATH_INFO", request.getPathInfo() },
        { "PATH_TRANSLATED", request.getPathTranslated() },
        { "QUERY_STRING", request.getQueryString() },
        { "REMOTE_ADDR", request.getRemoteAddr() },
        { "REMOTE_HOST", request.getRemoteHost() },
        { "REMOTE_USER", request.getRemoteUser() },
        { "REQUEST_METHOD", request.getMethod() },
        { "SCRIPT_NAME", request.getServletPath() },
        { "SERVER_NAME", request.getServerName() },
        { "SERVER_PORT",
          String.valueOf(request.getServerPort()) },
        { "SERVER_PROTOCOL", request.getProtocol() },
        { "SERVER_SOFTWARE",
          getServletContext().getServerInfo() }
      };
    String title = "Servlet: Zmienne CGI";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
```

```
                "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n" +
                "<TABLE BORDER=1 ALIGN=\"CENTER\">\n" +
                "<TR BGCOLOR=\"#FFAD00\">\n" +
                "<TH>Nazwa zmiennej CGI<TH>Wartosc");
    for(int i=0; i<variables.length; i++) {
      String varName = variables[i][0];
      String varValue = variables[i][1];
      if (varValue == null)
        varValue = "<I>Brak</I>";
      out.println("<TR><TD>" + varName + "<TD>" + varValue);
    }
    out.println("</TABLE></BODY></HTML>");
  }

  /** Działa tak samo dla żądań GET i POST. */

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```



Rysunek 5.1 Standardowe zmienne CGI dla typowego żądania

Rozdział 6.
Generacja odpowiedzi: Kody statusu

Gdy serwer WWW odpowiada na żądania przesłane przez przeglądarkę lub innego klienta, to generowana przez niego odpowiedź składa się zazwyczaj z wiersza statusu, kilku nagłówków odpowiedzi, pustego wiersza oraz dokumentu. Oto najprostszy z możliwych przykładów:

```
HTTP/1.1 200 OK
Content-Type: text/plain

Witam!!
```

Wiersz statusu składa się z określenia używanej wersji protokołu HTTP (w powyższym przykładzie jest to `HTTP/1.1`), kodu statusu będącego liczbą całkowitą (w naszym przykładzie jest to `200`) oraz bardzo krótkiego komunikatu opisującego kod statusu (w naszym przykładzie jest to `OK`). W większości przypadków wszystkie nagłówki odpowiedzi, za wyjątkiem nagłówka `Content-Type` określającego typ MIME przesyłanego dokumentu, są opcjonalne. Niemal wszystkie odpowiedzi zawierają jakiś dokument. Nie jest to jednak regułą, gdyż są odpowiedzi, które nie zawierają żadnych dokumentów. Przykładem mogą tu być odpowiedzi na żądanie `HEAD`, które nie mogą zawierać dokumentów. Poza tym istnieje wiele kodów statusu, które określają niepowodzenie i nie zawierają w odpowiedzi żadnego dokumentu bądź jedynie krótki dokument z informacją o błędzie.

Serwlety mogą wykonywać wiele ważnych zadań operując na wierszu statusu oraz nagłówkach odpowiedzi. Na przykład, mogą przekierować użytkownika do innej witryny, wskazać, że dołączony dokument jest obrazem, plikiem Adobe Acrobat lub dokumentem HTML, zażądać hasła w celu uzyskania dostępu do dokumentu, i tak dalej. W tym rozdziale przedstawię różne kody statusu oraz zadania, które można wykonać przy ich użyciu. W następnym rozdziale zajmiemy się nagłówkami odpowiedzi.

6.1 Określanie kodów statusu

Zgodnie z tym co napisałem wcześniej, wiersz statusu odpowiedzi HTTP składa się z określenia wersji protokołu HTTP, kodu statusu oraz ze skojarzonego z nim komunikatu. Ponieważ komunikat jest bezpośrednio skojarzony z kodem statusu a wersję protokołu HTTP określa serwer WWW, serwlet musi jedynie określić kod statusu. Do tego celu służy metoda `setStatus` interfejsu `HttpServletResponse`. Jeśli odpowiedź zawiera jakiś specjalny kod statusu *oraz* dokument, to koniecznie musisz określić kod statusu zanim zaczniesz generować treść dokumentu przy użyciu metod klasy `PrintWriter`. Konieczność określenia kodu statusu przed wygenerowaniem zawartości przesyłanego dokumentu wynika z faktu, iż odpowiedź HTTP składa się z wiersza statusu, nagłówków odpowiedzi, pustego wiersza i dokumentu; przy czym wszystkie te elementy *muszą* być przesłane w podanej kolejności. Kolejność samych nagłówków odpowiedzi jest dowolna, a serwlety buforują je i przesyłają wszystkie nagłówki za jednym razem; a zatem dozwolone jest określanie

kodu statusu po podaniu nagłówków odpowiedzi. Jednak serwlety nie muszą buforować samych dokumentów, gdyż użytkownicy mogą chcieć, aby w przypadku długich stron wyświetlane były wyniki cząstkowe. W założeniach specyfikacji Java Servlet 2.1 dane generowane przy użyciu klasy `PrintWriter` w ogóle nie są buforowane. Specyfikacja 2.2 pozwala na częściowe buforowanie danych, jednak wielkość bufora nie została określona. Rozmiar bufora można pobrać przy użyciu metody `getBufferSize` interfejsu `HttpServletResponse` i określić przy użyciu metody `setBufferSize`. W serwerach zgodnych ze specyfikacją 2.2 można określać kod statusu do momentu gdy bufor wyjściowy zostanie całkowicie wypełniony i przesłany do przeglądarki lub innego klienta. Jeśli nie jesteś pewien czy zawartość buforu została już przesłana, możesz to sprawdzić za pomocą metody `isCommitted`.

Metoda
Nie zapomnij określić kodu statusu przed przesłaniem jakiegokolwiek fragmentu dokumentu do klienta.

Metoda `setStatus` pobiera jeden argument typu `int` (kod statusu); jednak zamiast stosowania zwyczajnych liczb całkowitych lepiej jest posługiwać się stałymi zdefiniowanymi w interfejsie `HttpServletResponse`. Nazwy tych stałych zostały określone na podstawie standardowych komunikatów protokołu HTTP 1.1 skojarzonych z kodami statusu. Nazwy zapisywane są wyłącznie wielkimi literami, rozpoczynają się od liter `SC` (skrót od angielskich słów: *Status Code* — kod statusu) a w wszystkie odstępy są w nich zamieniane na znaki podkreślenia (_). Przykładowo, jeśli komunikat dla kodu `404` to „`Not Found`" (nie odnaleziono), to odpowiadająca mu stała będzie miała nazwę `SC_NOT_FOUND`. W specyfikacji Java Servlet 2.1 istnieją jednak trzy wyjątki od tej reguły. Otóż, stała skojarzona z kodem `302` została określona na podstawie komunikatu protokołu HTTP 1.0 (`Moved Temporarily`), a nie na podstawie komunikatu protokołu HTTP 1.1 (`Found`), natomiast stałe odpowiadające kodom `307` (`Temporary Redirect`) oraz `416` (`Requested Range Not Satsfiable`) w ogóle nie zostały zdefiniowane. W specyfikacji 2.2 dodano stałą dla kodu `416`, jednak pozostałe dwie niespójności wciąż pozostały.

Choć ogólną metodą określania kodu statusu jest wywołanie o postaci `request.setStatus(int)`, to istnieją dwie częste sytuacje, w których można użyć innych metod interfejsu `HttpServletResponse`. Należy jednak pamiętać, że obie przedstawione poniżej metody zgłaszają wyjątek `IOException`, natomiast metoda `setStatus` nie robi tego.

- **`public void sendError(int kod, String komunikat)`**

  Ta metoda przesyła kod statusu (zazwyczaj jest to kod `404`) oraz krótki komunikat, który jest automatycznie umieszczany w dokumencie HTML i przesyłany do klienta.

- **`public void sendRedirect(String url)`**

  Metoda `sendRedirect` generuje kod statusu `302` oraz nagłówek odpowiedzi `Location` podający adres URL nowego dokumentu. W przypadku specyfikacji Java Servlet 2.1 musi to być adres bezwzględny. Specyfikacja 2.2 zezwala na podawanie adresów względnych, a system, przed umieszczeniem adresu w nagłówku `Location`, przekształca go do postaci bezwzględnej.

Określenie kodu statusu wcale nie musi oznaczać, że nie trzeba zwracać dokumentu. Przykładowo, w przypadku odpowiedzi `404` większość serwerów automatycznie generuje krótki dokument z komunikat „File Not Found" (nie odnaleziono pliku), niemniej jednak serwlet może samemu określić zawartość tego dokumentu. Pamiętaj, że jeśli chcesz samodzielnie generować treść odpowiedzi, to zanim to zrobisz będziesz musiał wywołać metodą `setStatus` bądź `sendError`.

6.2 Kody statusu protokołu HTTP 1.1 oraz ich przeznaczenie

W tej części rozdziału przedstawię wszystkie kody statusu, których można używać w serwletach komunikujących się z klientami przy użyciu protokołu HTTP 1.1, oraz skojarzone z nimi standardowe komunikaty. Doskonałe zrozumienie tych kodów może w ogromnym stopniu rozszerzyć możliwości tworzonych serwletów, a zatem powinieneś przynajmniej przejrzeć dostępne kody i ich opisy, aby orientować się jakie możliwości są dostępne. Później, gdy będziesz gotów by wykorzystać te możliwości, możesz wrócić do tego rozdziału i znaleźć potrzebne, szczegółowe informacje. Informacje o kodach statusów, w skróconej, tabelarycznej postaci, zostały także podane w dodatku A, pt.: „Krótki przewodnik po serwletach i JSP".

Pełna specyfikacja protokołu HTTP 1.1 została podana w pliku RFC 2616, który można znaleźć na witrynie *http://www.rfc-editor.org/*. Kody charakterystyczne dla protokołu HTTP 1.1 zostały wyróżnione, gdyż wiele przeglądarek obsługuje wyłącznie protokół HTTP 1.0. Kody te należy przesyłać wyłącznie do klientów, które obsługują protokół HTTP 1.1. To czy program, który przesłał żądanie obsługuje protokół HTTP 1.1, można sprawdzić przy użyciu wywołania `request.getProtocol()`.

Dalsza część tego podrozdziału zawiera opis kodów statusu dostępnych w protokole HTTP 1.1. Kody te dzielą się na pięć głównych kategorii:

- **100-199**
  Kody należące do tej kategorii, są kodami informacyjnymi; oznaczają one, że klient powinien zareagować wykonując jakąś inną czynność.
- **200-299**
  Kody należące do tego kategorii oznaczają, że czynność została wykonana poprawnie.
- **300-399**
  Kody należące do tej kategorii oznaczają, że plik zostały gdzieś przeniesiony i zazwyczaj zawierają nagłówek `Location` określający jego nowy adres.
- **400-499**
  Kody należące do tej kategorii oznaczają błąd klienta.
- **500-599**
  Kody należące do tej kategorii oznaczają błąd serwera.

Nazwy stałych zdefiniowanych w interfejsie `HttpServletResponse` zostały określone na podstawi krótkich komunikatów skojarzonych z kodami statusu. Przy tworzeniu serwletów, niemal zawsze należy określać kody statusu przy wykorzystaniu tych stałych. Na przykład, zamiast wywołania `response.setStatus(204)` lepiej jest użyć wywołania o postaci `response.setStatus(response.SC_NO_CONTENT)`, gdyż jest ono bardziej zrozumiałe i mniej podatne na błędy typograficzne. Należy jednak zwrócić uwagę, iż na poszczególnych serwerach komunikaty mogą być nieco odmienne, a programy zwracają uwagę wyłącznie na numeryczny kod statusu. A zatem, czasami można napotkać wiersz statusu o postaci `HTTP/1.1 200 Document Follows` zamiast `HTTP/1.1 200 OK`.

100 (**Continue** — kontynuuj)

Jeśli serwer otrzyma żądanie z nagłówkiem `Expect` o wartości `100-continue`, będzie ono oznaczało, że klient pyta czy w następnym żądaniu może przesłać treść dokumentu. W takiej sytuacji serwer powinien odpowiedzieć zwracając kod statusu `100` (`SC_CONTINUE`), aby klient kontynuował, bądź też zwracając kod statusu `417`, aby poinformować program, że dokument nie zostanie przyjęty. Ten kod statusu został wprowadzony w protokole HTTP 1.1.

101 (**Switching Protocols** — zamiana protokołów)

Kod statusu `101` (`SC_SWITCH_PROTOCOLS`) oznacza, że serwer postąpi zgodnie z żądaniem wyrażonym w nagłówku `Upgrade` i zacznie stosować inny protokół. Ten kod statusu został wprowadzony w protokole HTTP 1.1.

200 (**OK**)

Kod statusu `200` (`SC_OK`) oznacza, że wszystko jest w porządku. W odpowiedzi na żądanie `GET` i `POST` zostanie przesłany dokument. Jest to domyślny kod statusu stosowany w serwletach — jeśli nie użyjesz metody `setStatus`, automatycznie zostanie wygenerowany kod statusu `200`.

201 (**Created** — utworzony)

Kod statusu `201` (`SC_CREATED`) oznacza, że w odpowiedzi na żądanie serwer utworzył nowy dokument; jego URL jest określany przy użyciu nagłówka `Location`.

202 (**Accepted** — przyjęty)

Kod statusu `202` (`SC_ACCEPTED`) informuje, że żądanie jest przetwarzane, jednak proces jego obsługi jeszcze nie został zakończony.

203 (**Non-Authoritative Information** — informacja niemiarodajna)

Kod statusu `203` (`SC_NON_AUTHORITATIVE_INFORMATION`) informuje, że dokument zostanie zwrócony w standardowy sposób, lecz niektóre z nagłówków żądania mogą być nieprawidłowe, gdyż została użyta kopia dokumentu. Ten kod statusu został wprowadzony w protokole HTTP 1.1.

204 (**No Content** — brak zawartości)

Kod statusu `204` (`SC_NO_CONTENT`) informuje, że przeglądarka ma wciąż wyświetlać poprzedni dokument, gdyż nowy dokument nie jest dostępny. Ten kod statusu jest przydatny w sytuacjach gdy użytkownik cyklicznie odświeża stronę klikając przycisk *Odśwież* i istnieje możliwość określenia czy poprzednia strona wyświetlona w przeglądarce jest aktualna. Na przykład, serwlet może wykonywać następujące czynności:

```
int wersjaStrony =
  Integer.parseInt(request.getParameter("wersjaStrony"));
if (wersjaStrony >= aktualnaWersja) {
  response.setStatus(response.SC_NO_CONTENT);
} else {
  // stwórz normalną stronę
}
```

Jednak metody tej nie da się zastosować do stron automatycznie odświeżanych przy użyciu nagłówka odpowiedzi `Refresh` lub odpowiadającego mu znacznika HTML `<META HTTP-EQUIV="Refresh" ...>`. Wynika to z faktu, iż przesłanie kodu statusu `204` powoduje przerwanie odświeżania strony. W takiej sytuacji wciąż jednak będzie działać odświeżanie strony realizowane przy użyciu JavaScriptu. Więcej szczegółowych informacji na ten temat znajdziesz w podrozdziale 7.2, pt.: „Nagłówki odpowiedzi protokołu HTTP 1.1 oraz ich znaczenie”.

205 (**Reset Content** — odtwórz zawartość do stanu początkowego)

Kod statusu `205` (`SC_RESET_CONTENT`) oznacza, że nie ma żadnego nowego dokumentu, lecz przeglądarka powinna ponownie wyświetlić aktualnie prezentowaną stronę. Użycie tego kodu statusu pozwala wymusić na przeglądarce wyczyszczenie pól formularza. Kod ten został wprowadzony w protokole HTTP 1.1.

206 (**Partial Content** — zawartość częściowa)

Kod statusu `206` (`SC_PARTIAL_CONTENT`) jest przesyłany w momencie gdy serwer wykona jedno z żądań podanych w nagłówku `Range`. Kod ten został wprowadzony w protokole HTTP 1.1.

300 (**Multiple Choice** — wiele możliwości wyboru)

Kod statusu `300` (`SC_MULTIPLE_CHOICE`) oznacza, że żądany dokument można odnaleźć w kilku różnych miejscach, których adresy zostaną podane w zwróconym dokumencie. Jeśli serwer jest w stanie podać preferowane miejsce z którego należy pobrać dokument, to jego adres powinien zostać podany w nagłówku odpowiedzi `Location`.

301 (**Moved Permanently** — przeniesiony na stałe)

Kod statusu `301` (`SC_MOVED_PERMANENTLY`) oznacza, że żądany dokument został przeniesiony; nowy adres URL tego dokumentu jest podawany w nagłówku odpowiedzi `Location`. Przeglądarka powinna automatycznie pobrać dokument z jego nowego miejsca.

302 (**Found** — odnaleziony)

Kod statusu `302` przypomina kod `301`, z tą różnicą iż adres URL podany w nagłówku odpowiedzi `Location` należy traktować jako tymczasowy, a nie jako stały. Notatka: w protokole HTTP 1.0 komunikat skojarzony z tym kodem statusu miał postać `Moved Temporarily` a nie `Found`; odpowiadająca mu stała zdefiniowana w interfejsie `HttpServletResponse` to `SC_MOVED_TEMPORARILY`, a nie `SC_FOUND`.

Notatka
Stała reprezentująca kod statusu o wartości `302` to `SC_MOVED_TEMPORARILY` a nie `SC_FOUND`.

Kod statusu `302` jest niezwykle przydatny, gdyż przeglądarki automatycznie przechodzą pod nowy adres URL podany w nagłówku odpowiedzi `Location`. W praktyce jest on tak przydatny, iż została zdefiniowana specjalna metoda, która go generuje — `sendRedirect`. Wykorzystanie wywołania o postaci `request.sendRedirect(url)` ma kilka zalet w porównaniu z wywołaniami `response.setStatus(response.SC_MOVED_TEMPORARILY)` oraz `response.setHeader("Location", url)`. Po pierwsze, jest ono krótsze i prostsze. Po wtóre, używając metody `sendRedirect` serwlet automatycznie tworzy stronę WWW zawierającą adres pod który przeglądarka zostaje przekierowana; strona ta jest przeznaczona dla starszych typów przeglądarek, które nie są w stanie automatycznie obsługiwać przekierowań. W końcu, w specyfikacji Java Servlet 2.2 (dostępnej w J2EE), metoda ta pozwala na stosowanie względnych adresów URL, automatycznie zamieniając je na adresy bezwzględne. W serwerach zgodnych ze specyfikacją 2.1, w metodzie `sendRedirect` można podawać wyłącznie adresy bezwzględne.

Jeśli kierujesz użytkownika do innej strony należącej do tej samej witryny, to przed podaniem jej adresu powinieneś przekształcić go przy użyciu metody `encodeURL` interfejsu `HttpServletResponse`. To proste zabezpieczenie na wypadek gdybyś korzystał ze śledzenia sesji bazującego na przepisywaniu adresów URL. Przepisywanie adresów URL to sposób śledzenia użytkowników, którzy odwiedzają witrynę i mają wyłączoną obsługę cookies. Jest on implementowany poprzez dodawanie dodatkowych informacji na końcu adresu URL, przy czym mechanizmy śledzenia sesji, którymi dysponują serwlety, automatycznie zajmują się wszelkimi szczegółami. Śledzenie sesji zostanie szczegółowo opisane w rozdziale 9. Warto jednak wyrobić sobie nawyk używania metody `encodeURL`, aby później można było wzbogacić witrynę o śledzenie sesji jak najmniejszym kosztem.

Metoda
Jeśli kierujesz użytkownika do innej strony należącej do tej samej witryny, to przygotuj się na wypadek gdybyś w przyszłości chciał korzystać ze śledzenia sesji. W tym celu zamiast wywołania:

```
response.sendRedirect(url)
```

użyj wywołania postaci:

```
response.sendRedirect(response.encodeURL(url)).
```

Czasami ten kod statusu jest używany zamiennie z kodem 301. Na przykład, gdy błędnie zażądasz strony *http://host/~użytkownik* (pomijając znak ukośnika na końcu adresu), to niektóre serwery odpowiedzą kodem statusu 301 a inne, kodem 302. Z technicznego punktu widzenia, przeglądarki powinne automatycznie obsłużyć przekierowanie wyłącznie wtedy, gdy początkowe żądanie było żądaniem GET. Więcej informacji na ten temat znajdziesz w opisie kodu statusu 307.

303 (**See Other** — patrz inny)

Kod statusu 304 (SC_SEE_OTHER) przypomina kody 301 oraz 302, z tą różnicą, że jeśli oryginalne żądanie było żądaniem POST, to nowy dokument (którego adres został podany w nagłówku odpowiedzi Location) powinien zostać pobrany przy użyciu żądania GET. Ten kod statusu został wprowadzony w protokole HTTP 1.1.

304 (**Not Modified** — niezmieniony)

Gdy program przechowuje dokument w pamięci podręcznej, może przeprowadzić warunkowe żądanie posługując się nagłówkiem If-Modified-Since, oznaczającym, że dokument ma być zwrócony wyłącznie wtedy, gdy został zmieniony po określonej dacie. Kod statusu 304 (SC_NOT_MODIFIED) oznacza, że wersja dokumentu przechowywana w pamięci podręcznej jest aktualna i to jej należy użyć. Jeśli dokument przechowywany w pamięci podręcznej nie będzie aktualny, serwer powinien zwrócić jego aktualną wersję wraz z kodem statusu 200. Zazwyczaj serwlety nie powinne stosować tego kodu bezpośrednio. Zamiast tego należy zaimplementować w serwlecie metodę getLastModified i pozwolić, aby domyślna metoda service obsługiwała żądania warunkowe na podstawie daty modyfikacji. Przykład zastosowania tej metody został przedstawiony w podrozdziale 2.8, pt.: „Przykład wykorzystania inicjalizacji serwletu i daty modyfikacji strony".

305 (**Use Proxy** — użyj serwera pośredniczącego)

Kod statusu 305 (SC_USE_PROXY) informuje, że żądany dokument powinien być pobrany przy wykorzystaniu serwera pośredniczącego, którego adres został podany w nagłówku odpowiedzi Location. Kod ten został wprowadzony w protokole HTTP 1.1.

307 (**Temporary Redirect** — chwilowo przekieruj)

Zasady obsługi kodu 307 przez przeglądarki niczym się nie różnią od metod obsługi kodu 302. Kod ten został dodany do protokołu HTTP 1.1, gdyż wiele przeglądarek po otrzymaniu odpowiedzi z kodem statusu 302 błędnie wykonywało przekierowanie nawet jeśli początkowe żądanie było żądanie POST. Po przesłaniu takiego żądania przeglądarki mają wykonywać przekierowanie wyłącznie jeśli zwrócony kod statusu będzie miał wartość 303. Ten nowy kod statusu ma wyjaśnić wszelkie niejednoznaczności — po otrzymaniu kodu 303 należy wykonać przekierowanie niezależnie od tego czy początkowe żądanie było żądaniem GET czy POST; jeśli jednak został zwrócony kod statusu 307 to przekierowanie należy wykonać wyłącznie jeśli początkowe żądanie było żądaniem GET. Notatka: Z niewiadomych powodów w interfejsie HttpServletResponse nie została zdefiniowana żadna stała odpowiadająca kodowi statusu 307. Ten kod statusu został wprowadzony w protokole HTTP 1.1.

Notatka
W interfejsie HttpServletResponse nie została zdefiniowana stała SC_TEMPORARY_REDIRECT, a zatem kod statusu 307 trzeba podawać jawnie.

400 (**Bad Request** — błędne żądanie)

Kod statusu 400 (SC_BAD_REQUEST) oznacza, że w żądaniu nadesłanym przez klienta wystąpił błąd składni.

401 (**Unauthorized** — nieupoważniony)

Kod statusu `401` (`SC_UNAUTHORIZED`) oznacza, że klient próbował pobrać stronę chronioną hasłem, bez poprawnego podania wymaganych informacji w nagłówku `Authorization`. Ta odpowiedź musi zawierać nagłówek `WWW-Authenticate`. Przykład zastosowania tego kodu statusu znajdziesz w podrozdziale 4.5, pt.: „Ograniczanie dostępu do stron WWW".

403 (**Forbidden** — zabroniony)

Kod statusu `403` (`SC_FORBIDDEN`) oznacza, że serwer odmawia przesłania żądanego zasobu, bez względu na informacje związane z autoryzacją użytkownika. Często się zdarza, że ten kod statusu jest zwracany gdy na serwerze zostaną błędnie określone prawa dostępu do plików i katalogów.

404 (**Not Found** — nie odnaleziono)

Niesławny kod statusu `404` (`SC_NOT_FOUND`) informuje, że nie został odnaleziony zasób o podanym adresie. Wartość ta jest standardową odpowiedzią oznaczającą „nie ma takiej strony". Kod `404` jest tak przydatny i często stosowany, iż w interfejsie `HttpServletResponse` została zdefiniowana specjalna metoda upraszczająca jego generację — `sendError(komunikat)`. Przewaga metody `sendError` nad `setStatus` polega na tym, iż metoda `sendError` automatycznie generuje stronę zawierającą komunikat o błędzie. Niestety Internet Explorer 5 domyślnie ignoruje strony z komunikatami o błędach przesyłane z serwera i wyświetla własne odpowiedniki tych stron (nawet pomimo faktu, iż działanie takie jest sprzeczne ze specyfikacją protokołu HTTP). Aby wyłączyć ten domyślny sposób działania, należy wybrać z menu głównego Internet Explorera opcje *Narzędzia⇨Opcje internetowe*, przejść na zakładkę *Zaawansowane* i upewnić się, że pole wyboru *Pokaż przyjazne komunikaty o błędach HTTP* nie jest zaznaczone. Niestety, niewielu użytkowników wie o tym aspekcie działania Internet Explorera 5, który uniemożliwia im oglądanie wszelkich stron z informacjami o błędach generowanych przez serwery WWW. Inne najpopularniejsze przeglądarki oraz Internet Explorer 4, poprawnie obsługują strony z informacjami o błędach. Przykłady takich stron możesz zobaczyć na rysunkach 6.3 oraz 6.4

Ostrzeżenie
Domyślnie Internet Explorer 5 ignoruje strony z komunikatami o błędach generowane przez serwery WWW.

405 (**Method Not Allowed** — niedozwolona metoda)

Kod statusu `405` (`SC_NOT_ALLOWED`) oznacza, że konkretne żądanie (`GET`, `POST`, `HEAD`, `PUT`, `DELETE`, itd.) nie może zostać użyte do pobrania wskazanego zasobu. Ten kod statusu został wprowadzony w protokole HTTP 1.1.

406 (**Not Acceptable** — nieakceptowany format)

Kod statusu `406` (`SC_NOT_ACCEPTABLE`) oznacza, że typ MIME żądanego zasobu nie jest zgodny z typami określonymi w nagłówku `Accept`. Nazwy oraz znaczenie najczęściej używanych typów MIME znajdziesz w tabeli 7.1 w podrozdziale 7.2, pt.: „Nagłówki odpowiedzi protokołu HTTP 1.1 oraz ich znaczenie". Ten kod statusu został wprowadzony w protokole HTTP 1.1.

407 (**Proxy Authentication Required** — wymagana autoryzacja na serwerze pośredniczącym)

Kod statusu `407` (`SC_PROXY_AUTHENTICATION_REQUIRED`) przypomina kod `401` lecz jest używany przez serwery pośredniczące. Oznacza on, że klient musi potwierdzić swą tożsamość na serwerze pośredniczącym. Serwer taki przesyła do klienta nagłówek odpowiedzi `Proxy-Authenticate`, co powoduje, że program ponownie nawiązuje połączenie przesyłając przy tym

nagłówek żądania `Proxy-Authorization`. Ten kod statusu został wprowadzony w protokole HTTP 1.1.

408 (Request Timeout — limit czasu żądania)

Kod statusu `408` (`SC_REQUEST_TIMEOUT`) oznacza, że klient zbyt długo wysyłał żądanie. Kod ten został wprowadzony w protokole HTTP 1.1.

409 (Conflict — konflikt)

Zazwyczaj kod statusu `409` (`SC_CONFLICT`) jest skojarzony z żądaniami `PUT`. Jest on zwracany, na przykład, w takich sytuacjach jak próba przesłania na serwer nieodpowiedniej wersji pliku. Ten kod statusu został wprowadzony w protokole HTTP 1.1.

410 (Gone — niedostępny)

Kod statusu `410` (`SC_GONE`) informuje, że żądany dokument został przeniesiony lecz jego aktualny adres nie jest znany. Kod `410` różni się od kodu `404`, gdyż oznacza, że dokument został przeniesiony na stałe, a nie jest chwilowo niedostępny z nieznanych powodów. Ten kod statusu został wprowadzony w protokole HTTP 1.1.

411 (Length Required — wymagana długość)

Kod statusu `411` (`SC_LENGTH_REQUIRED`) oznacza, że serwer nie będzie w stanie obsłużyć żądania (zazwyczaj żądania `POST` z dołączonym dokumentem) jeśli program nie prześle nagłówka `Content-Length` zawierającego informacje o ilości informacji przekazanych na serwer. Ten kod statusu został wprowadzony w protokole HTTP 1.1.

412 (Precondition Failed — niespełnione wymagania)

Kod statusu `412` (`SC_PRECONDITION_FAILED`) informuje, że niektóre z wymagań wstępnych określonych w nagłówkach żądania nie zostały spełnione. Kod ten został wprowadzony w protokole HTTP 1.1.

413 (Request Entity Too Large — żądany zasób zbyt duży)

Kod statusu `413` (`SC_REQUEST_ENTITY_TOO_LARGE`) informuje, że żądany dokument jest zbyt duży, aby serwer chciał go obsłużyć w danym momencie. Jeśli serwer dopuszcza możliwość obsłużenia żądania w późniejszym terminie, może dołączyć do odpowiedzi nagłówek `Retry-After`. Ten kod statusu został wprowadzony w protokole HTTP 1.1.

414 (Request URI Too Long — żądany URI zbyt długi)

Kod statusu `414` (`SC_REQUEST_URI_TOO_LONG`) jest stosowany w sytuacjach gdy podany URI jest zbyt długi. W tym kontekście, „URI" oznacza część adresu URL rozpoczynającą się po nazwie komputera i numerze portu. Na przykład, w adresie URL: *http://www.y2k.com:8080/ale/mamy/miny*, URI to */ale/mamy/miny*. Ten kod statusu został wprowadzony w protokole HTTP 1.1.

415 (Unsupported Media Type — nieobsługiwany typ danych)

Kod statusu `415` (`SC_UNSUPPORTED_MEDIA_TYPE`) oznacza, że do żądania był dołączony plik, lecz serwer nie wie jak należy obsługiwać pliki tego typu. Ten kod statusu został wprowadzony w protokole HTTP 1.1.

416 (Requested Range Not Satisfiable — nieprawidłowy zakres)

Kod statusu `416` oznacza, że klient dołączył do żądania nagłówek `Range`, który nie może być zaakceptowany. Został on wprowadzony w protokole HTTP 1.1. Ciekawe, że w definicji interfejsu

`HttpServletResponse` specyfikacji Java Servlet 2.1, została pominięta stała odpowiadająca temu kodowi statusu.

Notatka
W specyfikacji Java Servlet 2.1, w interfejsie `HttpServletResponse` nie została zdefiniowana stała `SC_REQUESTED_RANGE_NOT_SATISFIABLE`; a zatem, aby wygenerować ten kod statusu należy jawnie podać liczbę `416`. Stała odpowiadająca temu kodowi statusu jest już dostępna w specyfikacjach 2.2 i późniejszych.

417 (**Expectation Failed** — oczekiwania niemożliwe do spełnienia)

Jeśli serwer otrzyma nagłówek żądania `Expect` o wartości `100-continue`, będzie on oznaczał, że klient pyta, czy w następnym żądaniu może przesłać dokument. W takiej sytuacji serwer powinien zwrócić kod statusu `417` informując, że dokument nie zostanie przyjęty, bądź kod statusu `100` (`SC_CONTINUE`) informujący, że program może przysyłać dokument. Ten kod statusu został wprowadzony w protokole HTTP 1.1.

500 (**Internal Server Error** — wewnętrzny błąd serwera)

Kod statusu `500` (`SC_INTERNAL_SERVER_ERROR`) to ogólny kod oznaczający, że „serwer nie za bardzo wie co należy zrobić". Bardzo często jest on zwracany z programów CGI lub (nie daj Boże) serwletów, które nie zostały wykonane poprawnie lub zwróciły nieprawidłowo sformatowane nagłówki.

501 (**Not Implemented** — niezaimplementowane możliwości)

Kod statusu `501` (`SC_NOT_IMPLEMENTED`) informuje, że serwer nie dysponuje możliwościami funkcjonalnymi koniecznymi do obsłużenia żądania. Kod ten jest stosowany na przykład w sytuacjach gdy klient prześle polecenie (takie jak `PUT`) którego serwer nie obsługuje.

502 (**Bad Gateway** — zła brama)

Kod statusu `502` (`SC_BAD_GATEWAY`) jest używany przez serwery działające jako serwery pośredniczące; oznacza on, że serwer otrzymał od innego serwer błędne żądanie.

503 (**Service Unavailable** — usługa niedostępna)

Kod statusu `503` (`SC_SERVICE_UNAVAILABLE`) oznacza, że serwer nie jest w stanie odpowiedzieć ze względu na przeciążenie, bądź czynności związane z jego utrzymaniem. Przykładowo, serwlet może zwrócić ten kod statusu jeśli w danej chwili są używane wszystkie wątki lub połączenia z bazą danych. Serwer może dołączyć do odpowiedzi nagłówek `Retry-After`, aby poinformować program kiedy ten może ponownie przesłać żądanie.

504 (**Gateway Timeout** — przekroczony czas oczekiwania bramy)

Kod statusu `504` (`SC_GATEWAY_TIMEOUT`) jest stosowany przez serwery działające jako bramy lub serwery pośredniczące; oznacza on, że serwer początkowy nie uzyskał na czas odpowiedzi od zdalnego serwera. Ten kod statusu został wprowadzony w protokole HTTP 1.1.

505 (**HTTP Version Not Supported** — nieobsługiwana wersja HTTP)

Kod statusu `515` (`SC_HTTP_VERSION_NOT_SUPPORTED`) oznacza, że serwer nie obsługuje wersji protokołu HTTP podanej w wierszu żądania. Ten kod został wprowadzony w protokole HTTP 1.1.

6.3 Interfejs użytkownika obsługujący różne serwisy wyszukiwawcze

Listing 6.1 przedstawia przykład serwletu wykorzystującego jedne z dwóch najpopularniejszych kodów statusu — `302` (`Found`) oraz `404` (`Not Found`). Kod statusu `302` jest generowany przez metodę `sendRedirect` interfejsu `HttpServletResponse`, natomiast do generacji kodu `404` służy metoda `sendError`.

W tej aplikacji, użytkownik w pierwszej kolejności powinien wyświetlić formularz HTML (został on przedstawiony na rysunku 6.1, a jego kod źródłowy na listingu 6.3), który pozwala określić łańcuch zapytania, ilość wyników wyświetlanych na jednej stronie oraz mechanizm wyszukiwawczy jakiego należy użyć. Po przesłaniu formularza, serwlet pobiera wartości tych trzech parametrów, tworzy adresu URL umieszczając w nim informacje zapisane w sposób odpowiedni dla wybranego mechanizmu wyszukiwawczego (patrz klasa `SearchSpec` przedstawiona na listingu 6.2) i kieruje przeglądarkę użytkownika pod stworzony adresu URL (patrz rysunek 6.2). Jeśli użytkownik wybierze zły mechanizm wyszukiwawczy lub nieprawidłowo określi kryteria wyszukiwania, zwracana jest strona informująca o popełnionym błędzie (patrz rysunki 6.3 oraz 6.4).

Listing 6.1 SearchEngines.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;

/** Servlet który pobiera zapytanie, ilość wyników na stronę
 *  oraz nazwę serwisu wyszukiwawczego, a następnie
 *  konstruuje zapytanie i przesyła je do wybranego serwisu.
 *  Przykład ilustruje wykorzystanie kodów statusu.
 *  Servlet przesyła kod 302 (generowany przy użyciu metody
 *  sendRedirect) jeśli wybrana nazwa serwisu jest znana,
 *  oraz kod statusu 404 (przy użyciu metody sendError) w
 *  przeciwnym przypadku.
 */

public class SearchEngines extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    String searchString = request.getParameter("searchString");
    if ((searchString == null) ||
        (searchString.length() == 0)) {
      reportProblem(response, "Brak zapytania.");
      return;
    }
    /* Klasa URLEncoder zamienia odstępy na znaki "+", a
     * pozostałe znaki, które nie są znakami alfanumerycznymi
     * na wyrażenia o postaci "%XY"; gdzie XY wartością danego
     * znaku w kodzie ASCII (lub ISO Latin-1) zapisaną w formie
     * liczby szesnastkowej. Przeglądarki zawsze automatycznie
     * kodują wartości podawane w formularzach w ten
     * sposób, a zatem metoda getParameter dekoduje je
     * automatycznie. My jednak przekazujemy te dane do innego
     * serwera, a zatem musimy je ponownie zakodować.
     */
    searchString = URLEncoder.encode(searchString);
    String numResults = request.getParameter("numResults");
    if ((numResults == null) ||
        (numResults.equals("0")) ||
        (numResults.length() == 0)) {
      numResults = "10";
    }
    String searchEngine =
      request.getParameter("searchEngine");
    if (searchEngine == null) {
      reportProblem(response, "Brak nazwy serwisu wyszukiwawczego.");
```

```
      return;
    }
    SearchSpec[] commonSpecs = SearchSpec.getCommonSpecs();
    for(int i=0; i<commonSpecs.length; i++) {
      SearchSpec searchSpec = commonSpecs[i];
      if (searchSpec.getName().equals(searchEngine)) {
        String url =
          searchSpec.makeURL(searchString, numResults);
        response.sendRedirect(url);
        return;
      }
    }
    reportProblem(response, "Nieznany serwis wyszukiwawczy.");
  }

  private void reportProblem(HttpServletResponse response,
                             String message)
      throws IOException {
    response.sendError(response.SC_NOT_FOUND,
                       "<H2>" + message + "</H2>");
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

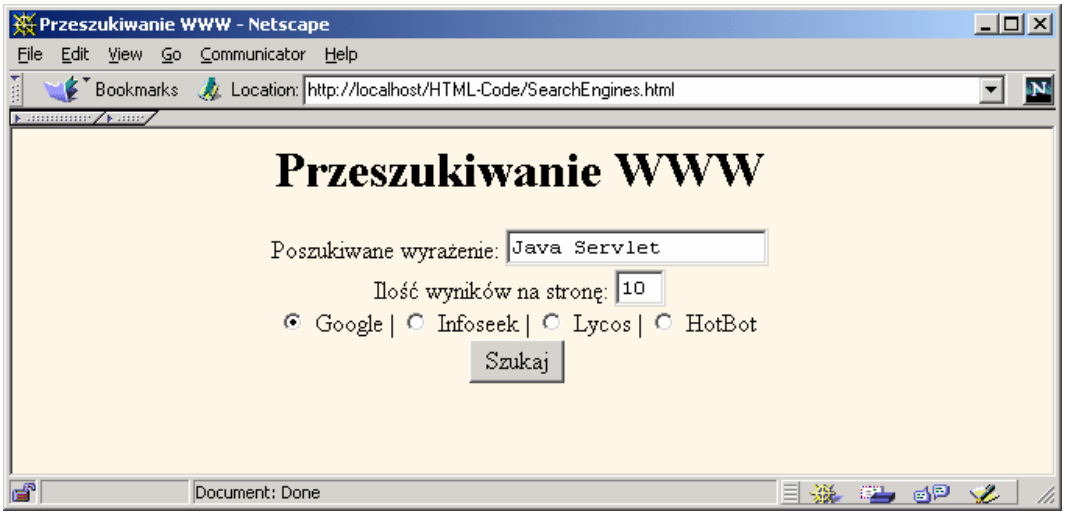
Listing 6.2 SearchSpec.java

```
package coreservlets;

/** Niewielka klasa zawierająca metody tworzenia
 *  łańcuchów zapytania charakterystycznych dla
 *  różnych mechanizmów wyszukiwawczych.
 */

public class SearchSpec {
  private String name, baseURL, numResultsSuffix;

  private static SearchSpec[] commonSpecs =
    { new SearchSpec("google",
                     "http://www.google.com/search?q=",
                     "&num="),
      new SearchSpec("infoseek",
                     "http://infoseek.go.com/Titles?qt=",
                     "&nh="),
      new SearchSpec("lycos",
                     "http://lycospro.lycos.com/cgi-bin/" +
                        "pursuit?query=",
                     "&maxhits="),
      new SearchSpec("hotbot",
                     "http://www.hotbot.com/?MT=",
                     "&DC=")
    };

  public SearchSpec(String name,
                    String baseURL,
                    String numResultsSuffix) {
    this.name = name;
    this.baseURL = baseURL;
    this.numResultsSuffix = numResultsSuffix;
  }

  public String makeURL(String searchString,
                        String numResults) {
    return(baseURL + searchString +
           numResultsSuffix + numResults);
  }

  public String getName() {
    return(name);
  }

  public static SearchSpec[] getCommonSpecs() {
    return(commonSpecs);
```

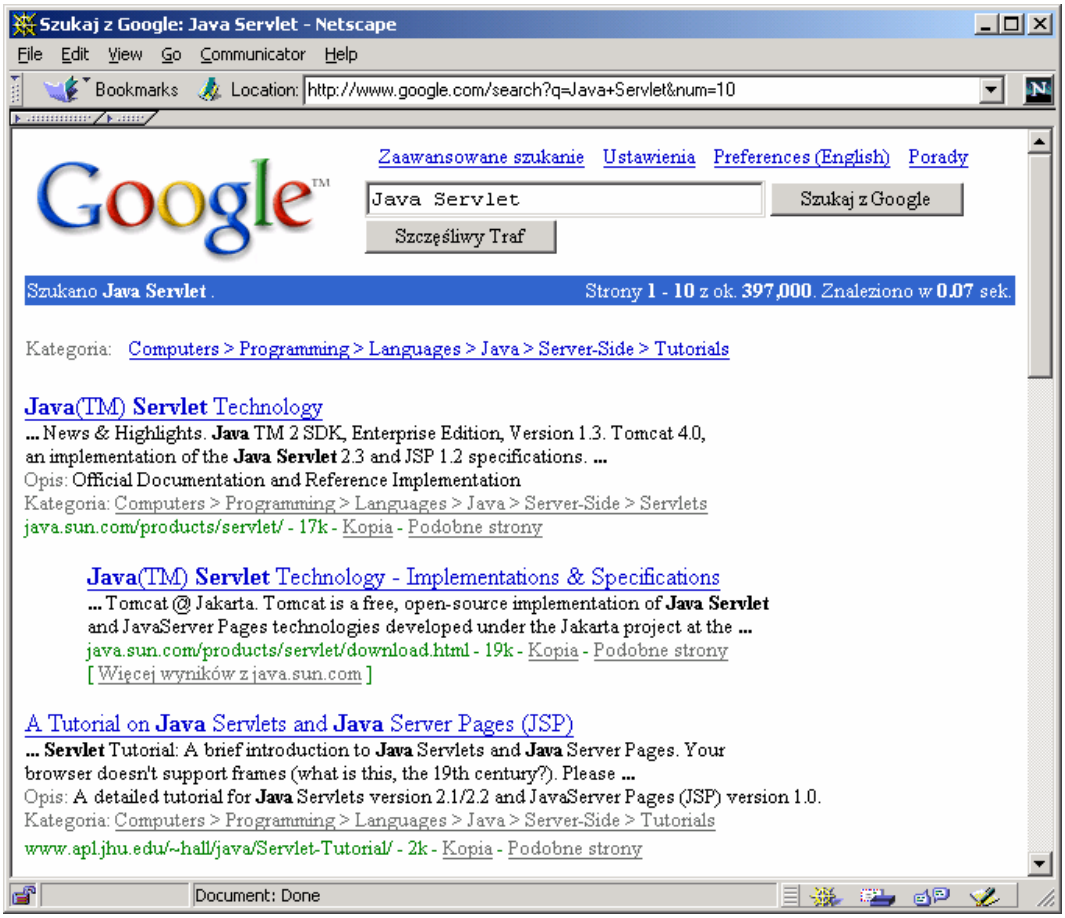
```
  }
}
```

Listing 6.3 SearchEngines.html

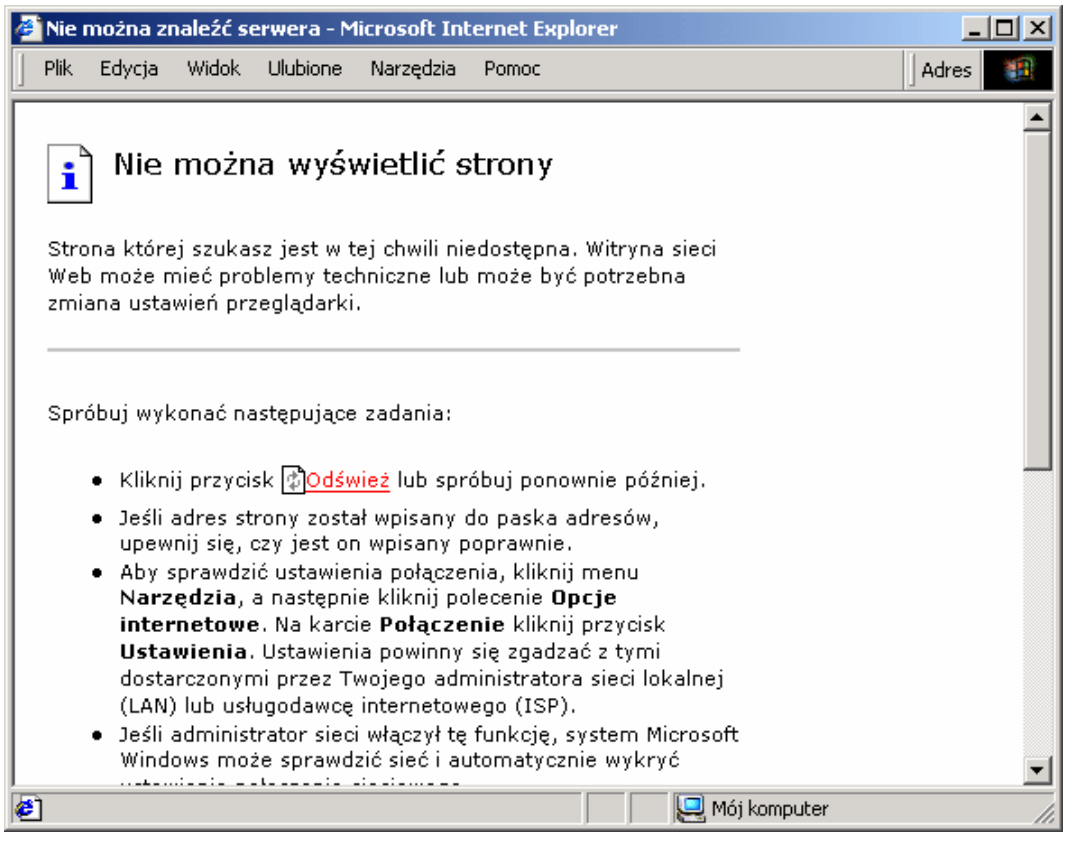
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<!--
Interfejs użytkownika dla servletu przekierowującego żądania
do wskazanych mechanizmów wyszukiwawczych.
-->
<HTML>
<HEAD>
  <TITLE>Przeszukiwanie WWW</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">Przeszukiwanie WWW</H1>

<FORM ACTION="/servlet/coreservlets.SearchEngines">
  <CENTER>
    Poszukiwane wyrażenie:
    <INPUT TYPE="TEXT" NAME="searchString"><BR>
    Ilość wyników na stronę:
    <INPUT TYPE="TEXT" NAME="numResults"
                       VALUE=10 SIZE=3><BR>
    <INPUT TYPE="RADIO" NAME="searchEngine"
                        VALUE="google">
    Google |
    <INPUT TYPE="RADIO" NAME="searchEngine"
                        VALUE="infoseek">
    Infoseek |
    <INPUT TYPE="RADIO" NAME="searchEngine"
                        VALUE="lycos">
    Lycos |
    <INPUT TYPE="RADIO" NAME="searchEngine"
                        VALUE="hotbot">
    HotBot
    <BR>
    <INPUT TYPE="SUBMIT" VALUE="Szukaj">
  </CENTER>
</FORM>

</BODY>
</HTML>
```



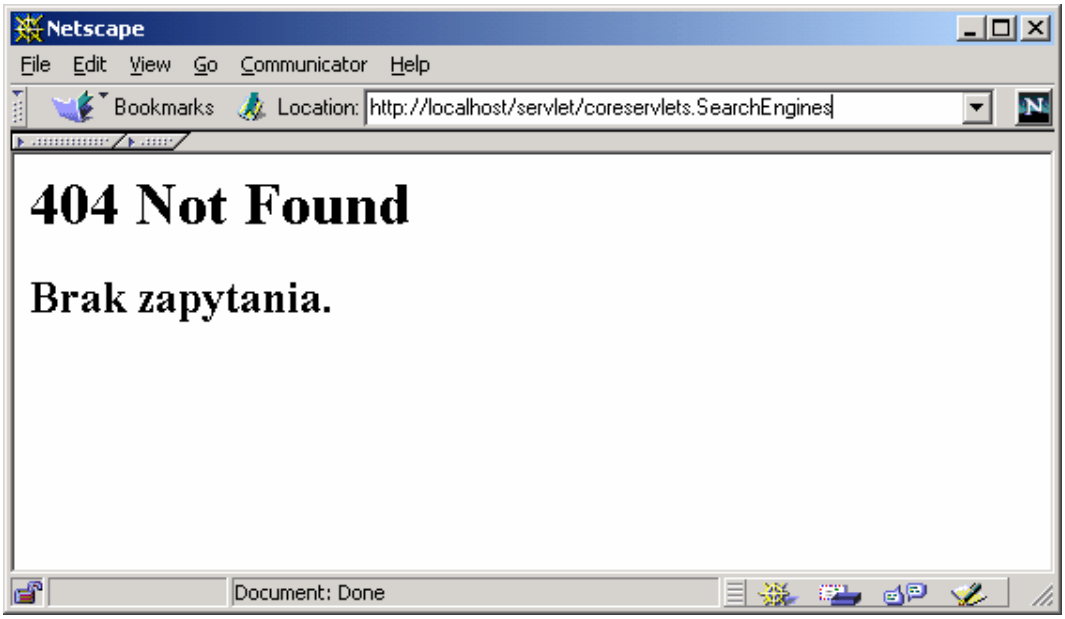
Rysunek 6.1 Formularz stanowiący interfejs użytkownika dla serwletu SearchEngines. Jego kod źródłowy został przedstawiony na listingu 6.3



Rysunek 6.2 Wyniki wykonania serwletu SearchEngines po przekazaniu danych przedstawionych na rysunku 6.1



Rysunek 6.3 Wyniki wykonania serwletu SearchEngines w przypadku gdy nie zostały określone żadne kryteria wyszukiwania. Internet Explorer 5 wyświetla własną stronę z komunikatem o błędzie, nawet pomimo faktu, iż serwlet generuje własną stronę informacyjną



Rysunek 6.4 Wyniki wykonania serwletu SearchEngines w przypadku gdy nie zostały określone żadne kryteria wyszukiwania. W Netscape Navigatorze została wyświetlona strona wygenerowana przez serwlet.

Rozdział 7.
Generacja odpowiedzi: Nagłówki odpowiedzi HTTP

Odpowiedź generowana przez serwer WWW składa się zazwyczaj z wiersza statusu, jednego lub kilku nagłówków odpowiedzi, pustego wiersza oraz dokumentu. Aby w jak największym stopniu wykorzystać możliwości serwletów, będziesz musiał wiedzieć nie tylko jak generować dokument, lecz także jak należy efektywnie używać kodów statusu i nagłówków odpowiedzi.

Określanie nagłówków odpowiedzi HTTP często jest ściśle powiązane z określaniem kodów statusu, którym poświęciłem cały poprzedni rozdział. Na przykład, wszystkim kodom statusu informującym że „dokument został przeniesiony" (o wartościach z zakresu od `300` do `307`) towarzysz nagłówek odpowiedzi `Location`, natomiast kod `401` (`Unauthorized`) zawsze występuje wraz z nagłówkiem `WWW-Authenticate`. Niemniej jednak określanie nagłówków odpowiedzi może mieć ważne znaczenie także w sytuacjach, gdy żadne niezwykłe kody statusu nie są stosowane. Nagłówki odpowiedzi mogą służyć do tworzenia cookies, określania daty modyfikacji strony (informacje te są wykorzystywane przez przeglądarki przy przechowywaniu stron WWW a pamięci podręcznej), nakazania przeglądarce, aby odświeżyła stronę po określonym czasie, do podania wielkości pliku, dzięki czemu będzie można wykorzystać trwałe połączenie HTTP, do określenia typu generowanego dokumentu oraz wykonywania wielu innych czynności.

7.1 Określanie nagłówków odpowiedzi z poziomu serwletów

Najbardziej ogólną metodą określania nagłówków odpowiedzi jest zastosowanie metody `setHeader` interfejsu `HttpServletResponse`. Metoda ta wymaga podania dwóch argumentów, będących łańcuchami znaków — pierwszym z nich jest nazwa nagłówka, a drugim jego wartość. Podobnie jak w przypadku kodów statusu, także nagłówki odpowiedzi należy określać przed rozpoczęciem generacji samego dokumentu. W przypadku specyfikacji Java Servlet 2.1 oznacza to, że nagłówki odpowiedzi należy określić przed pierwszym wykorzystaniem metod klas `PrintWriter` bądź `OutputStream`, służących do generacji treści dokumentu. W specyfikacji 2.2 serwletów (dostępnej w J2EE) klasa `PrintWriter` może korzystać z buforu, a zatem nagłówki odpowiedzi można określać aż do momentu pierwszego opróżnienia jego zawartości. Więcej informacji na ten temat znajdziesz w podrozdziale 6.1, pt.: „Określanie kodów statusu".

Metoda
Pamiętaj, aby podać nagłówki odpowiedzi przed przesłaniem zawartości dokumentu do klienta.

Oprócz ogólnej metody `setHeader`, interfejs `HttpServletResponse` posiada dwie bardziej specjalizowane metody służące do określania nagłówków zawierających daty oraz liczby całkowite. Oto te metody:

- **`setDateHeader(String nagłówek, long milisekundy)`**

  Użycie tej metody zaoszczędza nam problemów związanych z konwersją dat stosowanych w języku Java i wyrażonych w postaci milisekund jakie upłynęły od początku 1970 roku (wartość ta jest zwracana przez metody `System.currentTimeMillis`, `Date.getTime` oraz `Calendar.getTimeInMillis`), do postaci łańcucha znaków zawierającego datę zapisaną w formacie GMT.

- **`setIntHeader(String nagłówek, int wartośćNagłówka)`**

  Ta metoda zaoszczędza nam niewielkich problemów związanych z koniecznością skonwertowania liczby całkowitej do postaci łańcucha znaków, nim jej wartość będziemy mogli użyć w nagłówku odpowiedzi.

Protokół HTTP pozwala, aby w jednej odpowiedzi kilkukrotnie pojawiał się nagłówek o tej samej nazwie; czasami nawet będziesz wolał dodać nowy niż zastępować już istniejący nagłówek. Na przykład, często się zdarza, że w odpowiedzi pojawia się kilka nagłówków `Accept` lub `Set-Cookie`, które odpowiednio określają kilka różnych obsługiwanych typów MIME lub kilka różnych cookies. W specyfikacji Java Servlet 2.1 metody `setHeader`, `setDateHeader` oraz `setIntHeader` wyłącznie *dodają* nowe nagłówki, a zatem nie ma sposobu aby usunąć nagłówek, który już został podany. W specyfikacji 2.2 metody `setHeader`, `setDateHeader` oraz `setIntHeader` *zastępują* inne istniejące nagłówki o tej samej nazwie, natomiast metody `addHeader`, `addDateHeader` oraz `addIntHeader` dodają nowy nagłówek, niezależnie od tego czy w odpowiedzi zostały już zdefiniowane inne nagłówki o tej samej nazwie czy nie. Metoda `containsHeader` pozwala sprawdzić czy nagłówek o podanej nazwie już został podany; możesz jej używać jeśli fakt istnienia nagłówka ma dla Ciebie jakieś znaczenie.

Interfejs `HttpServletResponse` zawiera także kilka metod upraszczających generację najczęściej stosowanych nagłówków. Poniżej pokrótce przedstawiłem te metody:

- **`setContentType`**

  Ta metoda generuje nagłówek odpowiedzi `Content-Type` i jest używana w przeważającej większości serwletów. Przykład jej zastosowania przedstawiłem w podrozdziale 7.5, pt.: „Wykorzystanie serwletów do generacji obrazów GIF”.

- **`setContentLength`**

  Ta metoda generuje nagłówek odpowiedzi `Content-Length`, który jest przydatny, gdy przeglądarka obsługuje trwałe połączenia HTTP. Przykład jej zastosowania znajdziesz w podrozdziale 7.4.

- **`addCookie`**

  Ta metoda dodaje cookie do nagłówka `Set-Cookie`. Nie ma metody `setCookie`, gdyż jest całkowicie normalne, że w jednej odpowiedzi może się znaleźć kilka wierszy `Set-Cookie`. Cookies zostaną szczegółowo omówione w rozdziale 8.

- **`sendRedirect`**

  Zgodnie z tym co podałem w poprzednim rozdziale, metoda `sendRedirect` generuje nagłówek odpowiedzi `Location` oraz kod statusu `302`. Przykład wykorzystania tej metody znajdziesz w podrozdziale 6.3, pt.: „Interfejs użytkownika obsługujący różne serwisy wyszukiwawcze”.

7.2 Nagłówki odpowiedzi protokołu HTTP 1.1 oraz ich znaczenie

Ta część rozdziału zawiera omówienie nagłówków odpowiedzi protokołu HTTP 1.1. Dobra znajomość i zrozumienie tych nagłówków może poprawić efektywność działania tworzonych serwletów; z tego względu powinieneś przynajmniej pobieżnie przejrzeć podane tu informacje, aby zorientować się jakie możliwości są dostępne. Później, gdy będziesz gotów aby wykorzystać te możliwości, możesz uważnie przeczytać tę część rozdziału i zdobyć potrzebne, szczegółowe informacje. Skrócone omówienie przedstawionych tu nagłówków znajduje się także w dodatku A, pt.: „Krótki przewodnik po serwletach i JSP".

Nagłówki odpowiedzi protokołu HTTP 1.1 stanową nadzbiór nagłówków odpowiedzi protokołu HTTP 1.1. Wszelkie dodatkowe informacje na ich temat można znaleźć w specyfikacji protokołu HTTP 1.1 podanej w pliku RFC 2616. Oficjalne wersje plików RFC można znaleźć w wielu miejscach na Internecie, aktualna lista takich witryn znajduje się pod adresem *http://www.rfc-editor.org/*. Przy zapisywaniu nazw nagłówków wielkość liter nie odgrywa znaczenia, jednak tradycyjnie każde słowo wchodzące w skład nazwy nagłówka zaczyna się z dużej litery.

Wiele starszych przeglądarek korzysta wyłącznie z protokołu HTTP 1.0, dlatego też należy zachować szczególną uwagę przy tworzeniu serwletów, których zachowanie zależy od nagłówków odpowiedzi dostępnych tylko w protokole HTTP 1.1. Dotyczy to zwłaszcza tych sytuacjach, gdy serwlety działają na Internecie a nie na lokalnym intranecie. Najlepszym rozwiązaniem jest sprawdzenie wersji używanego protokołu HTTP zanim użyjesz nowych nagłówków, można się w tym celu posłużyć wywołaniem `request.getProtocol()`.

Accept-Ranges

Ten nagłówek, wprowadzony w protokole HTTP 1.1, informuje czy akceptujesz nagłówki żądania `Range`. Zazwyczaj w nagłówku tym podaje się wartość `bytes` oznaczającą, że nagłówki `Range` są akceptowane, bądź wartość none w przeciwnym przypadku.

Age

Ten nagłówek stosowany jest przez serwery pośredniczące, aby poinformować kiedy dokument został wygenerowany przez serwer z którego pochodzi. Nagłówek ten został wprowadzony w protokole HTTP 1.1 i rzadko kiedy jest stosowany w serwletach.

Allow

Nagłówek `Allow` określa metody żądań (na przykład: `GET`, `POST`, `HEAD`, itp.) obsługiwanych przez serwer. Nagłówek ten musi zostać podany w przypadku zwracania odpowiedzi zawierającej kod statusu `405` (`Method Not Allowed` — niedozwolona metoda). Domyślna metoda `service` serwletów, automatycznie generuje ten nagłówek w przypadku obsługi żądań `OPTIONS`.

Cache-Control

Ten bardzo przydatny nagłówek przekazuje przeglądarce lub innemu klientowi informacje o okolicznościach w jakich dokument zawarty w odpowiedzi może być bezpiecznie przechowany w pamięci podręcznej. Nagłówek ten może przybierać poniższe wartości:

- `public` — dokument może być przechowywany w pamięci podręcznej, nawet jeśli normalne reguły (na przykład: reguły dotyczące stron chronionych hasłem) wskazują, że nie powinien być.
- `private` — dokument jest przeznaczony dla jednego użytkownika i może być przechowywany tylko w „prywatnej" pamięci podręcznej, z której wyłącznie ten jeden użytkownik będzie mógł go pobrać.

- `no-cache` — dokument nigdy nie powinien być przechowywany w pamięci podręcznej (czyli nie powinien być zwracany przy obsłudze kolejnych żądań). Serwer może także użyć nagłówka o postaci "`no-cache=nagłówek1,nagłówek2,...,nagłówekN`", aby określić nagłówki, które powinne być usunięte z odpowiedzi, jeśli później zostanie zwrócona odpowiedź przechowywana w pamięci podręcznej. Przeglądarki zazwyczaj nie przechowują w pamięci podręcznej dokumentów wygenerowanych w odpowiedzi na żądanie zawierające informacje podane w formularzu. Niemniej jednak, jeśli serwlet generuje różne odpowiedzi nawet jeśli żądanie nie zawiera informacji z formularza, to konieczne jest przekazanie przeglądarce informacji, że odpowiedź nie powinna być zapisywana w pamięci podręcznej. Starsze typy przeglądarek używają do tego celu nagłówka odpowiedzi `Pragma`, dlatego też typowym rozwiązaniem stosowanym w serwletach jest podanie obu nagłówków, tak jak pokazałem na poniższym przykładzie.
```
response.setHeader("Cache-Control", "no-cache");
response.setHeader("Pragma", "no-cache");
```
- `no-store` — dokument nigdy nie powinien być przechowywany w pamięci podręcznej, a nawet nie powinien być zapisywany w pliku tymczasowym na dysku. Ten nagłówek stosowany jest, aby zapobiec nierozważnemu tworzeniu kopii istotnych informacji.
- `must-revalidate` — za każdym razem gdy klient chce użyć kopii dokumentu przechowywanej w pamięci podręcznej, powinien sprawdzić jej ważność na serwerze, który wygenerował ten dokument.
- `proxy-revalidate` — ta wartość jest bardzo podobna do poprzedniej, z tym, że dotyczy pamięci podręcznych wykorzystywanych przez wielu użytkowników.
- `max-age=xxx` — dokument powinien być uznany za nieważny po upłynięciu *xxx* sekund. To bardzo wygodna alternatywa dla nagłówka odpowiedzi `Expires`, jednak można z niej korzystać wyłącznie w programach korzystających z protokołu HTTP 1.1. Jeśli w odpowiedzi zostanie podany nagłówek `Cache-Control` z wartością `max-age` jak i `Expires`, to nagłówek `Cache-Control` będzie miał wyższy priorytet.
- `s-max-age=xxx` — pamięci podręczne wykorzystywane przez większą ilość użytkowników powinne uznać, że dokument jest nieważny po upłynięciu *xxx* sekund.

Connection

Przekazanie wartości `close` w tym nagłówku odpowiedzi informuje przeglądarkę, aby nie korzystała z trwałych połączeń HTTP. Z technicznego punktu widzenia, jeśli klient korzysta z protokołu HTTP 1.1 i *nie* zostanie użyty nagłówek `Connection: close`, to trwałe połączenia są stosowane domyślnie. (Trwałe połączenia HTTP będą także wykorzystane w programach korzystających z protokołu HTTP 1.0, jeśli zostanie podany nagłówek `Connection: keep-alive`.) Użycie trwałych połączeń HTTP wymaga jednak podania nagłówka odpowiedzi `Content-Length` i z tego względu serwlety nie muszą jawnie stosować nagłówka `Connection`. Jeśli bowiem nie chcesz stosować trwałych połączeń, wystarczy pominąć w odpowiedzi nagłówek `Content-Length`. Przykład wykorzystania trwałych połączeń w serwletach został przedstawiony w podrozdziale 7.4, pt.: „Stosowanie trwałych połączeń HTTP”.

Content-Encoding

Ten nagłówek określa sposób w jaki strona została zakodowana na czas przesyłania. Przeglądarka powinna odtworzyć kodowanie i przywrócić dokument do oryginalnego stanu, zanim podejmie decyzję co z nim należy dalej zrobić. Kompresja dokumentów przy użyciu algorytmu gzip jest w stanie w ogromnym stopniu skrócić czas ich transmisji; przykład wykorzystania takiego rozwiązania przedstawiłem w podrozdziale 4.4, pt.: „Przesyłanie skompresowanych stron WWW”.

Content-Language

Ten nagłówek określa język, w jakim dokument został napisany. Wartością tego nagłówka powinien być jeden ze standardowych kodów języków, takich jak: `pl`, `en`, `en-us`, `da`, itp. Więcej szczegółowych informacji na ten temat znajdziesz w pliku RFC 1766 (pliki RFC możesz znaleźć na WWW, na przykład na witrynie *http://www.rfc-editor.org/*.)

Content-Length

Ten nagłówek określa wielkość dokumentu przekazywanego w odpowiedzi (wyrażoną w bajtach). Informacja ta jest potrzebna wyłącznie w sytuacji, gdy przeglądarka korzysta z trwałych połączeń HTTP. Więcej informacji o sposobie określania, kiedy przeglądarka korzysta z trwałych połączeń znajdziesz w opisie nagłówka odpowiedzi `Connection`. Jeśli chcesz korzystać z trwałych połączeń w serwlecie (oczywiście, o ile na to pozwala przeglądarka), to powinieneś zapisać cały dokument w strumieniu `ByteArrayOutputStream`, określić jego długość, zapisać ją w nagłówku `Content-Type` przy użyciu metody `response.setContentLength`, a następnie przesłać dokument za pomocą wywołania o postaci `byteArrayStream.writeTo(response.getOutputStream))`. Przykład praktycznego wykorzystania trwałych połączeń HTTP znajdziesz w podrozdziale 7.4.

Content-Location

Ten nagłówek określa alternatywny adres żądanego dokumentu. Jest to nagłówek informacyjny, a odpowiedzi które go zawierają, zawierają także żądany dokument. A zatem odpowiedzi te różnią się od odpowiedzi zawierających nagłówek `Location`, do których nie jest dołączany żaden dokumentu. Nagłówek ten został wprowadzony w protokole HTTP 1.1.

Content-MD5

Nagłówek odpowiedzi `Content-MD5` zawiera kod MD5 obliczony dla przesyłanego dokumentu. Kod ten daje możliwość sprawdzenia integralności danych, z której mogą korzystać przeglądarki jeśli chcą upewnić się, że otrzymały kompletny dokument w niezmienionej postaci. Szczegółowe informacje na temat kodu MD5 znajdziesz w pliku RFC 1864. Nagłówek ten został wprowadzony w protokole HTTP 1.1.

Content-Range

Ten nowy nagłówek, wprowadzony w protokole HTTP 1.1, jest przesyłany w odpowiedziach zawierających fragmenty zwracanego dokumentu i określa jaka część tego dokumentu jest aktualnie przesyłana. Na przykład, gdyby wartość tego nagłówka miała postać `bytes 500-999/2345`, oznaczałoby to, że odpowiedź zawiera bajty od 500-nego do 999-go, a cały dokument liczy 2345 bajtów.

Content-Type

Nagłówek `Content-Type` określa typ MIME przesyłanego dokumentu. Nagłówka tego używa się tak często, że interfejs `HttpServletResponse` zawiera specjalną metodę, która służy do jego generacji. Metoda tą jest `setContentType`. W przypadku oficjalnie zarejestrowanych typów MIME, ich nazwy zapisywane są w postaci `typgłówny/podtyp`; niezarejestrowane typy mają natomiast nazwy postaci `typgłówny/x-podtyp`. Domyślnym typem MIME stosowanym przez serwlety jest `text/plain`, lecz zazwyczaj serwlety jawnie definiują typ `text/html`. Oczywiście, można także używać innych typów MIME. Przykład z podrozdziału 7.5, pt.: „Wykorzystanie serwletów do generacji obrazów GIF" przedstawia serwlet, który na podstawie podanych informacji generuje obrazy GIF wykorzystując przy tym typ MIME `image/gif`; natomiast w podrozdziale 11.2, pt.:

„Atrybut contentType", przedstawię serwlet oraz strony JSP, które dzięki wykorzystaniu typu MIME `application/vnd.ms-excel` generują arkusz kalkulacyjny programu Microsoft Excel.

Tabela 7.1 przedstawia typy MIME, najczęściej stosowane w serwletach.

Tabela 7.1 Najczęściej stosowane typy MIME.

Typ	Znaczenie
`application/msword`	Dokument programu Microsoft Word
`application/octet-stream`	Dane binarne lub dane, które nie zostały zapisane w żaden konkretny sposób
`application/pdf`	Plik Adob Acrobat (*.pdf*)
`application/postscript`	Plik PostScript
`application/vnd.lotus-notes`	Plik Lotus Notes
`application/vnd.ms-excel`	Arkusz kalkulacyjny programu Microsoft Excel
`application/vnd.ms-powerpoint`	Prezentacja Microsoft Powerpoint
`application/x-gzip`	Archiwum Gzip
`application/x-java-archive`	Plik JAR
`application/x-java-serialized-object`	Serializowany obiekt języka Java
`application/x-java-vm`	Kody bajtowe Javy (*.class*)
`application/zip`	Archiwum Zip
`audio/basic`	Plik dźwiękowy w formacie *.au* lub *.snd*
`audio/x-aiff`	Plik dźwiękowy AIFF
`audio/x-wav`	Plik dźwiękowy Microsoft Windows
`audio/midi`	Plik dźwiękowy MIDI
`text/css`	Kaskadowy arkusz stylów
`text/html`	Dokument HTML
`text/plain`	Dokument tekstowy
`image/gif`	Obraz w formacie GIF
`image/jpeg`	Obraz w formacie JPEG
`image/png`	Obraz w formacie PNG
`image/tiff`	Obraz w formacie TIFF
`image/x-xbitmap`	Obraz z formie mapy bitowej X Window
`video/mpeg`	Klip wideo w formacie MPEG
`video/quicktime`	Klip wideo w formacie QuickTime

Wiele najczęściej stosowanych typów MIME zostało podanych w plikach RFC 1521 oraz 1522 (możesz je znaleźć na witrynie *http://www.rfc-editor.org/*). Jedna cały czas rejestrowane są nowe typy MIME; dlatego w tym przypadku lepiej jest korzystać z listy typów generowanej dynamicznie. Listę oficjalnie zarejestrowanych typów można znaleźć pod adresem *http://www.isi.edu/in-notes/iana/assignments/media-types/media-types*. Informacje o najczęściej stosowanych, niezarejestrowanych typach MIME można natomiast znaleźć pod adresem *http://www.ltsw.se/knbase/internet/mime.htp*.

Date

Ten nagłówek określa aktualną datę zapisaną w formacie GMT. Jeśli chcesz określić tę datę z poziomu serwletu, powinieneś posłużyć się metodą `setDateHeader`. Metoda ta oszczędza problemów związanych z zapisaniem daty w łańcuchu znaków i jego odpowiednim sformatowaniem — w przypadku wykorzystania wywołania `response.setHeader("Date", "...")` czynności te musiałbyś wykonać samodzielnie. Niemniej jednak większość serwerów automatycznie generuje ten nagłówek, a zatem w serwletach nie będziesz musiał tego robić.

ETag

Ten nowy nagłówek wprowadzony w protokole HTTP 1.1 nadaje zwracanym dokumentom nazwy, których klient będzie mógł później użyć do odwołania się do danego dokumentu (na przykład, przy wykorzystaniu nagłówka żądania `If-Match`).

Expires

Ten nagłówek określa kiedy przesłany dokument ma być uznany za przestarzały i w związku z tym usunięty z pamięci podręcznej. W serwletach można używać tego nagłówka przy przesyłaniu dokumentów których zawartość zmienia się relatywnie często; w ten sposób można zapobiec wyświetlaniu przestarzałych dokumentów pobieranych z pamięci podręcznej przeglądarki. Na przykład, poniższy fragment kodu informuje przeglądarkę, iż dokument nie ma być przechowywany w pamięci podręcznej dłużej niż 10 minut:

```
long aktualnyCzaas = System.currentTime();
long dziesiecMinut  = 10 * 60 * 1000;        // wyrażony w milisekundach
response.setDateHeader("Expires", aktualnyCzas + dziesiecMinut);
```

Patrz także omówienie wartości `max-age` nagłówka odpowiedzi `Cache-Control`.

Last-Modified

Ten bardzo przydatny nagłówek określa datę i czas ostatniej modyfikacji dokumentu. Klient może zapisać dokument w pamięci podręcznej i w późniejszych żądaniach przekazywać tę datę na serwer w nagłówku `If-Modified-Since`. Żądanie takie jest traktowane jako warunkowe żądanie `GET` — wykonanie takiego żądania spowoduje zwrócenie dokumentu wyłącznie wtedy, gdy data podana w nagłówku `Last-Modified` jest późniejsza od daty podanej w nagłówku `If-Modified-Since`. Jeśli data podana w nagłówku `Last-Modified` nie jest późniejsza od daty podanej w nagłówku `If-Modified-Since`, to serwer zwraca kod statusu `304` (`Not Modified`), a program powinien wyświetlić dokument przechowywany w pamięci podręcznej. Jeśli chcesz samemu wygenerować ten nagłówek, to użyj w tym celu metody `setDateHeader`, która zaoszczędzi Ci problemów związanych z zapisywaniem daty w formacie GMT. W przeważającej większości sytuacji wystarczy jednak zaimplementować metodę `getLastModified` i pozwolić, aby żądania zawierające nagłówek `If-Modified-Since` były obsługiwane przez standardową metodę `service`. Przykład praktycznego wykorzystania tego nagłówka został przedstawiony w podrozdziale 2.8, pt.: „Przykład wykorzystania inicjalizacji serwletu i daty modyfikacji strony”.

Location

Nagłówek `Location` informuje przeglądarkę o nowym adresie żądanego dokumentu; powinien on być umieszczany we wszystkich odpowiedziach zawierających kod statusu z zakresu `300` do `399`. W wyniku otrzymania takiej odpowiedzi przeglądarka automatycznie nawiązuje połączenie i pobiera nowy dokument. Nagłówek `Location` jest zazwyczaj określany w sposób niejawny wraz z kodem statusu `302`; służy do tego metoda `sendRedirect` interfejsu

`HttpServletResponse`. Przykład zastosowania tego nagłówka odpowiedzi przedstawiłem w podrozdziale 6.3, pt.: „Interfejs użytkownika obsługujący różne serwisy wyszukiwawcze".

Pragma

Podanie nagłówka `Pragma` o wartości `no-cache` informuje programy korzystające z protokołu HTTP 1.0, iż nie należy zapisywać dokumentu w pamięci podręcznej. Jednak przeglądarki korzystające z protokołu HTTP 1.0 nie obsługiwały tego nagłówka w spójny sposób. W protokole HTTP 1.1, znacznie pewniejszą alternatywą dla tego nagłówka jest nagłówek `Cache-Control` z wartością `no-cache`.

Refresh

Ten nagłówek określa po ilu sekundach przeglądarka powinna zażądać aktualizacji strony. Na przykład, aby poinformować przeglądarkę, by zażądała aktualizacji dokumentu po 30 sekundach możesz użyć poniższego wywołania:

```
response.setIntHeader("Refresh", 30);
```

Zwróć uwagę, iż nagłówek ten nie oznacza wcale cyklicznego odświeżania strony — informuje on jedynie kiedy powinna nastąpić *następna* aktualizacja. Jeśli chcesz odświeżać stronę w sposób cykliczny, będziesz musiał umieszczać nagłówek `Refresh` we wszystkich kolejnych odpowiedziach. Aby przerwać odświeżanie strony należy wygenerować odpowiedź o kodzie statusu `204` (`No content` — brak zawartości). Przykład zastosowania tego nagłówka odpowiedzi znajdziesz w podrozdziale 7.3, pt.: „Trwałe przechowywanie stanu serwletu i automatyczne odświeżanie stron".

Przeglądarka może nie tylko odświeżać aktualnie wyświetloną stronę, lecz także pobrać stronę o podanym adresie. W tym celu, po liczbie określającej ilość sekund należy umieścić średnik oraz adres URL dokumentu, który ma zostać wyświetlony. Na przykład, aby poinstruować przeglądarkę, iż po 5 sekundach należy pobrać dokument *http://host/sciezka*, należy posłużyć się wywołaniem o postaci:

```
response.setHeader("Refresh", "5; URL=http://host/sciezka");
```

Możliwość ta jest przydatna przy tworzeniu „okienek informacyjnych", czyli stron zawierających obrazek lub komunikat, wyświetlanych ma moment przed załadowaniem właściwej strony.

Warto wiedzieć, że bardzo często nagłówek ten jest określany przy wykorzystaniu następującego znacznika HTML

```
<META HTTP-EQUIV="Refresh" CONTENT="5; URL=http://host/sciezka">
```

umieszczanego w nagłówków (sekcji `HEAD`) dokumentu HTML, a nie bezpośrednio, jako nagłówek generowany przez serwer. Taki sposób wykorzystania tego nagłówka pojawił się, gdyż autorzy statycznych dokumentów HTML bardzo często potrzebowali możliwości automatycznego przekierowywania lub odświeżania dokumentów. Niemniej jednak, w przypadku serwletów, bezpośrednia generacja tego nagłówka jest znacznie prostszym i bardziej zrozumiałym rozwiązaniem.

W zasadzie nagłówek ten nie jest oficjalną częścią protokołu HTTP 1.1, lecz stanowi jego rozszerzenie, obsługiwane zarówno przez Netscape Navigatora jak i Internet Explorera.

Retry-After

Nagłówek ten można stosować wraz z kodem statusu `503` (`Service Unavailable` — usługa niedostępna), aby poinformować przeglądarkę kiedy może ponowić żądanie.

Server

Ten nagłówek określa używany serwer WWW. Serwlety zazwyczaj nie określają jego wartości — robi to sam serwer WWW.

Set-Cookie

Nagłówek `Set-Cookie` określa cookie skojarzone z daną stroną. Każde cookie wymaga osobnego nagłówka `Set-Cookie`. W serwletach nie powinno się stosować wywołania `response.setHeader("Set-Cookie", "...")`, lecz zamiast niego należy używać metody `addCookie` interfejsu `HttpServletResponse`. Więcej informacji na temat cookies podam w rozdziale 8, pt.: „Obsługa cookies". Początkowo nagłówek ten stanowił rozszerzenie protokołu HTTP wprowadzone przez firmę Netscape, aktualnie jednak jest on powszechnie wykorzystywane, w tym także przez przeglądarki Netscape Navigator oraz Internet Explorer.

Trailer

Ten rzadko stosowany nagłówek odpowiedzi został wprowadzony w protokole HTTP 1.1. Określa on pola nagłówka umieszczane w stopce wiadomości, które są przesyłane przy wykorzystaniu kodowania typu „chunked"[10]. Szczegółowe informacje na ten temat znajdziesz w punkcie 3.6 specyfikacji protokołu HTTP 1.1 (plik RFC 2616). Pamiętasz zapewne, że pliki RFC można znaleźć na witrynie *http://www.rfc-editor.org/*.

Transfer-Encoding

Podanie tego nagłówka z wartością `chunked` informuje, że do przesłania dokumentu zostało wykorzystanie kodowanie typu „chunked". Szczegółowe informacje na ten temat znajdziesz w punkcie 3.6 specyfikacji protokołu HTTP 1.1.

Upgrade

Ten nagłówek odpowiedzi stosowany jest gdy wcześniej klient nadesłał nagłówek żądania `Upgrade`, prosząc serwer, aby ten zaczął korzystać z jednego spośród kilku nowych, dostępnych protokołów. Jeśli serwer zgodzi się na zmianę protokołu, przesyła odpowiedź zawierającą kod statusu `101` (`Switching Protocols`) oraz nagłówek odpowiedzi `Upgrade` określający jaki protokół zostanie wykorzystany. Zazwyczaj to negocjowanie protokołów jest obsługiwane przez sam serwer, a nie przez serwlet.

Vary

Ten rzadko stosowany, nowy nagłówek protokołu HTTP 1.1 informuje klienta jakich nagłówków można użyć w celu określenia czy przesłany w odpowiedzi dokument może być przechowywany w pamięci podręcznej.

Via

Nagłówek `Via` jest używany przez bramy oraz serwery pośredniczące w celu podania serwerów przez jakie przeszło żądanie. Nagłówek ten został wprowadzony w protokole HTTP 1.1.

Warning

Ten nowy i rzadko stosowany ogólny nagłówek odpowiedzi ostrzega klienty przed błędami związanymi z przechowywaniem dokumentu w pamięci podręcznej lub jego kodowaniem.

[10] przyp. tłum. Przy tym sposobie kodowania przesyłanego dokumentu, jego zawartość jest dzielona na niewielkie fragmenty (ang.: „chunk"), o określonej wielkości.

WWW-Authenticate

Ten nagłówek jest zawsze dołączany do odpowiedzi zawierających kod statusu `401` (`Unauthorized`). Przekazuje on przeglądarce informacje o tym, jaki typ autoryzacji oraz obszar należy podać w nagłówku `Authorization`. Bardzo często serwlety nie obsługują autoryzacji użytkowników samodzielnie, lecz pozwalają, aby dostęp do stron chronionych hasłem był obsługiwany przez odpowiednie mechanizmy serwera WWW (na przykład pliki *.htaccess*). Przykład serwletu wykorzystującego te nagłówki przedstawiłem w podrozdziale 4.5, pt.: „Ograniczanie dostępu do stron WWW".

7.3 Trwałe przechowywanie stanu serwletu i automatyczne odświeżanie stron

W tej części rozdziału przedstawię przykład serwletu, który zwraca listę losowo wybranych, dużych liczb pierwszych. W przypadku naprawdę dużych liczb (na przykład 150-o cyfrowych) wykonanie niezbędnych obliczeń może zająć dużo czasu; dlatego też serwlet niezwłocznie zwraca początkowe wyniki, potem jednak nie przerywa obliczeń lecz je kontynuuje wykorzystując w tym celu wątek o niskim priorytecie. Niski priorytet wątku używanego do przeprowadzania obliczeń umożliwia zachowanie efektywności działania serwera WWW. Jeśli w chwili nadesłania żądania obliczenia nie zostały jeszcze zakończone, serwlet przesyła odpowiedź zawierającą nagłówek `Refresh`, która informuje przeglądarkę, iż po upłynięciu podanego czasu należy zażądać nowej strony.

Przykład ten przedstawia nie tylko znaczenie nagłówków odpowiedzi protokołu HTTP, lecz także dwie, bardzo przydatne możliwości serwletów. Po pierwsze, przykład demonstruje, że jeden serwlet może zostać wykorzystany do obsługi wielu jednoczesnych połączeń, z których każde jest obsługiwane przez inny wątek. W ten sposób, gdy jeden wątek kończy obliczenia przeznaczone dla pewnego klienta, inny program może nawiązać połączenie i uzyskać częściowe wyniki.

Po drugie, przykład prezentuje jak łatwo serwlet może zachować swój stan pomiędzy kolejnymi żądaniami — właściwość ta jest trudna do zaimplementowania w tradycyjnych programach CGI, oraz w wielu technologiach stanowiących alternatywę dla CGI. W naszym przypadku tworzony jest tylko jeden egzemplarz serwletu, a każde żądanie powoduje stworzenie nowego wątku, który wywołuje metodę `service` serwletu (z kolei ta metoda wywołuje metodę `doGet` lub `doPost`). A zatem, wspólne dane mogą być umieszczone w zwyczajnych zmiennych instancyjnych (polach) serwletu. W ten sposób, serwlet może uzyskać dostęp od przeprowadzanych w danej chwili obliczeń kiedy tylko przeglądarka zażąda wyświetlenia strony. Serwlet może także przechowywać listę kilku ostatnio pobieranych wyników i zwracać je błyskawicznie, jeśli zostanie nadesłane nowe żądanie o takich samych parametrach jak jedno żądań z ostatnio obsłużonych. Oczywiście twórców serwletów obowiązują normalne zasady synchronizacji dostępu do wspólnych danych. Serwlety mogą przechowywać trwałe dane w inny sposób — w obiekcie `ServletContext`, który można uzyskać przy użyciu metody `getServletContext`. Interfejs `ServletContext` udostępnia dwie metody — `setAttribute` oraz `getAttribute` — które pozwalają zapisywać i pobierać dowolne dane skojarzone z podanymi kluczami. Różnica pomiędzy przechowywaniem danych w zmiennych instancyjnych a przechowywaniem ich w obiekcie `ServletContext` polega na tym, iż obiekt `ServletContext` jest wspólnie wykorzystywany przez wszystkie serwlety wykonywane przez dany mechanizm obsługi serwletów (bądź wchodzące w skład jednej aplikacji WWW, oczywiście jeśli serwer udostępnia tę możliwość).

Listing 7.1 przedstawia główną klasę serwletu. Na samym początku serwlet otrzymuje żądanie zawierające dwa parametry — `numPrimes` (ilość liczb pierwszych) oraz `numDigits` (ilość cyfr w liczbach pierwszych). Wartości te są pobierane od użytkownika i przesyłane do serwletu w

standardowy sposób, czyli przy użyciu prostego formularza HTML. Kod tego formularza został przedstawiony na listingu 7.2, a jego wygląd pokazałem na rysunku 7.1. Następnie, przy użyciu metody `Integer.parseInt`, wartości tych parametrów są konwertowane do postaci liczb całkowitych (patrz listing 7.5). Te wartości są następnie wykorzystywane w metodzie `findPrimeList` do przeszukiwania wektora (obiektu `Vector`) wyników zakończonych oraz trwających obliczeń i sprawdzenia czy są dostępne jakieś wyniki odpowiadające podanym parametrom. Jeśli takie wyniki są dostępne to zostaje użyta obliczona wcześniej wartość (obiekt `PrimeList`); w przeciwnym przypadku tworzony jest nowy obiekt `PrimeList`, który następnie zostaje zapisany w wektorze (obiekcie `Vector`) aktualnie realizowanych obliczeń; obiekt ten prawdopodobnie zastąpi najstarszy element tego wektora. Kolejnym krokiem jest sprawdzenie czy obiekty `PrimeList` zakończył obliczanie liczb pierwszych, które będą w nim przechowywane. Jeśli obliczenia nie zostały jeszcze zakończone, to do przeglądarki jest przesyłana odpowiedź zawierająca nagłówek `Refresh`, która informuje, że zaktualizowane wyniki należy pobrać po pięciu sekundach. Jeśli tylko są dostępne jakieś wyniki, to są one wyświetlane na stronie w formie listy wypunktowanej.

Listingi 7.3 (*PrimeList.java*) oraz 7.4 (*Primes.java*) przedstawiają pomocnicze klasy wykorzystywane w serwlecie. Kod umieszczony w pliku *PrimeList.java* obsługuje działający w tle wątek, służący do generacji listy liczb pierwszych dla konkretnej pary parametrów. Plik *Prime.java* zawiera algorytm niskiego poziomu służący do wyboru liczby losowej o określonej długości i odszukania liczb pierwszych większych od tej wartości. W obliczeniach wykorzystywane są wbudowane metody klasy `BigInteger`; użyty algorytm określający czy dana liczba jest liczbą pierwszą jest algorytmem probabilistycznym i dlatego istnieje szansa popełnienia błędu. Istnieje jednak możliwość określenia prawdopodobieństwa popełnienia tego błędu, a w przedstawionym przykładzie używam w tym celu wartości `100`. Jeśli założymy, że algorytm użyty w przeważającej większości implementacji języka Java jest testem Millera-Rabina, to prawdopodobieństwo błędnego określenia czy liczba złożona jest liczbą pierwszą, wynosi 2^{100}. Wartość ta jest bez wątpienia mniejsza od prawdopodobieństwa wystąpienia błędu sprzętowego lub losowego zdarzenia powodującego zwrócenie przez algorytm deterministyczny błędnej odpowiedzi. A zatem użyty w przykładzie algorytm można uznać za deterministyczny.

Listing 7.1 PrimeNumbers.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

/** Servlet obsługuje żądania generacji strony zawierającej n
 *  liczb pierwszych, z których każda ma m cyfr.
 *  Obliczenia wykonywane są przez wątek o niskim priorytecie
 *  zwracający wyłącznie obliczenia które w danej chwili
 *  zostały już zakończone. Jeśli wyniki nie są kompletne,
 *  servlet generuje nagłówek odpowiedzi Refresh, informując
 *  przeglądarkę, że należy powtórnie zażądać wyników po
 *  pewnym czasie. Servlet przechowuje także krótką listę
 *  ostatnio wygenerowanych liczb pierwszych, które zwraca
 *  od razu jeśli tylko ktoś poda takie same parametry n i m
 *  jakie były wykorzystane w jednym z ostatnio zakończonych
 *  obliczeń.
 */

public class PrimeNumbers extends HttpServlet {
  private Vector primeListVector = new Vector();
  private int maxPrimeLists = 30;

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    int numPrimes =
```

```
    ServletUtilities.getIntParameter(request,
                                     "numPrimes", 50);
  int numDigits =
    ServletUtilities.getIntParameter(request,
                                     "numDigits", 120);
  PrimeList primeList =
    findPrimeList(primeListVector, numPrimes, numDigits);
  if (primeList == null) {
    primeList = new PrimeList(numPrimes, numDigits, true);
    // Wiele żądań to tego servletu wspólnie używa tych
    // samych zmiennych instancyjnych (pola) klasy PrimeNumbers.
    // Dlatego trzeba synchronizować wszystkie próby dostępu do
    // pól servletu.
    synchronized(primeListVector) {
      if (primeListVector.size() >= maxPrimeLists)
        primeListVector.removeElementAt(0);
      primeListVector.addElement(primeList);
    }
  }
  Vector currentPrimes = primeList.getPrimes();
  int numCurrentPrimes = currentPrimes.size();
  int numPrimesRemaining = (numPrimes - numCurrentPrimes);
  boolean isLastResult = (numPrimesRemaining == 0);
  if (!isLastResult) {
    response.setHeader("Refresh", "5");
  }
  response.setContentType("text/html; charset=ISO-8859-2");
  PrintWriter out = response.getWriter();
  String title = "Kilka " + numDigits + "-cyfrowych liczb pierwszych";
  out.println(ServletUtilities.headWithTitle(title) +
              "<BODY BGCOLOR=\"#FDF5E6\">\n" +
              "<H2 ALIGN=CENTER>" + title + "</H2>\n" +
              "<H3>Odnalezione liczby pierwsze o długości " + numDigits +
              " cyfr lub większej: " + numCurrentPrimes +
              ".</H3>");
  if (isLastResult)
    out.println("<B>Obliczenia zakończone.</B>");
  else
    out.println("<B>Ciągle szukam " + numPrimesRemaining +
                " pozostałych liczb<BLINK>...</BLINK></B>");
  out.println("<OL>");
  for(int i=0; i<numCurrentPrimes; i++) {
    out.println("  <LI>" + currentPrimes.elementAt(i));
  }
  out.println("</OL>");
  out.println("</BODY></HTML>");
}

public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException {
  doGet(request, response);
}

/* Metoda sprawdza czy istnieją jakaś aktualnie obliczana
 * lub już obliczona lista liczb pierwszych o podaje ilości
 * liczb oraz ilości cyfr w liczbie. Jeśli taka lista
 * istnieje to zwraca tę listę zamiast uruchamiania nowego
 * wątku obliczeniowego działającego w tle. Lista wyników
 * powinna być możliwie niewielka, aby nie zabierała zbyt
 * dużo pamięci na serwerze. Dostęp do tej listy należy
 * synchronizować, gdyż jednocześnie może się do niej
 * odwoływać wiele wątków.
 */

private PrimeList findPrimeList(Vector primeListVector,
                                int numPrimes,
                                int numDigits) {
  synchronized(primeListVector) {
    for(int i=0; i<primeListVector.size(); i++) {
      PrimeList primes =
        (PrimeList)primeListVector.elementAt(i);
      if ((numPrimes == primes.numPrimes()) &&
          (numDigits == primes.numDigits()))
        return(primes);
    }
    return(null);
```

```
    }
  }
}
```

Listing 7.2 PrimeNumbers.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Odnajdywanie dużych liczb pierwszych</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H2 ALIGN="CENTER">Odnajdywanie dużych liczb pierwszych</H2>
<BR><BR>
<CENTER>
<FORM ACTION="/servlet/coreservlets.PrimeNumbers">
  <B>Ilość liczb pierwszych jakie należy obliczyć:</B>
  <INPUT TYPE="TEXT" NAME="numPrimes" VALUE=25 SIZE=4><BR>
  <B>Ilość cyfr w każdej z tych liczb:</B>
  <INPUT TYPE="TEXT" NAME="numDigits" VALUE=150 SIZE=3><BR>
  <INPUT TYPE="SUBMIT" VALUE="Rozpocznij obliczenia">
</FORM>
</CENTER>
</BODY>
</HTML>
```

Listing 7.3 PrimeList.java

```
package coreservlets;

import java.util.*;
import java.math.BigInteger;

/** Tworzy wektor (klasa Vector) dużych liczb pierwszych
 *  wykorzystując do obliczeń wątek o niskim priorytecie
 *  działający w tle. Klasa udostępnia kilka prostych
 *  metod dostępu umożliwiających pracę wielowątkową.
 */

public class PrimeList implements Runnable {
  private Vector primesFound;
  private int numPrimes, numDigits;

  /** Metoda odnajduje liczby pierwsze, z których każda
   *  ma długość co najmniej numDigits liczb. Metodę
   *  można wywołać w taki sposób aby zwróciła wyniki
   *  dopiero po całkowitym zakończeniu obliczeń, lub
   *  aby zakończyła się bezzwłocznie - w takim przypadku
   *  będziesz mógł później sprawdzić na jakim etapie
   *  są w danej chwili obliczenia.
   */

  public PrimeList(int numPrimes, int numDigits,
                   boolean runInBackground) {
    // Używam klasy Vector a nie ArrayList
    // aby zachować zgodność z mechanizmami obsługi
    // korzystającymi z JDK 1.1
    primesFound = new Vector(numPrimes);
    this.numPrimes = numPrimes;
    this.numDigits = numDigits;
    if (runInBackground) {
      Thread t = new Thread(this);
      // Używam wątku o niskim priorytecie, aby zbytnio
      // nie obciążać serwera.
      t.setPriority(Thread.MIN_PRIORITY);
      t.start();
    } else {
      run();
    }
  }

  public void run() {
    BigInteger start = Primes.random(numDigits);
    for(int i=0; i<numPrimes; i++) {
      start = Primes.nextPrime(start);
      synchronized(this) {
```

```
          primesFound.addElement(start);
        }
      }
    }

    public synchronized boolean isDone() {
      return(primesFound.size() == numPrimes);
    }

    public synchronized Vector getPrimes() {
      if (isDone())
        return(primesFound);
      else
        return((Vector)primesFound.clone());
    }

    public int numDigits() {
      return(numDigits);
    }

    public int numPrimes() {
      return(numPrimes);
    }

    public synchronized int numCalculatedPrimes() {
      return(primesFound.size());
    }
}
```

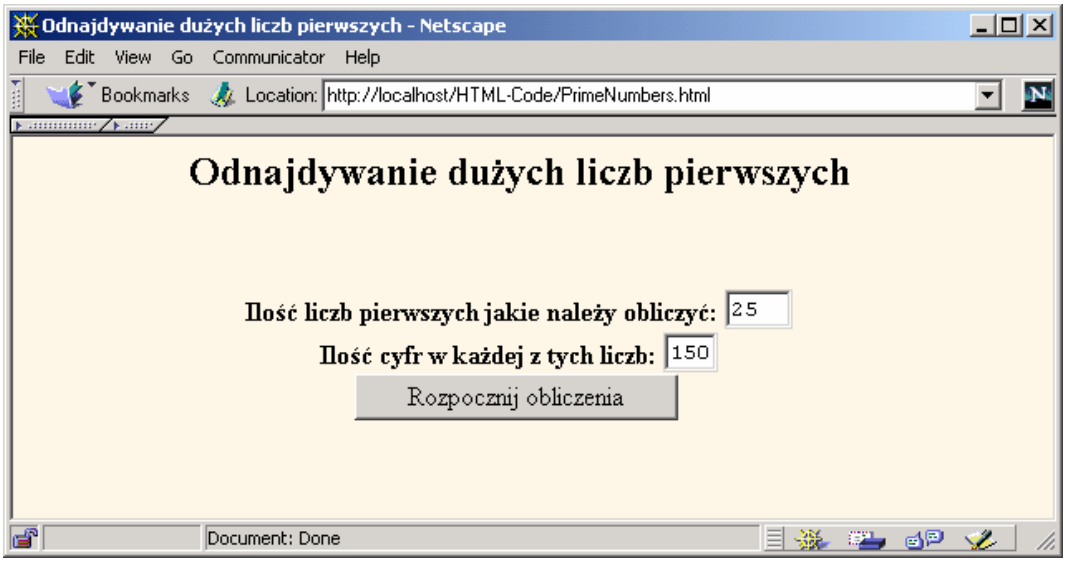
Listing 7.4 Primes.java

```
package coreservlets;

import java.math.BigInteger;

/** Kilka pomocniczych narzędzi służących do generacji
 *  dużych liczb losowych typu BigInteger i odnajdywania
 *  liczb pierwszych większych od podanej wartości
 *  BigInteger.
 */

public class Primes {
  // Zwróć uwagę, że stała BigInteger.ZERO została wprowadzona
  // w JDK 1.2, w tym przykładzie używam kodu JDK 1.1, aby
  // zapewnić zgodność z większością istniejących mechanizmów
  // obsługi servletów.
  private static final BigInteger ZERO = new BigInteger("0");
  private static final BigInteger ONE = new BigInteger("1");
  private static final BigInteger TWO = new BigInteger("2");

  /* Prawdopodobieństwo popełniania błędu przy określaniu
   * czy dana liczba jest liczbą pierwszą jest mniejsze
   * od 1/2^ERR_VAL. Można założyć że klasa BigInteger
   * używa algorytmu Millera-Rabina lub jego ekwiwalentu
   * i dlatego nie można go oszukać stosując liczby
   * Carmichael. Więcej szczegółów znajdziesz w rozdziale
   * 33.8 książki Introduction to Algorithms.
   */
  private static final int ERR_VAL = 100;

  public static BigInteger nextPrime(BigInteger start) {
    if (isEven(start))
      start = start.add(ONE);
    else
      start = start.add(TWO);
    if (start.isProbablePrime(ERR_VAL))
      return(start);
    else
      return(nextPrime(start));
  }

  private static boolean isEven(BigInteger n) {
    return(n.mod(TWO).equals(ZERO));
  }

  private static StringBuffer[] digits =
    { new StringBuffer("0"), new StringBuffer("1"),
```

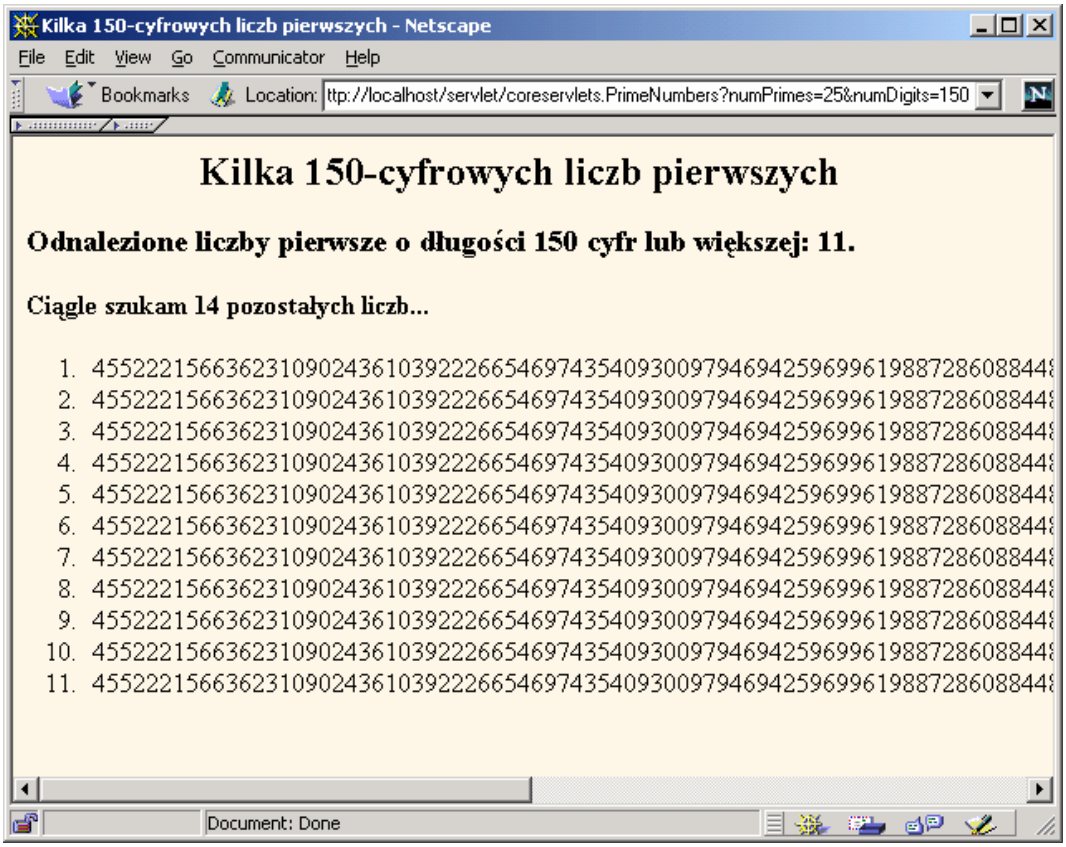
```
      new StringBuffer("2"), new StringBuffer("3"),
      new StringBuffer("4"), new StringBuffer("5"),
      new StringBuffer("6"), new StringBuffer("7"),
      new StringBuffer("8"), new StringBuffer("9") };

  private static StringBuffer randomDigit() {
    int index = (int)Math.floor(Math.random() * 10);
    return(digits[index]);
  }

  public static BigInteger random(int numDigits) {
    StringBuffer s = new StringBuffer("");
    for(int i=0; i<numDigits; i++) {
      s.append(randomDigit());
    }
    return(new BigInteger(s.toString()));
  }

  /** Prosty program uruchamiany z poziomu wiersza poleceń,
   *  umożliwiający przeprowadzanie testów. Podaj ilość
   *  cyfr, a program wybierze losową liczbę pierwszą
   *  o podanej długości oraz 50 liczb pierwszych większych od niej.
   */

  public static void main(String[] args) {
    int numDigits;
    if (args.length > 0)
      numDigits = Integer.parseInt(args[0]);
    else
      numDigits = 150;
    BigInteger start = random(numDigits);
    for(int i=0; i<50; i++) {
      start = nextPrime(start);
      System.out.println("Prime " + i + " = " + start);
    }
  }
}
```

Listing 7.5 ServletUtilities.java

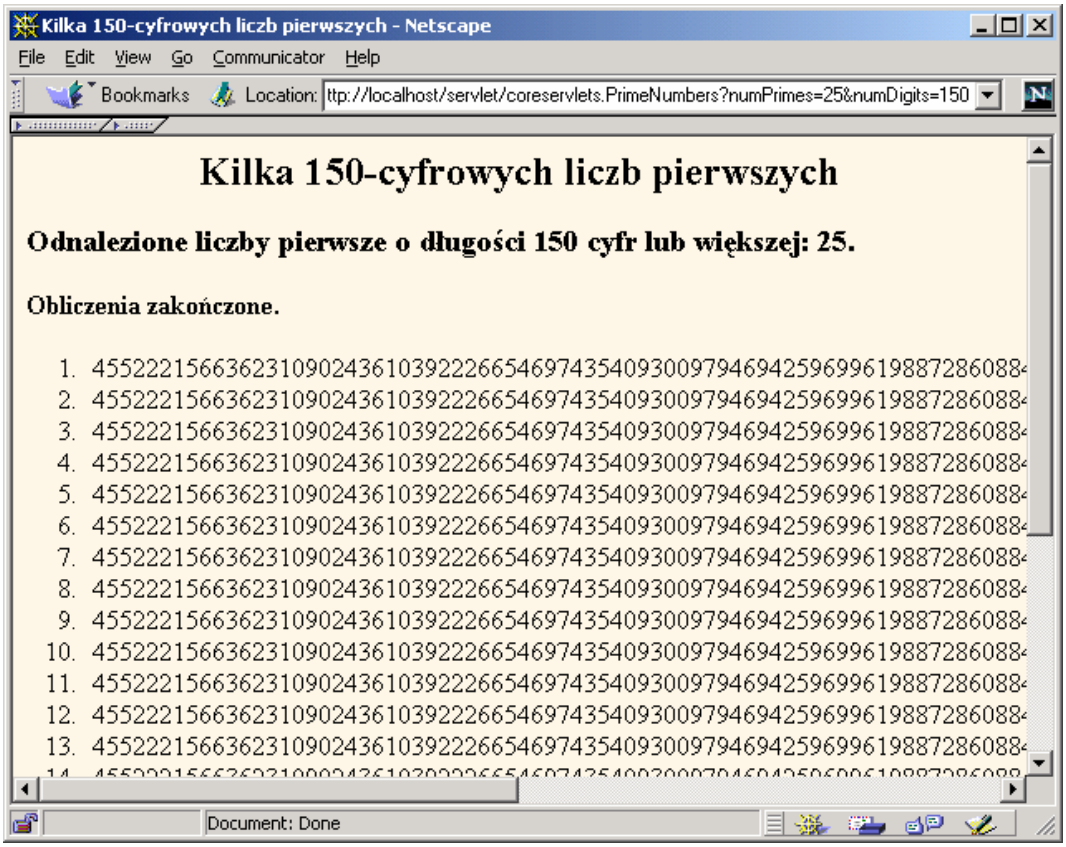
```
package coreservlets;

import javax.servlet.*;
import javax.servlet.http.*;

public class ServletUtilities {

  // ... -- inne narzędzia przedstawione w innych miejscach książki

  /** Odczytuje parametr o określonej nazwie, konwertuje jego
   *  wartość do postaci liczby całkowitej i zwraca ją.
   *  Zwraca określoną wartość domyślną w przypadku gdy parametr
   *  nie został podany lub gdy jego wartość nie jest poprawnie
   *  zapisaną liczbą całkowitą.
   */

  public static int getIntParameter(HttpServletRequest request,
                                    String paramName,
                                    int defaultValue) {
    String paramString = request.getParameter(paramName);
    int paramValue;
    try {
      paramValue = Integer.parseInt(paramString);
    } catch(NumberFormatException nfe) { // null lub zły format
      paramValue = defaultValue;
    }
    return(paramValue);
  }

  // ... -- inne narzędzia przedstawione w innych miejscach książki

}
```



Rysunek 7.1 Dokument PrimeNumbers.html stanowiący interfejs użytkownika służący do obsługi serwletu PrimeNumbers



Rysunek 7.2 Częściowe wyniki zwrócone przez serwlet PrimeNumbers. Wyniki takie mogą zostać wyświetlone gdy przeglądarka automatycznie odświeży stronę, lub gdy inna przeglądarka niezależnie poda parametry odpowiadające parametrom aktualnie obsługiwanych żądań lub żądań których obsługa została niedawno zakończona. W obu przypadkach przeglądarka automatycznie odświeży stronę, aby uzyskać zaktualizowane wyniki.



Rysunek 7.3 Końcowe wyniki zwrócone przez serwlet PrimeNumbers. Wyniki takie mogą zostać wyświetlone gdy przeglądarka automatycznie odświeży stronę, lub gdy inna przeglądarka niezależnie poda parametry odpowiadające parametrom aktualnie obsługiwanych żądań lub żądań których obsługa została niedawno zakończona. Po wyświetleniu końcowej wersji wyników przeglądarka nie będzie już odświeżać strony.

7.4 Stosowanie trwałych połączeń HTTP

Jednym z problemów występujących w protokole HTTP 1.0, była konieczność tworzenia nowych połączeń dla każdego żądania. W przypadku pobierania strony zawierającej bardzo dużo niewielkich obrazów lub apletów, narzut czasowy związany z koniecznością tworzenia wszystkich tych połączeń mógł być znaczący w porównaniu z całkowitym czasem pobierania dokumentu. Wiele przeglądarek i serwerów rozwiązuje ten problem przy wykorzystaniu trwałych połączeń HTTP (ang. „keep-alive"). W tym przypadku serwer podaje przeglądarce wielkość odpowiedzi wyrażoną w bajtach, a następnie przekazuje dokument i pozostawia połączenie otworzone na pewien okres czasu. Przeglądarka (lub inny klient) monitoruje ilość odbieranych danych i na tej podstawie jest w stanie określić moment zakończenia transmisji. Gdy to nastąpi program ponownie nawiązuje połączenie z tym samym gniazdem i używa go do wykonania kolejnych transakcji. Trwałe połączenia tego typu stały się standardem w protokole HTTP 1.1, a serwery posługujące się tym protokołem mają używać trwałych połączeń jeśli program jawnie tego nie zabronił (bądź to poprzez wysłanie nagłówka żądania `Connection: close`, bądź też niejawnie, poprzez określenie w żądaniu protokołu `HTTP/1.0` zamiast `HTTP/1.1` i jednoczesne pominięcie nagłówka żądania `Connection: keep-alive`).

Serwlety mogą wykorzystywać trwałe połączenia jeśli są wykonywane na serwerze, który jest w stanie takie połączenia obsługiwać. Serwer powinien samodzielnie obsługiwać większą część całego procesu, jednak nie ma możliwości określenia wielkości zwracanego dokumentu. A zatem to serwlet musi wygenerować nagłówek odpowiedzi `Content-Length` używając w tym celu metody

`response.setContentLength`. Serwlet może określić wielkość zwracanego dokumentu, buforując go w strumieniu `ByteArrayOutputStream`. Gdy dokument zostanie w całości zapisany, jego wielkość można określić przy użyci metody `size`. Aby przesłać dokument do przeglądarki należy wywołać metodę `writeTo` obiektu `ByteArrayOutputStream`, przekazując w jej wywołaniu strumień wyjściowy serwletu (obiekt `PrintWriter`).

Wykorzystanie trwałych połączeń może się opłacić wyłącznie w przypadku serwletów, które pobierają znaczną ilość niewielkich obiektów, przy czym obiekty te nie są generowane przez serwlet i w żaden inny sposób nie mogłyby skorzystać z mechanizmów obsługi trwałych połączeń jakimi dysponuje serwer. Jednak nawet w takim przypadku zyski jakie daje wykorzystanie trwałych połączeń zależą od stosowanego serwera WWW, a nawet od używanej przeglądarki. Na przykład, domyślne ustawienia Java Web Servera pozwalają na wykonanie wyłącznie pięciu połączeń na jednym gnieździe HTTP — w wielu aplikacjach wartość ta jest zbyt mała. Aby zmienić te ustawienia należy uruchomić konsolę administracyjną, wybrać opcję *Web Service* oraz *Service Tuning*, a następnie podać nową wartość w oknie *Connection Persistance*.

Na listingu 7.6 przedstawiłem serwlet generujący stronę zawierającą 100 znaczników `<IMG>` (strona ta została przedstawiona na rysunku 7.4). Każdy z tych znaczników odwołuje się do innego serwletu (`ImageRetriever`, jego kod przedstawiłem na listingu 7.7), który odczytuje plik GIF z dysku i przekazuje go do przeglądarki. Zarówno pierwszy serwlet jak i serwlet `ImageRetriever` korzystają z trwałych połączeń, chyba że w danych przekazanych z formularza znajdzie się parametr `usePersistance` o wartości `no`. Testy przeprowadzone przy użyciu przeglądarki Netscape Navigator 4.7 łączącej się połączeniem o prędkości 28,8 kbps z serwerem Java Web Server (przy czyli limit połączeń został podniesiony do wartości większej od 100) działającym w systemie Solaris wykazały, że wykorzystanie trwałych połączeń umożliwiło redukcję łącznego czasu pobierania strony o 15 do 20 procent.

Listing 7.6 PersistentConnection.java

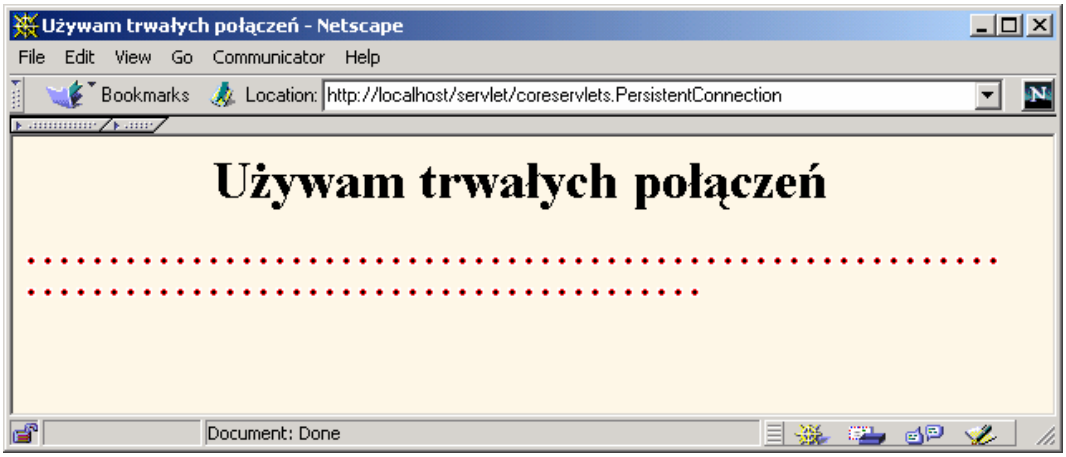
```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

/** Servlet ilustruje znaczenie trwałych połączeń HTTP
 *  dla stron zawierających wiele obrazów, apletów, lub
 *  inną zawartość, której pobranie wymagałoby utworzenia
 *  oddzielnego połączenia.
 */

public class PersistentConnection extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html; charset=ISO-8859-2");
    ByteArrayOutputStream byteStream =
      new ByteArrayOutputStream(7000);
    PrintWriter out = new PrintWriter(byteStream, true);
    String persistenceFlag =
      request.getParameter("usePersistence");
    boolean usePersistence =
      ((persistenceFlag == null) ||
       (!persistenceFlag.equals("no")));
    String title;
    if (usePersistence) {
      title = "Używam trwałych połączeń";
    } else {
      title = "Nie używam trwałych połączeń";
    }
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=\"CENTER\">" + title + "</H1>");
    int numImages = 100;
    for(int i=0; i<numImages; i++) {
      out.println(makeImage(i, usePersistence));
```

```
    }
    out.println("</BODY></HTML>");
    if (usePersistence) {
      response.setContentLength(byteStream.size());
    }
    byteStream.writeTo(response.getOutputStream());
  }

  private String makeImage(int n, boolean usePersistence) {
    String file =
      "/servlet/coreservlets.ImageRetriever?gifLocation=" +
      "/bullets/bullet" + n + ".gif";
    if (!usePersistence)
      file = file + "&usePersistence=no";
    return("<IMG SRC=\"" + file + "\"\n" +
           "     WIDTH=6 HEIGHT=6 ALT=\"\">");
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

Listing 7.7 ImageRetriever.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Servlet który odczytuje plik GIF z lokalnego systemu
 *  i przesyła go do klienta wykorzystując przy tym odpowiedni
 *  typ MIME. Servlet wykorzystuje nagłówek Content-Type aby
 *  umożliwić wykorzystanie trwałych połączeń HTTP, chyba
 *  że klient zabroni tego używając parametru
 *  "usePersistence=no". Klasa używana przez servlet
 *  PersistentConnection.
 *  <P>
 *  <i>Nie</i> należy instalować tego servletu w sposób trwały
 *  na publicznie dostępnych serwerach, gdyż umożliwia on
 *  dostęp do obrazów, które nie koniecznie znajdują się
 *  w drzewie katalogów dostępnych dla serwera WWW.
 */

public class ImageRetriever extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    String gifLocation = request.getParameter("gifLocation");
    if ((gifLocation == null) ||
        (gifLocation.length() == 0)) {
      reportError(response, "Podany obraz nie został odnaleziony");
      return;
    }
    String file = getServletContext().getRealPath(gifLocation);
    try {
      BufferedInputStream in =
        new BufferedInputStream(new FileInputStream(file));
      ByteArrayOutputStream byteStream =
        new ByteArrayOutputStream(512);
      int imageByte;
      while((imageByte = in.read()) != -1) {
        byteStream.write(imageByte);
      }
      in.close();
      String persistenceFlag =
      request.getParameter("usePersistence");
      boolean usePersistence =
        ((persistenceFlag == null) ||
         (!persistenceFlag.equals("no")));
      response.setContentType("image/gif");
      if (usePersistence) {
        response.setContentLength(byteStream.size());
      }
```

```
        byteStream.writeTo(response.getOutputStream());
      } catch(IOException ioe) {
        reportError(response, "Błąd: " + ioe);
      }
    }

    public void reportError(HttpServletResponse response,
                            String message)
        throws IOException {
      response.sendError(response.SC_NOT_FOUND,
                         message);
    }
  }
```



Rysunek 7.4 Wyniki wykonania serwletu PersistentConnection

7.5 Wykorzystanie servletów do generacji obrazów GIF

Choć serwlety bardzo często generują kod HTML, to bez wątpienia nie robią tego zawsze. Na przykład, w podrozdziale 11.2, pt.: „Atrybut contentType", przedstawiłem stronę JSP generującą arkusz kalkulacyjny programu Microsoft Excel i przekazującą go do przeglądarki użytkownika. W tej części rozdziału pokażę jak można generować obrazy GIF.

Po pierwsze chciałbym pokrótce opisać dwie podstawowe czynności jakie należy wykonać, aby z poziomu serwletu wygenerować zawartość multimedialną. Po pierwsze należy podać nagłówek odpowiedzi `Content-Type` przy użyciu metody `setContentType` udostępnianej przez interfejs `HttpServletResponse`. Po drugie należy przesłać dane zapisane w odpowiednim formacie. Format ten zależy oczywiście od typu dokumentu, jednak w większości przypadków będą to dane binarne, a nie łańcuchy znaków, jak w przypadku generacji dokumentów HTML. Oznacza to, że serwlety będą raczej korzystały z nieprzetworzonego strumienia wyjściowego, który można pobrać przy użyciu metody `getOutputStreamd`, a nie z strumienia `PrintWriter` (zwracanego przez metodę `getWriter`). Uwzględniając powyższe informacje, metody `doGet` oraz `doPost` serwletów, które nie generują dokumentów HTML, będą zawierały następujący fragment kodu:

```
response.setContentType("typ/podtyp");
OutputStream out = response.getOutputStream();
```

Powyżej przedstawiłem dwie, ogólne czynności jakie należy wykonać, aby wygenerować z serwletu zawartość inną niż kod HTML. Teraz przyjrzyjmy się bliżej czynnościom jakie trzeba wykonać aby wygenerować obraz GIF; oto one:

1. **Stwórz obraz (obiekt `Image`).**

   Obiekt `Image` tworzy się poprzez wywołanie metody `createImage` klasy `Component`. Programy wykonywane na serwerze nie powinne wyświetlać na ekranie żadnych okien, a

zatem muszą one jawnie nakazać systemowi stworzenie obiektu rodzimego okna systemu graficznego (proces ten jest zazwyczaj wykonywany automatycznie gdy okno jest wyświetlane). Zadanie to wykonywane jest dzięki wywołaniu metody `addNotify`. A zatem, poniżej przedstawiłem proces tworzenia w serwlecie obiektu `Image`:

```
Frame f = new Frame();
f.addNotify();
int width = ...;              // szerokość
int height = ...;             // wysokość
Image img = f.createImage(width, height);
```

2. **Narysuj zawartość obrazu.**

   Aby wykonać to zadanie należy wywołać metodę `getGraphics` obiektu `Image`, a następnie, w tradycyjny sposób, wykorzystać możliwości uzyskanego obiektu `Graphics`. Na przykład, w JDK 1.1 mógłbyś stworzyć zawartość obrazu korzystając z różnych metod `draw`*XXX* oraz `fill`*XXX* obiektu `Graphisc`. Jeśli używasz platformy Java 2, to możesz rzutować obiekt `Graphisc` do obiektu klasy `Graphics2D` i stworzyć obraz wykorzystując znacznie bogatszy zbiór metod służących do rysowania, transformacji współrzędnych, określania postaci czcionek oraz wypełniania obszarów określonym wzorem. Oto bardzo prosty przykład:

```
Graphics g = img.getGraphics();
g.fillRect(...);
g.drawRect(...);
```

3. **Wygeneruj nagłówek `Content-Type`.**

   Jak zapewne pamiętasz, to zadanie należy wykonać wywołując metodę `setContentType` interfejsu `HttpServletResponse`. Typ MIME odpowiadający obrazom GIF to `image/gif`.

```
response.setContentType("image/gif");
```

4. **Pobrać strumień wyjściowy.**

   Zgodnie z tym co napisałem wcześniej, generując dane binarne powinieneś pobrać strumień wyjściowy wywołując metodę `getOutputStream` interfejsu `HttpServletResponse` a nie metodę `getWriter`.

```
OutputStream out = response.getOutputStream();
```

5. **Przekarz zawartość obrazu zapisaną w formacie GIF do strumienia wyjściowego.**

   Gdybyś chciał wykonać to zadanie samemu, musiałbyś się nieźle namęczyć. Na szczęście dostępnych jest kilka gotowych klas, które mogą wykonać je za Ciebie. Jedną z najpopularniejszy klas dysponujących potrzebnymi nam możliwościami funkcjonalnymi, jest klasa `GifEncoder` Jefa Poskanzera. Klasę tę można znaleźć pod adresem *http://www.acme.com/java/*, a korzystanie z niej jest bezpłatne. Poniżej pokazałem jak należy użyć klasy `GifEncoder` do przesłania obrazu zapisanego w formacie GIF:

```
try {
  new GifEncoder(img, out).encode();
} catch(IOException ioe) {
  // Komunikat o błędzie
}
```

Na listingach 7.8 oraz 7.9 przedstawiłem serwlet który pobiera trzy parametry — `message` (komunikat), `fontName` (nazwa czcionki) oraz `fontSize` (wielkość czcionki) — i na ich podstawie generuje obraz GIF przestawiający komunikat wyświetlony określoną czcionką o określonej wielkości liter oraz cień tego komunikatu, wyświetlony szarą, pochyloną czcionką. Serwlet ten wykorzystuje kilka narzędzi dostępnych wyłącznie w platformie Java 2. Po pierwsze, daje on możliwość użycia dowolnych czcionek zainstalowanych na serwerze, a nie tylko standardowych czcionek dostępnych dla programów pisanych w JDK 1.1 (o nazwach `Serif`, `SansSerif`, `Monospaced`, `Dialog` oraz `DilogInput`).

Listing 7.8 ShadowedText.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.awt.*;

/** Servlet generujący obrazy GIF zawierające podany
 *  komunikat oraz jego cień.
 *  <P>
 *  <B>Działa wyłącznie w mechanizmach obsługi servletów
 *  korzystających z platformy Java 2, gdyż do rysowania
 *  obrazów używana jest technologia Java2D.</B>
 */

public class ShadowedText extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    String message = request.getParameter("message");
    if ((message == null) || (message.length() == 0)) {
      message = "Brak parametru 'message'";
    }
    String fontName = request.getParameter("fontName");
    if (fontName == null) {
      fontName = "Serif";
    }
    String fontSizeString = request.getParameter("fontSize");
    int fontSize;
    try {
      fontSize = Integer.parseInt(fontSizeString);
    } catch(NumberFormatException nfe) {
      fontSize = 90;
    }
    response.setContentType("image/gif");
    OutputStream out = response.getOutputStream();
    Image messageImage =
      MessageImage.makeMessageImage(message,
                                    fontName,
                                    fontSize);
    MessageImage.sendAsGIF(messageImage, out);
  }

  /** Formularz może przesyłać dane metodami GET lub POST. */

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

Listing 7.9 MessageImage.java

```
package coreservlets;

import java.awt.*;
import java.awt.geom.*;
import java.io.*;
import Acme.JPM.Encoders.GifEncoder;

/** Narzędzia służące do tworzenia obrazów prezentujących
 *  komunikat wraz z cieniem. Zawiera metody wykorzystujące
 *  GifEncoder Jefa Poskanzera (do generacji obrazu w formacie
 *  GIF).
 *  <P>
 *  <B>Nie działa w JDK 1.1, gdyż do tworzenia obrazów
 *  używana jest technologia Java2D.</B>
 */

public class MessageImage {

  /** Tworzy Image (obraz) przedstawiający komunikat
   *  oraz jego cień. Używany przez servlet ShadowedText
   *  oraz aplikację ShadowedTextFrame.
   */
```

```
  public static Image makeMessageImage(String message,
                                        String fontName,
                                        int fontSize) {
    Frame f = new Frame();
    // Połącz z rodzimymi zasobami aby utworzyć obraz.
    f.addNotify();
    // Upewnij się, że Java zna nazwy lokalnych czcionek.
    GraphicsEnvironment env =
      GraphicsEnvironment.getLocalGraphicsEnvironment();
    env.getAvailableFontFamilyNames();
    Font font = new Font(fontName, Font.PLAIN, fontSize);
    FontMetrics metrics = f.getFontMetrics(font);
    int messageWidth = metrics.stringWidth(message);
    int baselineX = messageWidth/10;
    int width = messageWidth+2*(baselineX + fontSize);
    int height = fontSize*7/2;
    int baselineY = height*8/10;
    Image messageImage = f.createImage(width, height);
    Graphics2D g2d =
      (Graphics2D)messageImage.getGraphics();
    g2d.setFont(font);
    g2d.translate(baselineX, baselineY);
    g2d.setPaint(Color.lightGray);
    AffineTransform origTransform = g2d.getTransform();
    g2d.shear(-0.95, 0);
    g2d.scale(1, 3);
    g2d.drawString(message, 0, 0);
    g2d.setTransform(origTransform);
    g2d.setPaint(Color.black);
    g2d.drawString(message, 0, 0);
    return(messageImage);
  }

  /** Korzysta z klasy GifEncoder w celu przesłania obrazu
   *  (zawartości obiektu Image) w formacie GIF89A
   *  do strumienia wyjściowego. Klasę GifEncoder można
   *  znaleźć pod adresem http://www.acme.com/java.
   */

  public static void sendAsGIF(Image image, OutputStream out) {
    try {
      new GifEncoder(image, out).encode();
    } catch(IOException ioe) {
      System.err.println("Błąd przy przesyłaniu obrazu GIF: " + ioe);
    }
  }
}
```

Po drugie, przy tworzeniu „cienia” komunikatu wykorzystywane są transformacje `translate`, `scale` oraz `shear`. Oznacza to, że serwlet będzie działać wyłącznie w środowisku obsługi serwletów działającym na platformie Java 2. Mógłbyś przypuszczać, że dotyczy to mechanizmów obsługi serwletów zgodnych ze specyfikacją Java Servlet 2.2, gdyż ta wersja serwletów jest stosowana w J2EE.

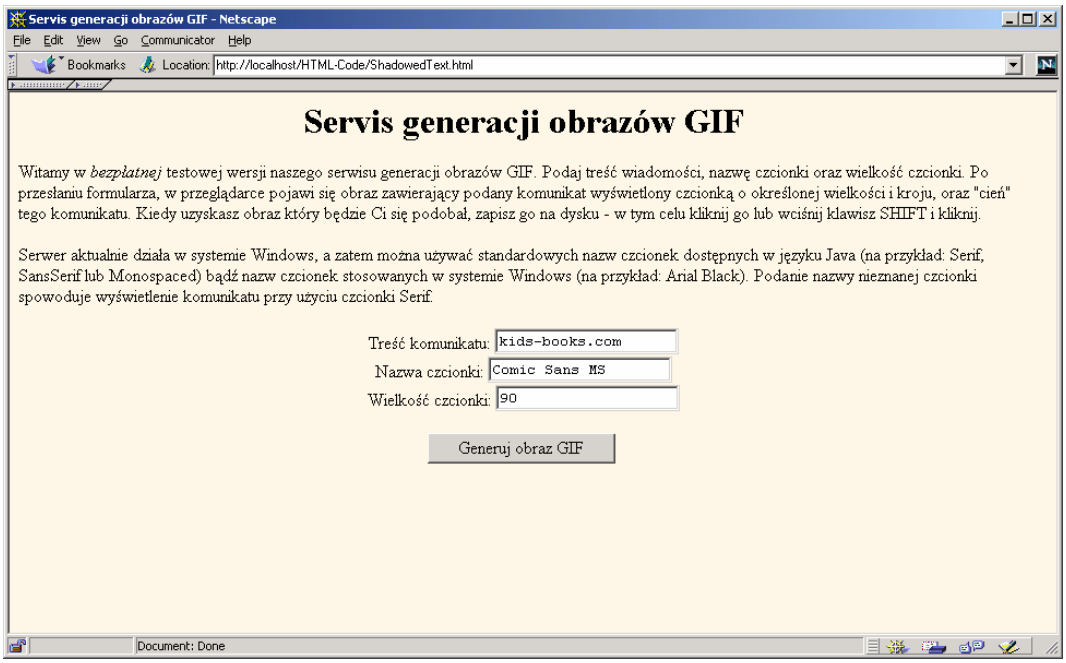
Jednak nawet jeśli korzystasz z mechanizmu obsługi serwletów zgodnego ze specyfikacją Java Servlet 2.1, to także, jeśli tylko możesz, powinieneś korzystać z platformy Java 2. W przypadku zadań wykonywanych na serwerze, platforma ta cechuje się bowiem znacznie większą efektywnością działania. Jednak wiele mechanizmów obsługi serwletów, zgodnych ze specyfikacją 2.1, jest wstępnie konfigurowanych w taki sposób, aby korzystać z JDK 1.1, a zmiana używanej wersji Javy nie zawsze jest prosta. Przykładem może tu być Java Web Server wykorzystujący JDK 1.1 dostarczane wraz z nim. Zarówno Tomcat jak i JSWDK automatycznie wykorzystują pierwszą wersję Javy zdefiniowaną w zmiennej środowiskowej `PATH`.

Listing 7.10 przedstawia formularz HTML stanowiący interfejs użytkownika naszego przykładowego serwletu. Przykładowe wyniki działania serwletu przedstawiłem na rysunkach od 7.5 do 7.8. Aby ułatwić Ci eksperymenty stworzyłem także interaktywną aplikację, której kod przedstawiłem na listingu 7.11. Aplikacja ta jest wywoływana z poziomu wiersza poleceń, a w jej wywołaniu należy podać komunikat, nazwę czcionki oraz jej wielkość. Wykonanie aplikacji

powoduje wyświetlenie okna (`JFrame`) przedstawiającego ten sam obraz, który zostałby wygenerowany przez serwlet. Przykładowy wygląd tej aplikacji przedstawiłem na rysunku 7.9.

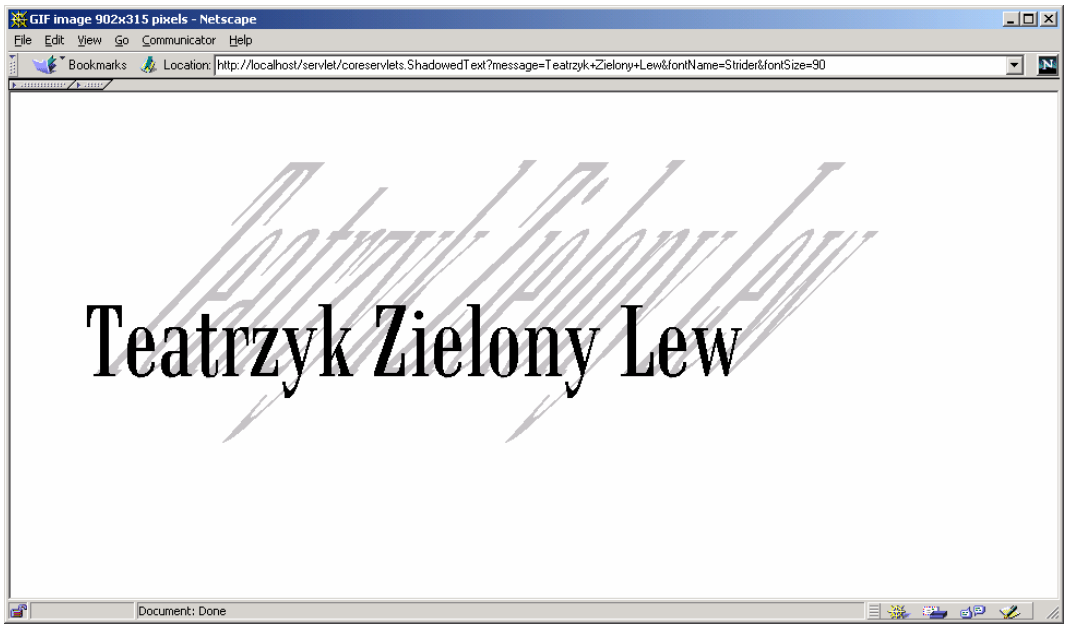
Listing 7.10 ShadowedText.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<!--
Interfejs użytkownika do servletu generującego obrazki.
-->
<HTML>
<HEAD>
  <TITLE>Servis generacji obrazów GIF</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">Servis generacji obrazów GIF</H1>
Witamy w <I>bezpłatnej</I> testowej wersji naszego serwisu generacji
obrazów GIF. Podaj treść wiadomości, nazwę czcionki oraz wielkość
czcionki. Po przesłaniu formularza, w przeglądarce pojawi się
obraz zawierający podany komunikat wyświetlony
czcionką o określonej wielkości i kroju, oraz "cień" tego komunikatu.
Kiedy uzyskasz obraz który będzie Ci się podobał, zapisz go na
dysku - w tym celu kliknij go lub wciśnij klawisz SHIFT i kliknij.
<P>
Serwer aktualnie działa w systemie Windows, a zatem można
używać standardowych nazw czcionek dostępnych w języku Java
(na przykład: Serif, SansSerif lub Monospaced) bądź nazw
czcionek stosowanych w systemie Windows (na przykład: Arial
Black). Podanie nazwy nieznanej czcionki spowoduje wyświetlenie
komunikatu przy użyciu czcionki Serif.

<FORM ACTION="/servlet/coreservlets.ShadowedText">
  <CENTER>
    Treść komunikatu:
    <INPUT TYPE="TEXT" NAME="message"><BR>
    Nazwa czcionki:
    <INPUT TYPE="TEXT" NAME="fontName" VALUE="Serif"><BR>
    Wielkość czcionki:
    <INPUT TYPE="TEXT" NAME="fontSize" VALUE="90"><BR><BR>
    <Input TYPE="SUBMIT" VALUE="Generuj obraz GIF">
  </CENTER>
</FORM>

</BODY>
</HTML>
```

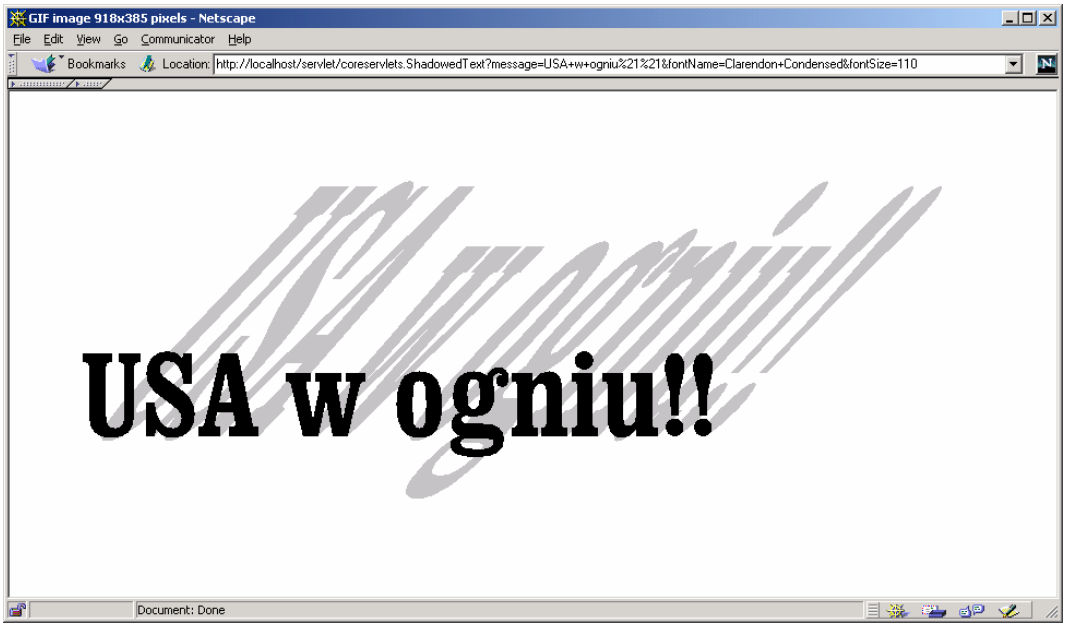


Rysunek 7.5 Interfejs użytkownika umożliwiający korzystanie z serwletu ShadowedText

Rysunek 7.6 Użycie serwletu ShadowedText do wygenerowania obrazu GIF stanowiącego logo witryny z książkami dla dzieci (wyniki przesłania formularza z rysunku 7.5)



Rysunek 7.7 Użycie serwletu ShadowedText do wygenerowania obrazu GIF stanowiącego nagłówek witryny prezentującej dokonania lokalnego teatru



Rysunek 7.8 Użycie serwletu ShadowedText do wygenerowania obrazu GIF stanowiącego nagłówek lokalnej internetowej gazety

Listing 7.11 ShadowedTextFrame.java

```
package coreservlets;

import java.awt.*;
import javax.swing.*;
import java.awt.geom.*;

/** Interaktywny interfejs umożliwiający testowanie klasy
 *  MessageImage. Podaj komunikat, nazwę czcionki,
 *  wielkość czcionki w wierszu poleceń. Konieczna
 *  platforma Java 2.
 */

public class ShadowedTextFrame extends JPanel {
  private Image messageImage;

  public static void main(String[] args) {
    String message = "Tekst z cieniem";
    if (args.length > 0) {
      message = args[0];
    }
    String fontName = "Serif";
    if (args.length > 1) {
      fontName = args[1];
    }
    int fontSize = 90;
    if (args.length > 2) {
      try {
        fontSize = Integer.parseInt(args[2]);
      } catch(NumberFormatException nfe) {}
    }
    JFrame frame = new JFrame("Tekst z cieniem");
    frame.addWindowListener(new ExitListener());
    JPanel panel =
      new ShadowedTextFrame(message, fontName, fontSize);
    frame.setContentPane(panel);
    frame.pack();
    frame.setVisible(true);
  }

  public ShadowedTextFrame(String message,
                           String fontName,
                           int fontSize) {
    messageImage = MessageImage.makeMessageImage(message,
                                                 fontName,
                                                 fontSize);
    int width = messageImage.getWidth(this);
```

```
    int height = messageImage.getHeight(this);
    setPreferredSize(new Dimension(width, height));
  }

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawImage(messageImage, 0, 0, this);
  }
}
```

Listing 7.12 ExitListener.java

```
package coreservlets;

import java.awt.*;
import java.awt.event.*;

/** Klasa dołączana do obiektu Frame lub JFrame
 *  najwyższego poziomu w aplikacji; umożliwia
 *  zamykanie okna aplikacji.
 */

public class ExitListener extends WindowAdapter {
  public void windowClosing(WindowEvent event) {
    System.exit(0);
  }
}
```

Rysunek 7.9 Wyniki wykonania aplikacji ShadowedTextFrame wywołanej przy użyciu polecenia ‘java coreservlets.ShadowedTextFrame "Serwlety to jest to" Haettenschweiler 100’

Rozdział 8.
Obsługa cookies

Cookies to niewielkie fragmenty informacji tekstowych, które najpierw serwer WWW przesyła do przeglądarki, a następnie, przy kolejnych odwołaniach do tej samej witryny lub domeny, przeglądarka przesyła w niezmienionej formie na serwer. Dzięki możliwości odczytania informacji przesłanych wcześniej do przeglądarki, serwer może udostępnić użytkownikom wiele ciekawych możliwości i ułatwień, takich jak dostosowanie zawartości witryny w sposób wcześniej wybrany przez użytkownika lub zezwalanie zarejestrowanych użytkownikom na dostęp do witryny bez konieczności ponownego podawania hasła. Większość przeglądarek unika przechowywania dokumentów skojarzonych z cookies w pamięci podręcznej, a zatem witryna może za każdym razem zwrócić inną zawartość.

W tym rozdziale przedstawię sposoby jawnego definiowania i odczytywania cookies z poziomu serwletów; a w następnym — sposoby wykorzystania narzędzi do obsługi sesji (które w niewidoczny sposób mogą używać cookies) do śledzenia poczynań użytkowników poruszających się po różnych strony witryny.

8.1 Korzyści stosowania cookies

W tej części rozdziału przedstawię cztery podstawowe sposoby wykorzystania cookies do rozszerzenia możliwości funkcjonalnych witryny i poprawienia jej atrakcyjności.

Identyfikacja użytkowników podczas trwania sesji na witrynach komercyjnych

Wiele internetowych witryn komercyjnych wykorzystuje rozwiązanie określane jako „koszyki” — dzięki niemu użytkownicy mogą wybrać towar, dodać go do „koszyka” i dalej kontynuować zakupy. Jednak połączenia HTTP są zazwyczaj zamykane po przesłaniu każdej ze stron. A zatem, gdy użytkownik wybierze nowy towar i doda go do koszyka, to w jaki sposób serwer może wiedzieć jaki to był użytkownik? Trwałe połączenia HTTP (patrz podrozdział 7.4) także nie rozwiązują tego problemu, gdyż zazwyczaj dotyczą one wyłącznie żądań przesyłanych w krótkich odstępach czasu; na przykład żądań przesyłanych gdy przeglądarka prosi o obrazy skojarzone z pobraną wcześniej stroną WWW. Poza tym wiele serwerów WWW oraz przeglądarek nie obsługuje trwałych połączeń. Narzędziem, które może rozwiązać ten problem są cookies. W rzeczywistości możliwość te są tak przydatne, iż serwlety dysponują specjalnymi narzędziami programistycznymi służącymi do śledzenia sesji, a twórcy serwletów nie muszą bezpośrednio operować na cookies, aby z nich skorzystać. Śledzenie sesji zostało omówione w rozdziale 9.

Unikanie konieczności podawania nazwy użytkownika i hasła

Korzystanie z usług i zasobów wielu dużych witryn wymaga uprzedniej rejestracji; jednak konieczność pamiętania i podawania nazwy użytkownika oraz hasła przy każdej wizycie, jest dość niewygodna. Cookies są doskonałą alternatywą dla witryn, które nie wymagają zabezpieczeń na wysokim poziomie. Gdy użytkownik zarejestruje się na witrynie, do jego przeglądarki jest przesyłane cookie zawierające unikalny identyfikator użytkownika (ID). Kiedy użytkownik później ponownie odwiedzi witrynę, cookie z jego identyfikatorem jest przesyłane na serwer, który sprawdza czy identyfikator należy do zarejestrowanego użytkownika i jeśli tak, to zezwala na dostęp bez konieczności podawania nazwy użytkownika i hasła. Witryna może także pamiętać adres użytkownika, numer jego karty kredytowej oraz inne informacje, które ułatwią wykonywanie wszelkich późniejszych transakcji.

Dostosowywanie witryny

Wiele „portali" pozwala użytkownikom na określanie wyglądu strony głównej. Może to dotyczyć wyboru prognozy pogody, którą użytkownik chce oglądać, wskazania akcji i wyników sportowych interesujących użytkownika, i tak dalej. Określanie wybranej zawartości strony podczas każdej wizyty na witrynie nie byłoby rozwiązaniem wygodnym dla użytkownika, i właśnie dlatego to witryna zapamiętuje wybrane ustawienia, wykorzystując do tego celu cookies. W prostych przypadkach, dostosowanie zawartości można zrealizować zapisując ustawienia bezpośrednio w cookie. Przykład takiego rozwiązania przedstawię w podrozdziale 8.6. Jednak w bardziej złożonych sytuacjach, witryna przesyła do przeglądarki jedynie unikalny identyfikator, natomiast preferencje użytkownika skojarzone z tym identyfikatorem są zapisywane w bazie danych działającej na serwerze.

Dobór reklam

Większość witryn prezentujących reklamy pobiera znacznie wyższe opłaty za wyświetlanie reklam „dobieranych" niż „losowych". Reklamodawcy zazwyczaj wolą zapłacić znacznie więcej za to, by ich reklamy trafiły do osób, których zainteresowania (dotyczące ogólnych kategorii produktów) są znane. Na przykład jeśli korzystasz z serwisu wyszukiwawczego i podasz w nim hasło „Java Servlets" to witryna obciąży reklamodawcę znacznie wyższymi kosztami w przypadku wyświetlenia reklamy środowiska służącego do tworzenia serwletów, niż biura podróży specjalizującego się w wycieczkach do Indonezji. Jednak z drugiej strony, jeśli poszukiwane hasło będzie miało postać „Java Hotele", to sytuacja będzie dokładnie odwrotna. Bez wykorzystania cookies, w momencie wejścia na witrynę (czyli w chwili gdy jeszcze nie zostało wykonane żadne wyszukiwanie) konieczne jest wyświetlenie losowej reklamy, podobnie jak w przypadku poszukiwania hasła, którego nie da się powiązać z żadną kategorią reklam. Cookies pozwalają jednak zapamiętać potrzebne dane — „Oho, ten użytkownik poprzednio poszukiwał takich informacji" — i wyświetlić specjalnie dobraną (czyli drogą) a nie losową (tanią) reklamę.

8.2 Niektóre problemy związane ze stosowaniem cookies

Cookies zostały stworzone po to by ułatwiać życie użytkownikom i poprawiać atrakcyjność witryn WWW. Pomimo wielu błędnych przeświadczeń, cookies wcale nie stanowią poważnego zagrożenia dla bezpieczeństwa użytkownika. Cookies nigdy nie są w żaden sposób interpretowane ani wykonywane, a zatem nie można ich użyć do przekazania wirusa lub zaatakowania komputera

użytkownika. Co więcej, przeglądarki zazwyczaj akceptują wyłącznie 20 cookies pochodzących z jednej witryny i nie więcej niż 300 cookies łącznie, a ponieważ pojedyncze cookie nie może przekraczać 4 kilobajtów wielkości, zatem nie można ich użyć do zapełnienia dysku na komputerze użytkownika lub przeprowadzenie ataku typu odmowa usługi.

Cookies nie zagrażają zatem *bezpieczeństwu* użytkownika, mogą jednak stanowić bardzo poważne zagrożenie dla jego *prywatności*. Po pierwsze, niektórym użytkownikom może się nie podobać fakt, że serwis będzie pamiętał, iż są to osoby poszukujące stron o pewnej tematyce. Na przykład, użytkownicy którzy poszukują ofert pracy bądź ważnych informacji medycznych mogą nie życzyć sobie, aby reklamy zdradzały te informacje ich współpracownikom, gdy oni będą czegoś szukać. Można sobie wyobrazić jeszcze gorszą sytuację gdy dwie witryny wspólnie wykorzystują informacje o użytkowniku korzystając z obrazków pobieranych z trzeciej witryny, która używa cookies i dzieli się informacjami w pierwszymi dwoma witrynami. (Na szczęście przeglądarka Netscape dysponuje opcją która pozwala odmówić przyjmowania cookies jeśli nie pochodzą one z witryny z którą nawiązano połączenie, a jednocześnie nie wyłącza całkowicie obsługi cookies.) Sztuczka polegająca na kojarzeniu cookies z obrazkami może zostać wykorzystana nawet we wiadomościach poczty elektronicznej, jeśli użytkownik korzysta z programu, który obsługuje pocztę w formacie HTML, potrafi obsługiwać cookies i jest skojarzony z przeglądarką. W takiej sytuacji, ktoś może Ci przesłać wiadomość zawierającą obrazy, do tych obrazów dołączyć cookies, a następnie identyfikować Cię (Twój adres poczty elektronicznej, itp.) gdy odwiedzisz jego witrynę. O rany...

Kolejny problem związany z prywatnością pojawia się na witrynach, które wykorzystują cookies przy obsłudze zbyt cennych informacji. Na przykład, jakaś duża księgarnia internetowa może przechowywać w cookies informacje o użytkownikach i zezwalać na dokonywanie zamówień bez konieczności powtórnego podawania informacji personalnych. Nie jest to szczególny problem, gdyż w takiej sytuacji numer karty kredytowej nigdy nie jest wyświetlany, a witryna przesyła zamówione książki na adres podany wraz z numerem karty kredytowej lub po podaniu nazwy użytkownika i hasła. W takiej sytuacji jedyną krzywdą jaką może Ci wyrządzić osoba korzystająca z Twojego komputera (lub ktoś kto ukradł Twój plik cookies) będzie przesłanie dużego zamówienia na Twój adres; a zamówienia zawsze można nie przyjąć. Jednak inne witryny mogą być mniej ostrożne i dać napastnikowi korzystającemu z cudzego komputera możliwość dostępu do cennych informacji personalnych. Co gorsze, niekompetentni twórcy witryn mogą umieszczać numery kart kredytowych lub inne ważne informacje bezpośrednio w cookies, a nie korzystać z niewinnych identyfikatorów, które stanowią jedynie połączenie z faktycznymi danymi o użytkowniku, przechowywanymi na serwerze. To bardzo niebezpieczne, gdyż większość użytkowników zostawiając swój komputer w pracy bez nadzoru, nie przypuszcza nawet, aby było równoznaczne z pozostawieniem numeru karty kredytowej na stole.

Z powyższych rozważań należy wyciągnąć dwojakie wnioski. Po pierwsze, z powodu realnych i oczywistych przyczyn, niektórzy użytkownicy wyłączają cookies. A zatem, nawet jeśli używasz cookies aby wzbogacić możliwości witryny, to jednak jej działanie nie powinno do nich *zależeć*. Po drugie, jako autor serwletów korzystających z cookies, nie powinieneś używać ich do obsługi wyjątkowo ważnych informacji, gdyż mogłoby to być ryzykowne dla użytkowników Twojej witryny; zwłaszcza jeśli ktoś niepowołany mógłby uzyskać dostęp do ich komputer lub pliku cookies.

8.3 Narzędzia obsługi cookies dostępne w servletach

Aby przesłać cookies do przeglądarki użytkownika, serwlet musi wykonać kilka czynności. Po pierwsze, musi stworzyć jedno lub kilka cookies o określonych nazwach i wartościach, używając w tym celu wywołania `new Cookie(nazwa, wartość)`. Następnie może podać wszelkie dodatkowe atrybuty cookie przy użyciu metod `setXxx` (można je później odczytać przy użyciu metod `getXxx`). W końcu serwlet musi dodać cookies do nagłówków odpowiedzi przy pomocy

wywołania `response.addCookie(cookie)`. Aby odczytać cookies nadesłane przez przeglądarkę, serwlet powinien skorzystać z wywołania `request.getCookies`. Metoda `getCookies` zwraca tablicę obiektów `Cookie` reprezentujących cookies jakie przeglądarka skojarzyła z daną witryną. (Jeśli w żądaniu nie zostały umieszczone żadne cookies, to tablica ta jest pusta, nie jest jednak równa `null`.) W większości przypadków serwlet przegląda tę tablicę, aż do momentu odszukania nazwy cookie (pobieranej przy użyciu metody `getName`), którego chciał użyć. Po odszukaniu odpowiedniego cookie, serwlet może pobrać jego wartość przy użyciu metody `getValue` klasy `Cookie`. Każde z tych zagadnień zostanie bardziej szczegółowo omówione w dalszej części rozdziału.

Tworzenie cookies

Cookies tworzy się wywołując konstruktor klasy `Cookie`. Konstruktor ten wymaga podania dwóch argumentów —łańcuchów znaków określających odpowiednio nazwę oraz wartość tworzonego cookie. Ani nazwa, ani wartość cookie nie może zawierać odstępów ani poniższych znaków:

```
[ ] ( ) = , " / ? @ : ;
```

Atrybuty cookies

Zanim dodasz cookie do nagłówków odpowiedzi, możesz podać jego różne cechy. Służą do tego metody `set`*`Xxx`*, gdzie *`Xxx`* to nazwa atrybutu którego wartość chcesz określić. Każdej z metod `set`*`Xxx`* odpowiada metoda `get`*`Xxx`* służąca do pobierania wartości odpowiedniego atrybutu. Za wyjątkiem nazwy i wartości, wszystkie atrybuty dotyczą wyłącznie cookies przekazywanych z serwera do przeglądarki; atrybuty te nie są określane dla cookies przesyłanych w przeciwnym kierunku. Skróconą wersję przedstawionych poniżej informacji znajdziesz także w dodatku A — „Krótki przewodnik po serwletach i JSP”.

public String getComment()
public void String setComment(String komentarz)

Pierwsza z tych metod pobiera, a druga określa komentarz skojarzony z danym cookie. W przypadku cookies działających według pierwszej wersji protokołu opracowanego przez firmę Netscape (protokół ten na numer wersji 0, patrz opis metod `getVersion` oraz `setVersion` podany w dalszej części rozdziału) komentarz ten jest używany wyłącznie na serwerze i wyłącznie w celach informacyjnych; nie jest on przesyłany do przeglądarki.

public String getDomain()
public void setDomain(String wzorzecDomeny)

Pierwsza z tych metod pobiera, a druga określa domenę której dotyczy dane cookie. Zazwyczaj przeglądarka zwraca cookies wyłącznie do komputera, którego nazwa ściśle odpowiada nazwie komputera z którego cookies zostały wysłane. Metody `setDomain` można użyć, aby poinstruować przeglądarkę, że cookies można także przekazywać do innych komputerów należących do tej samej domeny. Aby uniemożliwić tworzenie cookies, które będą przesyłane do komputera należącego do innej domeny niż ta, do której należy komputer który stworzył cookie, nazwa domeny podawana w metodzie `setDomain` musi zaczynać się od kropki „.” (na przykład: *.prenhall.com*), a dodatkowo musi zawierać dwie kropki w przypadku domen, które nie określają kraju (np.: *.com*, *.edu* lub *.gov*) i trzy kropki w przypadku gdy domena zawiera określenie kraju (np.: *.edu.pl*, *.gov.pl*, itp.). Na przykład, cookies zdefiniowane przez serwer *bali.vacations.com* nie będą zazwyczaj przesyłane przez przeglądarkę do stron należących do domeny *mexico.vacations.com*. Gdyby jednak witryna tworząca cookies chciała aby takie przesyłanie

cookies było możliwe, to serwlet powinien posłużyć się wywołaniem o postaci `cookie.setDomain(".vacations.com")`.

public int getMaxAge()
public void setMaxAge(int długośćIstnienia)

Pierwsza z tych metod zwraca a druga określa czas (wyrażony w sekundach) po upłynięciu którego cookie zostanie uznane za przestarzałe. Atrybut ten domyślnie posiada wartość ujemną, która oznacza, że cookie będzie przechowywane wyłącznie do końca bieżącej sesji przeglądarki (czyli do momentu gdy użytkownik ją zamknie) i nie będzie zapisywane na dysku. Na listingu 8.4 przedstawiłem klasę `LongLivedCookie` (klasę potomną klasy `Cookie`), definiującą cookie, którego czas istnienia jest automatycznie określany na jeden rok. Podanie wartości `0` w wywołaniu metody `setMaxAge` informuje przeglądarkę, że należy usunąć cookie.

public String getName()
public void setName(String nazwaCookie)

Pierwsza z tych metod zwraca a druga określa nazwę cookie. Nazwa oraz wartość, są dwoma elementami cookies na które zawsze trzeba zwracać uwagę. Niemniej jednak metoda `setName` jest stosowana rzadko, gdyż nazwa jest podawana w wywołaniu konstruktora, podczas tworzenia cookie. Z drugiej strony metoda `getName` jest używana niezwykle często, gdyż za jej pomocą pobieramy nazwy niemal wszystkich cookies przekazywanych z przeglądarki na serwer. Ponieważ metoda `getCookies` interfejsu `HttpServletRequest` zwraca tablicę obiektów `Cookie`, dlatego często stosowanym rozwiązaniem jest pobieranie po kolei nazw wszystkich cookies przechowywanych w tej tablicy, aż do momentu odszukania tego, które nas interesuje. Proces pobierania wartości cookie o określonej nazwie został zaimplementowany w metodzie `getCookieValue` przedstawionej na listingu 8.3.

public String getPath()
public void setPath(String ścieżka)

Pierwsza z tych metod zwraca a druga określa ścieżkę z jaką jest skojarzone cookie. Jeśli żadna ścieżka nie zostanie podana, to przeglądarka będzie umieszczać cookie wyłącznie w żądaniach dotyczących stron znajdujących się w tym samym katalogu serwera, w którym znajdowała się strona która utworzyła cookie, bądź w jego podkatalogach. Na przykład, jeśli serwer przesłał cookie podczas obsługi żądania dotyczącego strony *http://witryna.handlowa.com/zabawki/zb.html* to przeglądarka prześle cookie z powrotem wraz z żądaniem dotyczącym strony *http://witryna.handlowa.com/zabawki/rowery/dladzieci.html*, lecz nie z żądaniem dotyczącym strony *http://witryna.handlowa.com/cd/klasyka.html*. Metoda `setPath` może służyć do określenia bardziej ogólnych warunków zwracania cookies. Na przykład, wywołanie postaci `jakiesCookie.setPath("/")` informuje, że to cookie powinno być przesyłane do wszystkich stron na danym serwerze. Ścieżka podawana w wywołaniu tej metody musi zawierać aktualną stronę. Oznacza to, że można podać ścieżkę bardziej ogólną niż ścieżka domyślna; nie można natomiast podawać ścieżek bardziej szczegółowych. Na przykład, serwlet o ścieżce dostępu *http://witryna/sklep/obsluga/zadanie* może podać ścieżkę */sklep* (gdyż ścieżka ta zawiera w sobie ścieżkę */sklep/obsluga/*), lecz nie może użyć ścieżki */sklep/obsluga/zwroty/* (gdyż nie zawiera ona w sobie ścieżki */sklep/obsluga/*).

public boolean getSecure()
public void setSecure(boolean flagaBezpieczne)

Pierwsza z tych metod zwraca a druga określa wartość logiczną informującą czy cookie powinno być przesyłane wyłącznie przy wykorzystaniu szyfrowanego połączenia (na przykład połączenia SSL). Wartość domyślna tego atrybutu — `false` — oznacza, że cookie powinno być przesyłane dowolnymi połączeniami.

public String getValue()
public void setValue(String wartośćCookie)

Metoda `getValue` zwraca wartość skojarzoną z cookie, natomiast metoda `setValue` określa ją. Jak wiadomo, nazwa oraz wartość są dwoma elementami cookies, które niemal zawsze są używane; jednak zdarza się nazwa cookie pełni funkcję flagi logicznej i w takich przypadkach jego wartość jest pomijana (dotyczy to na przykład sytuacji, gdy znaczenie ma sam fakt, że cookie o konkretnej nazwie istnieje).

public int getVersion()
public void setVersion(int wersja)

Pierwsza z tych metod zwraca, a druga określa wersję protokołu cookies, z którym dane cookie jest zgodne. Domyślna wersja — 0 — jest zgodna z oryginalną specyfikacją firmy Netscape (*http://www.netscape.com/newsref/std/cookie_spec.html*). Wersja 1 bazuje na pliku RFC 2109 (możesz go znaleźć na witrynie *http:///www.rfc-editor.org/*), lecz aktualnie nie zyskała jeszcze dużej popularności.

Umieszczanie cookies w nagłówkach odpowiedzi

Cookie można umieścić w nagłówku odpowiedzi `Set-Cookie` przy użyciu metody `addCookie` interfejsu `HttpServletResponse`. Zwróć uwagę, że metoda ta nosi nazwę `addCookie` a nie `setCookie`, gdyż jej wywołanie nie usuwa wcześniej wygenerowanych nagłówków `Set-Cookie`, lecz dodaje do nich nowy. Oto przykład:

```
Cookie mojeCookie = new Cookie( "user", "uid1234" );
mojeCookie.setMaxAge( 60 * 60 * 24 * 365 ); // jeden rok
response.addCookie( mojeCookie );
```

Odczytywanie cookies nadesłanych przez przeglądarkę

Aby przesłać cookie do przeglądarki, należy utworzyć obiekt `Cookie`, a następnie wygenerować nagłówek odpowiedzi `Set-Cookie` przy użyciu metody `addCookie`. Aby odczytać cookies, które zostały nadesłane *przez* przeglądarkę, należy się posłużyć metodą `getCookies` interfejsu `HttpServletRequest`. Metoda ta zwraca tablicę obiektów `Cookie` reprezentujących wartości nadesłane przez przeglądarkę w nagłówkach żądania `Cookie`. Jeśli żądanie nie zawiera żadnych cookies, to tablica ta jest pusta. Kiedy uzyskasz już tę tablicę, to zazwyczaj będziesz ją analizował wywołując metodę `getName` dla każdego jej elementu, aż do chwili gdy odnajdziesz cookie o poszukiwanej nazwie. Po odnalezieniu poszukiwanego cookie zapewne pobierzesz jego wartość przy użyciu metody `getValue` i przetworzysz ją w odpowiedni sposób. Ten proces jest wykonywany tak często, iż w podrozdziale 8.5 przedstawiłem dwie metody upraszczające pobieranie całego cookie o podanej nazwie lub samej jego wartości.

8.4 Przykłady generacji i odczytywana cookies

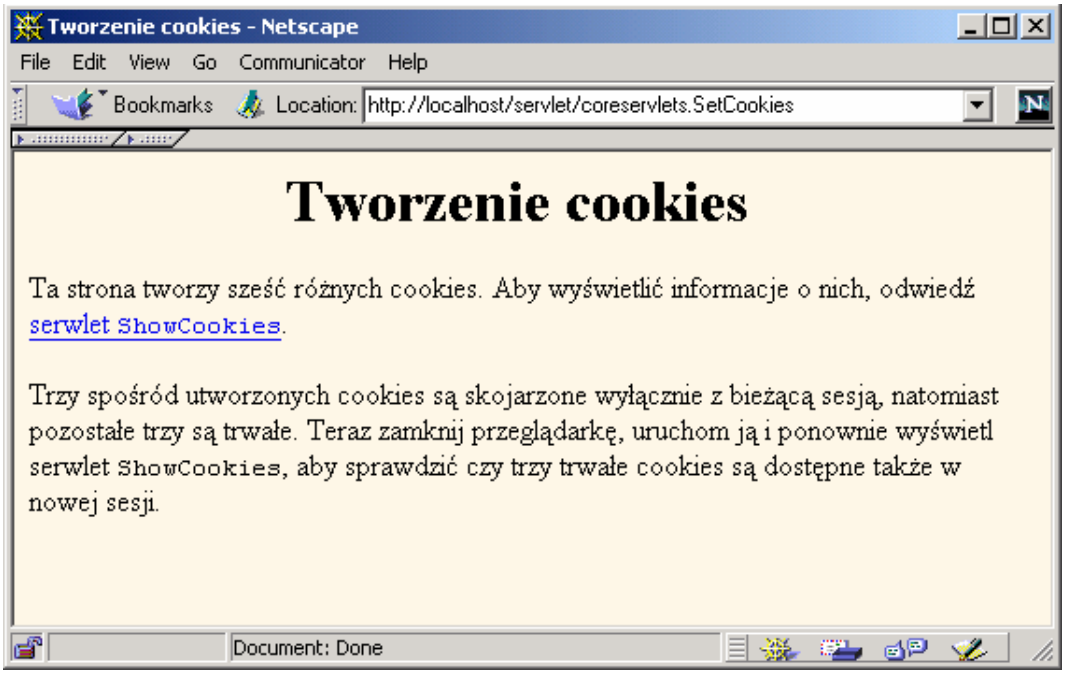
Listing 8.1 przedstawia kod źródłowy serwletu `SetCookies` generującego sześć cookies; wyniki jego wykonania zostały przedstawione na rysunku 8.1. Trzy spośród sześciu generowanych cookies mają domyślną datę wygaśnięcia ważności, co oznacza, że będą one używane wyłącznie do momentu gdy użytkownik zamknie przeglądarkę. Przy tworzeniu pozostałych trzech cookies została użyta metoda `setMaxAge`, dzięki czemu cookies te będą istniał przez godzinę od chwili utworzenia, niezależnie od tego czy w tym czasie użytkownik zamknie i ponownie uruchomi

przeglądarkę lub nawet ponownie uruchomi komputer, aby w ten sposób rozpocząć nową sesję przeglądarki.

Listing 8.2 przedstawia serwlet wyświetlający tablicę z informacjami o wszystkich nadesłanych do niego cookies. Na rysunku 8.2 przedstawiłem wyniki wykonania tego serwletu, bezpośrednio po wykonaniu serwletu `SetCookies`. Rysunek 8.3 przedstawia natomiast wyniki wykonania serwletu `ShowCookies`, po wykonaniu serwletu `SetCookies` oraz zamknięciu i ponownym uruchomieniu przeglądarki.

Listing 8.1 SetCookies.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Servlet tworzy sześć cookies: połowa z nich będzie
 *  istnieć tylko do końca aktualnej sesji (niezależnie
 *  od czasu jej trwania), a druga połowa będzie istnieć
 *  dokładnie przez godzinę od czasu utworzenia
 *  (niezależnie od tego czy przeglądarka zostanie
 *  zamknięta i ponownie uruchomiona, czy nie).
 */

public class SetCookies extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    for(int i=0; i<3; i++) {
      // Domyślna wartość maxAge wynosi -1, co oznacza
      // że cookie będzie istniało wyłącznie w czasie
      // trwania bieżącej sesji przeglądarki.
      Cookie cookie = new Cookie("Cookie-Sesyjne-" + i,
                                 "Cookie-Wartość-S" + i);
      response.addCookie(cookie);
      cookie = new Cookie("Cookie-Trwałe-" + i,
                          "Cookie-Wartość-T" + i);
      // Cookie będzie ważne przez godzinę, niezależnie od
      // tego czy użytkownik zamknie przeglądarkę i ją
      // ponownie otworzy czy też nie.
      cookie.setMaxAge(3600);
      response.addCookie(cookie);
    }
    response.setContentType("text/html; charset=ISO-8859-2");
    PrintWriter out = response.getWriter();
    String title = "Tworzenie cookies";
    out.println
      (ServletUtilities.headWithTitle(title) +
       "<BODY BGCOLOR=\"#FDF5E6\">\n" +
       "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n" +
       "Ta strona tworzy sześć różnych cookies.\n" +
       "Aby wyświetlić informacje o nich, odwiedź \n" +
       "<A HREF=\"/servlet/coreservlets.ShowCookies\">\n" +
       "serwlet <CODE>ShowCookies</CODE></A>.\n" +
       "<P>\n" +
       "Trzy spośród utworzonych cookies są skojarzone wyłącznie \n" +
       "z bieżącą sesją, natomiast pozostałe trzy są trwałe.\n" +
       "Teraz zamknij przeglądarkę, uruchom ją i ponownie wyświetl\n" +
       "serwlet <CODE>ShowCookies</CODE>, aby sprawdzić czy \n" +
       "trzy trwałe cookies są dostępne także w nowej sesji.\n" +
       "</BODY></HTML>");
  }
}
```



Rysunek 8.1 Wyniki wykonania serwletu SetCookies

Listing 8.2 ShowCookies.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Serwlet tworzy tabelę z informacjami dotyczącymi
 *  cookies skojarzonymi z aktualną stroną.
 */

public class ShowCookies extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html; charset=ISO-8859-2");
    PrintWriter out = response.getWriter();
    String title = "Aktywne cookies";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n" +
                "<TABLE BORDER=1 ALIGN=\"CENTER\">\n" +
                "<TR BGCOLOR=\"#FFAD00\">\n" +
                "  <TH>Nazwa cookie\n" +
                "  <TH>Wartość cookie");
    Cookie[] cookies = request.getCookies();
    Cookie cookie;
    for(int i=0; i<cookies.length; i++) {
      cookie = cookies[i];
      out.println("<TR>\n" +
                  "  <TD>" + cookie.getName() + "\n" +
                  "  <TD>" + cookie.getValue());
    }
    out.println("</TABLE></BODY></HTML>");
  }
}
```

Rysunek 8.2 Wyniki wykonania serwletu ShowCookies w ciągu godziny po wykonaniu serwletu SetCookies i w trakcie trwania tej samej sesji przeglądarki

Rysunek 8.3 Wyniki wykonania serwletu ShowCookies w ciągu godziny po wykonaniu serwletu SetCookies i po rozpoczęciu nowej sesji przeglądarki

8.5 Proste narzędzia do obsługi cookies

W tej części rozdziału przedstawiłem dwie proste lecz przydatne metody, ułatwiające obsługę cookies.

Odnajdywanie cookie o określonej nazwie

Listing 8.3 przedstawia fragment pliku *ServletUtilities.java* zawierający metody ułatwiające pobieranie cookie oraz samej wartości cookie o podanej nazwie. Metoda `getCookieValue` przegląda tablicę dostępnych obiektów klasy `Cookie`, i zwraca wartość cookie o podanej nazwie. Jeśli cookie o podanej nazwie nie zostanie odnalezione, to metoda zwraca podaną wartość domyślną. Poniżej przedstawiłem typowy, wykorzystywany przeze mnie sposób obsługi cookies:

```
Cookie[] cookies = request.getCookies();
String kolor =
    ServletUtilities.getCookieValue(cookies, "kolor", "black");
String czcionka =
    ServletUtilities.getCookieValue(cookies, "czcionka", "Arial");
```

Także metoda `getCookie` przegląda tablicę w poszukiwaniu cookie o podanej nazwie; jednak zwraca cały obiekt `Cookie`, a nie samą wartość. Metoda ta jest wykorzystywana w sytuacjach gdy chcesz wykonać jakieś czynności na cookie, a jego wartość Cię chwilowo nie interesuje.

Listing 8.3 ServletUtilities.java

```
package coreservlets;

import javax.servlet.*;
import javax.servlet.http.*;


public class ServletUtilities {

  // ... inne metody przedstawione w innych miejscach książki ...

  public static String getCookieValue(Cookie[] cookies,
                                       String cookieName,
                                       String defaultValue) {
    if (cookies != null) {
      for(int i=0; i<cookies.length; i++) {
        Cookie cookie = cookies[i];
        if (cookieName.equals(cookie.getName()))
          return(cookie.getValue());
      }
    }
    return(defaultValue);
  }

  public static Cookie getCookie(Cookie[] cookies,
                                 String cookieName) {
    if (cookies != null) {
      for(int i=0; i<cookies.length; i++) {
        Cookie cookie = cookies[i];
        if (cookieName.equals(cookie.getName()))
          return(cookie);
      }
    }
    return(null);
  }

  // ... inne metody przedstawione w innych miejscach książki ...

}
```

Tworzenie cookies o długim czasie istnienia

Listing 8.4 przedstawia niewielką klasę, której możesz użyć zamiast klasy `Cookie` jeśli chcesz aby tworzone cookies istniały dłużej niż do końca bieżącej sesji przeglądarki. Przykład serwletu korzystającego z tej klasy został przedstawiony na listingu 8.5.

Listing 8.4 LongLivedCookie.java

```
package coreservlets;
```

```
import javax.servlet.http.*;

/** Cookie które będzie istnieć dokładnie przez 1 rok.
 *  Cookies tworzone w domyślny sposób istnieją tylko
 *  do końca bieżącej sesji przeglądarki.
 */

public class LongLivedCookie extends Cookie {
  public static final int SECONDS_PER_YEAR = 60*60*24*365;

  public LongLivedCookie(String name, String value) {
    super(name, value);
    setMaxAge(SECONDS_PER_YEAR);
  }
}
```

8.6 Interfejs wyszukiwawczy z możliwością zapamiętywania ustawień

Listing 8.5 przedstawia serwlet o nazwie `CustomizedSearchEngines`. Jest to nowa wersja serwletu `SearchEngines` przedstawionego w podrozdziale 6.3. Nowa wersja serwletu (podobnie jak poprzednia) odczytuje informacje podane przez użytkownika w formularzu HTML (patrz rysunek 8.5) i przesyła je do odpowiedniego mechanizmu wyszukiwawczego. Dodatkowo serwlet `CustomizedSearchEngines` przekazuje do przeglądarki cookies zawierające informacje o realizowanym wyszukiwaniu. Dzięki temu, gdy użytkownik ponownie wyświetli formularz służący do obsługi serwletu (nawet po zamknięciu i ponownym uruchomieniu przeglądarki), to zostaną w nim wyświetlone ustawienia z ostatniego wyszukiwania.

Aby zaimplementować taki interfejs użytkownika, formularz wyszukiwawczy musi być generowany dynamicznie, wykorzystanie statycznej strony WWW nie jest w tej sytuacji wystarczające. Kod serwletu generującego formularz wyszukiwawczy został przedstawiony na listingu 8.6, a wyniki jego działania — na rysunku 8.4. Serwlet generujący formularz odczytuje wartości przekazane za pośrednictwem cookies i używa ich jaki początkowych wartości pól formularza. Zwróć uwagę, iż serwlet ten nie mógłby przekazać cookies bezpośrednio do przeglądarki. Wynika to z faktu, że informacje przechowywane w cookies nie są znane aż do chwili gdy użytkownik poda je w polach formularza i prześle na serwer, to jednak może nastąpić dopiero po zakończeniu działania serwletu, który generuje formularz wyszukiwawczy.

Ten przykład wykorzystuje klasę `LongLivedCookie`, przedstawioną w poprzedniej części rozdziału. Klasa ta tworzy obiekty `Cookie`, w których data wygaśnięcia jest ustawiana na rok od chwili utworzenia, dzięki czemu przeglądarka będzie ich używać tych nawet w kolejnych sesjach.

Listing 8.5 CustomizedSearchEngines.java

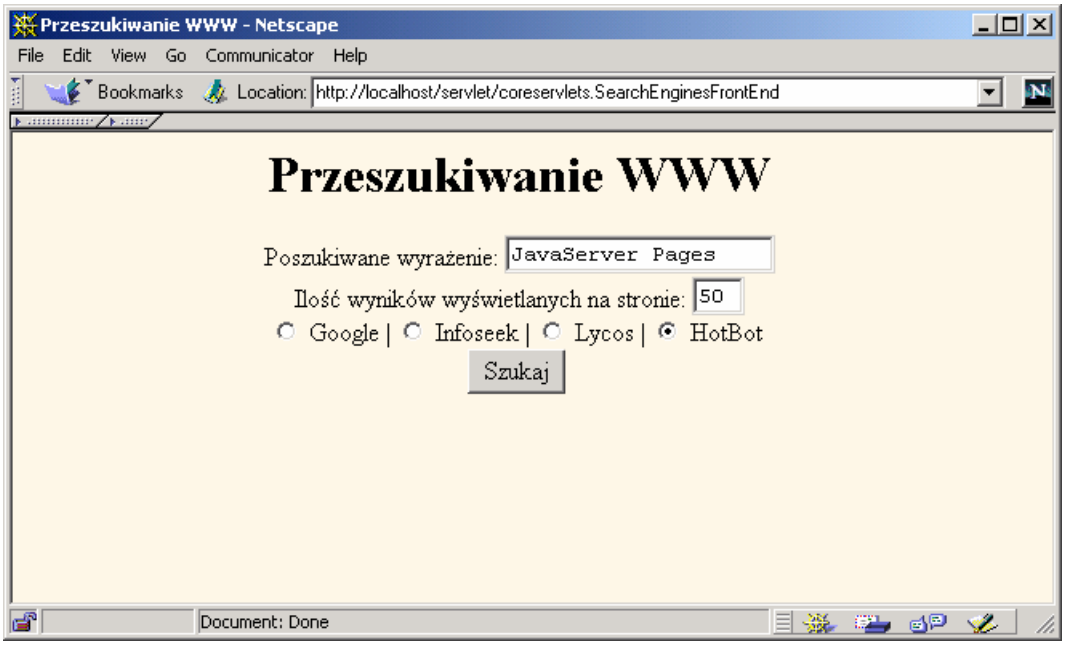
```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;

/** Zmodyfikowana wersja serwletu SearchEngine, która
 *  wykorzystuje cookies w celu zapamiętania opcji
 *  wybranych przez użytkownika. Zapamiętane wartości
 *  są następnie wykorzystywane przez serwlet
 *  SearchEngineFrontEnd w celu inicjalizacji pól
 *  formularza wyszukiwawczego.
 */

public class CustomizedSearchEngines extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
```

```
    String searchString = request.getParameter("searchString");
    if ((searchString == null) ||
        (searchString.length() == 0)) {
      reportProblem(response, "Brak łańcucha zapytania.");
      return;
    }
    Cookie searchStringCookie =
      new LongLivedCookie("searchString", searchString);
    response.addCookie(searchStringCookie);
    // Klasa URLEncoder zamienia odstępy na znaki "+"
    // oraz inne znaki które nie są znakami alfanumerycznymi
    // na wyrażenia %XY, gdzie XY to wartość znaku w kodzie
    // ASCII (lub ISO Latin-1) zapisana w formie liczby
    // szesnastkowej.
    // Przeglądarki zawsze kodują wartości wpisane w polach
    // formularzy w ten właśnie sposób, a zatem metoda
    // getParameter serwletów dekoduje je automatycznie.
    // Jednak my przekazujemy te wartości na inny serwer,
    // i dlatego musimy je ponownie zakodować.
    searchString = URLEncoder.encode(searchString);
    String numResults = request.getParameter("numResults");
    if ((numResults == null) ||
        (numResults.equals("0")) ||
        (numResults.length() == 0)) {
      numResults = "10";
    }
    Cookie numResultsCookie =
      new LongLivedCookie("numResults", numResults);
    response.addCookie(numResultsCookie);
    String searchEngine = request.getParameter("searchEngine");
    if (searchEngine == null) {
      reportProblem(response, "Brak nazwy serwisu wyszukiwawczego.");
      return;
    }
    Cookie searchEngineCookie =
      new LongLivedCookie("searchEngine", searchEngine);
    response.addCookie(searchEngineCookie);
    SearchSpec[] commonSpecs = SearchSpec.getCommonSpecs();
    for(int i=0; i<commonSpecs.length; i++) {
      SearchSpec searchSpec = commonSpecs[i];
      if (searchSpec.getName().equals(searchEngine)) {
        String url =
          searchSpec.makeURL(searchString, numResults);
        response.sendRedirect(url);
        return;
      }
    }
    reportProblem(response, "Nieznany mechanizm wyszukiwawczy.");
  }

  private void reportProblem(HttpServletResponse response,
                             String message)
      throws IOException {
    response.sendError(response.SC_NOT_FOUND,
                       "<H2>" + message + "</H2>");
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

Listing 8.6 SearchEnginesFrontEnd.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;

/** Dynamicznie generowana, zmodyfikowana wersja
 *  dokumentu HTML SearchEngines.html. Serwlet
 *  ten używa cookies, aby zapamiętać opcje
 *  wybrane przez użytkownika.
```

```
 */

public class SearchEnginesFrontEnd extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    Cookie[] cookies = request.getCookies();
    String searchString =
      ServletUtilities.getCookieValue(cookies,
                                      "searchString",
                                      "Java Programming");
    String numResults =
      ServletUtilities.getCookieValue(cookies,
                                      "numResults",
                                      "10");
    String searchEngine =
      ServletUtilities.getCookieValue(cookies,
                                      "searchEngine",
                                      "google");
    response.setContentType("text/html; charset=ISO-8859-2");
    PrintWriter out = response.getWriter();
    String title = "Przeszukiwanie WWW";
    out.println
      (ServletUtilities.headWithTitle(title) +
       "<BODY BGCOLOR=\"#FDF5E6\">\n" +
       "<H1 ALIGN=\"CENTER\">Przeszukiwanie WWW</H1>\n" +
       "\n" +
       "<FORM ACTION=\"/servlet/" +
         "coreservlets.CustomizedSearchEngines\">\n" +
       "<CENTER>\n" +
       "Poszukiwane wyrażenie:\n" +
       "<INPUT TYPE=\"TEXT\" NAME=\"searchString\"\n" +
       "       VALUE=\"" + searchString + "\"><BR>\n" +
       "Ilość wyników wyświetlanych na stronie:\n" +
       "<INPUT TYPE=\"TEXT\" NAME=\"numResults\"\n" +
       "       VALUE=" + numResults + " SIZE=3><BR>\n" +
       "<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n" +
       "       VALUE=\"google\"" +
       checked("google", searchEngine) + ">\n" +
       "Google |\n" +
       "<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n" +
                "       VALUE=\"infoseek\"" +
       checked("infoseek", searchEngine) + ">\n" +
       "Infoseek |\n" +
       "<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n" +
       "       VALUE=\"lycos\"" +
       checked("lycos", searchEngine) + ">\n" +
       "Lycos |\n" +
       "<INPUT TYPE=\"RADIO\" NAME=\"searchEngine\"\n" +
       "       VALUE=\"hotbot\"" +
       checked("hotbot", searchEngine) + ">\n" +
       "HotBot\n" +
       "<BR>\n" +
       "<INPUT TYPE=\"SUBMIT\" VALUE=\"Szukaj\">\n" +
       "</CENTER>\n" +
       "</FORM>\n" +
       "\n" +
       "</BODY>\n" +
       "</HTML>\n");
  }

  private String checked(String name1, String name2) {
    if (name1.equals(name2))
      return(" CHECKED");
    else
      return("");
  }
}
```



Rysunek 8.4 Wyniki wykonania serwletu SearchEnginesFrontEnd. Podane opcje wyszukiwania staną się domyślnymi wartościami pól przy kolejnym wyświetleniu formularza

Rysunek 8.5 Wyniki wykonania serwletu CustomizedSearchEngines

Rozdział 9.
Śledzenie sesji

Ten rozdział pokazuje w jaki sposób mechanizmy śledzenia sesji dostępne w serwletach można wykorzystać do przechowywania informacji o użytkownikach poruszających się po witrynie WWW0.

9.1 Potrzeba śledzenia sesji

HTTP jest protokołem „bezstanowym" — oznacza to, że za każdym razem gdy przeglądarka pobiera stronę WWW tworzone jest niezależne połączenie z serwerem, który automatycznie nie przechowuje żadnych kontekstowych informacji skojarzonych z przeglądarką, która to połączenie nawiązała. Nawet w serwerach, które wykorzystują trwałe połączenia HTTP i za pomocą jednego połączenia obsługują wiele żądań zgłaszanych w niewielkich odstępach czasu, nie ma żadnych wbudowanych mechanizmów ułatwiających przechowywanie takich kontekstowych informacji. Brak takiego kontekstu przysparza wielu problemów. Na przykład, kiedy klienci internetowego sklepu dodają wybrany towar do swojego wirtualnego koszyka, skąd serwer będzie wiedział co się już w nim znajduje? Podobnie, gdy klient zdecyduje się przejrzeć wybrane towary i zapłacić za nie, w jaki sposób serwer ma określić który z istniejących koszyków należy do danego klienta?

Istnieją trzy podstawowe sposoby rozwiązywania problemów tego typu — cookies, przepisywanie adresów URL oraz zastosowanie ukrytych pól formularzy.

Cookies

Cookies można wykorzystać do przechowywania informacji o sesji w internetowym sklepie, a każde kolejne połączenie może powodować odszukanie bieżącej sesji i pobranie informacji o niej, przechowywanych na serwerze. Na przykład, serwlet mógłby wykonywać następujące czynności:

```
String idSesji = utworzUnikalnyIdentyfikatorString();
Hashtable infoSesji = new Hashtable();
Hashtable tablicaGlobalna = odszukajTabliceZDanymiSesji();
tablicaGlobalna.put(idSesji, infoSesji);
Cookie cookieSesji = new Cookie("SESSIONID", idSesji);
sessionCookie.setPath("/");
response.addCookie(cookieSesji);
```

Dzięki temu, podczas obsługi kolejnych żądań, na podstawie wartości cookie `"SESSIONID"` serwer będzie mógł pobrać z tablicy `tablicaGlobalna` tablicę asocjacyjną `infoSesji` zawierającą informacje o konkretnej sesji. To doskonałe rozwiązanie, które jest bardzo często wykorzystywane przy obsłudze sesji. Niemniej jednak wciąż warto by mieć do dyspozycji narzędzia programistyczne wyższego poziomu, które obsługiwałyby wszystkie szczegóły. Choć serwlety dysponują łatwymi w obsłudze narzędziami wysokiego poziomu przeznaczonymi do obsługi cookies (opisałem je w rozdziale 8), to wciąż jednak jest stosunkowo dużo drobnych czynności które należy wykonać samodzielnie. Do czynności tych można zaliczyć:

- pobranie cookie przechowującego identyfikator sesji (w końcu, cookies nadesłanych z przeglądarki może być kilka),
- określenie odpowiedniego czasu wygaśnięcia ważności cookie (sesje, które są nieaktywne przez ponad 24 godziny, prawdopodobnie powinne zostać zamknięte),
- skojarzenie tablic asocjacyjnych z każdym z żądań,
- generację unikalnego identyfikatora sesji.

Poza tym, ze względu na doskonale znane i oczywiste zagrożenia prywatności jakie wiążą się z wykorzystaniem cookies (patrz podrozdział 8.2), niektóre osoby wyłączają ich obsługę. A zatem, oprócz protokołu wysokiego poziomu, warto by mieć jakieś alternatywne rozwiązanie.

Przepisywanie adresów URL

Rozwiązanie to polega na dopisywaniu na końcu adresu URL pewnych dodatkowych danych, które identyfikują sesję i pozwalają serwerowi na skojarzenie przekazanego identyfikatora z przechowywanymi informacjami dotyczącymi danej sesji. Na przykład, w adresie URL *http://komputer/katalog/plik.html;sessionid=1234*, dołączone informacje o sesji mają postać *sessionid=1234*. Także to rozwiązanie jest bardzo dobre, a posiada tę dodatkową zaletę, iż można z niego korzystać nawet w przeglądarkach, które nie obsługują cookies oraz wtedy, gdy użytkownik wyłączył obsługę cookies w przeglądarce. Jednak także i to rozwiązanie nastręcza tych samych problemów co stosowanie cookies — zmusza program działający na serwerze do wykonywania wielu prostych lecz uciążliwych czynności. Poza tym należy zwrócić szczególną uwagę, aby każdy adres URL odwołujący się do danej witryny i przekazywany do przeglądarki użytkownika (nawet w sposób niejawny, na przykład, przy użyciu nagłówka `Location`), został uzupełniony do dodatkowe informacje. Poza tym, jeśli użytkownik zostawi sesję i później wróci na witrynę korzystając z zapamiętanego adresu lub połączenia, wszystkie informacje o sesji zostaną utracone.

Ukryte pola formularzy

W formularzach HTML można umieszczać pola o następującej postaci:

```
<INPUT TYPE="HIDDEN" NAME="sesja" VALUE="...">
```

Wykorzystanie takiego elementu oznacza, że podczas przesyłania formularza, podana nazwa elementu oraz jego wartość zostaną dołączone do pozostałych informacji przekazywanych na serwer. Więcej informacji na temat ukrytych pól formularzy znajdziesz w podrozdziale 16.9, pt.: „Pola ukryte”. W takich ukrytych polach formularzy można przechowywać informacje na temat sesji. Jednak wykorzystanie tej metody ma jedną podstawową wadę — każda strona witryny musi być generowana dynamicznie.

Śledzenie sesji w serwletach

Serwlety udostępniają doskonałe, techniczne rozwiązanie problemu śledzenia sesji — interfejs `HttpSession`. Możliwości funkcjonalne tego interfejsu bazują na wykorzystaniu cookies lub przepisywania adresów URL. W rzeczywistości większość serwerów wykorzystuje cookies jeśli tylko przeglądarka użytkownika je obsługuje, lecz automatycznie może wykorzystać przepisywanie adresów URL, jeśli przeglądarka nie obsługuje cookies lub gdy ich obsługa została jawnie wyłączona. W tym przypadku autorzy serwletów nie muszą zaprzątać sobie głowy niepotrzebnymi szczegółami, nie muszą jawnie obsługiwać cookies ani informacji dołączanych do adresów URL, a zyskują bardzo wygodne miejsce, w którym mogą przechowywać dowolne obiekty skojarzone z każdą sesją.

9.2 Narzędzia programistyczne do śledzenia sesji

Wykorzystanie sesji w serwletach jest bardzo proste i wiąże się z pobieraniem obiektu sesji skojarzonego z obsługiwanym żądaniem, tworzeniem nowego obiektu sesji jeśli okaże się to konieczne, pobieraniem informacji skojarzonych z sesją, zapisywaniem informacji w obiekcie sesji oraz z usuwaniem przerwanych lub zakończonych sesji. Dodatkowo, jeśli przekazujesz do użytkownika jakiekolwiek adresy URL odwołujące się do Twojej witryny i jeśli jest stosowane przepisywanie adresów URL, to do każdego adresu URL będziesz musiał dołączyć informacje o sesji.

Pobieranie obiektu HttpSession skojarzonego z bieżącym żądaniem

Obiekt `HttpSession` można pobrać przy użyciu metody `getSession` interfejsu `HttpServletRequest`. W niewidoczny sposób, system pobiera identyfikator użytkownika z cookie lub z informacji dołączonych do adresu URL, a następnie używa go jako klucza przy pobieraniu odpowiedniego elementu z tablicy utworzonych wcześniej obiektów `HttpSession`. Jednak wszystkie te czynności są wykonywane w sposób niezauważalny dla programisty, który jedynie wywołuje metodę `getSession`. Jeśli wywołanie tej metody zwróci wartość `null`, będzie to oznaczało, że z danym użytkownikiem jeszcze nie jest skojarzona żadna sesja, a zatem można ją utworzyć. W takiej sytuacji sesje są tworzone tak często, iż dostępna jest specjalna opcja pozwalająca na utworzenie nowej sesji jeśli sesja dla danego użytkownika jeszcze nie istnieje. W tym celu wystarczy przekazać wartość `true` w wywołaniu metody `getSession`. A zatem, pierwszy krok przy korzystaniu z sesji ma zazwyczaj następującą postać:

```
HttpSession sesja = request.getSession(true);
```

Jeśli interesuje Cię czy sesja istniała już wcześniej czy też została właśnie utworzona, to możesz to sprawdzić za pomocą metody `isNew`.

Pobieranie informacji skojarzonych z sesją

Obiekty `HttpSession` są przechowywane na serwerze; są one automatycznie kojarzone z żądaniami przy wykorzystaniu niewidocznych mechanizmów, takich jak cookies lub przepisywanie adresów URL. Obiekty te posiadają wbudowaną strukturę danych, która umożliwia im przechowywanie dowolnej ilości kluczy oraz skojarzonych z nimi wartości. W specyfikacjach Java Servlet 2.1 oraz wcześniejszych, do pobrania wartości uprzednio zapisanej w sesji, używane jest wywołanie o postaci `sessin.getValue("nazwaAtrybutu")`. Metoda `getValue` zwraca obiekt typu `Object`, a zatem, aby przywrócić oryginalny typ danej zapisane w sesji, konieczne jest zastosowanie odpowiedniego rzutowania typów. Metoda zwraca wartość `null` jeśli nie ma atrybutu o podanej nazwie. Oznacza to, że przed wywołaniem jakichkolwiek metod obiektu skojarzonego z sesją, należy sprawdzić czy nie jest to wartość `null`.

W specyfikacji Java Servlet 2.2 metoda `getValue` została uznana za przestarzałą i zastąpiono ją metodą `getAttribute`. Nazwa tej metody lepiej odpowiada metodzie `setAttribute` (w specyfikacji 2.1 metodzie `getValue` odpowiadała metoda `putValue` a nie `setValue`). W przykładach przedstawionych w niniejszej książce będę jednak używał metody `getValue`, gdyż nie wszystkie dostępne na rynku komercyjne mechanizmy obsługi serwletów są zgodne ze specyfikacją 2.2.

Poniżej podałem przykład wykorzystania sesji; zakładam przy tym iż `ShoppingCart` to klasa zdefiniowana w celu przechowywania informacji o zamawianych towarach (implementację tej klasy znajdziesz w podrozdziale 9.4, pt.: „Internetowy sklep wykorzystujący koszyki i śledzenie sesji”).

```
HttpSession sesja = request.getSession(true);
```

```
ShoppingCart koszyk = (ShoppingCart)sesja.getValue("shoppingCart");
if (koszyk == null) { // w sesji nie ma jeszcze koszyka
  koszyk = new ShoppingCart();
  sesja.putValue("shoppingCart",koszyk);
}
zrobCosZKoszykiem(koszyk);
```

W większości przypadków będziesz znał nazwę atrybutu, a Twoim celem będzie pobranie wartości skojarzonej z tą nazwą. Istnieje także możliwość pobrania nazw wszystkich atrybutów skojarzonych z daną sesją — służy do tego metoda `getValueNames` zwracająca tablicę łańcuchów znaków. Ta metoda jest jedynym sposobem określenia nazw atrybutów w przypadku korzystania z mechanizmów obsługi serwletów zgodnych ze specyfikacją Java Servlet 2.1. W przypadku mechanizmów zgodnych ze specyfikacją 2.2 nazwy atrybutów można pobrać przy użyciu metody `getAttributeNames`. Metoda ta działa w bardziej spójny sposób, gdyż zwraca obiekt `Enumeration`, podobnie jak metody `getHeaderNames` oraz `getParameterNames` interfejsu `HttpServletRequest`.

Choć bez wątpienia najbardziej będą Cię interesowały dane bezpośrednio skojarzone z sesją, to jednak dostępne są także inne, interesujące informacje dotyczące sesji, które czasami mogą być użyteczne. Poniżej przedstawiłem metody interfejsu `HttpSession`:

public Object getValue(String nazwa)
public Object getAttribute(String nazwa)

Metody te pobierają z obiektu sesji wartość, która uprzednio została w nim zapisana. Obie metody zwracają wartość `null` jeśli z podaną nazwą nie jest skojarzona żadna wartość. W mechanizmach obsługi serwletów zgodnych ze specyfikacją Java Servlet 2.1 należy używać metody `getValue`. W mechanizmach zgodnych ze specyfikacją 2.2 można używać obu metod, jednak preferowane jest użycie metody `getAttribute` gdyż `getValue` została uznana za przestarzałą.

public void putValue(String nazwa, Object wartość)
public void setAttribute(String nazwa, Object wartość)

Te metody kojarzą wartość z nazwą. W mechanizmach obsługi serwletów zgodnych ze specyfikacją Java Servlet 2.1 należy używać metody `putValue`, natomiast w mechanizmach zgodnych ze specyfikacją 2.2 można stosować obie metody (choć `setAttribute` jest preferowana, gdyż `putValue` została uznana za przestarzałą). Jeśli obiekt przekazany w wywołaniu metody `putValue` lub `setAttribute` implementuje interfejs `HttpSessionBindingListener`, to po zapisaniu tego obiektu w sesji jest wywoływana jego metoda `valueBound`. Podobnie, jeśli obiekt implementuje interfejs `HttpSessionBindingListener`, to po jego usunięciu z sesji zostanie wywoływana metoda `valueUnbound` tego obiektu.

public void removeValue(String nazwa)
public void removeAttribute(String nazwa)

Obie te metody powodują usunięcie wartości skojarzonych z podaną nazwą. Jeśli usuwana wartość implementuje interfejs `HttpSessionBindingListener`, to wywoływana jest jej metoda `valueUnbound`. W mechanizmach obsługi serwletów zgodnych ze specyfikacją Java Servlet 2.1, należy używać metody `removeValue`. W mechanizmach zgodnych ze specyfikacją 2.2 preferowane jest użycie metody `removeAttribute`, choć w celu zapewnienia zgodności z poprzednimi wersjami oprogramowania można także używać metody `removeValue` (uznanej za przestarzałą).

public String[] getValueNames()
public Enumeration getAttributeNames()

Te metody zwracają nazwy wszystkich atrybutów w danej sesji. W mechanizmach obsługi serwletów zgodnych ze specyfikacją Java Servlet 2.1 należy używać metody `getValueNames`; w mechanizmach zgodnych ze specyfikacją 2.2 metoda ta jest wciąż dostępna lecz uznana za przestarzałą, z tego względu należy używać metody `getAttributeNames`.

public String getId()

Ta metoda zwraca unikalny identyfikator generowany dla każdej z sesji. Identyfikator ten jest czasami używany jako nazwa klucza, dotyczy to sytuacji, gdy z sesją jest skojarzona tylko jedna wartość lub gdy informacje o sesji są rejestrowane.

public boolean isNew()

Metoda zwraca wartość `true` jeśli klient (przeglądarka) jeszcze nigdy nie używała sesji (zazwyczaj dlatego, że sesja została właśnie utworzona, a odbierane żądania jeszcze się do niej nie odwoływały). Jeśli dana sesja już istnieje od jakiegoś czasu, metoda ta zwraca wartość `false`.

public long getCreationTime()

Ta metoda zwraca czas utworzenia sesji, wyrażony jako ilość milisekund jakie upłynęły od początku 1970 roku (GMT). Aby przekształcić tę wartość do postaci, którą można wydrukować, należy przekazać ją do konstruktora klasy `Date` lub posłużyć się metodą `setTimeInMilis` klasy `GregorianCalendar`.

public long getLastAccessTime()

Metoda zwraca czas ostatniego przesłania sesji z przeglądarki na serwer, czas ten jest wyrażony jako ilość milisekund jakie upłynęły od początku 1970 roku (GMT).

public int getMaxInactiveInterval()
public void setMaxInactiveInterval(int ilośćSekund)

Pierwsza z tych metod pobiera a druga określa długość okresu czasu (wyrażoną w sekundach) w którym klient musi nadesłać żądanie, aby sesja nie została automatycznie unieważniona. Jeśli w wywołaniu metody `setMaxInactiveInterval` zostanie podana wartość mniejsza od zera, to sesja nigdy nie zostanie automatycznie unieważniona. Należy pamiętać, iż limit czasu oczekiwania sesji jest przechowywany na serwerze i nie odpowiada dacie wygaśnięcia ważności cookies, która jest przesyłana do przeglądarki.

public void invalidate()

Wywołanie tej metody powoduje unieważnienie sesji i usunięcie z niej wszystkich skojarzonych z nią obiektów.

Kojarzenie informacji z sesją

Zgodnie z tym co podałem w poprzednim podrozdziale, informacje skojarzone z sesją można odczytać przy użyciu metod `getValue` (stosowanej w mechanizmach obsługi serwletów zgodnych ze specyfikacją Java Servlet 2.1) oraz `getAttribute` (stosowanej w mechanizmach obsługi serwletów zgodnych ze specyfikacją 2.2). Aby podać te informacje, w przypadku korzystania z mechanizmów obsługi serwletów zgodnych ze specyfikacją 2.1, należy posłużyć się metodą `putValue`, podając w jej wywołaniu klucz oraz wartość. W przypadku wykorzystania mechanizmów zgodnych ze specyfikacją Java Servlet 2.2, należy użyć metody `setAttribute`. Nazwa tej metody jest bardziej spójna z ogólnie stosowanym nazewnictwem, gdyż wykorzystuje notacje `set/get` stosowaną w komponentach JavaBeans. Aby wartości przechowywane w sesji mogły wywoływać jakieś skutki uboczne w chwili ich zapisywania w sesji, to używane obiekty muszą implementować interfejs `HttpSessionBindingListener`. Dzięki temu, za każdym razem gdy przy użyciu metody `putValue` lub `setAttribute`, skojarzysz jakiś obiekt z sesją, zaraz potem zostanie wywołana jego metoda `valueBound`.

Pamiętaj, że metody `putValue` oraz `setAttribute` zastępują wszelkie wartości skojarzone z podaną nazwą; jeśli chcesz usunąć tą wartość bez podawania zamiennika, powinieneś posłużyć się metodą `removeValue` (w przypadku korzystania z mechanizmów obsługi serwletów zgodnych ze specyfikacją Java Servlet 2.1) lub `removeAttribute` (w przypadku korzystania z mechanizmów obsługi serwletów zgodnych ze specyfikacją 2.2). Jeśli usuwane obiekty implementują interfejs `HttpSessionBindingListener`, to wywołanie którejkolwiek z tych metod spowoduje wywołanie metody `valueUnbound` usuwanego obiektu. Czasami będziesz chciał jedynie zastąpić istniejącą wartość atrybutu, w takim przypadku będziesz mógł użyć sposobu przedstawionego w drugim wierszu poniższego przykładu (określającego wartość atrybutu `"referringPage"`). Kiedy indziej będziesz chciał pobrać aktualną wartość atrybutu i zmodyfikować ją — patrz poniższy przykład i operacje wykonywane na atrybucie `"previousItem"`. Zakładam, że w poniższym przykładzie wykorzystywane są dwie klasy — klasa `ShoppingCart` dysponuje metodą `addItem`, a obiekty tej klasy służą do przechowywania informacji o wybranych towarach, klasa `Catalog` udostępnia statyczną metodę `getItem` zwracającą towar na podstawie podanego identyfikatora. Implementacje tych klas znajdziesz w podrozdziale 9.4, pt.: „Internetowy sklep wykorzystujący koszyki i śledzenie sesji".

```
HttpSession sesja = request.getSession(true);
sesja.putValue("referringPage", request.getHeader("Referer"));
ShoppingCart koszyk =
    (ShoppingCart) sesja.getValue("previousItem");
if (koszyk == null) { // w sesji nie ma jeszcze żadnego koszyka
  koszyk = new ShoppingCart();
  sesja.putValue("previousItem", koszyk);
}
String idTowaru = request.getParameter("itemID");
if (idTowaru != null) {
  koszyk.addItem(Catalog.getItem(idTowaru));
}
```

Zakańczanie sesji

Sesja automatycznie stanie się nieaktywna, jeśli odstęp pomiędzy kolejnymi żądaniami przekroczy czas określony przez metodę `getMaxInactiveInterval`. Gdy to nastąpi, automatycznie zostaną usunięte wszelkie skojarzenia obiektów z tą sesją. To z kolei sprawi, że obiekty implementujące interfejs `HttpSessionBindingListener` zostaną zawiadomione o fakcie ich usunięcia z sesji.

Wcale nie musisz czekać, aż zostanie przekroczony limit czasu oczekiwania sesji, możesz jawnie zażądać dezaktywacji sesji — służy do tego metoda `invalidate`.

Kodowanie adresów URL przesyłanych do przeglądarki

Jeśli przy śledzeniu sesji wykorzystywana jest metoda przepisywania adresów URL i jeśli przesyłasz do przeglądarki adresy URL odwołujące się do Twojej witryny, to będziesz musiał jawnie dodać do nich informacje o sesji. Pamiętasz zapewne, że serwlety automatycznie zaczną używać metody przepisywania adresów URL jeśli przeglądarka nie obsługuje cookies; dlatego też powinieneś zawsze kodować wszystkie adresy URL odwołujące się do Twojej witryny. Adresy URL odwołujące się do tej samej witryny można używać w dwóch miejscach. Pierwszym z nich są strony WWW generowane przez serwlety. Adresy umieszczane na takich stronach powinne być odpowiednio zakodowane przy użyciu metody `encodeURL` interfejsu `HttpServletResponse`. Metoda ta sama stwierdzi czy przepisywanie adresów jest aktualnie wykorzystywane i dopisze do adresu informacje o sesji tylko gdy będzie to konieczne. W przeciwnym przypadku metoda zwraca adresu URL w oryginalnej postaci.

Oto przykład:

```
String oryginalnyURL = jakisURL_wzglednyLubBezwzgledny;
String zakowowanyURL = response.encodeURL(oryginalnyURL);
out.println("<A HREF=\"" + zakodowanyURL + "\">...</A>);
```

Drugim miejscem, w którym można podawać adresy URL odwołujące się do tej samej witryny jest wywołanie metody `sendRedirect` (oznacza to, że adresy te zostaną umieszczone w nagłówku odpowiedzi `Location`). W tym przypadku obowiązują inne metody określania czy do adresu należy dodać informacje o sesji czy nie, a zatem nie można posłużyć się metodą `encodeURL`. Na szczęście interfejs `HttpServletResponse` udostępnia metodę `encodeRedirectURL`, której można użyć właśnie w takich sytuacjach. Oto przykład zastosowania tej metody:

```
String oryginalnyURL = jakisURL; // specyfikacja 2.2 pozwala na używanie
                                 // względnych adresów URL
String zakodowanyURL = response.encodeRedirectURL(oryginalnyURL);
response.sendRedirect(zakodowanyURL);
```

Często się zdarza, że tworząc serwlet nie jesteś w stanie przewidzieć czy w przyszłości stanie się on częścią grupy strony wykorzystujących śledzenie sesji, warto przewidzieć to zawczasu i kodować w serwlecie wszystkie adresy URL odwołujące się do tej samej witryny.

Metoda
Zaplanuj to zawczasu — wszystkie adresy URL odwołujące się do tej samej witryny przekształcaj przy użyciu metod `response.encodeURL` lub `response.encodeRedirectURL`, niezależnie od tego czy servlet wykorzystuje mechanizmy śledzenia sesji czy nie.

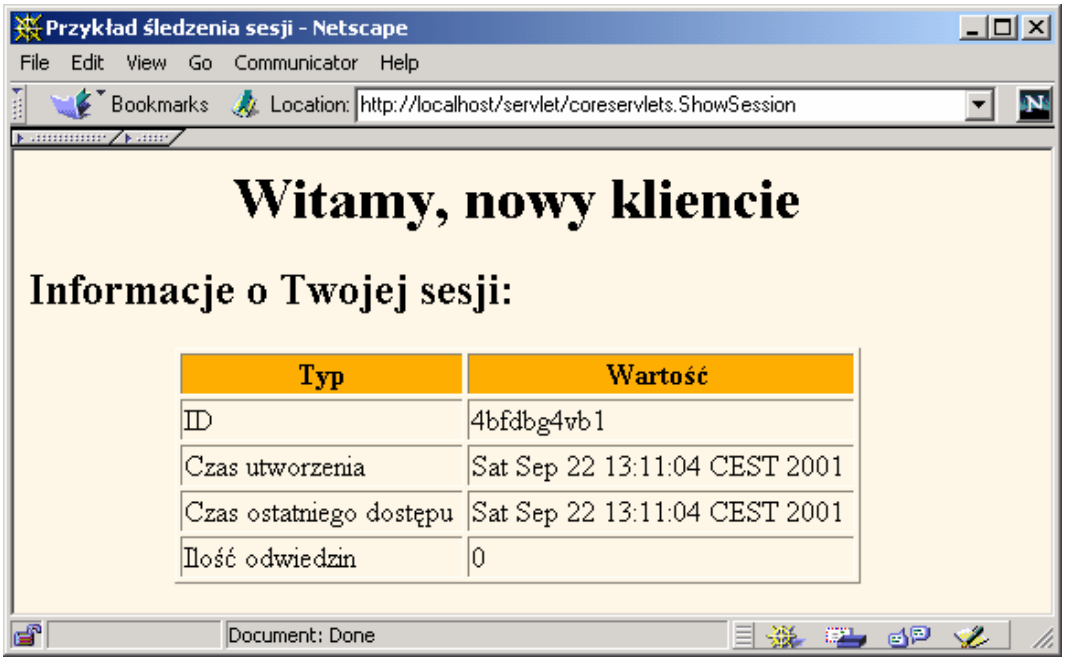
9.3 Servlet generujący indywidualny licznik odwiedzin dla każdego użytkownika

Listing 9.1 przedstawia prosty serwlet wyświetlający podstawowe informacje dotyczące sesji. Gdy przeglądarka nawiąże połączenie, serwlet pobiera istniejącą sesję lub, jeśli sesji jeszcze nie ma, to ją tworzy. Obie te czynności realizowane są przy użyciu jednego wywołania — `request.getSession(true)`. Następnie serwlet sprawdza czy w sesji został zapisany atrybut typu `Integer` o nazwie "`accessCount`". Jeśli serwlet nie odnajdzie tego atrybutu, to zakłada, że ilość wcześniej odwiedzonych strony wynosi `0`. Jeśli atrybut `accessCount` zostanie odnaleziony to serwlet pobiera jego wartość. Wartość ta jest następnie inkrementowana i ponownie kojarzona z sesją poprzez wywołanie metody `putValue`. I w reszcie, serwlet generuje niewielką tabelę HTML zawierającą informacje o sesji. Rysunek 9.1 przedstawia stronę wygenerowaną przez serwlet podczas pierwszej wizyty na witrynie, a rysunek 9.2 — po jej kilkukrotnym odświeżeniu.

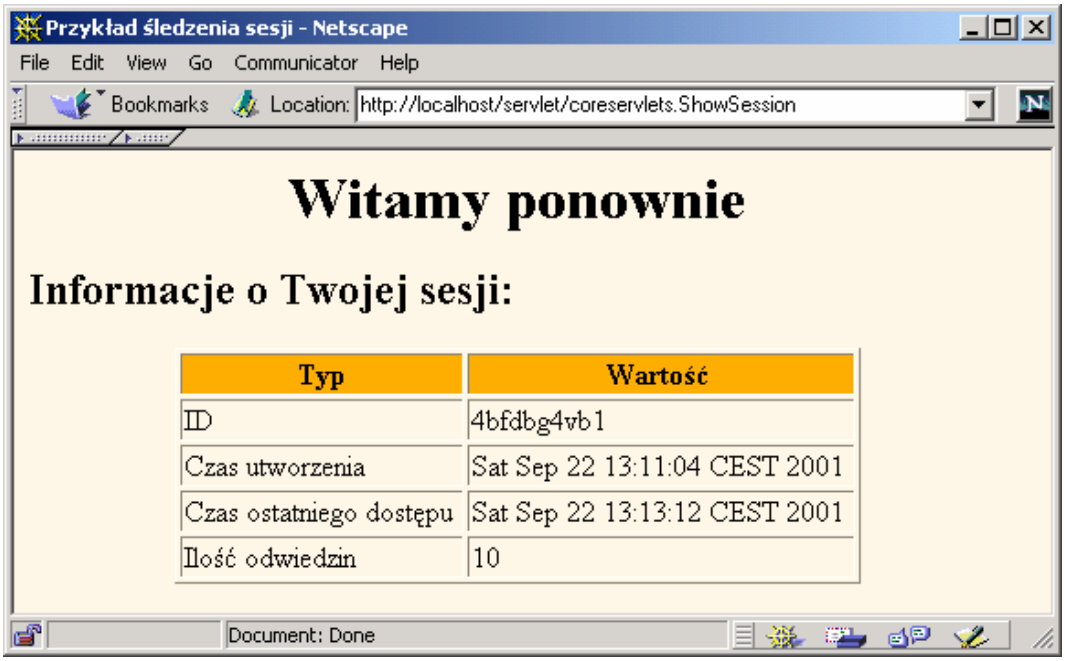
Listing 9.1 ShowSession.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;
import java.util.*;

/** Prosty przykład śledzenia sesji. Bardziej
 *  zaawansowanym przykładem jest implementacja koszyków
 *  ShoppingCart.java.
 */

public class ShowSession extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html; charset=ISO-8859-2");
    PrintWriter out = response.getWriter();
    String title = "Przykład śledzenia sesji";
    HttpSession session = request.getSession(true);
    String heading;
    /* W mechanizmach obsługi servletów zgodnych ze specyfikacją
```

```
     * Java Servlet 2.2 zamiast metody getValue należy używać
     * metody getAttribute.
     */
    Integer accessCount =
      (Integer)session.getValue("accessCount");
    if (accessCount == null) {
      accessCount = new Integer(0);
      heading = "Witamy, nowy kliencie";
    } else {
      heading = "Witamy ponownie";
      accessCount = new Integer(accessCount.intValue() + 1);
    }
    /* W mechanizmach obsługi servletów zgodnych ze specyfikacją
     * Java Servlet 2.2 zamiast metody putValue należy używać
     * metody setAttribute.
     */
    session.putValue("accessCount", accessCount);

    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=\"CENTER\">" + heading + "</H1>\n" +
                "<H2>Informacje o Twojej sesji:</H2>\n" +
                "<TABLE BORDER=1 ALIGN=\"CENTER\">\n" +
                "<TR BGCOLOR=\"#FFAD00\">\n" +
                "  <TH>Typ<TH>Wartość\n" +
                "<TR>\n" +
                "  <TD>ID\n" +
                "  <TD>" + session.getId() + "\n" +
                "<TR>\n" +
                "  <TD>Czas utworzenia\n" +
                "  <TD>" +
                new Date(session.getCreationTime()) + "\n" +
                "<TR>\n" +
                "  <TD>Czas ostatniego dostępu\n" +
                "  <TD>" +
                new Date(session.getLastAccessedTime()) + "\n" +
                "<TR>\n" +
                "  <TD>Ilość odwiedzin\n" +
                "  <TD>" + accessCount + "\n" +
                "</TABLE>\n" +
                "</BODY></HTML>");

  }

  /** Żądania GET i POST mają być obsługiwane jednakowo. */

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```



Rysunek 9.1 Strona wygenerowana po pierwszym wykonaniu serwletu ShowSession



Rysunek 9.2 Strona wygenerowana po jedenastym wykonaniu serwletu ShowSession

9.4 Internetowy sklep wykorzystujący koszyki i śledzenie sesji

W tym podrozdziale przedstawiłem rozbudowany przykład pokazujący w jaki sposób można stworzyć internetowy sklep wykorzystujący mechanizmy śledzenia sesji. W pierwszej części podrozdziału pokażę jak można stworzyć strony wyświetlające informacje o sprzedawanych towarach. Kod każdej ze stron prezentujących sprzedawane towary zawiera wyłącznie tytuł strony oraz identyfikatory towarów jakie maja się na niej pojawić. Sam kod HTML każdej z tych stron jest generowany automatycznie przez metody klasy bazowej, na podstawie opisów towarów

przechowywanych w katalogu. Druga część podrozdziału przedstawia stronę obsługującą zamawianie towarów. Strona ta kojarzy każdego z użytkowników z koszykiem i pozwala użytkownikowi na zmodyfikowanie ilości każdego z zamówionych towarów. Do skojarzenia użytkowników z koszykami, strona ta używa mechanizmów śledzenia sesji. W końcu, w trzeciej części podrozdziału przedstawiłem implementację koszyka — struktury danych reprezentującej poszczególne towary oraz zamówienia — oraz katalogu towarów.

Tworzenie interfejsu użytkownika

Listing 9.2 przedstawia abstrakcyjną klasę bazową używaną przy tworzeniu serwletów, które mają prezentować sprzedawane towary. Serwlet ten pobiera identyfikatory towarów, odszukuje je w katalogu, a następnie pobiera z niego nazwy i ceny towarów, i wyświetla na stronie umożliwiającej zamawianie. Listingi 9.3 oraz 9.4 pokazują jak łatwo można stworzyć strony z informacji o towarach, posługując się klasą bazową z listingu 9.2. Wygląd stron zdefiniowanych na listingach 9.3 oraz 9.4 został przedstawiony na rysunkach 9.3 oraz 9.4.

Listing 9.2 CatalogPage.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

/** Klasa bazowa stron prezentujących towary z katalogu.
 *  Servlety dziedziczące od tej klasy bazowej muszą
 *  określić wyświetlane elementy katalogu oraz tytuł
 *  strony <i>zanim</i> servlet zostanie wykonany.
 *  W tym celu w metodzie init takiego servletu należy
 *  wywołać metody setItems oraz setTitle klasy bazowej.
 */

public abstract class CatalogPage extends HttpServlet {
  private Item[] items;
  private String[] itemIDs;
  private String title;

  /** Dysponując tablicą identyfikatorów towarów, odszukaj
   *  je w katalogu (Catalog) i zapisz odpowiadające im
   *  obiekty Item do tablicy items. Obiekty Item zawierają
   *  krótki opis, pełny opis oraz cenę towaru, a ich
   *  unikalnym kluczem jest identyfikator towaru.
   *  <P>
   *  Servlety dziedziczące po klasie CatalogPage
   *  <b>muszą</b> wywoływać tę metodę (zazwyczaj
   *  w metodzie init) zanim servlet zostanie wywołany.
   */

  protected void setItems(String[] itemIDs) {
    this.itemIDs = itemIDs;
    items = new Item[itemIDs.length];
    for(int i=0; i<items.length; i++) {
      items[i] = Catalog.getItem(itemIDs[i]);
    }
  }

  /** Określa tytuł strony, który jest wyświetlany na
   *  stronie wynikowej w nagłówku <H1>.
   *  <P>
   *  Servlety dziedziczące po klasie CatalogPage
   *  <b>muszą</b> wywoływać tę metodę (zazwyczaj
   *  w metodzie init) zanim servlet zostanie wywołany.
   */

  protected void setTitle(String title) {
    this.title = title;
  }
```

```
  /** Metoda w pierwszej kolejności wyświetla tytuł, a następnie
   *  dla każdego towaru z katalogu, który ma być przedstawiony
   *  na danej stronie, wyświetla jego krótki opis (w nagłówku
   *  <H2>), cenę (w nawiasach) oraz pełny opis poniżej.
   *  Poniżej każdego towaru wyświetlany jest przycisk umożliwiający
   *  złożenie zamówienia dotyczącego danego towaru - informacje
   *  przesyłane są do servletu OrderPage.
   *  <P>
   *  Aby zobaczyć kod HTML generowany przez tę metodę, należy
   *  wykonać servlet KidsBooksPage lub TechBooksPage (obie
   *  te klasy dziedziczą po abstrakcyjnej klasie CatalogPage)
   *  i wybrać opcję "Wyświetl kod źródłowy" (lub jej ekwiwalent)
   *  w przeglądarce.
   */

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html; charset=ISO-8859-2");
    if (items == null) {
      response.sendError(response.SC_NOT_FOUND,
                         "Brak towarów.");
      return;
    }
    PrintWriter out = response.getWriter();
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=\"CENTER\">" + title + "</H1>");
    Item item;
    for(int i=0; i<items.length; i++) {
      out.println("<HR>");
      item = items[i];
      // Wyświetl informacje o błędzie jeśli klasa potomna
      // podała identyfikator towaru którego nie ma w katalogu
      if (item == null) {
        out.println("<FONT COLOR=\"RED\">" +
                    "Nieznany identyfikator towaru " + itemIDs[i] +
                    "</FONT>");
      } else {
        out.println();
        String formURL =
          "/servlet/coreservlets.OrderPage";
        // Adresy URL odwołujące się do tej samej witryny należy
        // przekształcić przy użyciu metody encodeURL.
        formURL = response.encodeURL(formURL);
        out.println
          ("<FORM ACTION=\"" + formURL + "\">\n" +
           "<INPUT TYPE=\"HIDDEN\" NAME=\"itemID\" " +
           "       VALUE=\"" + item.getItemID() + "\">\n" +
           "<H2>" + item.getShortDescription() +
           " ($" + item.getCost() + ")</H2>\n" +
           item.getLongDescription() + "\n" +
           "<P>\n<CENTER>\n" +
           "<INPUT TYPE=\"SUBMIT\" " +
           "VALUE=\"Dodaj do koszyka\">\n" +
           "</CENTER>\n<P>\n</FORM>");
}
    }
    out.println("<HR>\n</BODY></HTML>");
  }

  /** Żądania POST i GET mają być obsługiwane tak samo. */

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

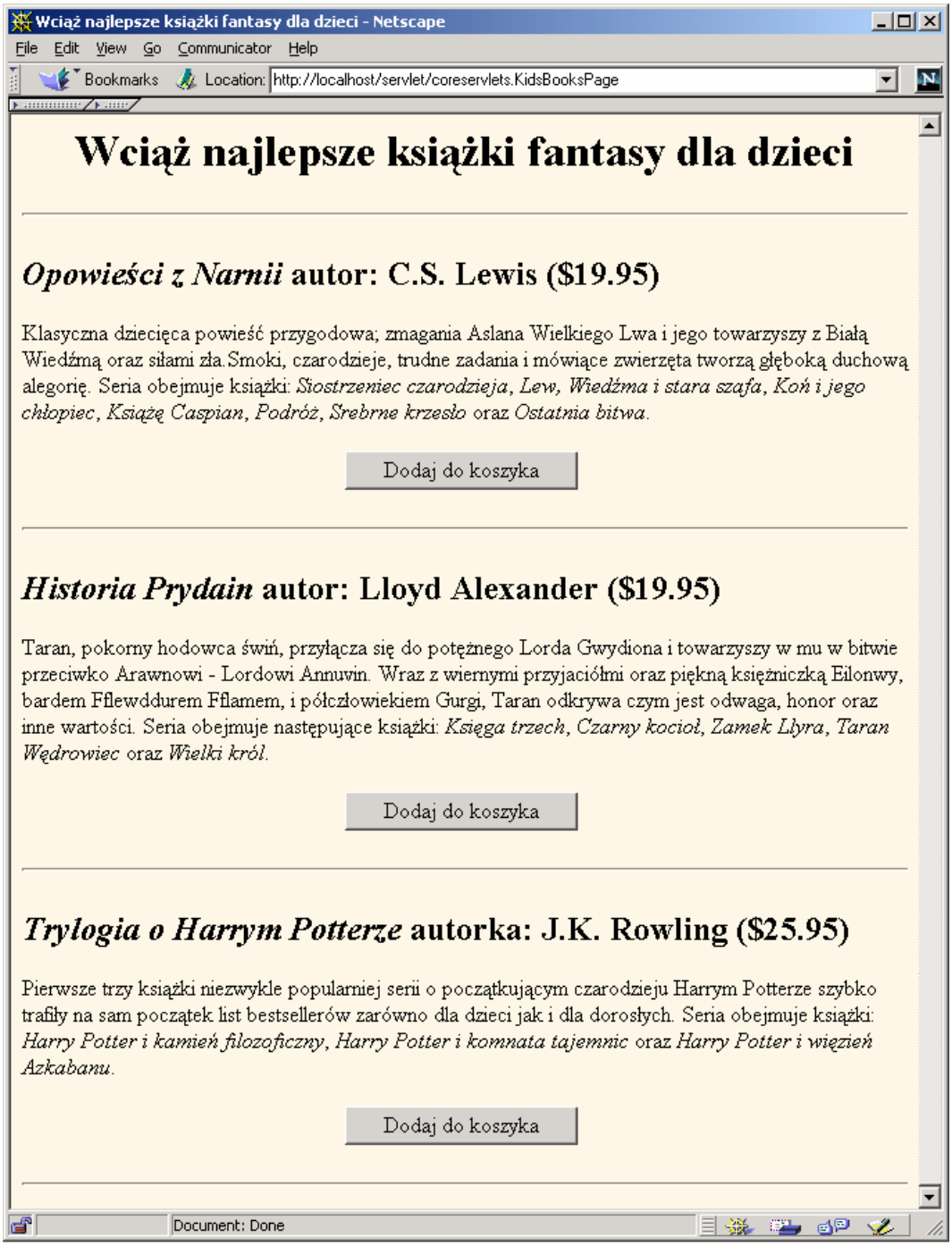
Listing 9.3 KidsBooksPage.java

```
package coreservlets;

/** Klasa potomna servletu CatalogPage wyświetlająca
 *  stronę WWW umożliwiającą zamówienie trzech znanych
 *  serii książek dla dzieci.
```

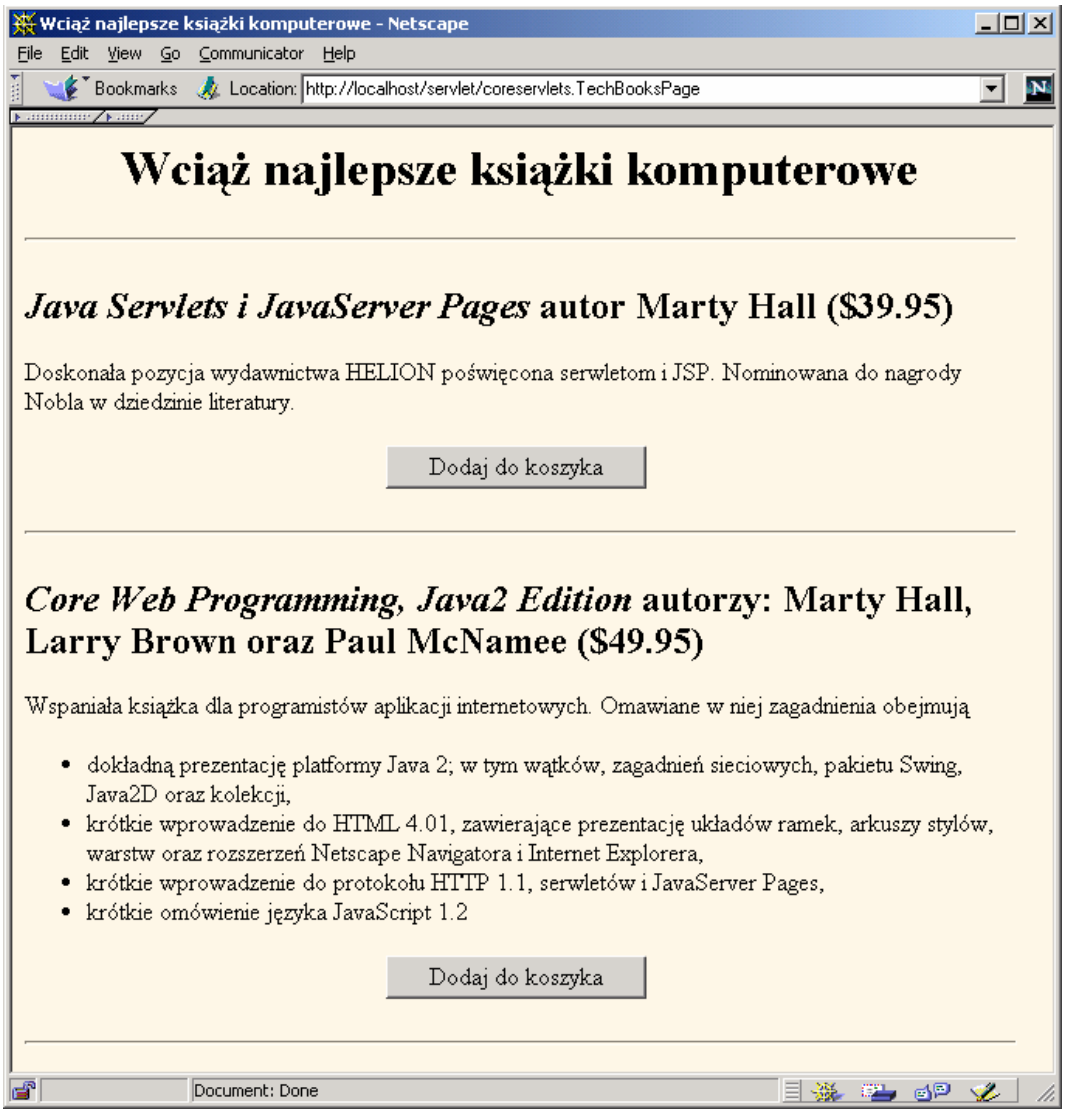
```
 *  Zamówienia są przesyłane do servletu OrderPage.
 */

public class KidsBooksPage extends CatalogPage {
  public void init() {
    String[] ids = { "lewis001", "alexander001", "rowling001" };
    setItems(ids);
    setTitle("Wciąż najlepsze książki fantasy dla dzieci");
  }
}
```

Listing 9.4 TechBooksPage.java

```
package coreservlets;

/** Klasa potomna servletu CatalogPage wyświetlająca
 *  stronę WWW umożliwiającą zamówienie dwóch
 *  doskonałych książek komputerowych.
 *  Zamówienia są przesyłane do servletu OrderPage.
 */

public class TechBooksPage extends CatalogPage {
  public void init() {
    String[] ids = { "hall001", "hall002" };
    setItems(ids);
    setTitle("Wciąż najlepsze książki komputerowe");
  }
}
```



Rysunek 9.3 Wyniki wykonania serwletu KidsBooksPage



Rysunek 9.4 Wyniki wykonania serwletu TechBooksPage

Obsługa zamówień

Listing 9.5 przedstawia kod serwletu służącego do obsługi zamówień nadsyłanych przez różne strony katalogowe (przedstawione w poprzedniej części podrozdziału). Serwlet ten kojarzy każdego użytkownika z koszykiem, wykorzystując przy tym mechanizmy śledzenia sesji. Ponieważ każdy użytkownik dysponuje osobnym koszykiem, jest mało prawdopodobne, że wiele wątków będzie jednocześnie próbowało uzyskać dostęp do tego samego koszyka. Niemniej jednak, gdybyś popadł w paranoję, mógłbyś wyobrazić sobie kilka sytuacji, w których mógłby nastąpić jednoczesny dostęp do tego samego koszyka. Na przykład, gdyby ten sam użytkownik miał jednocześnie otworzonych kilka okien przeglądarki i niemal w tym samym czasie wysyłał zamówienia bądź aktualizacje z kilku różnych okien. A zatem, aby zapewnić w miarę wysoki poziom bezpieczeństwa, kod naszego serwletu będzie synchronizował dostęp do koszyków na podstawie obiektu sesji. W ten sposób inne wątki korzystające z tej samej sesji nie będą mogły równocześnie uzyskać dostępu do przechowywanych w niej informacji, choć wciąż będzie możliwa równoczesna obsługa żądań nadsyłanych przez różnych użytkowników. Typowe wyniki wykonania tego serwletu zostały przedstawione na rysunkach 9.5 oraz 9.6.

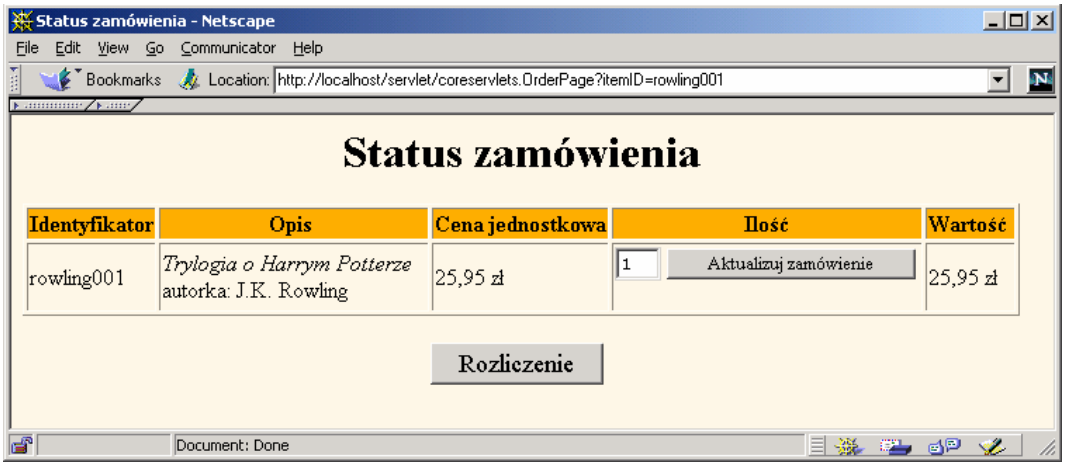
Listing 9.5 OrderPage.java

```
package coreservlets;
```

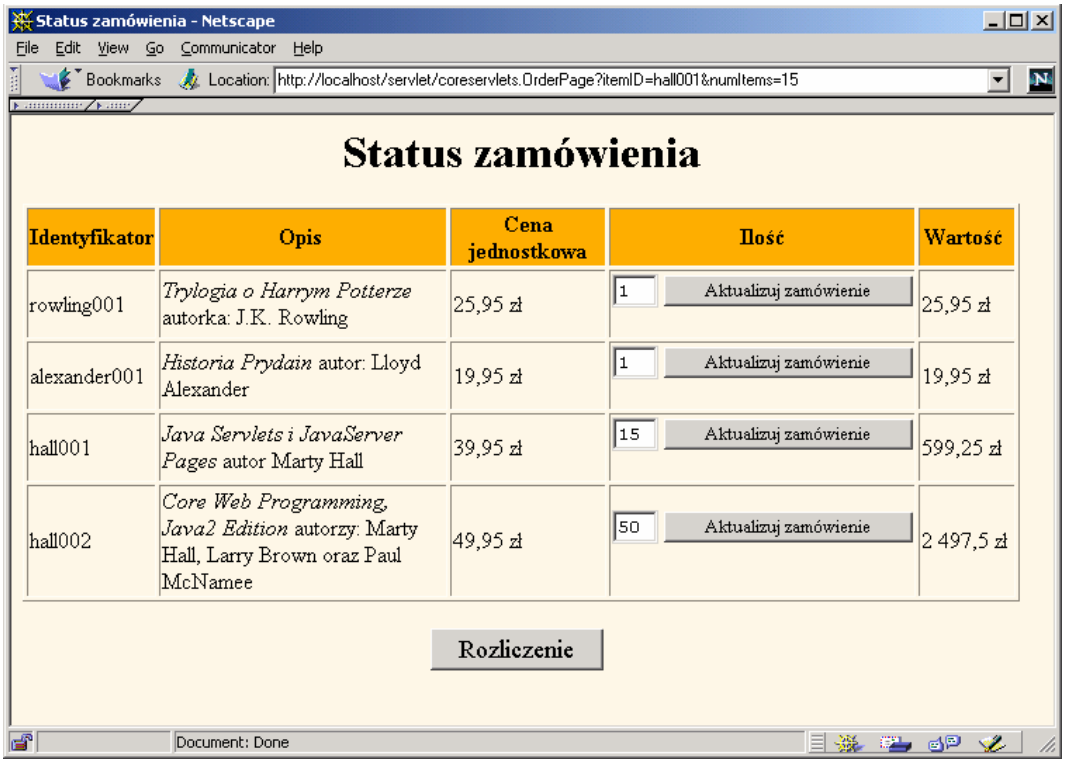
```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import java.text.NumberFormat;

/** Wyświetla informacje o towarach, które aktualnie
 *  znajdują się w koszyku (ShoppingCart). Przeglądarki
 *  mają swoje własne sesje, na podstawie których
 *  określane jest przynależność koszyków. Jeśli to
 *  jest pierwsza wizyta na stronie umożliwiającej
 *  składnie zamówień, to tworzony jest nowy koszyk.
 *  Użytkownicy zazwyczaj trafiają na tę stronę ze stron
 *  prezentujących towary, które można zamawiać, dlatego
 *  też ta strona dodaje nowy element do koszyka. Jednak
 *  użytkownicy mogą zapamiętać adres tej strony i wyświetlać
 *  ją posługując się listą ulubionych stron; mogą także
 *  wrócić na nią klikając przycisk "Aktualizuj zamówienie"
 *  po zmienieniu ilości egzemplarzy jednego z zamawianych
 *  towarów.
 */

public class OrderPage extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    HttpSession session = request.getSession(true);
    ShoppingCart cart;
    synchronized(session) {
      cart = (ShoppingCart)session.getValue("shoppingCart");
      // Dla nowych użytkowników tworzone są nowe koszyki.
      // Użytkownicy, którzy już dysponują koszykami
      // używają tych, które zostały dla nich wcześniej utworzone .
      if (cart == null) {
        cart = new ShoppingCart();
        session.putValue("shoppingCart", cart);
      }
      String itemID = request.getParameter("itemID");
      if (itemID != null) {
        String numItemsString =
          request.getParameter("numItems");
        if (numItemsString == null) {
          // Jeśli w żądaniu został podany identyfikator (ID) lecz
          // nie liczba, to oznacza to, że użytkownik trafił
          // tutaj klikając przycisk "Dodaj do koszyka" na jednej
          // ze stron prezentującej towary z katalogu.
          cart.addItem(itemID);
        } else {
          // Jeśli w żądaniu został podany zarówno identyfikator
          // (ID) jak i liczba, to oznacza to, że użytkownik
          // trafił na stronę klikając przycisk "Aktualizuj
          // zamówienie" po zmianie ilości egzemplarzy jednego
          // z zamawianych towarów. Zwróć uwagę, iż podanie wartości
          // 0 jako liczby egzemplarzy zamawianego towaru sprawi, że
          // dany towar zostanie usunięty z koszyka.
          int numItems;
          try {
            numItems = Integer.parseInt(numItemsString);
          } catch(NumberFormatException nfe) {
            numItems = 1;
          }
          cart.setNumOrdered(itemID, numItems);
        }
      }
    }
    // Pokaż status zamówienia niezależnie od tego czy użytkownik
    // je zmodyfikował czy nie.
    response.setContentType("text/html; charset=ISO-8859-2");
    PrintWriter out = response.getWriter();
    String title = "Status zamówienia";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=\"CENTER\">" + title + "</H1>");
    synchronized(session) {
      Vector itemsOrdered = cart.getItemsOrdered();
      if (itemsOrdered.size() == 0) {
```

```
      out.println("<H2><I>Brak towarów w koszyku...</I></H2>");
    } else {
      // Jeśli w koszyku jest co najmniej jeden towar, to
      // wyświetl tabelę z informacjami o nim.
      out.println
        ("<TABLE BORDER=1 ALIGN=\"CENTER\">\n" +
         "<TR BGCOLOR=\"#FFAD00\">\n" +
         "  <TH>Identyfikator<TH>Opis\n" +
         "  <TH>Cena jednostkowa<TH>Ilość<TH>Wartość");
      ItemOrder order;

      // Zaokrąglamy do dwóch miejsc po przecinku,
      // wstawiamy znak dolara (lub innej waluty), itd.
      // wszystko zgodnie z bieżącymi ustawieniami lokalnymi.
      NumberFormat formatter =
        NumberFormat.getCurrencyInstance();

      String formURL =
        "/servlet/coreservlets.OrderPage";
      // Adresy URL odwołujące się do stron tej samej witryny
      // przekształcamy przy użyciu metody encodeURL.
      formURL = response.encodeURL(formURL);

      // Dla każdego towaru umieszczonego w koszyku
      // tworzymy wiersz tabeli zawierający identyfikator
      // towaru (ID), opis, jego cenę jednostkową,
      // ilość zamówionych egzemplarzy oraz łączną cenę.
      // Ilość zamawianych egzemplarzy wyświetlamy w
      // polu tekstowym, tak aby użytkownik mógł ją zmienić.
      // Dodatkowo, obok pola, wyświetlamy przycisk
      // "Aktualizuj zamówienie", który powoduje ponowne
      // przesłanie tej samej strony na serwer, przy czym
      // zmieniana jest ilość zamawianych egzemplarzy
      // danego towaru.
      for(int i=0; i<itemsOrdered.size(); i++) {
        order = (ItemOrder)itemsOrdered.elementAt(i);
        out.println
          ("<TR>\n" +
           "  <TD>" + order.getItemID() + "\n" +
           "  <TD>" + order.getShortDescription() + "\n" +
           "  <TD>" +
           formatter.format(order.getUnitCost()) + "\n" +
           "  <TD>" +
           "<FORM ACTION=\"" + formURL + "\">\n" +
           "<INPUT TYPE=\"HIDDEN\" NAME=\"itemID\"\n" +
           "       VALUE=\"" + order.getItemID() + "\">\n" +
           "<INPUT TYPE=\"TEXT\" NAME=\"numItems\"\n" +
           "       SIZE=3 VALUE=\"" +
           order.getNumItems() + "\">\n" +
           "<SMALL>\n" +
           "<INPUT TYPE=\"SUBMIT\"\n "+
           "       VALUE=\"Aktualizuj zamówienie\">\n" +
           "</SMALL>\n" +
           "</FORM>\n" +
           "  <TD>" +
           formatter.format(order.getTotalCost()));
      }
      String checkoutURL =
        response.encodeURL("/Checkout.html");
      // Pod tabelą wyświetlany jest przycisk "Rozliczenie"
      out.println
        ("</TABLE>\n" +
         "<FORM ACTION=\"" + checkoutURL + "\">\n" +
         "<BIG><CENTER>\n" +
         "<INPUT TYPE=\"SUBMIT\"\n" +
         "       VALUE=\"Rozliczenie\">\n" +
         "</CENTER></BIG></FORM>");
    }
    out.println("</BODY></HTML>");
  }
}

/** Żądania GET i POST są obsługiwane w identyczny sposób */

public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException {
```

```
    doGet(request, response);
  }
}
```



Rysunek 9.5 Wyniki wygenerowane przez serwlet OrderPage po kliknięciu przycisku Dodaj do koszyka na stronie KidsBooksPage



Rysunek 9.6 Wyniki wygenerowane przez serwlet OrderPage po zamówieniu kilku dodatkowych książek i wprowadzeniu kilku zmian w zamówieniu

To czego nie widać: Implementacja koszyka i katalogu towarów

Listing 9.6 przedstawia implementację koszyka. Obiekt koszyka — `ShoppingCart` — zawiera obiekt `Vector` przechowujący informacje o zamówionych towarach i dysponuje metodami umożliwiającymi dodawanie i aktualizację zamówienia. Na listingu 9.7 został przedstawiony kod obiektu reprezentującego element katalogu towarów. Klasa przedstawiona na listingu 9.8

reprezentuje status zamówienia konkretnego towaru. I w końcu listing 9.9 przedstawia implementację katalogu towarów.

Listing 9.6 ShoppingCart.java

```
package coreservlets;

import java.util.*;

/** Klasa implementująca koszyk - jest to struktura danych
 *  służąca do przechowywania informacji o zamawianych
 *  towarach.
 *  Servlet OrderPage kojarzy jeden z tych koszyków z
 *  każdą sesją.
 */

public class ShoppingCart {
  private Vector itemsOrdered;

  /** Tworz pusty koszyk */

  public ShoppingCart() {
    itemsOrdered = new Vector();
  }

  /** Zwraca Vector obiektów ItemOrder zawierających
   *  informacje o zamówionych towarach i ich ilości.
   */

  public Vector getItemsOrdered() {
    return(itemsOrdered);
  }

  /** Przegląda koszyk i sprawdza czy znajduje się już w
   *  nim zamówienie dotyczące towary o podanym
   *  identyfikatorze. Jeśli takie zamówienie zostanie odnalezione
   *  to ilość egzemplarzy danego towaru jest inkrementowana.
   *  Jeśli nie ma zamówienie dotyczącego podanego towaru,
   *  to metoda pobiera z katalogu (Catalog) informacje o nim
   *  i dodaje do koszyka odpowiedni obiekt.
   */

  public synchronized void addItem(String itemID) {
    ItemOrder order;
    for(int i=0; i<itemsOrdered.size(); i++) {
      order = (ItemOrder)itemsOrdered.elementAt(i);
      if (order.getItemID().equals(itemID)) {
        order.incrementNumItems();
        return;
      }
    }
    ItemOrder newOrder = new ItemOrder(Catalog.getItem(itemID));
    itemsOrdered.addElement(newOrder);
  }

  /** Przegląda koszyk w poszukiwaniu wpisu dotyczącego
   *  towaru o podany identyfikatorze. Jeśli podana liczba
   *  jest większa od zera to zostaje ona użyta do określenia
   *  ilości zamówionych egzemplarzy danego towaru. Jeśli
   *  przekazana liczba ma wartość 0 (lub mniejszą od zera,
   *  co może nastąpić w przypadku błędu użytkownika) to
   *  wpis reprezentujący dany towar jest usuwany z koszyka.
   */

  public synchronized void setNumOrdered(String itemID,
                                         int numOrdered) {
    ItemOrder order;
    for(int i=0; i<itemsOrdered.size(); i++) {
      order = (ItemOrder)itemsOrdered.elementAt(i);
      if (order.getItemID().equals(itemID)) {
        if (numOrdered <= 0) {
          itemsOrdered.removeElementAt(i);
        } else {
          order.setNumItems(numOrdered);
        }
        return;
```

```
      }
    }
    ItemOrder newOrder =
      new ItemOrder(Catalog.getItem(itemID));
    itemsOrdered.addElement(newOrder);
  }
}
```

Listing 9.7 Item.java

```
package coreservlets;

/** Opisuje element katalogu dla internetowego sklepu.
 *  identyfikator (itemID) unikalnie identyfikuje każdy
 *  element, krótki opis (shortDescription) zawiera
 *  krótkie informacje o towarze (takie jak nazwę
 *  autora i tytuł książki), a długi opis (longDescription)
 *  to kilka zdań dokładniej opisujących dany towar; w końcu
 *  cena (cost) to jednostkowa cena towaru.
 *  Zarówno krótki jak i długi opis może zawierać kod HTML.
 */

public class Item {
  private String itemID;
  private String shortDescription;
  private String longDescription;
  private double cost;

  public Item(String itemID, String shortDescription,
              String longDescription, double cost) {
    setItemID(itemID);
    setShortDescription(shortDescription);
    setLongDescription(longDescription);
    setCost(cost);
  }

  public String getItemID() {
    return(itemID);
  }

  protected void setItemID(String itemID) {
    this.itemID = itemID;
  }

  public String getShortDescription() {
    return(shortDescription);
  }

  protected void setShortDescription(String shortDescription) {
    this.shortDescription = shortDescription;
  }

  public String getLongDescription() {
    return(longDescription);
  }

  protected void setLongDescription(String longDescription) {
    this.longDescription = longDescription;
  }

  public double getCost() {
    return(cost);
  }

  protected void setCost(double cost) {
    this.cost = cost;
  }
}
```

Listing 9.8 ItemOrder.java

```
package coreservlets;

/** Kojarzy element katalogu (Item) z konkretnym zamówieniem
 *  poprzez zapamiętanie informacji i ilości zamawianych
```

```
 *  egzemplarzy danego towaru oraz ich łącznej wartości.
 *  Udostępnia także przydatne metody umożliwiające
 *  operowanie na informacjach przechowywanych w obiekcie Item
 *  bez konieczności jego pobierania.
 */

public class ItemOrder {
  private Item item;
  private int numItems;

  public ItemOrder(Item item) {
    setItem(item);
    setNumItems(1);
  }

  public Item getItem() {
    return(item);
  }

  protected void setItem(Item item) {
    this.item = item;
  }

  public String getItemID() {
    return(getItem().getItemID());
  }

  public String getShortDescription() {
    return(getItem().getShortDescription());
  }

  public String getLongDescription() {
    return(getItem().getLongDescription());
  }

  public double getUnitCost() {
    return(getItem().getCost());
  }

  public int getNumItems() {
    return(numItems);
  }

  public void setNumItems(int n) {
    this.numItems = n;
  }

  public void incrementNumItems() {
    setNumItems(getNumItems() + 1);
  }

  public void cancelOrder() {
    setNumItems(0);
  }

  public double getTotalCost() {
    return(getNumItems() * getUnitCost());
  }
}
```

Listing 9.9 Catalog.java

```
package coreservlets;

/** Katalog zawierający informacje o towarach
 *  dostępnych w internetowym sklepie.
 */

public class Catalog {
  // Normalnie te informacje byłyby przechowywane i pobierane
  // z bazy danych.
  private static Item[] items =
    { new Item("hall001",
               "<I>Java Servlets i JavaServer Pages</I> " +
                 " autor Marty Hall",
               "Doskonała pozycja wydawnictwa HELION poświęcona " +
                 "serwletom i JSP.\n" +
```

```
            "Nominowana do nagrody Nobla w dziedzinie literatury.",
          39.95),
    new Item("hall002",
          "<I>Core Web Programming, Java2 Edition</I> " +
            " autorzy: Marty Hall, Larry Brown oraz " +
            "Paul McNamee",
          "Wspaniała książka dla programistów aplikacji " +
            "internetowych. Omawiane w niej zagadnienia obejmują \n" +
            "<UL><LI>dokładną prezentację platformy Java 2; " +
            "w tym wątków, zagadnień sieciowych, pakietu Swing, \n" +
            "Java2D oraz kolekcji,\n" +
            "<LI>krótkie wprowadzenie do HTML 4.01, " +
            "zawierające prezentację układów ramek, arkuszy stylów, \n" +
            "warstw oraz rozszerzeń Netscape Navigatora i " +
            "Internet Explorera,\n" +
            "<LI>krótkie wprowadzenie do protokołu HTTP 1.1, " +
            "serwletów i JavaServer Pages,\n" +
            "<LI>krótkie omówienie języka JavaScript 1.2\n" +
            "</UL>",
          49.95),
    new Item("lewis001",
          "<I>Opowieści z Narnii</I> autor: C.S. Lewis",
          "Klasyczna dziecięca powieść przygodowa; zmagania " +
            "Aslana Wielkiego Lwa i jego towarzyszy\n" +
            "z Białą Wiedźmą oraz siłami zła." +
            "Smoki, czarodzieje, trudne zadania \n" +
            "i mówiące zwierzęta tworzą głęboką duchową " +
            "alegorię. Seria obejmuje książki\n" +
            "<I>Siostrzeniec czarodzieja</I>,\n" +
            "<I>Lew, Wiedźma i stara szafa</I>,\n" +
            "<I>Koń i jego chłopiec</I>,\n" +
            "<I>Książę Caspian</I>,\n" +
            "<I>Podróż</I>,\n" +
            "<I>Srebrne krzesło</I> oraz \n" +
            "<I>Ostatnia bitwa</I>.",
          19.95),
    new Item("alexander001",
          "<I>Historia Prydain</I> autor: Lloyd Alexander",
          "Taran, pokorny hodowca świń, przyłącza się do " +
            "potężnego Lorda Gwydiona i towarzyszy w mu w \n" +
            "bitwie przeciwko Arawnowi - Lordowi Annuvin. Wraz " +
            "z wiernymi przyjaciółmi oraz piękną księżniczką \n" +
            "Eilonwy, bardem Fflewddurem Fflamem, " +
            "i półczłowiekiem Gurgi, Taran odkrywa " +
            "czym jest odwaga, honor oraz inne wartości.\n" +
            "Seria obejmuje następujące książki: \n" +
            "<I>Księga trzech</I>, \n" +
            "<I>Czarny kocioł</I>, \n" +
            "<I>Zamek Llyra</I>, \n" +
            "<I>Taran Wędrowiec</I> oraz \n" +
            "<I>Wielki król</I>.",
          19.95),
    new Item("rowling001",
          "<I>Trylogia o Harrym Potterze</I> autorka: " +
            "J.K. Rowling",
          "Pierwsze trzy książki niezwykle popularniej serii o " +
            "początkującym czarodzieju Harrym Potterze \n" +
            "szybko trafiły na sam początek list bestsellerów " +
            "zarówno dla dzieci jak i dla dorosłych. Seria \n" +
            "obejmuje książki: \n" +
            "<I>Harry Potter i kamień " +
              "filozoficzny</I>, \n" +
            "<I>Harry Potter i komnata " +
              "tajemnic</I> oraz \n" +
            "<I>Harry Potter i " +
              "więzień Azkabanu</I>.",
          25.95)
  };

public static Item getItem(String itemID) {
  Item item;
  if (itemID == null) {
    return(null);
  }
  for(int i=0; i<items.length; i++) {
    item = items[i];
    if (itemID.equals(item.getItemID())) {
```

```
            return(item);
          }
        }
        return(null);
      }
    }
```

Rozdział 10.
Elementy skryptowe JSP

Technologia Java Server Pages (w skrócie: JSP) pozwala na mieszanie zwyczajnego, statycznego kodu HTML oraz informacji generowanych dynamicznie przez serwlety. W pierwszej kolejności tworzy się normalny dokument HTML korzystając z klasycznych narzędzi do tego przeznaczonych; a potem dodaje się kod mający dynamicznie generować zawartość strony, zapisując go pomiędzy specjalnymi znacznikami — najczęściej `<%` oraz `%>`. Poniższy przykład przedstawia fragment strony JSP, która po odwołaniu się do adresu URL o postaci *http://host/OrderConfirmation.jsp?title=Java+Servlet+i+Java+Server+Pages*, spowoduje wyświetlenie tekstu „Dziękujemy za zamówienie książki *Java Server i Java Server Pages*”:

```
Dziękujemy za zamówienie książki <I><%= request.getParameter("title") %></I>
```

Rozdzielenie statycznego kodu HTML oraz informacji generowanych dynamicznie ma kilka zalet w porównaniu z wykorzystaniem samych serwletów, a konkretne rozwiązania stosowane w technologii Java Server Pages mają kilka zalet w porównaniu z konkurencyjnymi technologiami, takimi jak ASP, PHP czy też ColdFusion. Zalety te opisałem szczegółowo w podrozdziale 1.4. — „Zalety JSP”. Jednak, najprościej rzecz biorąc są one związane z tym, iż technologia JSP jest szeroko obsługiwana i jej wykorzystanie nie wiąże się z koniecznością używania żadnego konkretnego systemu operacyjnego ani serwera WWW. Poza tym, technologia ta daje pełny dostęp do wszystkich możliwości serwletów oraz języka Java i nie zmusza do nauki nowego, wyspecjalizowanego języka programowania o mniejszych możliwościach.

Proces udostępniania stron JSP na Internecie jest znacznie prostszy od udostępniania serwletów. Zakładając, że dysponujesz serwerem WWW obsługującym technologię JSP, wystarczy zapisać dokument z rozszerzeniem *.jsp* i umieścić go w dowolnym miejscu, gdzie możesz umieszczać zwyczajne dokumenty HTML. Nie jest konieczna żadna kompilacja, stosowanie jakichkolwiek pakietów, ani określanie wartości zmiennej środowiskowej `CLASSPATH`. Niemniej jednak, choć *środowisko* użytkownika nie wymaga określania jakichkolwiek specjalnych ustawień, to *serwer* będzie musiał być skonfigurowany w taki sposób, aby mieć dostęp do plików klasowych serwletów i JSP oraz do kompilatora Javy. Szczegółowe informacje na ten temat znajdziesz w podrozdziale 1.5. — „Instalacja i konfiguracja”.

Choć kod który piszesz tworząc stronę JSP bardziej przypomina zwyczajny kod HTML niż serwlet, to jednak dokumenty JSP są automatycznie — i w sposób niewidoczny dla programisty — przekształcane do postaci serwletów, a statyczny kod HTML jest po prostu przekazywany do strumienia wyjściowego skojarzonego z metodą `service` serwletu. Ta translacja jest zazwyczaj wykonywana w momencie odebrania pierwszego żądania dotyczącego danej strony. Aby pierwszy użytkownik, który będzie chciał wykonać stronę nie musiał czekać na jej przekształcenie i skompilowanie, programista może samemu wyświetlić tę stronę, tuż po jej zainstalowaniu na serwerze. Wiele serwerów pozwala także na tworzenie nazw umownych (ang.: *alias*), dzięki którym adres URL pozornie wskazujący na dokument HTML, w rzeczywistości odwołuje się do serwletu lub strony JSP.

W zależności od konfiguracji serwera istnieje także możliwość wyświetlenia kodu źródłowego serwletu wygenerowanego na podstawie strony JSP. W przypadku serwera Tomcat 3.0,

należy przypisać atrybutowi `isWorkDirPersistant` wartość `true` (domyślnie jest mu przypisywana wartość `false`). Atrybut ten jest umieszczony w pliku *katalog_instalacyjny/server.xml*. Po zmianie wartości tego parametru i ponownym uruchomieniu serwera, kody źródłowe kompilowanych dokumentów JSP będzie można znaleźć w katalogu *katalog_instalacyjny/work/host_numerportu*. W przypadku serwera JSWDK 1.0.1, konieczna będzie zmiana wartości parametru `workDirIsPersistent` z `false` na `true`; parametr ten jest zapisany w pliku *katalog_instalacyjny/webserver.xml*. Po dokonaniu tej modyfikacji kody źródłowe kompilowanych dokumentów JSP będzie można znaleźć w katalogu *katalog_instalacyjny/work/%3Anumer_portu%2F*. Java Web Server 2.0 jest domyślnie konfigurowany w taki sposób, że kody źródłowe automatycznie generowanych serwletów są zachowywane; można je znaleźć w katalogu *katalog_instalacyjny/tmpdir/default/pagecompile/jsp/_JSP*.

Koniecznie należy podać jedno ostrzeżenie dotyczące procesu automatycznego przekształcania dokumentów JSP. Otóż, jeśli w kodzie dokumentu JSP dynamicznie generującego informacje, zostanie popełniony błąd, to serwer może nie być w stanie poprawnie skompilować wygenerowanego serwletu. W przypadku pojawienia się takiego krytycznego błędu na etapie przekształcania strony, serwer wyświetli stronę WWW zawierającą opis problemu. Niestety Internet Explorer 5 zastępuje strony błędów generowane przez serwer swoimi własnymi, które uważa za bardziej „przyjazne dla użytkownika". Testując strony JSP konieczne będzie wyłączenie tej „opcji". W tym celu, z menu głównego Internet Explorera należy wybrać opcję *Narzędzia⇨Opcje internetowe*, następnie przejść na zakładkę *Zaawansowane* i usunąć znacznik z pola wyboru *Pokaż przyjazne komunikaty o błędach HTTP*.

Ostrzeżenie
Testując strony JSP upewnij się, że Internet Explorer nie będzie wyświetlał „przyjaznych" komunikatów o błędach HTTP.

Oprócz zwyczajnego kodu HTML, w dokumentach JSP mogą się pojawiać trzy typy „konstrukcji" JSP — *elementy skryptowe*, *dyrektywy* oraz *akcje*. Elementy skryptowe pozwalają na podawanie kodu napisanego w języku Java, który stanie się częścią wynikowego serwletu. Dyrektywy pozwalają natomiast na określanie ogólnej struktury generowanego serwletu, a akcje — na wskazywanie istniejących komponentów, których należy użyć lub na inną kontrolę działania mechanizmów obsługi JSP. Aby ułatwić tworzenie elementów skryptowych, programiści piszący strony JSP mają dostęp do kilku predefiniowanych zmiennych (takich jak zmienna `request` z przykładu przedstawionego na samym początku tego rozdziału; więcej informacji na temat tych zmiennych znajdziesz w podrozdziale 10.5.). W dalszej części tego rozdziału omówię elementy skryptowe JSP, natomiast zagadnienia związane z dyrektywami i akcjami przedstawię w kolejnych rozdziałach. Skrócony opis składni JSP znajdziesz także w dodatku A — „Krótki przewodnik po serwletach i JSP".

W niniejszej książce omawiam wersje 1.0 oraz 1.1 specyfikacji Java Server Pages. Należy wiedzieć, że w specyfikacji JSP 1.0 wprowadzono wiele zmian, przez co w ogromny sposób różni się ona od specyfikacji 0.92. Wprowadzone zmiany są korzystne, lecz ze względu na nie nowsze strony JSP niemal w ogóle nie są zgodne z mechanizmami JSP bazującymi na specyfikacji 0.92; podobnie zresztą jak starsze strony JSP niemal w ogóle nie są zgodne z mechanizmami JSP bazującymi na specyfikacji 1.0. Zmiany pomiędzy specyfikacjami JSP 1.0 oraz 1.1 są już znacznie mniejsze — główną innowacją wprowadzoną w specyfikacji 1.1 są ułatwienia w definiowaniu nowych znaczników oraz możliwość generacji serwletów bazujących na specyfikacji Java Servlet 2.2. Strony JSP bazujące na specyfikacji 1.1, które nie wykorzystują znaczników definiowanych przez programistę ani jawnych odwołań do możliwości serwletów charakterystycznych dla specyfikacji Java Servlet 2.2, są zgodne z JSP 1.0. Wszystkie strony JSP bazujące na technologii 1.0 są całkowicie i bez wyjątków zgodne ze specyfikacją JSP 1.1.

10.1 Elementy skryptowe

Elementy skryptowe JSP pozwalają na wstawianie kodu do serwletu który zostanie wygenerowany na podstawie strony JSP. Elementy te mogą przybierać trzy postacie:

1. *wyrażeń* — `<%= wyrażenie %>` — których wartość jest obliczana i wstawiana do kodu generowanego przez serwlet,
2. *skryptletów* — `<% kod %>` — których kod jest wstawiany do metody `_jspService` serwletu (wywoływanej przez metodę `service`),
3. *deklaracji* — `<%! kod %>` — których kod wstawiany jest wewnątrz klasy serwletu poza jakimikolwiek metodami.

   W kolejnych podrozdziałach dokładniej opiszę każdy z tych elementów skryptowych.

Tekst szablonu

W bardzo wielu przypadkach znaczną częścią strony JSP jest statyczny kod HTML, określany jako *tekst szablonu*. Niemal zawsze tekst szablonu wygląda jak zwyczajny kod HTML, jest tworzony zgodnie z zasadami składni HTML i jest bez żadnych modyfikacji przekazywany do przeglądarki przez serwlet utworzony w celu obsługi strony. Tekst szablonu nie tylko wygląda jak zwyczajny kod HTML — można go także tworzyć za pomocą tych samych narzędzi, które są normalnie używane do tworzenia stron WWW. Na przykład, większość przykładowych stron JSP przedstawionych w tej książce stworzyłem w programie HomeSite firmy Allaire.

Istnieją dwa, drobne wyjątki od reguły mówiącej, że „tekst szablonu jest przekazywany do przeglądarki bez żadnych modyfikacji". Po pierwsze, aby wygenerować łańcuch znaków `<%`, w kodzie strony JSP należy zapisać go w postaci `<\%`. I po drugie, jeśli w kodzie strony JSP chcesz umieścić komentarze, które nie mają się pojawiać w kodzie HTML generowanej strony WWW, to należy je zapisać w następującej postaci:

```
<%-- komentarz w kodzie JSP --%>
```

Zwyczajne komentarze HTML:

```
<!-- komentarz HTML -->
```

są przekazywane do strony wynikowej.

10.2 Wyrażenia JSP

Wyrażenia JSP są stosowane w celu wstawiania wartości bezpośrednio do kodu HTML generowanego przez serwlet. Są one zapisywane w następującej postaci:

```
<%= wyrażenie zapisane w języku Java %>
```

Podane wyrażenie jest obliczane, a jego wynik jest następnie konwertowany do postaci łańcucha znaków i umieszczany w wynikowym kodzie HTML. Obliczenie wartości wyrażenia odbywa się w czasie wykonywania serwletu (po otrzymaniu żądania), co oznacza, że można w nim wykorzystać wszystkie informacje przekazane w żądaniu. Na przykład, przedstawiony poniżej fragment kodu wyświetla datę i czas otrzymania żądania:

```
Aktualny czas to: <%= new java.util.Date() %>
```

Predefiniowane zmienne

JSP udostępnia kilka predefiniowanych zmiennych, które ułatwiają tworzenie wyrażeń. Te niejawne obiekty omówię bardziej szczegółowo w podrozdziale 10.5; tu jedynie przedstawię cztery spośród nich, które mają największe znaczenie przy tworzeniu wyrażeń:

- **`request`** — obiekt `HttpServletRequest`,

- **response** — obiekt HttpServletResponse,
- **session** — obiekt HttpSession, reprezentujący sesję skojarzoną z danym żądaniem (chyba że obsługa sesji została wyłączona przy użyciu atrybutu session dyrektywy page — patrz podrozdział 11.4),
- **out** — obiekt PrintWriter (a konkretnie obiekt klasy JspWriter wyposażonej w mechanizmy buforowania) służący do przekazywania informacji do przeglądarki.

  Oto przykład użycia tych zmiennych:

```
Nazwa Twojego komputera: <%= request.getRemoteHost() %>
```

Składnia XML stosowana w wyrażeniach

Twórcy dokumentów XML mogą zapisywać wyrażenia JSP w poniższy, alternatywny sposób:

```
<jsp:expression>
wyrażenie zapisane w języku Java
</jsp:expression>
```

Zwróć uwagę, iż w odróżnieniu od elementów HTML, w elementach XML wielkość liter odgrywa znaczenie.

Zastosowanie wyrażeń jako wartości atrybutów

Jak się przekonasz w dalszej części książki, JSP zawiera wiele elementów, których parametry można określać przy wykorzystaniu składni XML. W przedstawionym poniżej przykładzie, łańcuch znaków "Maria" zostaje przekazany do metody setFirstName obiektu skojarzonego ze zmienną autor. Nie przejmuj się, jeśli nie rozumiesz znaczenia tego kodu — omówię je dokładnie w rozdziale 13. — „Wykorzystanie komponentów JavaBeans w dokumentach JSP". Podając ten przykład chciałem jedynie przedstawić sposób wykorzystania atrybutów name, property oraz value.

```
<jsp:setProperty name="autor"
                 property="firstName"
                 value="Maria" />
```

Większość atrybutów wymaga, aby wartość była łańcuchem znaków zapisanym w znakach apostrofu lub cudzysłowach (jak na powyższym przykładzie). Jednak w kilku przypadkach istnieje możliwość wykorzystania wyrażeń JSP, których wartości są obliczane w trakcie obsługi żądania. Jednym z nich jest atrybut value elementu jsp:setProperty. Oznacza to, że przedstawiony poniżej fragment kodu jest całkowicie poprawny:

```
<jsp:setProperty name="uzytkownik"
                 property="id"
                 value='<%= "UserID" + Math.random() %>' />
```

Atrybuty, w których można stosować wyrażenia przetwarzane w czasie obsługi żądania przedstawiłem w tabeli 10.1.

Tabela 10.1 Atrybuty, w których można stosować wyrażenia JSP.

Nazwa elementu	Nazwa atrybutu
jsp:setProperty — patrz podrozdział 13.3. — „Określanie wartości właściwości komponentów".	name, value
jsp:include — patrz rozdział 12. — „Dołączanie plików i apletów do dokumentów JSP".	page
jsp:forward — patrz rozdział 15. — „Integracja serwletów i dokumentów JSP".	page

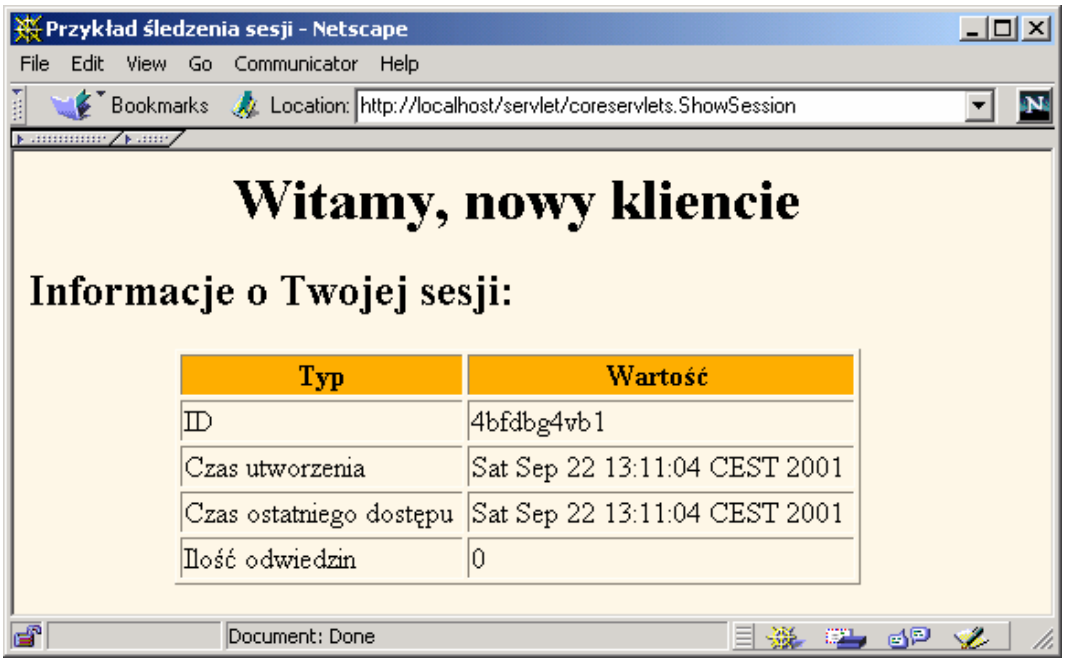
`jsp:param` — patrz rozdział 12. — „Dołączanie plików i apletów do dokumentów JSP”.	`value`

Przykład

Na listingu 10.1 przedstawiłem przykładową stronę JSP, a na rysunku 10. 1 — wyniki jej wykonania. Zwróć uwagę, iż w nagłówku strony (w znaczniku `<HEAD>`) umieściłem znaczniki `<META>` oraz dołączyłem arkusz stylów. Dołączanie tych elementów stanowi dobry zwyczaj, jednak istnieją dwa powody dla których są one często pomijane w stronach generowanych przez zwyczajne serwlety. Po pierwsze, stosowanie koniecznych wywołań metody `println` jest w serwletach męczące. Problem ten jest znacznie prostszy w przypadku tworzenia dokumentów JSP; na przykład, znaczniki te można dołączać do dokumentu przy użyciu mechanizmów wielokrotnego używania kodu, udostępnianych przez używany program do edycji dokumentów HTML. Po drugie, w serwletach nie można używać najprostszej postaci względnych adresów URL (tych, które odwołują się do plików przechowywanych w tym samym katalogu). Wynika to z faktu, że katalogi z serwletami nie są tak samo kojarzone z adresami URL jak katalogi zawierające normalne dokumenty HTML. Jednak dokumenty JSP są umieszczane na serwerze w tych samych katalogach co strony WWW, dzięki czemu adresy URL są przetwarzane poprawnie. A zatem, arkusze stylów oraz dokumenty JSP mogą być przechowywane w tym samym katalogu. Kod źródłowy użytego arkusza stylów, jak również kody źródłowe wszystkich przykładów podanych w niniejszej książce można znaleźć pod adresem *ftp://ftp.helion.pl/przyklady/jsjsp.zip*.

Listing 10.1 Expressions.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Wyrażenia JSP</TITLE>
<META NAME="author" CONTENT="Marty Hall">
<META NAME="keywords"
      CONTENT="JSP,wyrażenia,Java Server Pages,serwlety">
<META NAME="description"
      CONTENT="Krótki przykład wyrażeń JSP.">
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<H2>Wyrażenia JSP</H2>
<UL>
  <LI>Aktualny czas: <%= new java.util.Date() %>
  <LI>Nazwa komputera: <%= request.getRemoteHost() %>
  <LI>Identyfikator sesji: <%= session.getId() %>
  <LI>Parametr <CODE>testParam</CODE>:
      <%= request.getParameter("testParam") %>
</UL>
</BODY>
</HTML>
```



Rysunek 10.1 Przykładowe wyniki wykonania strony Expressions.jsp

10.3 Skryptlety JSP

Jeśli chcesz wykonać coś bardziej skomplikowanego do wstawienia wartości wyrażenia, będziesz mógł posłużyć się skryptletami JSP, które pozwalają na wstawienie dowolnego kodu do metody `_jspService`. (Metoda ta jest wywoływana przez metodę `service` serwletu.) Skryptlety mają następującą postać:

```
<% kod napisany w języku Java %>
```

Skryptlety mają dostęp do tych samych automatycznie definiowanych zmiennych co wyrażenia JSP (są to zmienne `request`, `response`, `session`, `out`, itp.; opiszę je szczegółowo w podrozdziale 10.5). A zatem, jeśli chcesz aby jakieś informacje pojawiły się na wynikowej stronie WWW, powinieneś posłużyć się zmienną `out`, tak jak pokazałem na poniższym przykładzie:

```
<%
String daneZapytania = request.getQueryString();
out.println( "Dane przesłane żądaniem GET: " + daneZapytania );
%>
```

W tym konkretnym przypadku, ten sam efekt można uzyskać znacznie łatwiej posługując się wyrażeniem JSP:

```
Dane przesłane żądaniem GET: <%= request.getQueryData() %>
```

Jednak przy użyciu skryptletów można wykonać wiele zadań, których nie można zrealizować przy użyciu samych wyrażeń JSP. Dotyczy to między innymi określania nagłówków odpowiedzi oraz kodu statusu, wywoływania efektów ubocznych (takich jak zapis informacji w dziennikach serwera, czy też aktualizacja baz danych) oraz wykonywania kodu zawierającego pętle, wyrażenia warunkowe lub inne instrukcje złożone. Na przykład, poniższy fragment kodu informuje, że strona zostanie przesłana do przeglądarki jako zwyczajny tekst, a nie jako dokument HTML (to domyślny typ informacji generowanych przez serwlety):

```
<% response.setContentType("text/plain"); %>
```

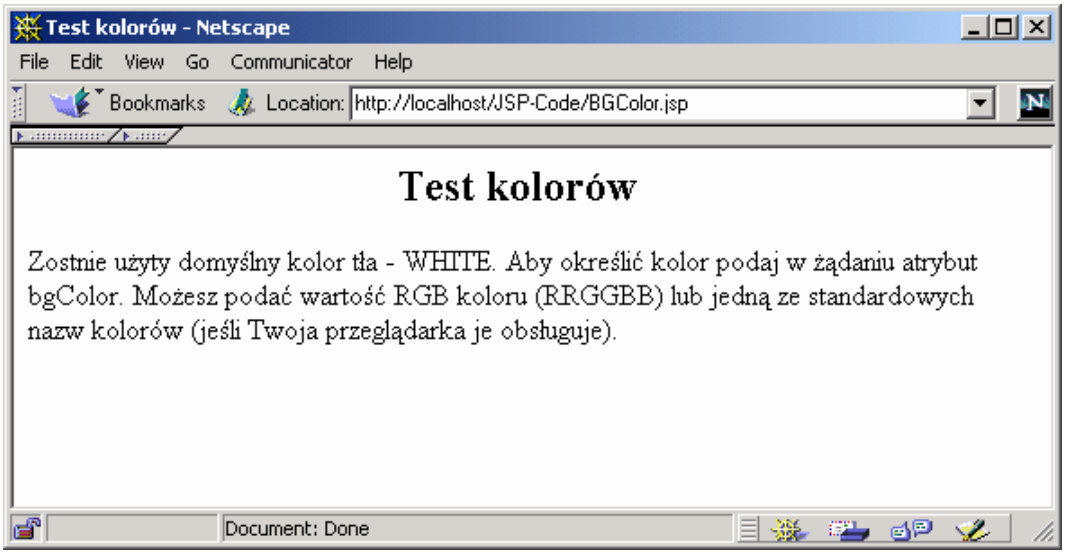
Niezwykle ważne jest to, iż nagłówki odpowiedzi oraz kod statusu można określać w dowolnym miejscu strony JSP, choć pozornie narusza to zasadę, głoszącą iż informacje tego typu muszą zostać wygenerowane przed przekazaniem do przeglądarki jakiejkolwiek treści generowanego dokumentu. Takie określanie kodu statusu oraz nagłówków odpowiedzi jest możliwe

dzięki temu, że strony JSP wykorzystują specjalny typ obiektów `PrintWriter` (konkretnie rzecz biorąc używają obiektów `JspWriter`), które buforują generowany dokument przed jego przesłaniem do przeglądarki. Istnieje jednak możliwość zmiany sposobu buforowania, więcej informacji na ten temat podałem w podrozdziale 11.6 przy okazji omawiania atrybutu `autoflush` dyrektywy `page`.

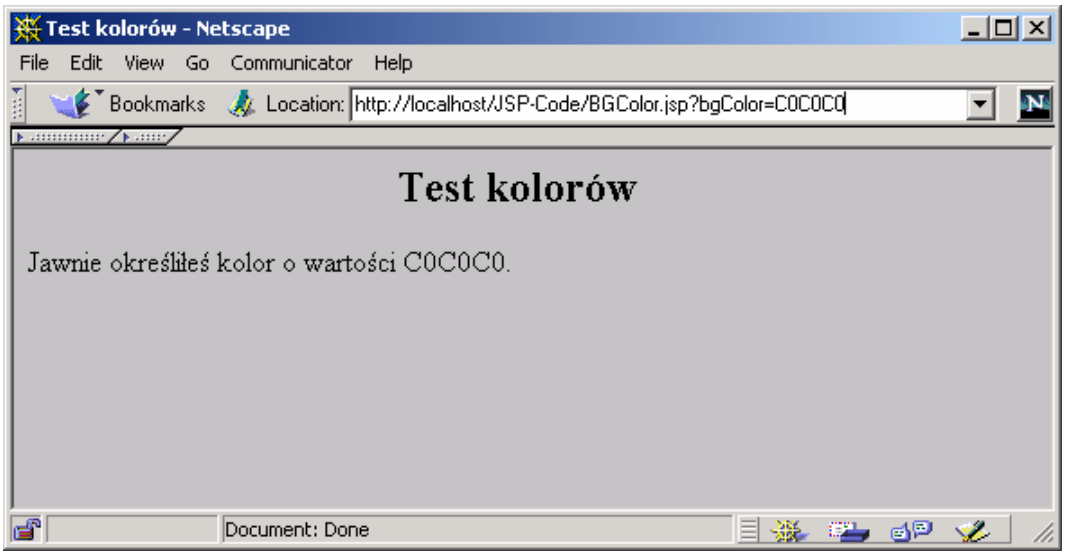
Na listingu 10.2 przedstawiłem przykład kodu który jest zbyt skomplikowany, aby można go było wykonać przy wykorzystaniu wyrażeń JSP. Przykład ten prezentuje stronę JSP, która określa kolor tła generowanego dokumentu HTML na podstawie atrybutu `bgColor`. Wyniki wykonania tej strony zostały przedstawione na rysunkach 10.2, 10.3 oraz 10.4.

Listing 10.2 BGColor.jsp

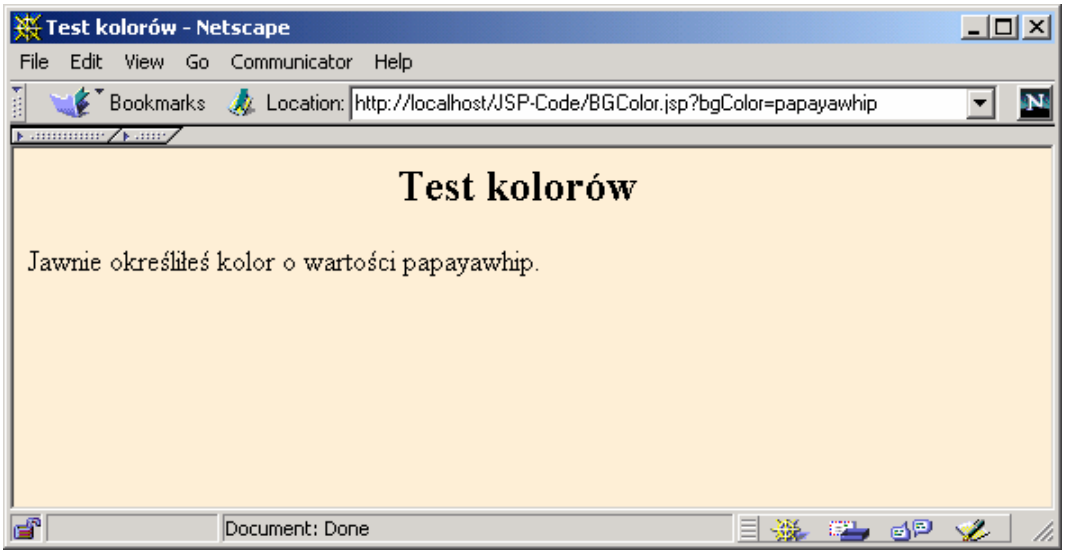
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Test kolorów</TITLE>
</HEAD>

<%
String bgColor = request.getParameter("bgColor");
boolean hasExplicitColor;
if (bgColor != null) {
  hasExplicitColor = true;
} else {
  hasExplicitColor = false;
  bgColor = "WHITE";
}
%>
<BODY BGCOLOR="<%= bgColor %>">
<H2 ALIGN="CENTER">Test kolorów</H2>

<%
if (hasExplicitColor) {
  out.println("Jawnie określiłeś kolor o wartości " +
              bgColor + ".");
} else {
  out.println("Zostnie użyty domyślny kolor tła - WHITE. " +
     "Aby określić kolor podaj w żądaniu atrybut bgColor. " +
     "Możesz podać wartość RGB koloru (RRGGBB) lub jedną ze " +
     "standardowych nazw kolorów (jeśli Twoja przeglądarka je obsługuje).");
}
%>

</BODY>
</HTML>
```



Rysunek 10.2 Domyślne wyniki wykonania strony BGColor.jsp



Rysunek 10.3 Wyniki wykonania strony BGColor.jsp w przypadku przekazania parametru bgColor o wartości C0C0C0



Rysunek 10.4 Wyniki wykonania strony BGColor.jsp w przypadku przekazania parametru bgColor o wartości papayawhip

Wykorzystanie skryptletów do warunkowego wykonania fragmentu strony JSP

Skryptlety są także wykorzystywane w celu warunkowego dołączania kodu HTML i wykonywania instrukcji JSP. W tym przypadku najważniejsze znaczenie ma fakt, iż kod umieszczony wewnątrz skryptletu zostaje wstawiony w metodzie `_jspService` wygenerowanego serwletu, dokładnie w takiej postaci w jakiej został zapisany, a statyczny kod HTML (tekst szablonu) otaczający skryptlet jest zamieniany na wywołania metody `println`. Oznacza to, że skryptlety nie muszą zawierać pełnych instrukcji Javy, a otworzone bloki kodu mogą mieć wpływ na przetwarzanie statycznego kodu HTML oraz kodu JSP umieszczonego poza skryptletem. Na przykład, przeanalizujmy działanie poniższego fragmentu strony JSP, zawierającego zarówno tekst szablonu jak i skryptlety:

```
<% if (Math.random() < 0.5) { %>
<B>Miłego</B> dnia!
<% } else { %>
Mam nadzieję, że będziesz miał <B>pecha</B>!
```

```
<% } %>
```

Po przekształceniu do postaci serwletu, powyższy fragment kodu będzie wyglądał mniej więcej tak:

```
if (Math.random() < 0.5) {
  out.println("<B>Miłego</B> dnia!");
} else {
  out.println("Mam nadzieję, że będziesz miał <B>pecha</B>!");
}
```

Specjalna składnia skryptletów

Powinieneś jeszcze wiedzieć o dwóch sprawach. Otóż jeśli w skryptlecie chcesz użyć łańcucha znaków `%>` to musisz zapisać go w postaci `%\>`. Poza tym, warto wiedzieć, że istnieje alternatywny — XML-owy — odpowiednik zapisu `<% kod skryptletu %>`, oto on:

```
<jsp:scriptlet>
kod skryptletu
</jsp:scriptlet>
```

10.4 Deklaracje JSP

Deklaracje JSP pozwalają definiować metody i pola, które zostaną umieszczone w klasie serwletu (jednak *poza* metodą `_jspService` wywoływaną przez metodę `service` podczas obsługi żądania). Poniżej przedstawiłem ogólną postać deklaracji:

```
<%! kod w języku Java %>
```

Deklaracje nie generują żadnych informacji wyjściowych, a zatem są zazwyczaj używane w połączeniu z wyrażeniami JSP i skryptletami. Na przykład, poniżej przedstawiłem fragment kodu wyświetlający ilość odwołań do strony od czasu uruchomienia serwera (lub od momentu zmiany i ponownego załadowania klasy serwletu do pamięci serwera). Przypominasz sobie zapewne, że żądania dotyczące tego samego serwletu powodują tworzenie niezależnych wątków wywołujących metodę `service` jednej kopii serwletu. Nie powodują one natomiast tworzenia wielu niezależnych kopii danego serwletu; chyba, że implementuje on interfejs `SingleThreadModel`. Więcej informacji na temat tego interfejsu znajdziesz w podrozdziale 2.6 (pt.: „Cykl życiowy serwletów") oraz 11.3 (pt.: „Atrybut isThreadSafe"). Oznacza to, że zmienne instancyjne (pola) serwletu są wspólnie wykorzystywane przez wszystkie, równocześnie obsługiwane żądania odwołujące się do tego samego serwletu. A zatem, przedstawiona poniżej zmienna `accessCount` nie musi być deklarowana jako statyczna:

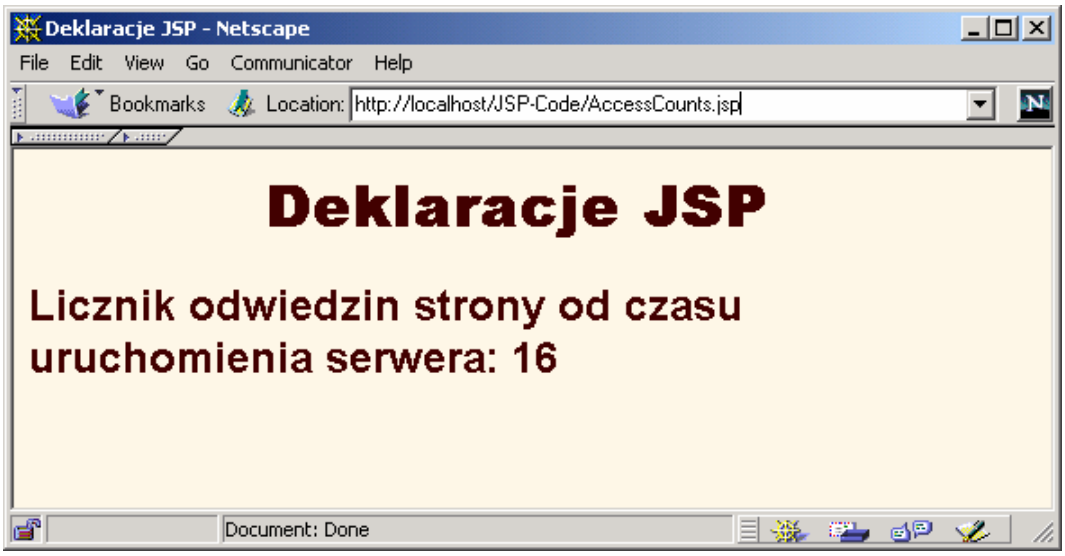
```
<%! private int accessCount = 0 %>
Licznik odwiedzin strony od czasu uruchomienia serwera:
<%= ++accessCount %>
```

Powyższy fragment kodu został wykorzystany na stronie JSP przedstawionej na listingu 10.3. Wyniki jej wykonania można zobaczyć na rysunku 10.5.

Listing 10.3 AccessCount.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Deklaracje JSP</TITLE>
<META NAME="author" CONTENT="Marty Hall">
<META NAME="keywords"
      CONTENT="JSP,deklaracje,Java Server Pages,serwlety">
<META NAME="description"
      CONTENT="Krótki przykład deklaracji JSP.">
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>
```

```
<BODY>
<H1>Deklaracje JSP</H1>

<%! private int accessCount = 0; %>
<H2>Licznik odwiedzin strony od czasu uruchomienia serwera:
<%= ++accessCount %></H2>

</BODY>
</HTML>
```



Rysunek 10.5 Wyniki wykonania strony AccessCount.jsp po 15 wcześniejszych odwołaniach do niej.

Specjalna składnia zapisu deklaracji

W deklaracjach, podobnie jak w skryptletach, aby użyć łańcucha znaków `%>` należy zapisać go jako `%\>`. Istnieje także alternatywny — XML-owy — sposób zapisu deklaracji:

```
<jsp:declaration>
kod deklaracji
</jsp:declaration>
```

10.5 Predefiniowane zmienne

Aby uprościć kod wyrażeń JSP i skryptletów, można w nich używać ośmiu automatycznie definiowanych zmiennych, nazywanych także *obiektami niejawnymi*. Zmiennych tych nie można stosować w deklaracjach JSP (patrz podrozdział 10.4), gdyż generowany przez nie kod jest umieszczany poza metodą `_jspService`. Predefiniowane zmienne to: `request`, `response`, `out`, `session`, `application`, `config`, `pageContext` oraz `page`. Poniżej zamieściłem dokładniejszy opis każdej z nich.

request

Ta zmienna reprezentuje obiekt `HttpServletRequest` skojarzony z żądaniem; daje ona dostęp do parametrów żądania, informacji o jego typie (np.: `GET` lub `POST`) oraz otrzymanych nagłówkach żądania (w tym także o cookies). Gwoli ścisłości należy zaznaczyć, iż jeśli żądanie zostało przesłane protokołem innym niż HTTP, to zmienna `request` może zawierać obiekt, którego typ jest rozszerzeniem interfejsu `ServletResponse` lecz nie `HttpServletResponse`. Jednak aktualnie bardzo niewiele serwerów obsługuje serwlety, które mogą przyjmować żądania przesyłane innymi protokołami niż HTTP.

response

Ta zmienna reprezentuje obiekt `HttpServletResponse` skojarzony z odpowiedzią, która zostanie przesłana do przeglądarki użytkownika. Zwróć uwagę, iż strumień wyjściowy (patrz zmienna `out`) jest zazwyczaj buforowany, dzięki czemu w stronach JSP można podawać nagłówki odpowiedzi oraz kod statusu, choć w serwletach, po wygenerowaniu dowolnego fragmentu dokumentu, nie można było tego robić.

out

Predefiniowana zmienna `out` to obiekt `PrintWriter`, używany do przesyłania wyników do przeglądarki. Aby obiekt `response` zachował przydatność, to zmienna `out` jest w rzeczywistości obiekt klasy `JspWriter` będącej zmodyfikowaną wersją klasy `PrintWriter` wyposażoną w możliwości buforowania. Wielkość buforu można określać za pomocą atrybutu `buffer` dyrektywy `page` (patrz podrozdział 11.5). Zwróć także uwagę, iż zmienna ta jest niemal wyłącznie używana w skryptletach, gdyż wartości wyrażeń JSP są automatycznie umieszczane w strumieniu wyjściowym (przez co nie trzeba ich jawnie generować przy użyciu zmiennej `out`).

session

Ta zmienna zawiera obiekt `HttpSession` skojarzony z daną sesją. Przypominasz sobie zapewne, że sesję są tworzone automatycznie, a zatem zmienna ta jest kojarzona z obiektem nawet jeśli w nadesłanym żądaniu nie ma żadnego odwołania do sesji. Jedynym wyjątkiem są sytuacje, gdy obsługa sesji zostanie wyłączona przy użyciu atrybutu `session` dyrektywy `page` (patrz podrozdział 11.4). W takim przypadku odwołanie do zmiennej `session` spowoduje powstanie błędów podczas procesu translacji strony JSP do postaci serwletu.

application

Ta zmienna zawiera obiekt `ServletContext`, który można także uzyskać za pomocą wywołania metody `getServletConfig().getContext()`. W tym obiekcie serwlety oraz strony JSP mogą przechowywać trwałe informacje, zamiast używać do tego celu zmiennych instancyjnych. Interfejs `ServletContext` udostępnia metody `setAttribute` oraz `getAttribute`, które pozwalają na zachowanie i pobranie dowolnych danych skojarzonych z podanymi kluczami. Różnica pomiędzy przechowywanie danych w obiekcie `ServletContext` a w zmiennych instancyjnych polega na tym, iż obiekt ten jest wspólnie wykorzystywany przez wszystkie serwlety działające w danym mechanizmie obsługi serwletów (lub w danej aplikacji WWW, jeśli używany serwer dysponuję taką możliwością). Więcej informacji na temat interfejsu `ServletContext` znajdziesz w podrozdziale 13.4. (pt.: „Wspólne wykorzystywanie komponentów”) oraz w rozdziale 15. — „Integracja serwletów i JSP”.

config

Ta zmienna zawiera obiekt `ServletConfig` dla danej strony.

pageContext

W JSP została wprowadzona nowa klasa o nazwie `PageContext`, której celem jest udzielenie dostępu do wielu atrybutów strony oraz stworzenie wygodnego miejsca do przechowywania wspólnie wykorzystywanych informacji. Zmienna `pageContext` zawiera obiekt klasy `PageContext` skojarzony z bieżącą stroną. Zastosowanie tego obiektu opisałem dokładniej w podrozdziale 13.4. — pt.: „Wspólne wykorzystywanie komponentów”.

page

Ta zmienna jest synonimem słowa kluczowego `this` i raczej nie jest często stosowana w języku Java. Została ona stworzona „na wszelki wypadek" gdyby serwlety i strony JSP mogły być tworzone w jakimś innym języku programowania.

Rozdział 11.
Dyrektywa page: Strukturalizacja generowanych serwletów

Dyrektywy JSP mają wpływ na ogólną strukturę serwletu generowanego na podstawie strony JSP. Przedstawione poniżej wzory pokazują dwie możliwe formy zapisu dyrektyw. Wartości atrybutów można także zapisywać zarówno w cudzysłowach jak i apostrofach, nie można jednak pominąć otaczających je znaków (niezależnie od tego czy będą to apostrofy czy cudzysłowy). Aby umieścić cudzysłów lub apostrof wewnątrz wartości atrybutu, należy poprzedzić go znakiem odwrotnego ukośnika — `\"` (aby umieścić cudzysłów) oraz `\'` (aby umieścić apostrof).

```
<%@ dyrektywa atrybut="wartość" %>
<%@ dyrektywa atrybut1="wartość1"
              atrybut2="wartość2"
              ...
              atrybutN="wartośćN" %>
```

Przy tworzeniu stron JSP można stosować trzy dyrektywy — `page`, `include` oraz `taglib`. Dyrektywa `page` umożliwia kontrolę struktury serwletu, poprzez importowanie klas, modyfikowanie klasy bazowej serwletu, określanie typu zawartości , itp. Dyrektywę tę można umieścić w dowolnym miejscu dokumentu. Zastosowanie dyrektywy `page` będzie tematem niniejszego rozdziału. Druga dyrektywa JSP — `include` — umożliwia wstawianie plików do klasy serwletu na etapie przekształcania strony JSP do postaci serwletu. Dyrektywę tę należy umieszczać w tym miejscu strony JSP, w jakim chcesz wstawić plik; sposoby jej użycia opisałem szczegółowo w rozdziale 12. — „Dołączanie plików i apletów do dokumentów JSP". W specyfikacji 1.1 technologii JSP pojawiła się trzecia dyrektywa — `taglib`. Można jej używać do definiowania własnych znaczników. Więcej informacji na ten temat znajdziesz w rozdziale 14. — „Tworzenie bibliotek znaczników".

Dyrektywa page umożliwia zdefiniowanie następujących atrybutów — `import`, `contentType`, `isThreadSafe`, `session`, `buffer`, `autoflush`, `extends`, `info`, `errorPage`, `isErrorPage` oraz `language`. Pamiętaj, że przy podawaniu nazw tych atrybutów jest uwzględniana wielkość liter. Wszystkie powyższe atrybut omówię szczegółowo w dalszych częściach rozdziału.

11.1 Atrybut import

Atrybut `import` dyrektywy `page` określa pakiet jaki powinien zostać zaimportowany do serwletu wygenerowanego na podstawie strony JSP. Jeśli jawnie nie określisz żadnych klas, które mają być zaimportowane, to serwlet automatycznie zaimportuje klasy `java.lang.*`, `javax.servlet.*`, `javax.servlet.jsp.*`, `javax.servlet.http.*` i być może także kilka innych klas, charakterystycznych dla serwletów. Nigdy nie twórz stron JSP, których działanie ma polegać na automatycznie importowanych klasach charakterystycznych dla serwera. Atrybutu `import` dyrektywy `page` można używać na dwa sposoby:

```
<%@ page import="pakiet.klasa" %>
```

```
<%@ page import="pakiet.klasa1,...,pakiet.klasaN" %>
```

Przykładowo, poniższa dyrektywa informuje, że klasy należące do pakietu `java.util` powinne być dostępne bez konieczności jawnego określania jego nazwy.

```
<%@ page import="java.util.*" %>
```

Atrybut `import` dyrektywy `page` jest jedynym atrybutem, który może się wielokrotnie pojawić w tym samym dokumencie. Choć dyrektywę page można umieszczać w dowolnym miejscu dokumentu, to jednak tradycyjne polecenia importu umieszcza się na samym początku dokumentu lub bezpośrednio przed pierwszym fragmentem kodu, w jakim importowana klasa zostaje użyta.

Katalogi służące do przechowywania własnych klas

Jeśli importujesz klasy, które nie należą do standardowych pakietów `java` lub `javax.servlet`, musisz mieć pewność, że klasy te zostały poprawnie zainstalowane na serwerze. Większość serwerów dysponujących możliwością automatycznego przeładowywania serwletów nie pozwala, aby dokumenty JSP odwoływały się do klas, których pliki są przechowywane w katalogach umożliwiających automatyczne odświeżanie. Nazwa katalogu używanego do przechowywania plików klasowych serwletów zależy od używanego serwera, a precyzyjnych informacji na ten temat należy szukać w jego dokumentacji. Nazwy katalogów używanych przez serwery Apache Tomcat 3.0, JSWDK 1.0.1 oraz Java Web Server 2.0 zostały przedstawione w tabeli 11.1. Wszystkie te serwery wykorzystują pliki JAR przechowywane w katalogu *lib*; wszystkie wymagają także ponownego uruchomienia w przypadku modyfikacji plików przechowywanych w tym katalogu.

Tabela 11.1 Katalogi używane przy instalacji klas.

Serwer	Katalog (względem katalogu instalacyjnego)	Przeznaczenie	Automatyczna aktualizacja klas w przypadku ich zmiany?	Katalog dostępny z poziomu stron JSP?
Tomcat 3.0	*webpages/WEB-INF/ classes*	Standardowe położenie plików klasowych serwletów	Nie	Tak
Tomcat 3.0	*classes*	Alternatywne położenie plików klasowych serwletów	Nie	Tak
JSWDK 1.0.1	*webpages/WEB-INF/ servlets*	Standardowe położenie klas serwletów	Nie	Tak
JSWDK 1.0.1	*classes*	Alternatywne położenie klas serwletów	Nie	Tak
Java Web Server 2.0	*servlets*	Położenie klas często modyfikowanych serwletów	**Tak**	**Nie**
Java Web Server 2.0	*classes*	Położenie klas rzadko modyfikowanych serwletów	Nie	Tak

Przykład

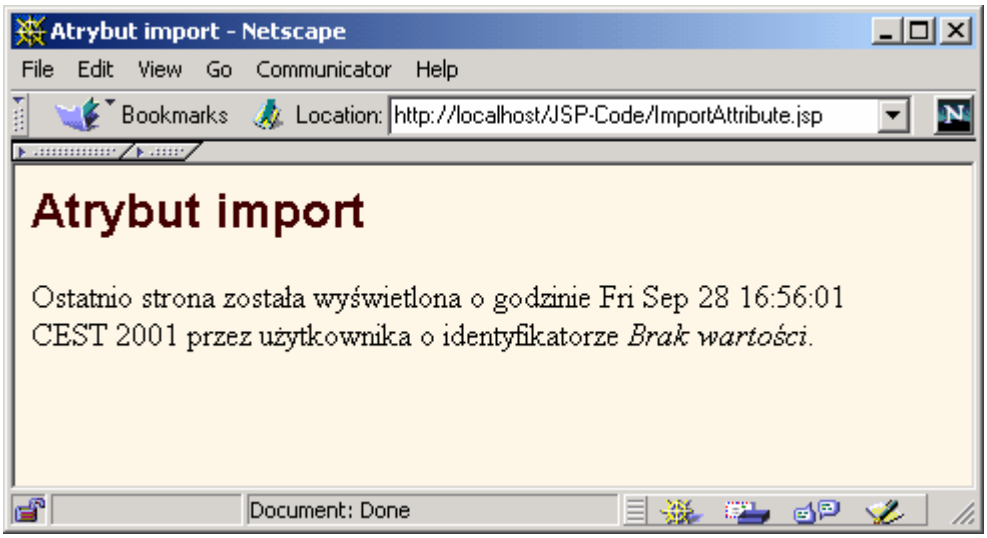
Listing 11.1 przedstawia stronę JSP wykorzystującą trzy klasy, które standardowo nie są importowane — `java.util.Date`, `coreservlets.ServletUtilities` (patrz listing 8.3) oraz `coreservlets.LingLivedCookie` (patrz listing 8.4). Aby uprościć proces odwoływania się do tych klas, w przykładowej stronie JSP użyłem dyrektywy `page` o następującej postaci:

```
<%@ page import="java.util.*,coreservlets.*" %>
```

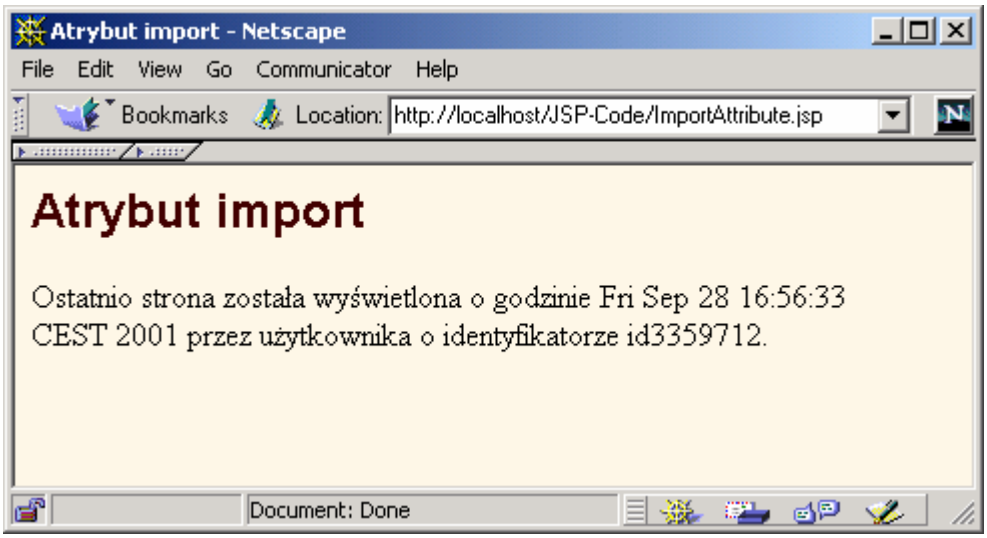
Typowe wyniki wykonania strony z listingu 11.1 przedstawiłem na rysunkach 11.1 oraz 11.2.

Listing 11.1 ImportAttribute.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Atrybut import</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<H2>Atrybut import</H2>
<%-- dyrektywa page --%>
<%@ page import="java.util.*,coreservlets.*" %>

<%-- Deklaracja JSP (patrz podrozdział 10.4) --%>
<%!
private String randomID() {
  int num = (int)(Math.random()*10000000.0);
  return("id" + num);
}

private final String NO_VALUE = "<I>Brak wartości</I>";
%>

<%-- Skryptlet (patrz podrozdział 10.3) --%>
<%
Cookie[] cookies = request.getCookies();
String oldID =
  ServletUtilities.getCookieValue(cookies, "userID", NO_VALUE);
String newID;
if (oldID.equals(NO_VALUE)) {
  newID = randomID();
} else {
  newID = oldID;
}
LongLivedCookie cookie = new LongLivedCookie("userID", newID);
response.addCookie(cookie);
%>

<%-- Wyrażenie JSP (patrz podrozdział 10.2) --%>
Ostatnio strona została wyświetlona o godzinie <%= new Date() %>
przez użytkownika o identyfikatorze <%= oldID %>.

</BODY>
</HTML>
```



Rysunek 11.1 Wyniki pierwszego wywołania strony ImportAttribute.jsp



Rysunek 11.2 Wyniki powtórnego wyświetlenia strony ImportAttribute.jsp

11.2 Atrybut contentType

Atrybut `contentType` określa wartość nagłówka odpowiedzi `Content-Type`. Pamiętasz zapewne, że nagłówek ten określa typ MIME dokumentu przesyłanego z serwera do przeglądarki. Więcej informacji na jego temat możesz znaleźć w tabeli 7.1 (pt.: „Najczęściej stosowane typy MIME") zamieszczonej w podrozdziale 7.2. — „Nagłówki odpowiedzi protokołu HTTP 1.1 oraz ich znaczenie".

Poniżej przedstawiłem dwa możliwe sposoby stosowania atrybutu `contentType`:

```
<%@ page contentType="typ-MIME" %>
<%@ page contentType="typ-MIME; charset=zbiórZnaków" %>
```

Na przykład, użycie dyrektywy o następującej postaci:

```
<%@ page contentType="text/plain" %>
```

da takie same wyniki co użycie poniższego skryptletu

```
<% response.setContentType("text/plain"); %>
```

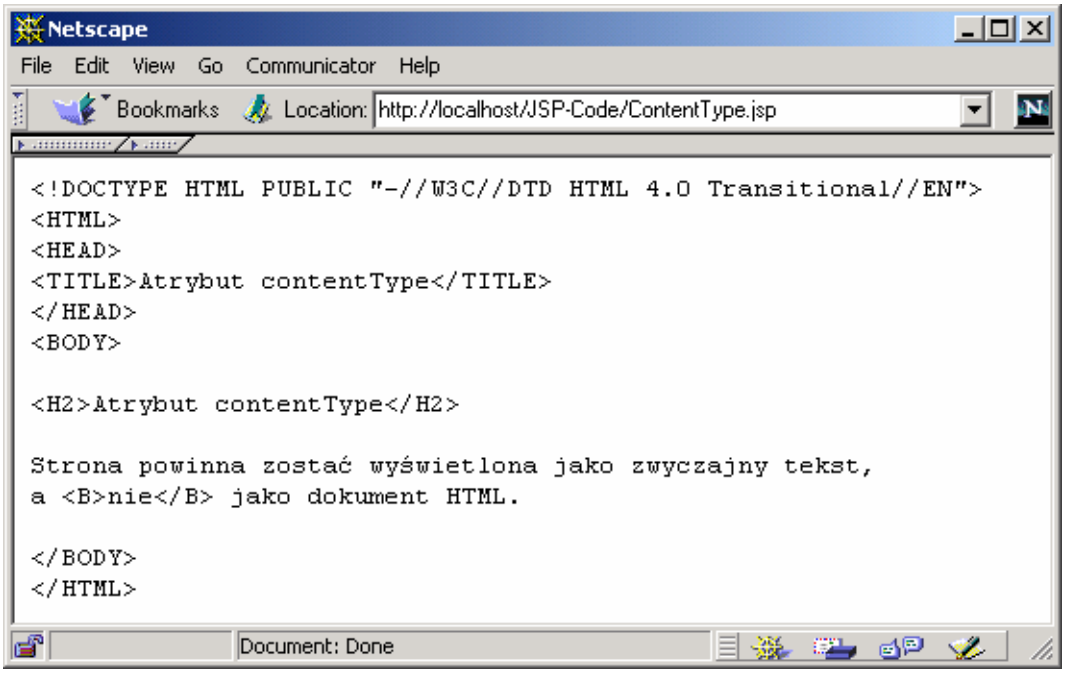
Domyślnym typem MIME dokumentów JSP jest `text/html` (a standardowo używanym zbiorem znaków jest `ISO-8859-1`), co odróżnia je od zwyczajnych serwletów, których domyślnym typem MIME jest `text/plain`.

Generacja zwyczajnych dokumentów tekstowych

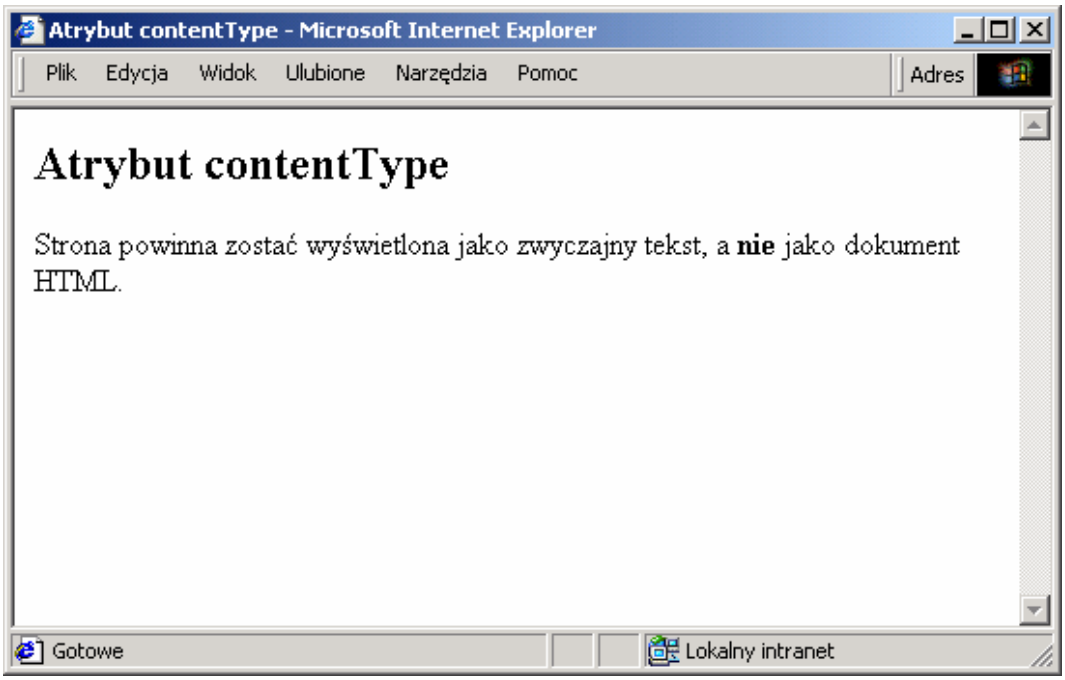
Na listingu 11.2 przedstawiłem stronę JSP która pozornie generuje dokument HTML, lecz jego typ określa jako `text/plain`. W takich przypadkach przeglądarki powinny wyświetlić dane tekstowe (bez ich interpretowania), tak jak pokazuje rysunek 11.3 przedstawiający Netscape Navigatora. Internet Explorer widoczny na rysunku 11.4 interpretuje stronę, jak gdyby była ona zwyczajnym dokumentem HTML.

Listing 11.2 ContentType.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Atrybut contentType</TITLE>
</HEAD>
<BODY>

<H2>Atrybut contentType</H2>
<%@ page contentType="text/plain" %>
Strona powinna zostać wyświetlona jako zwyczajny tekst,
a <B>nie</B> jako dokument HTML.

</BODY>
</HTML>
```



Rysunek 11.3 Zwyczajny dokument tekstowy, Netscape nie musi interpretować znaczników HTML



Rysunek 11.4 Internet Explorer interpretuje znaczniki HTML zapisane w zwyczajnych dokumentach tekstowych

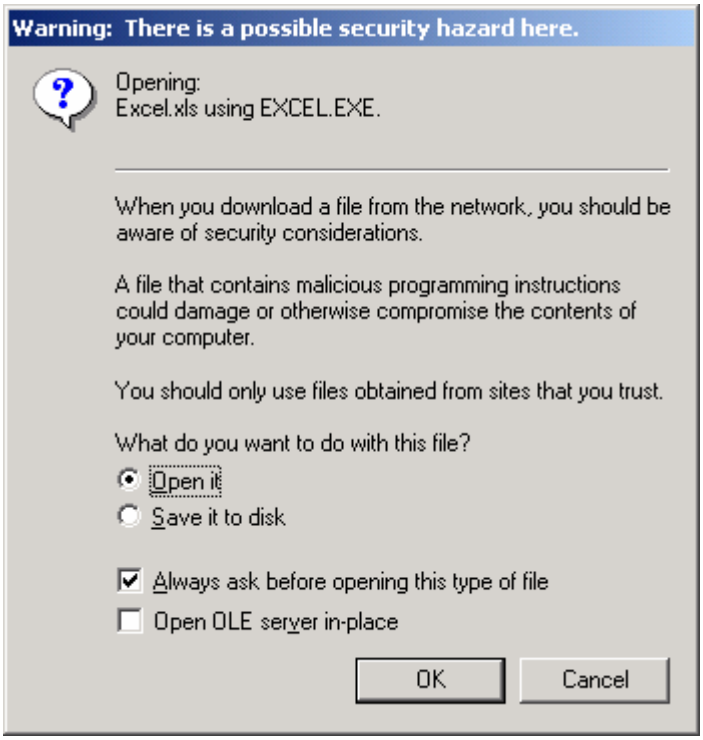
Generacja arkuszy kalkulacyjnych programu Microsoft Excel

Aby wygenerować arkusz kalkulacyjny programu Microsoft Excel należy zdefiniować typ zwracanych informacji jako `application/vnd.ms-excel`, a następnie sformatować komórki generowanego arkusza na jeden z dwóch dostępnych sposobów.

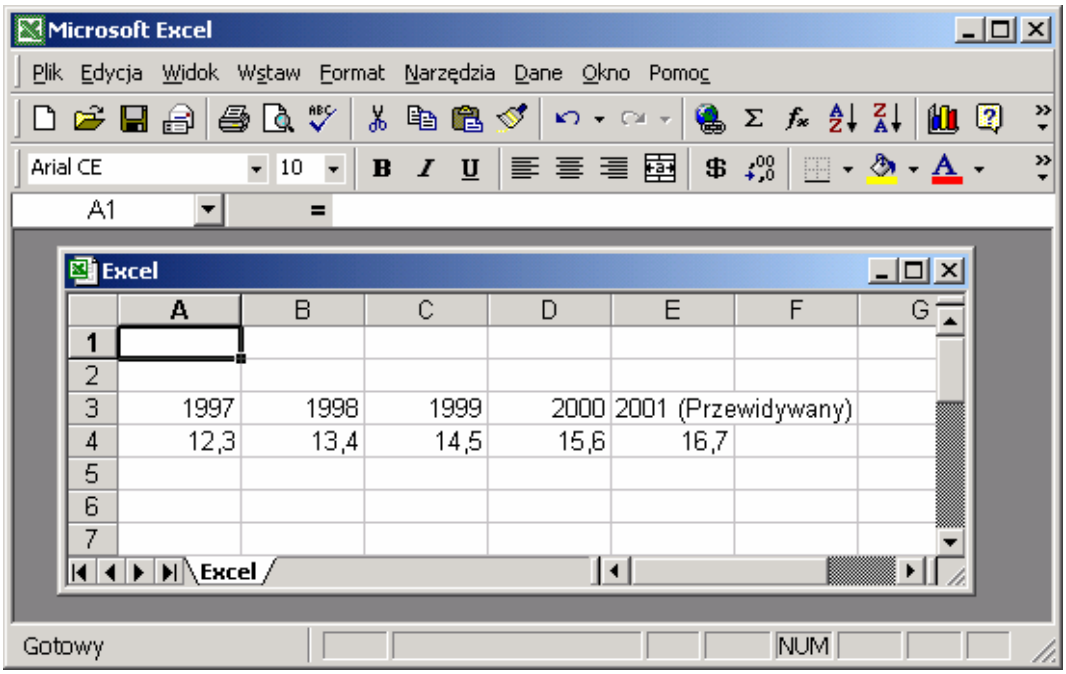
Jednym ze sposobów sformatowania zawartości arkusza jest zapisanie wartości jego poszczególnych wierszy w osobnych wierszach generowanego dokumentu i oddzielenie wartości poszczególnych komórek znakami tabulacji. Przykład prostej strony JSP generującej arkusz kalkulacyjny przedstawiłem na listingu 11.3; rysunki 11.5 oraz 11.6 przedstawiają rezultaty odwołania się do tej strony z przeglądarki Netscape Navigator działającej na komputerze, na który został zainstalowany program Microsoft Excel. Oczywiście, w prawdziwej aplikacji wartości poszczególnych komórek tego arkusza kalkulacyjnego byłyby generowane dynamicznie, być może przy użyciu wyrażeń JSP lub skryptletów odwołujących się do informacji przechowywanych w bazie danych i pobieranych przy wykorzystaniu JDBC (więcej informacji na temat JDBC znajdziesz w rozdziale 18).

Listing 11.3 Excel.jsp

```
<%@ page contentType="application/vnd.ms-excel" %>
<%-- Zwróć uwagę, iż wartości komórek są oddzielone
     znakami tabulacji a nie odstępami (spacjami). --%>
1997   1998   1999   2000   2001 (Przewidywany)
12.3   13.4   14.5   15.6   16.7
```



Rysunek 11.5 Jeśli używasz domyślnych ustawień przeglądarki, to Netscape Navigator zapyta się co należy zrobić z pobranym arkuszem kalkulacyjnym



Rysunek 11.6 Wyniki odwołania się do strony Excel.jsp, na komputerze, na którym jest zainstalowany program Microsoft Excel

Zawartość arkusza kalkulacyjnego można także przedstawić w formie zwyczajnej tabeli HTML, gdyż najnowsze wersje programu Microsoft Excel są w stanie poprawnie zinterpretować taką tabelę, o ile zostanie podany odpowiedni typ MIME. Możliwość ta podsuwa prosty pomysł polegający na zwracaniu bądź to dokumentu HTML bądź arkusza kalkulacyjnego, w zależności od preferencji użytkownika. W obu przypadkach należy użyć tej samej tabeli HTML, a jeśli użytkownik zażąda zwrócenia arkusza kalkulacyjnego, trzeba będzie określić typ wyników jako

application/vnd.ms-excel. Niestety, w trakcie implementacji takiego rozwiązania wychodzą na jaw pewne braki dyrektywy page. Otóż wartości jej atrybutów nie mogą być obliczane w czasie wykonywania strony, a sama dyrektywa nie może być wykonywana warunkowo (w sposób przypominający warunkową generację tekstu szablonów). A zatem, przedstawiony poniżej fragment strony JSP spowoduje, że zawsze będzie generowany arkusz kalkulacyjny, niezależnie do wyników zwróconych przez metodę checkUserRequest:

```
<% boolean uzyjExcela = checkUserRequest(request); %>
<% if (uzyjExcela) { %>
<%@ page contentType="application/vnd.ms-excel" %>
<% } %>
```

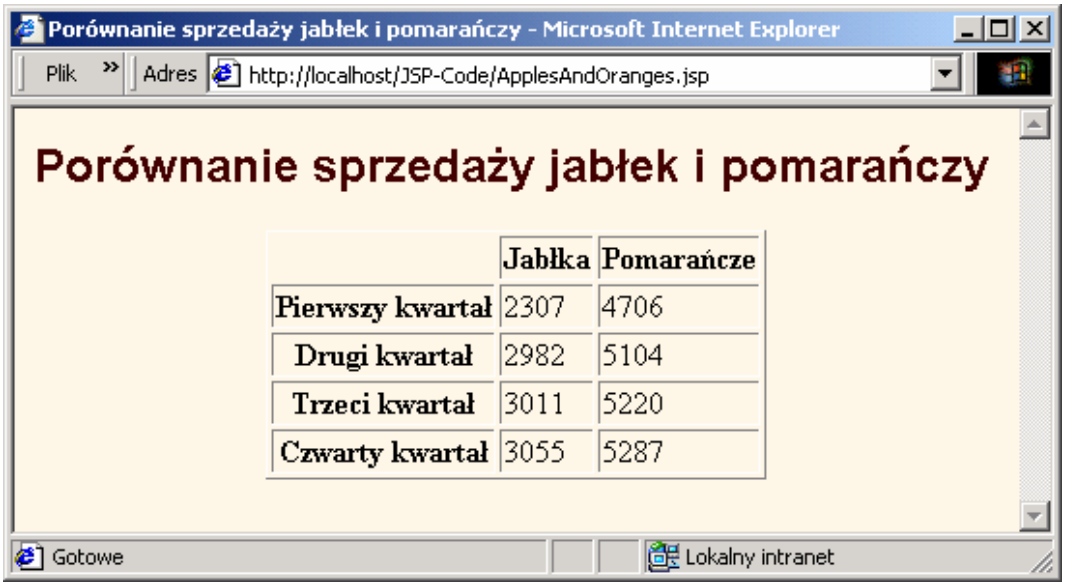
Na szczęście problem warunkowego określania typu generowanych wyników można rozwiązać w bardzo prosty sposób — wystarczy użyć skryptletu oraz metody określania nagłówka Content-Type znanej z serwletów. Oto przykład:

```
<%
String format = request.getParameter("format");
if ((format != null) && (format.equels("excel"))) {
  response.setContentType("application/vnd.ms-excel");
}
%>
```

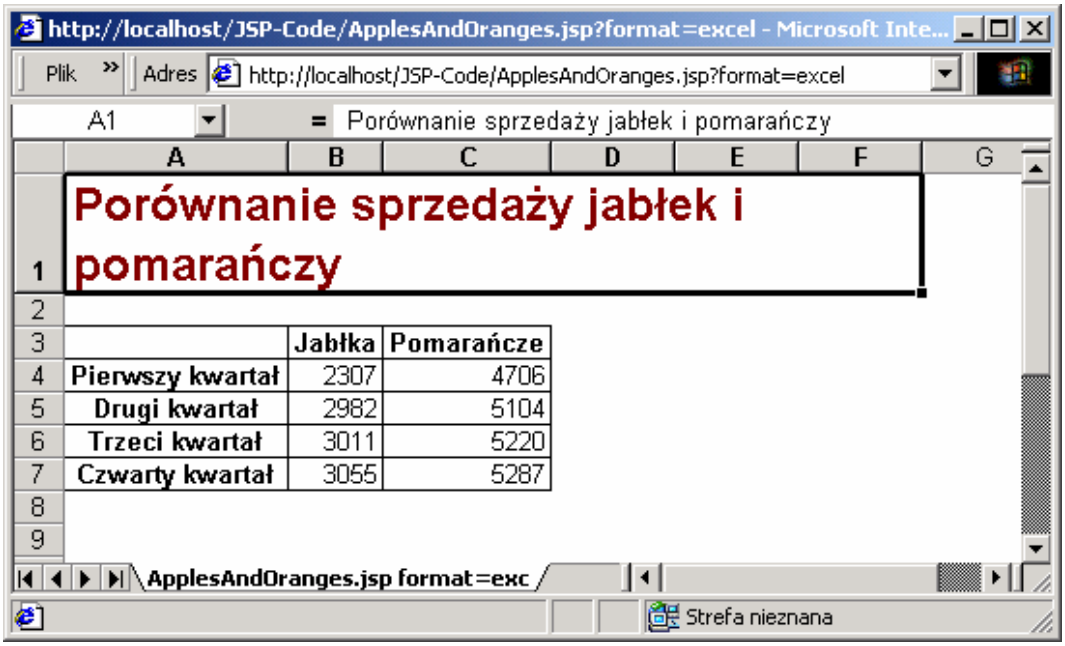
Powyższa metoda została wykorzystana na stronie JSP przedstawionej na listingu 11.4. Rysunki 11.7 oraz 11.8 przedstawiają wyniki jej wykonania w Internet Explorerze. Także w tym przypadku, w normalnej aplikacji dane były generowane dynamicznie. Bardzo prosty przykład tworzenia tabel HTML (których można używać jako dokumentu HTML bądź arkusza kalkulacyjnego), na podstawie informacji pochodzących z bazy danych, zaprezentowałem w podrozdziale 18.3. — „Narzędzia ułatwiające korzystanie z JDBC”.

Listing 11.4 ApplesAndOranges.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Porównanie sprzedaży jabłek i pomarańczy</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<CENTER>
<H2>Porównanie sprzedaży jabłek i pomarańczy</H2>

<%
String format = request.getParameter("format");
if ((format != null) && (format.equals("excel"))) {
  response.setContentType("application/vnd.ms-excel");
}
%>

<TABLE BORDER=1>
  <TR><TH></TH><TH>Jabłka<TH>Pomarańcze
  <TR><TH>Pierwszy kwartał<TD>2307<TD>4706
  <TR><TH>Drugi kwartał<TD>2982<TD>5104
  <TR><TH>Trzeci kwartał<TD>3011<TD>5220
  <TR><TH>Czwarty kwartał<TD>3055<TD>5287
</TABLE>

</CENTER>
</BODY>
</HTML>
```



Rysunek 11.7 Domyślnie strona ApplesAndOranges.jsp generuje dokument HTML



Rysunek 11.8 W przypadku użycia parametru format=excel, strona ApplesAndOranges.jsp wygeneruje arkusz kalkulacyjny programu Microsoft Excel

11.3 Atrybut isThreadSafe

Atrybut `isThreadSafe` dyrektywy `page` określa czy serwlet wygenerowany na podstawie strony JSP będzie implementować interfejs `SingleThreadModel` czy nie. Atrybutowi `isThreadSafe` można przypisać dwie wartości:

```
<%@ page isThreadSafe="true" %> <%-- sposób domyślny --%>
<%@ page isThreadSafe="false" %>
```

W standardowych serwletach jednoczesne odwołania do nich spowodują utworzenie wielu wątków, które będą współbieżnie wykonywać metodę `service` jednej kopii serwletu. Zakłada się przy tym, że serwlet został napisany w sposób „wielowątkowy” (umożliwiający takie współbieżne działanie); to znaczy, że serwlet synchronizuje dostęp do wartości pól, dzięki czemu nieokreślona

kolejność wykonywania poszczególnych wątków nie wpłynie na spójność informacji przechowywanych w polach serwletu. W niektórych przypadkach (przykładowo, podczas tworzenia liczników odwiedzin) fakt, że dwóch użytkowników pomyłkowo uzyska te same informacje nie będzie miał większego znaczenia; jednak w innych — na przykład przy operowaniu na identyfikatorach użytkowników — użycie tych samych wartości może doprowadzić do tragedii. Na przykład, przedstawiony poniżej fragment kodu nie jest „bezpieczny z punktu widzenia wielowątkowego wykonywania kodu", gdyż wykonywanie wątku może zostać przerwane po odczytaniu wartości zmiennej `idNum` jednak przed jej aktualizacją, co może doprowadzić do utworzenia dwóch użytkowników o tym samym identyfikatorze.

```
<%! private int idNum = 0; %>
<%
String userID = "userID" + idNum;
out.println("Twój identyfikator to: " + userID + "." );
idNum = idNum + 1;
%>
```

W tym przypadku, kod operujący na zmiennej `idNum` powinien zostać umieszczony w bloku `synchronized`, którego składnię przedstawiłem poniżej:

```
synchronized(jakiśObiekt) { ... }
```

Oznacza to, że jeśli jakiś wątek rozpocznie wykonywanie kodu umieszczonego wewnątrz bloku, to żaden inny wątek nie będzie mógł rozpocząć wykonywania tego samego bloku (ani żadnego innego bloku zdefiniowanego przy użyciu tego samego obiektu) aż do momentu, gdy pierwszy wątek zakończy wykonywanie bloku. A zatem, przedstawiony powyżej fragment kodu powinien zostać zapisany w następujący sposób:

```
<%! private int idNum = 0; %>
<%
synchronized(this) {
  String userID = "userID" + idNum;
  out.println("Twój identyfikator to: " + userID + "." );
  idNum = idNum + 1;
}
%>
```

Taki jest standardowy sposób działania serwletów — wiele jednocześnie odebranych żądań, jest obsługiwanych przez wiele wątków, które współbieżnie korzystają z jednej kopii serwletu. Jednak, jeśli serwlet implementuje interfejs `SingleThreadModel`, to system gwarantuje, że nie będzie współbieżnych odwołań do tej samej kopii serwletu. System może spełnić to założenie na dwa sposoby. Pierwszy z nich polega na kolejkowaniu żądań i kolejnym przekazywaniu ich do tej samej kopii serwletu. Drugi sposób bazuje na stworzeniu grupy kopii danego serwletu, przy czym każda z nich w danej chwili obsługuje tylko jedno żądanie.

Dyrektywa `<%@ page isThreadSafe="false" %>` oznacza, że kod strony nie jest bezpieczny z punktu widzenia działania wielowątkowego i dlatego wynikowy serwlet powinien implementować interfejs `SingleThreadModel`. (Więcej informacji na temat interfejsu `SingleThreadModel` znajdziesz w podrozdziale 2.6. — „Cykl życiowy serwletów"). Domyślnie, atrybut `isThreadSafe` dyrektywy `page` ma wartość `true`. Oznacza to, że system zakłada, iż programista napisał kod w sposób bezpieczny z punktu widzenia działania wielowątkowego, dzięki czemu można wykorzystać bardziej efektywny sposób obsługi żądań, polegający na tworzeniu wielu wątków jednocześnie odwołujących się do jednego egzemplarza serwletu. Jeśli w serwlecie są wykorzystywane zmienne instancyjne służące do przechowywania trwałych informacji, to dyrektywy `<%@ page isThreadSafe="false" %>` należy używać z wielką ostrożnością. W szczególności zauważ, że mechanizmy obsługi serwletów mogą (lecz nie muszą) w takim przypadku tworzyć wiele kopii serwletu, przez co nie ma żadnej pewności że wartości zmiennych instancyjnych będą unikalne. Oczywiście, w takim przypadku rozwiązaniem jest zastosowanie statycznych (`static`) zmiennych instancyjnych.

11.4 Atrybut session

Atrybut `session` dyrektywy `page` określa czy dana strona JSP będzie należała do sesji HTTP. Atrybut ten może przybierać dwie wartości:

```
<%@ page session="true" %> <%-- wartość domyślna --%>
<%@ page session="false" %>
```

Przypisanie atrybutowi `session` wartości `true` (domyślnej) oznacza, że predefiniowana zmienna `session` (typu `HttpSession`) powinna zostać skojarzona z istniejącą sesją jeśli taka istnieje, a jeśli żadnej sesji jeszcze nie ma, to należy ją utworzyć i skojarzyć ze zmienną `session`. Przypisanie atrybutowi wartości `false` sprawi, że sesje nie będą automatycznie używane, a wszelkie próby odwołania się do zmiennej `session` spowodują powstanie błędów podczas przekształcania dokumentu JSP do postaci serwletu.

11.5 Atrybut buffer

Atrybut `buffer` dyrektywy `page` określa wielkość bufor wyjściowego używanego przez predefiniowaną zmienną `out` (klasy `JspWriter`, będącej zmodyfikowaną wersją klasy `PrintWriter`). Wartość tego atrybutu można określać na dwa sposoby:

```
<%@ page buffer="wielkośćkb" %>
<%@ page buffer="none" %>
```

Serwer może używać buforu o rozmiarze większym niż podany, nie jest natomiast dopuszczalne użycie mniejszego buforu. Na przykład, użycie dyrektywy o postaci `<%@ page buffer="32kb" %>` oznacza, że treść generowanego dokumentu powinna być buforowana; przy czym zawartość buforu ma być przesyłana do przeglądarki dopiero gdy zostanie zgromadzonych 32 kb informacji, chyba że wcześniej generacja strony zostanie zakończona. Domyślny rozmiar buforu zależy od używanego serwera, jednak nie może być on mniejszy od 8 kb. Jeśli chcesz wyłączyć buforowanie wyników, będziesz musiał zachować dużą ostrożność, gdyż w takim przypadku wszystkie próby określania nagłówków odpowiedzi oraz kodu statusu w stronach JSP, muszą być wykonywane zanim zostanie wygenerowana jakakolwiek treść wynikowego dokumentu.

11.6 Atrybut autoflush

Atrybut `autoflush` określa czy w momencie wypełnienia buforu jego zawartość ma zostać automatycznie przesłana do przeglądarki, czy też ma zostać zgłoszony wyjątek. Atrybutowi temu można przypisać dwie wartości:

```
<%@ page autoflush="true" %> <%-- wartość domyślna --%>
<%@ page autoflush="false" %>
```

W przypadku przypisania wartości `none` atrybutowi `buffer` dyrektywy `page`, przypisanie wartości `false` atrybutowi `autoflush` nie jest dozwolone.

11.7 Atrybut extends

Atrybut `extends` określa klasę bazową serwletu jaki zostanie wygenerowany na podstawie strony JSP. Jego wartość definiuje się w następujący sposób:

```
<%@ page extends="pakiet.klasa" %>
```

Atrybutu tego należy używać z niezwykłą ostrożnością, gdyż serwer może wykorzystywać własną klasę bazową serwletów.

11.8 Atrybut info

Atrybut `info` określa łańcuch znaków jaki zostanie zwrócony przez serwlet w wyniku wywołania metody `getServletInfo`. Wartość tego atrybutu podaje się w następujący sposób:

```
<%@ page info="jakieś informacje o serwlecie " %>
```

11.9 Atrybut errorPage

Atrybut `errorPage` dyrektywy `page` określa stronę JSP, która ma zostać użyta do obsługi zgłoszonych wyjątków (na przykład, obiektów klas potomnych klasy `Throwable`), które nie zostały obsłużone przez bieżącą stronę JSP. Poniżej przedstawiłem sposób określania wartości tego atrybutu:

```
<%@ page errorPage="względny_adres_URL" %>
```

Zgłoszony wyjątek będzie automatycznie dostępny na wskazanej stronie obsługi błędów, jako zmienna `exception`. Przykłady użycia tego atrybutu dyrektywy page przedstawiłem na listingach 11.5 oraz 11.6.

11.10 Atrybut isErrorPage

Atrybut `isErrorPage` określa czy bieżąca strona jest używana przez inne dokumenty JSP jako strona obsługi błędów. Atrybutowi temu można przypisać dwie wartości:

```
<%@ page isErrorPage="true" %>
<%@ page isErrorPage="false" %> <%-- wartość domyślna --%>
```

Listing 11.5 przedstawia przykład strony JSP obliczającej szybkość na podstawie parametrów określających przejechany dystans i czas. Strona w żaden sposób nie sprawdza czy zostały podane wartości obu parametrów, ani czy są one poprawne. A zatem, podczas wykonywania strony może pojawić się błąd. Niemniej jednak, dyrektywa page użyta na stronie *ComputeSpeed.jsp* informuje, że wszystkie błędy jakie się na tej stronie pojawią, mają zostać obsłużone przez stronę *SpeedErrors.jsp* (jej kod przedstawiłem na listingu 11.6). Dzięki użyciu strony obsługi błędów, w przypadku ich pojawienia się użytkownik nie będzie musiał oglądać typowych informacji o błędach generowanych przez JSP. Na rysunkach 11.9 oraz 11.10 przedstawiłem wyniki wykonania strony *ComputeSpeed.jsp* w przypadku podania poprawnych oraz błędnych parametrów wejściowych.

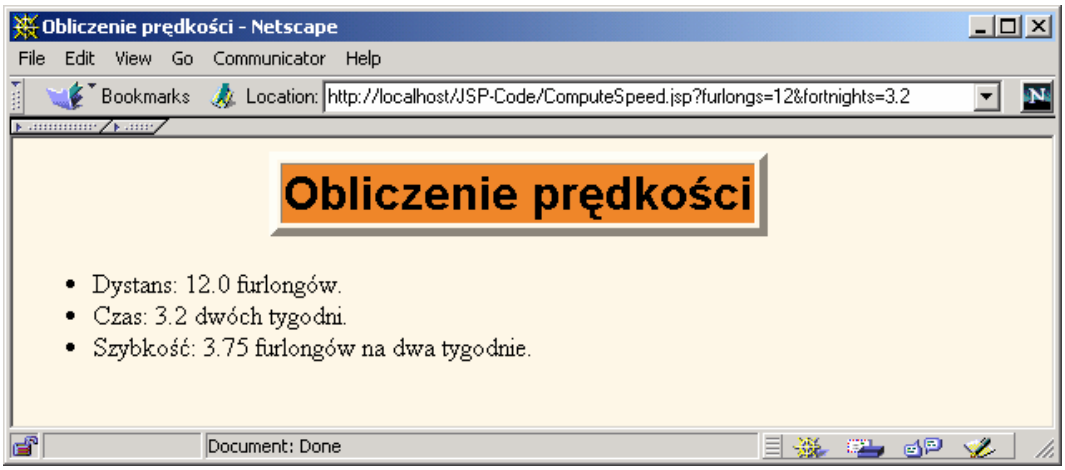
Listing 11.5 ComputeSpeed.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Obliczenie prędkości</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>

<%@ page errorPage="SpeedErrors.jsp" %>

<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      Obliczenie prędkości</TABLE>

<%!
// Zwróć uwagę na brak bloków try/catch przechwytujących
// wyjątki NumberFormatException generowanych w przypadkach
// gdy wartości parametrów nie zostaną podane lub gdy zostaną
// zapisane w niewłaściwym formacie.
```

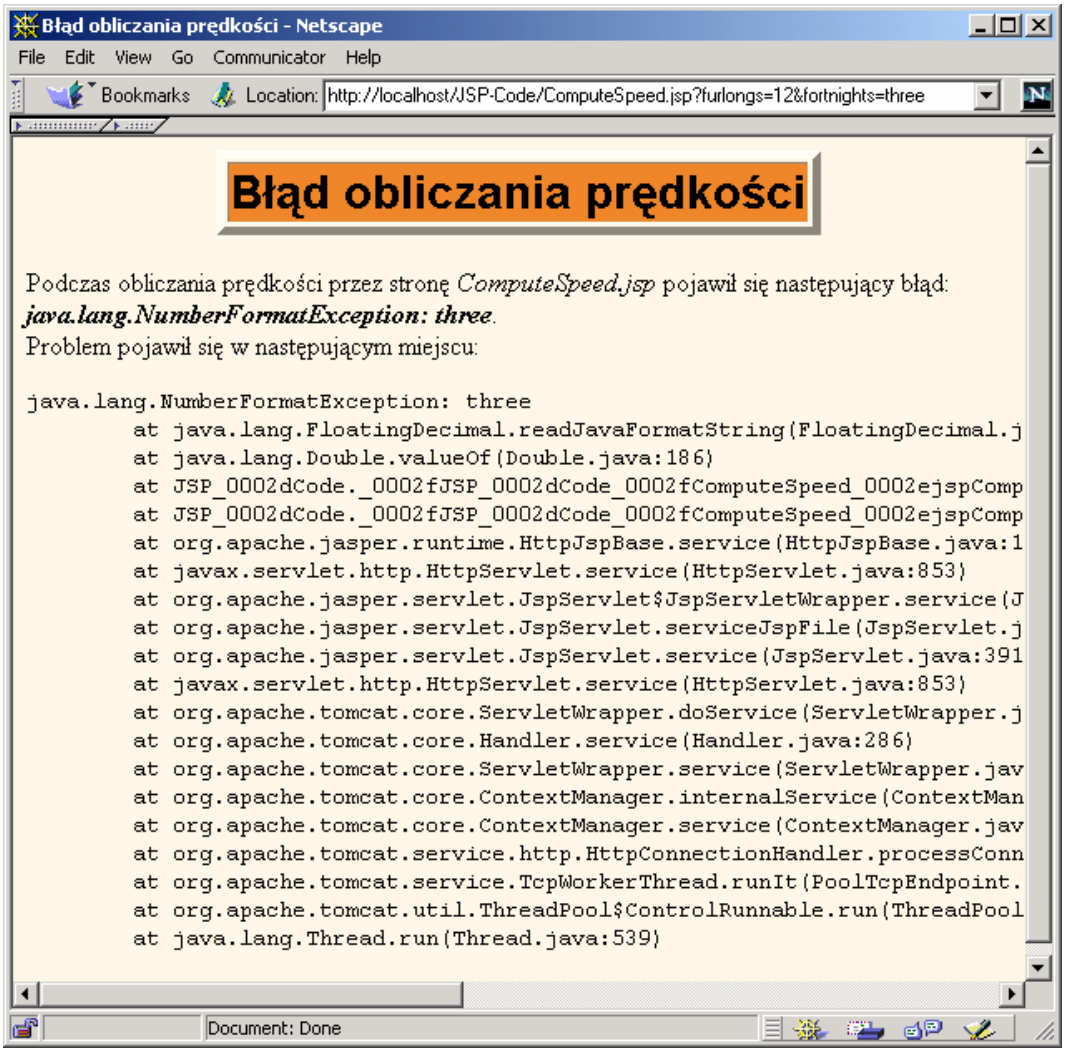
```
private double toDouble(String value) {
  return(Double.valueOf(value).doubleValue());
}
%>

<%
double furlongs = toDouble(request.getParameter("furlongs"));
double fortnights = toDouble(request.getParameter("fortnights"));
double speed = furlongs/fortnights;
%>

<UL>
  <LI>Dystans: <%= furlongs %> furlongów.
  <LI>Czas: <%= fortnights %> dwóch tygodni.
  <LI>Szybkość: <%= speed %> furlongów na dwa tygodnie.
</UL>

</BODY>
</HTML>
```

Listing 11.6 SpeedErrors.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<!--
Przykład strony obsługi błędów (ta strona jest używana przez
stronę ComputeSpeed.jsp
-->
<HTML>
<HEAD>
<TITLE>Błąd obliczania prędkości</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>

<%@ page isErrorPage="true" %>

<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      Błąd obliczania prędkości</TABLE>
<P>
Podczas obliczania prędkości przez stronę <I>ComputeSpeed.jsp</I>
pojawił się następujący błąd:
<B><I><%= exception %></I></B>. <BR>
Problem pojawił się w następującym miejscu:
<FONT SIZE="-1">
<PRE>
<% exception.printStackTrace(new PrintWriter(out)); %>
</PRE>
</FONT>

</BODY>
</HTML>
```



Rysunek 11.9 Wyniki wykonania strony ComputeSpeed.jsp po przekazaniu poprawnych parametrów



Rysunek 11.10 Wyniki wykonania strony ComputeSpeed.jsp po przekazaniu do niej nieprawidłowych danych

11.11 Atrybut language

Atrybut `language` będzie kiedyś określać zastosowany język programowania, na przykład:

```
<%@ page language="cobol" %>
```

Jak na razie jednak możesz zapomnieć o tym atrybucie, gdyż jego domyślną a jednocześnie jedyną dopuszczalną wartością jest `java`.

11.12 Składnia XML zapisu dyrektyw

JSP pozwala na podawanie dyrektyw w alternatywny — XML-owy — sposób:

```
<jsp:directive.typ_dyrektywy atrybut="wartość_atrybutu" />
```

Na przykład, oba przedstawione poniżej metody zapisu dyrektywy `page` dają ten sam efekt:

```
<%@ page import="java.util.*" %>
```

oraz

```
<jsp:directive.page import="java.util.*" />
```

Rozdział 12.
Dołączanie plików i apletów do dokumentów JSP

Technologia JSP udostępnia trzy podstawowe sposoby dołączania zewnętrznych elementów do dokumentów JSP.

Dyrektywa `include` pozwala na wykorzystywanie pasków nawigacyjnych, tabel oraz wszelkich innych elementów w wielu dokumentach. Dołączane elementy mogą zawierać kod JSP i dlatego są wstawiane zanim strona zostanie przekształcona do postaci serwletu. Zastosowanie tej dyrektywy omówię szczegółowo w podrozdziale 12.1.

Choć dołączanie elementów zawierających kod JSP daje ogromne możliwości, to jednak może się zdarzyć, że będziesz wolał poświęcić niektóre z nich z nich w zamian za wygodę jaką daje możliwość aktualizacji dołączanego elementu bez konieczności wprowadzania zmian w głównej stronie JSP. Na przykład, na witrynie WWW mojego kościoła publikowane są ogłoszenia dotyczące pomocy przy odgarnianiu śniegu. Strona jest aktualizowana w niedzielę o godzinie 6:30 rano, czyli wtedy, gdy śnieg ma być odgarniany. Nie należy oczekiwać, że to twórca witryny będzie własnoręcznie dokonywać tych aktualizacji — w tym czasie zapewne smacznie sobie śpi. Znacznie prostszym rozwiązaniem jest przesłanie na serwer zwyczajnego pliku tekstowego i umieszczenie jego zawartości na stronie przy wykorzystaniu elementu `jsp:include`. To rozwiązanie omówię w podrozdziale 12.2.

Choć niniejsza książka jest poświęcona głównie tworzeniu programów działających na serwerze, to jednak wciąż dużą rolę odgrywają aplety — niewielkie programy pisane w języku Java i wykonywane w przeglądarkach WWW. Rozwiązanie to jest najczęściej wykorzystywane w szybkich, firmowych aplikacjach intranetowych. Element `jsp:plugin` umożliwia umieszczanie na stronach JSP apletów korzystających z Java Plug-in. To rozwiązanie omówiłem w podrozdziale 12.3.

12.1 Dołączanie plików w czasie przekształcania strony

Dyrektywa `include` służy do dołączania plików do głównej strony JSP w czasie gdy jest ona przekształcana do postaci serwletu (co zazwyczaj ma miejsce po odebraniu pierwszego żądania dotyczącego tej strony). Składnia tej dyrektywy ma następującą postać:

```
<%@ include file="względy_adres_URL" %>
```

Fakt, że dyrektywa ta powoduje dołączenie pliku podczas przekształcania strony a nie podczas obsługi żądania (jak czyni znacznik akcji `jsp:include` opisany w podrozdziale 12.2), ma dwie konsekwencje.

Po pierwsze, dołączana jest zawartość wskazanego pliku. To odróżnia dyrektywę `include` od znacznika akcji `jsp:include`, którego użycie sprawia, że serwer wykonuje wskazany plik i wstawia wygenerowane przez niego *wyniki*. Oznacza to, że stosując dyrektywę include można dołączać kod JSP (na przykład, deklaracje pól bądź metod), który będzie miał wpływ na wyniki wykonania całej strony.

Po drugie, jeśli dołączany plik zostanie zmieniony, to także trzeba będzie zmodyfikować wszystkie strony JSP, w których jest on używany. Serwer mogą wykrywać kiedy dołączany plik zostanie zmieniony (i w rezultacie automatycznie rekompilować serwlet), lecz niestety działanie takie nie jest wymagane. W praktyce, bardzo niewiele serwerów udostępnia tą możliwość. Co więcej, nie ma żadnego prostego sposobu na wydanie polecenia „a teraz skompiluj tę stronę JSP". Pewnym rozwiązaniem jest zmiana daty modyfikacji dokumentu JSP. Niektóre systemy operacyjne udostępniają polecenie pozwalające na określenie daty modyfikacji pliku, bez konieczności jego edycji (na przykład, w systemach uniksowych jest to polecenie `touch`). Jednak najprostszym rozwiązaniem jest umieszczenie na początku głównej strony JSP komentarza JSP. Komentarz ten należy zmieniać zawsze, gdy zostanie zmodyfikowany jeden z plików dołączanych do strony. W komentarzu tym możesz, na przykład, umieścić datę modyfikacji strony, jak pokazałem na poniższym przykładzie:

```
<%-- Navbar.jsp zmodyfikowany 3.2.2001 --%>
<%@ include file="Navbar.jsp" %>
```

Ostrzeżenie
Jeśli zmienisz plik dołączany plik JSP, to będziesz musiał uaktualnić daty modyfikacji wszystkich dokumentów JSP, które używają tego pliku.

Na listingu 12.1 przedstawiłem przykład fragmentu strony, który zawiera informacje o kontakcie oraz statystykę odwiedzin bieżącej strony. Fragment ten może być umieszczany u dołu wszystkich stron w obrębie danej witryny. Listing 12.2 przedstawia stronę, która dołącza kod z listingu 12.1; wyniki jej wykonania pokazałem na rysunku 12.1.

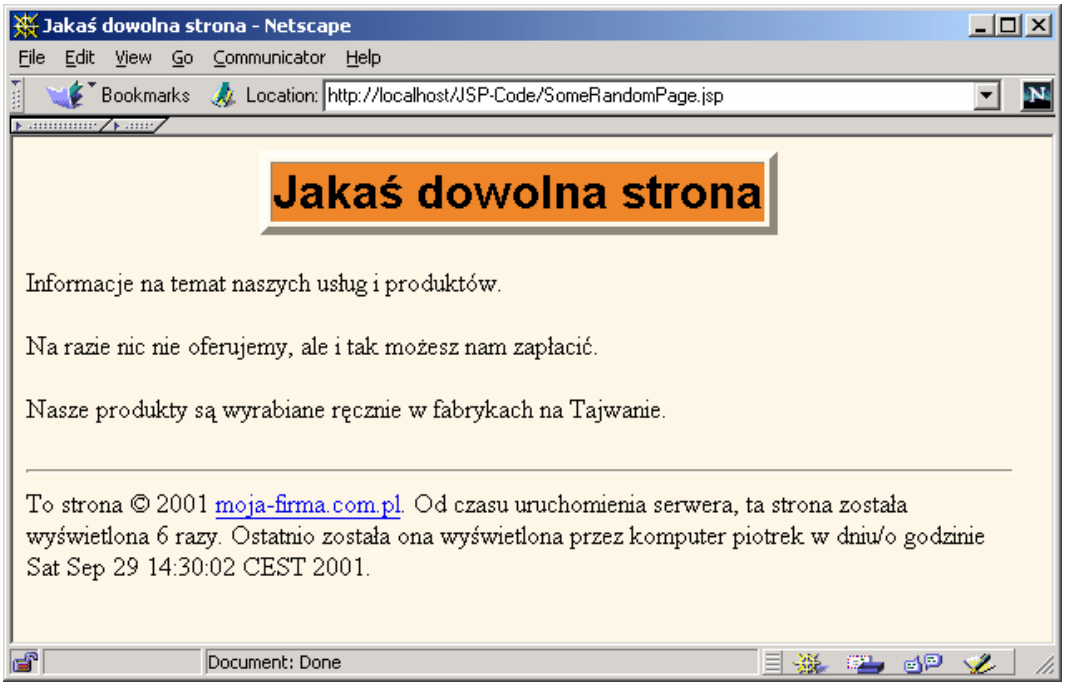
Listing 12.1 ContactSection.jsp

```
<%@ page import="java.util.Date" %>

<%-- Poniższe zmienne staną się częścią każdego serwletu
          wygenerowanego na podstawie strony JSP do której
          niniejszy plik zostanie dołączony --%>
<%!
private int accessCount = 0;
private Date accessDate = new Date();
private String accessHost = "<I>brak danych</I>";
%>

<P>
<HR>
To strona &copy; 2001
<A HREF="http//www.moja-firma.com.pl/">moja-firma.com.pl</A>.
Od czasu uruchomienia serwera, ta strona została wyświetlona
<%= ++accessCount %> razy. Ostatnio została ona wyświetlona przez
komputer <%= accessHost %> w dniu/o godzinie <%= accessDate %>.

<% accessHost = request.getRemoteHost(); %>
<% accessDate = new Date(); %>
```

Listing 12.2 SomeRandomPage.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Jakaś dowolna strona</TITLE>
<META NAME="author" CONTENT="J. Super Hacker">
<META NAME="keywords"
      CONTENT="glupoty,nie ważne,coś tam">
<META NAME="description"
```

```
      CONTENT="Dowolna strona.">
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      Jakaś dowolna strona</TABLE>
<P>
Informacje na temat naszych usług i produktów.
<P>
Na razie nic nie oferujemy, ale i tak możesz nam zapłacić.
<P>
Nasze produkty są wyrabiane ręcznie w fabrykach na Tajwanie.

<%@ include file="ContactSection.jsp" %>

</BODY>
</HTML>
```



Rysunek 12.1 Wyniki wykonania strony SomeRandomPage.jsp

12.2 Dołączanie plików podczas obsługi żądań

Dyrektywa `include` przedstawiona w poprzedniej części rozdziału pozwala na dołączanie dokumentów zawierających kod JSP, do wielu różnych stron. Dołączanie kodu JSP jest bardzo przydatną możliwością, lecz dyrektywa `include` narzuca konieczność aktualizowania daty modyfikacji strony, za każdym razem gdy zmieni się jeden z dołączanych do niej plików. Jest to dość znaczące utrudnienie. Znacznik akcji `jsp:include` powoduje dołączanie plików w czasie obsługi żądania, dzięki czemu nie stwarza konieczności aktualizacji głównej strony, gdy zostanie zmieniona zawartość jednego z dołączanych do niej plików. Z drugiej strony, w czasie obsługi żądania strona JSP została już przekształcona do postaci serwletu, a zatem dołączany plik nie może zawierać kodu JSP.

Metoda

Jeśli dołączane pliki mają zawierać kod JSP, to powinieneś skorzystać z dyrektywy `include`. W pozostałych przypadkach użyj znacznika akcji `jsp:include`.

Choć dołączany plik nie może *zawierać* kodu JSP, to jednak może on zostać wygenerowany przez zasoby *używające* JSP. Oznacza to, że dołączany zasób, do którego odwołuje się podany adres URL, jest w normalny sposób interpretowany przez serwer — czyli może być serwletem lub stroną JSP. Właśnie w taki sposób działa metoda `include` klasy `RequestDispatcher`, która jest używana przez serwlety w celu dołączania plików. Szczegółowe informacje na ten temat znajdziesz w podrozdziale 15.3. — „Dołączanie danych statycznych bądź dynamicznych".

Znacznik akcji `jsp:include` wymaga podania dwóch atrybutów (patrz poniższy przykład) — `page` (zawierającego względny adres URL dołączanego pliku) oraz `flush` (atrybut ten musi mieć wartość `true`).

```
<jsp:include page="względny_URL" flush="true" />
```

Choć zazwyczaj będziesz w ten sposób dołączał dokumenty HTML i pliki tekstowe, to jednak nie ma żadnych ograniczeń dotyczących rozszerzenia dołączanego pliku. W serwerze Java Web Server 2.0 jest jednak błąd, który powoduje przerwanie przetwarzania strony jeśli dołączany plik nie ma rozszerzenia *.html* lub *.htm* (czyli, jeśli jest to na przykład plik tekstowy z rozszerzeniem *.txt*). Serwery Tomcat oraz JSWDK, jak również większość serwerów komercyjnych nie ma takich ograniczeń.

Ostrzeżenie
Ze względu na błąd, Java Web Server pozwala wyłącznie na dołączanie plików z rozszerzeniami *.html* oraz *.htm*.

Na listingu 12.3 przedstawiłem prosty przykład strony zawierającej krótkie informacje na temat opublikowanych nowości. Twórcy witryny mogą zmieniać informacje o nowościach zamieszczone w plikach do *Item1.html* do *Item4.html* (patrz listingi 12.4 do 12.7). Zmiana zawartości któregokolwiek z tych plików nie pociąga za sobą konieczności zmiany strony, do której są one dołączane. Wyniki działania strony *WhatsNew.jsp* przedstawiłem na rysunku 12.2.

Listing 12.3 WhatsNew.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<%@ page contentType="text/html; charset=ISO-8859-2" %>
<HTML>
<HEAD>
<TITLE>Co nowego</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>

<CENTER>
<TABLE BORDER=5>
  <TR><TH CLASS="TITLE">
      Co nowego na witrynie NowościJsp.com</TABLE>
</CENTER>
<P>

Oto krótkie podsumowanie czterech najważniejszych spośród
wielu nowych informacji zamieszczonych na naszej witrynie:
<OL>
  <LI><jsp:include page="news/Item1.html" flush="true" />
  <LI><jsp:include page="news/Item2.html" flush="true" />
  <LI><jsp:include page="news/Item3.html" flush="true" />
  <LI><jsp:include page="news/Item4.html" flush="true" />
</OL>
</BODY>
</HTML>
```

Listing 12.4 Item1.html

```
<B>Pokorny Bill Gates.</B> Wczoraj, w sposób całkowicie zaskakujący
i niespodziewany, prezes firmy Microsoft Bil Gates zaprezentował
akt bezinteresownego humanitaryzmu.
```

```
<A HREF="http://www.microsoft.com/Never.html">Więcej informacji...</A>
```

Listing 12.5 Item2.html

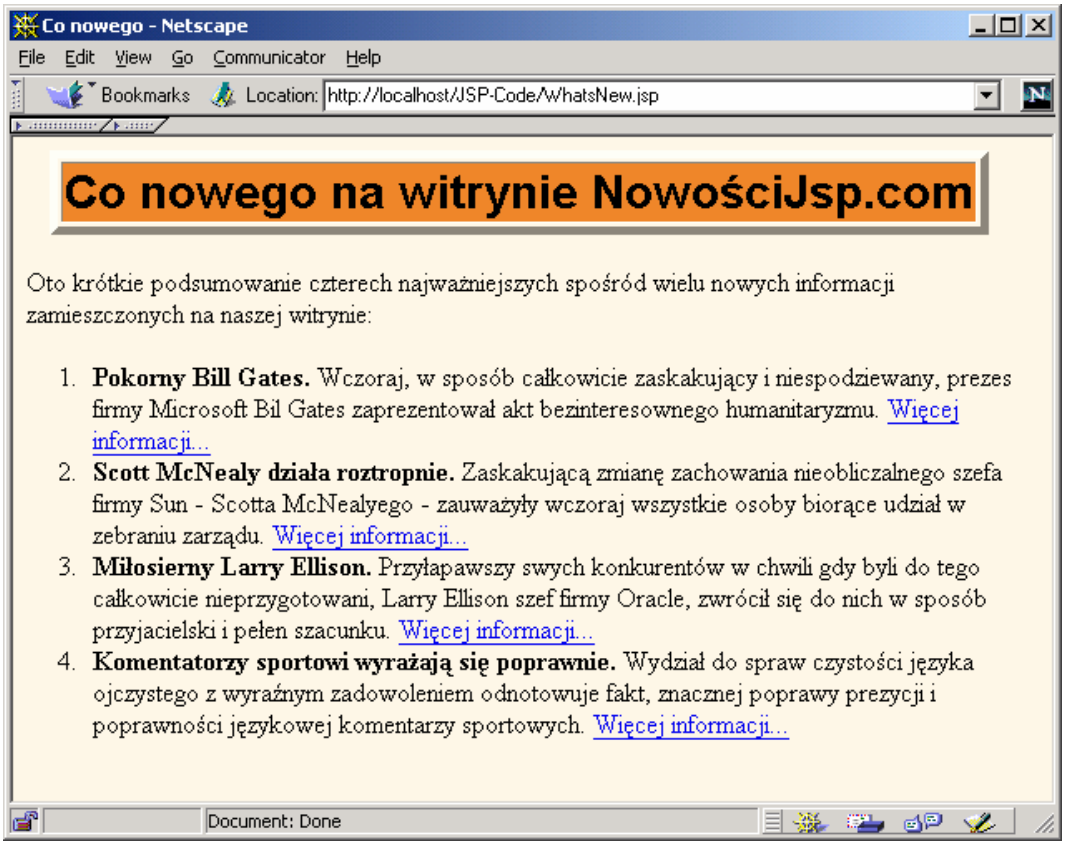
```
<B>Scott McNealy działa roztropnie.</B> Zaskakującą zmianę
zachowania nieobliczalnego szefa firmy Sun -
Scotta McNealyego - zauważyły wczoraj wszystkie osoby biorące
udział w zebraniu zarządu.
<A HREF="http://www.sun.com/Imposter.html">Więcej informacji...</A>
```

Listing 12.6 Item3.html

```
<B>Miłosierny Larry Ellison.</B> Przyłapawszy swych konkurentów
w chwili gdy byli do tego całkowicie nieprzygotowani, Larry
Ellison szef firmy Oracle, zwrócił się do nich w sposób
przyjacielski i pełen szacunku.
<A HREF="http://www.oracle.com/Mistake.html">Więcej informacji...</A>
```

Listing 12.7 Item4.html

```
<B>Komentatorzy sportowi wyrażają się poprawnei.</B>
Wydział do spraw czystości języka ojczystego z wyraźnym
zadowoleniem odnotowuje fakt, znacznej poprawy prezycji
i poprawności językowej komentarzy sportowych.
<A HREF="http://www.espn.com/Slip.html">Więcej informacji...</A>
```



Rysunek 12.2 Wyniki wykonania strony WhatsNew.jsp

Dołączanie apletów korzystających z Java Plug-In

Aby umieszczać zwyczajne aplety na stronach JSP nie musisz stosować żadnej specjalnej składni — wystarczy posłużyć się standardowym elementem APPLET. Jednak aplety umieszczane na

stronach w taki sposób muszą korzystać z JDK 1.1 bądź 1.02, gdyż ani Netscape Navigator 4.x ani Internet Explorer 5.x nie są w stanie korzystać z platformy Java 2 (czyli także z JDK 1.2). Aplety takie mają kilka poważnych ograniczeń:

- aby korzystać z klas pakietu Swing, należy przesyłać wymagane pliki klasowe siecią. Proces ten jest długotrwały, a co gorsze nie można go wykonać w Internet Explorerze 3 oraz Netscape Navigatorze 3 .x oraz 4.01 – 4.05, gdyż przeglądarki te obsługują wyłącznie JDK 1.02, a Swing wymaga JDK 1.1,
- nie można używać technologii Java 2D,
- nie można korzystać z kolekcji dostępnych w Java 2.
- kod apletów jest wykonywany wolniej, gdyż większość kompilatorów platformy Java 2 zostało znacznie usprawnionych w porównaniu z ich wcześniejszymi wersjami.

Co więcej, dawne wersje przeglądarek obsługiwały różne komponenty AWT w niespójny sposób, przez co tworzenie apletów z zaawansowanym i skomplikowany interfejsem użytkownika była znacznie trudniejsze i uciążliwe niż mogło by być. Aby rozwiązać ten problem, firma Sun stworzyła specjalny plug-in przeznaczony dla przeglądarek Netscape Navigator i Internet Explorer, który pozwala na wykorzystywanie w apletach technologii dostępnych w platformie Java 2. Plug-in ten można pobrać z witryny firmy Sun — *http://java.sun.com/products/plugin*; jest on także dołączany do JDK w wersji 1.2.2 oraz następnych. Java Plug-in jest całkiem duży — ma wielkość kilku megabajtów — a zatem raczej nie należy oczekiwać, aby przeciętni użytkownicy WWW chcieli go pobierać i instalować tylko po to, aby móc wykonywać Twoje aplety. Jednak z drugiej strony, jego wykorzystanie jest całkiem rozsądną alternatywą dla szybkich, korporacyjnych intranetów; zwłaszcza, iż w przypadku jego braku, aplety mogą automatycznie zażądać od przeglądarki, aby pobrała go z serwera.

Niestety, normalny znacznik `APPLET` nie daje możliwości użycia plug-inu. Wynika to z faktu, iż przeglądarki są tworzone w taki sposób, by aplety umieszczane na stronach przy użyciu tego znacznika mogły być wykonywane wyłącznie przez wirtualne maszyny Javy wbudowane w przeglądarkę. A zatem, zamiast znacznika `APPLET` należy stosować inne rozwiązania — w Internet Explorerze stosowany jest długi i zawiły znacznik `OBJECT`, a w Netscape Navigatorze równie długi znacznik `EMBED`. Co grosze, zazwyczaj nie można z góry wiedzieć jaka przeglądarka zostanie użyta do pobrania strony. A zatem, konieczne jest umieszczenie w stronie obu znaczników jednocześnie (przy czym znacznik `EMBED` umieszczany jest w sekcji `COMMENT` znacznika `OBJECT`) lub określenie typu używanej przeglądarki na podstawie nagłówków żądania i wygenerowanie odpowiedniego znacznika. Choć napisanie odpowiedniego kodu nie nastręcza żadnych trudności, to jednak rozwiązanie takie jest męczące i czasochłonne.

Znacznik akcji `jsp:plugin` informuje serwer, iż należy wygenerować odpowiedni znacznik, który umieści aplet na stronie i wykona go przy użyciu plug-inu. Serwery nie mają narzuconego żadnego konkretnego sposobu udostępnienia powyższych możliwości funkcjonalnych, jednak większość z nich stosuje znaczniki `OBJECT` oraz `EMBED`.

Znacznik akcji jsp:plugin

W najprostszym przypadku znacznik akcji `jsp:plugin` wymaga podania czterech atrybutów — `type`, `code`, `width` oraz `height`. Atrybutowi `type` należy przypisać wartość `applet`, natomiast znaczenie i działanie pozostałych atrybutów jest takie samo jak w przypadku znacznika `APPLET`. Pamiętaj jednak o dwóch sprawach — w nazwach atrybutów uwzględniana jest wielkość liter, a wartości atrybutów zawsze muszą być umieszczone pomiędzy cudzysłowami lub apostrofami. A zatem, poniższy znacznik `APPLET`

```
<APPLET CODE="MojApplet.class"
        WIDTH=475 HEIGHT=350>
</APPLET>
```

można zastąpić znacznikiem

```
<jsp:plugin type="applet"
            code="MojApplet.class"
            width="475" height="350">
</jsp:plugin>
```

W znaczniku `jsp:plugin` można dodatkowo podać wiele innych, opcjonalnych atrybutów, których większość (choć nie wszystkie) odpowiadają atrybutom znacznika `APPLET`. Poniżej podałem pełną listę atrybutów znacznika `jsp:plugin`:

- **`type`**

  W przypadku apletów, atrybut ten powinien mieć wartość `applet`. Niemniej jednak Java Plug-In pozwala na umieszczanie na stronach WWW komponentów JavaBean. W takim przypadku atrybutowi temu należy przypisać wartość `bean`.

- **`code`**

  Ten atrybut w pełni odpowiada atrybutowi `CODE` znacznika `APPLET`. Określa on nazwę pliku klasowego głównej klasy apletu, będącej klasą potomną klasy `Applet` lub `JApplet`. Pamiętaj, że w znaczniku `jsp:plugin` nazwa `code` musi być zapisana małymi literami (ze względu na składnię języka XML). W przypadku znacznika `APPLET` wielkość liter w nazwach atrybutów nie ma znaczenia (gdyż język HTML nie zwraca uwagi na wielkości liter jakimi są zapisywane znaczniki i ich atrybuty).

- **`width`**

  Ten atrybut działa tak samo jak atrybut `WIDTH` znacznika `APPLET`. Określa on szerokość obszaru zarezerwowanego dla apletu, wyrażoną w pikselach. Pamiętaj, że wartość tego atrybutu należy zapisać w cudzysłowach lub apostrofach.

- **`height`**

  Ten atrybut działa tak samo jak atrybut `HEIGHT` znacznika `APPLET` i określa wysokość obszaru zarezerwowanego dla apletu, wyrażoną w pikselach. Pamiętaj, że wartość tego atrybutu należy zapisać w cudzysłowach lub apostrofach.

- **`codebase`**

  Ten atrybut działa tak samo jak atrybut `CODEBASE` znacznika `APPLET` i określa katalog bazowy apletów. Wartość atrybutu `code` podawana jest względem katalogu określonego przy użyciu atrybutu `codebase`. Podobnie jak w przypadku znacznika `APPLET`, także i tutaj, jeśli wartość tego atrybutu nie zostanie określona, to domyślnie zostanie użyty katalog bieżącej strony WWW. W przypadku JSP, jest to katalog w którym jest umieszczona oryginalna strona JSP, a nie katalog w jakim na danym serwerze są przechowywane pliki klasowe serwletów wygenerowanych na podstawie dokumentów JSP.

- **`align`**

  Ten atrybut działa tak samo jak atrybut `ALIGN` znaczników `APPLET` oraz `IMG` i określa wyrównanie apletu na stronie WWW. Atrybut ten może przybierać następujące wartości: `left`, `right`, `top`, `bottom` oraz `middle`. Określając wartości tego atrybutu dla znacznika `jsp:plugin` nie zapomnij zapisać ich w cudzysłowach lub apostrofach, gdyż w tym przypadku znaki te nie opcjonalne (w odróżnieniu od języka HTML gdzie można je pominąć).

- **`hspace`**

  Ten atrybut działa tak samo jak atrybut `HSPACE` znacznika `APPLET` — określa (w pikselach) szerokość pustego obszaru pozostawianego z prawej oraz z lewej strony apletu. Pamiętaj, że wartość tego atrybutu należy zapisać w cudzysłowach lub apostrofach.

- **`vspace`**

  Ten atrybut działa tak samo jak atrybut `VSPACE` znacznika `APPLET` — określa (w pikselach) wysokość pustego obszaru pozostawianego powyżej oraz poniżej apletu. Pamiętaj, że wartość tego atrybutu należy zapisać w cudzysłowach lub apostrofach.

- **`archive`**

Ten atrybut działa tak samo jak atrybut ARCHIVE znacznika APPLET, czyli określa plik JAR, z którego należy pobrać pliki klasowe apletu oraz wszystkie pozostałe pliki multimedialne używane przez niego.

- **name**

  Atrybut ten działa tak samo jak atrybut NAME znacznika APPLET, czyli określa nazwę apletu identyfikującą go w przypadku wymiany informacji pomiędzy różnymi apletami oraz w językach skryptowych (takich jak JavaScript).

- **title**

  Atrybut ten działa tak samo jak bardzo rzadko stosowany atrybut TITLE znacznika APPLET (oraz niemal wszystkich pozostałych znaczników języka HTML 4.0). Określa on tytuł znacznika, który może być wyświetlany na etykietach ekranowych lub użyty przy indeksowaniu.

- **jreversion**

  Atrybut ten określa wymaganą wersję JRE (ang.: Java Runtime Environment — środowiska wykonawczego Javy). Domyślna wartość tego atrybutu to 1.1.

- **iepluginurl**

  Ten atrybut określa URL z którego można pobrać Java Plug-In przeznaczony dla Internet Explorera. Użytkownicy, którzy jeszcze nie zainstalowali tego plug-ina zostaną zapytani czy należy go pobrać spod podanego adresu. Domyślna wartość tego atrybutu umożliwia pobranie plug-ina z witryny firmy Sun, jednak w przypadku aplikacji intranetowych można zażądać pobrania jego lokalnej kopii.

- **nspluginurl**

  Ten atrybut określa URL z którego można pobrać Java Plug-In przeznaczony dla Netscape Navigatora. Użytkownicy, którzy jeszcze nie zainstalowali tego plug-ina zostaną zapytani czy należy go pobrać spod podanego adresu. Domyślna wartość tego atrybutu umożliwia pobranie plug-ina z witryny firmy Sun, jednak w przypadku aplikacji intranetowych można zażądać pobrania jego lokalnej kopii.

Znaczniki akcji jsp:param oraz jsp:params

Znacznik akcji jsp:param jest stosowany wraz ze znacznikiem jsp:plugin. Sposób jego użycia przypomina wykorzystanie znacznika PARAM stosowanego wewnątrz znacznika APPLET i określającego nazwy i wartości, które można pobierać z wnętrza apletu przy użyciu metody getParameter. Istnieją jednak pewne różnice pomiędzy tymi znacznikami. Po pierwsze, znacznik akcji jsp:param zapisywany jest według zasad składni języka XML. Oznacza to, że nazwy jego atrybutów muszą być zapisane małymi literami, wartości muszą być umieszczone pomiędzy znakami cudzysłowu lub apostrofu, a znacznik musi zostać zamknięty przy użyciu kombinacji znaków /> a nie znaku >. Poza tym, wszystkie znaczniki akcji jsp:param muszą być umieszczone wewnątrz znacznika jsp:params.

A zatem, poniższy znacznik APPLET

```
<APPLET CODE="MojApplet.class"
        WIDTH=475 HEIGHT=350>
  <PARAM NAME="Param1" VALUE="Wartość1">
  <PARAM NAME="Param2" VALUE="Wartość2">
</APPLET>
```

należałoby zastąpić znacznikiem jsp:plugin o następującej postaci:

```
<jsp:plugin type="applet"
            code="MojApplet.class"
            width="475"  height="350">
  <jsp:params>
    <jsp:param name="Param1" value="Wartość1" />
    <jsp:param name="Param2" value="Wartość2" />
  </jsp:params>
```

```
</jsp:plugin>
```

Znacznik akcji jsp:fallback

Znacznik akcji `jsp:fallback` umożliwia podanie alternatywnego tekstu, jaki zostanie wyświetlony w przeglądarkach, które nie obsługują znaczników `OBJECT` ani `EMBED`. Znacznika tego można używać niemal tak samo jak tekstu alternatywnego umieszczanego w znaczniku `APPLET`. A zatem, poniższy znacznik `APPLET`

```
<APPLET CODE="MojApplet.class"
        WIDTH=475 HEIGHT=350>
  <B>Błąd: Wykonanie tego przykładu wymaga użycia przeglądarki
     potrafiącej obsługiwać aplety.</B>
</APPLET>
```

można zastąpić w następujący sposób:

```
<jsp:plugin type="applet"
            code="MojApplet.class"
            width="475" height="350">
  <jsp:fallback>
    <B>Błąd: Wykonanie tego przykładu wymaga użycia przeglądarki
       potrafiącej obsługiwać aplety.</B>
  </jsp:fallback>
</jsp:plugin>
```

Warto zapamiętać, iż w serwerze Java Web Server 2.0 występuje błąd, który sprawia, że strony JSP zawierające ten znacznik nie są poprawnie przekształcane do postaci serwletów. Serwery Tomcat, JSWDK oraz większość serwerów komercyjnych poprawnie obsługuje ten znacznik.

Ostrzeżenie
Java Web Server nie obsługuje poprawnie znacznika `jsp:fallback`.

Przykład: Generacja tekstu z cieniem

Listingi 7.9 oraz 7.11 przedstawione w podrozdziale 7.5 (pt.: „Wykorzystanie serwletów do generacji obrazów GIF") zawierają kod okna (klasy `JFrame`) wykorzystujące technologię Java 2D do stworzenia obrazu prezentującego podany łańcuch znaków, wyświetlony czcionką o określonym kroju i wielkości oraz cień tego łańcucha. Na listingach 12.10 oraz 12.11 przedstawiłem aplet, który przy użyciu komponentów pakietu Swing używa tego okna.

Nasz przykładowy aplet wykorzystuje zarówno pakiet Swing jak i technologię Java 2D, a zatem może być wykonany wyłącznie przy użyciu Java Plug-In. Na listingu 12.8 przedstawiłem stronę JSP, która wyświetla ten aplet przy użyciu znacznika `jsp:plugin`. Kod źródłowy strony WWW wygenerowanej przed tę stronę JSP przedstawiłem na listingu 12.9. Kilka typowych wyników wykonania tej strony można zobaczyć na rysunkach od 12.3 od 12.6.

Listing 12.8 ShadowedTextApplet.jsp[11]

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Zastosowanie znacznika jsp:plugin</TITLE>
```

[11] Aby strona *ShadowedTextApplet.jsp* mogła zostać poprawnie wykonana trzeba będzie poczynić pewne przygotowania. Otóż, strona ta korzysta z pliku klasowego apletu *ShadowedTextApplet.class* należącego do pakietu `coreservlets`; na podstawie wartości atrybutów znacznika `jsp:plugin` można określić, iż plik ten będzie poszukiwany w katalogu *coreservlets* znajdującym się wewnątrz katalogu, w którym jest umieszczona strona *ShadowedTextApplet.jsp*. W tym samym katalogu powinne się także znaleźć wszystkie pliki klasowe z których korzysta sam aplet, są to: *LabelPanel.class*, *MessageImage.class*, *ShadowedText.class*, *ShadowedTextFrame.class* oraz *WindowUtilities.class*. A zatem, aby przykład działał poprawnie, należy utworzyć katalog *coreservlets* w katalogu, w którym znajduje się strona *ShadowedTextApplet.jsp* i skopiować do niego wszystkie wspomniane wcześniej pliki klasowe.

```
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>

<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      Zastosowanie znacznika jsp:plugin</TABLE>
<P>
<CENTER>
<jsp:plugin type="applet"
            code="coreservlets.ShadowedTextApplet.class"
            width="475" height="350">
  <jsp:params>
    <jsp:param name="MESSAGE" value="Tutaj podaj komunikat" />
  </jsp:params>
</jsp:plugin>
</CENTER>

</BODY>
</HTML>
```

Listing 12.9 Kod źródłowy dokumentu WWW wygenerowanego przez stronę ShadowedTextApplet.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Zastosowanie znacznika jsp:plugin</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>

<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      Zastosowanie znacznika jsp:plugin</TABLE>
<P>
<CENTER>
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93" width="475"  height="350"
codebase="http://java.sun.com/products/plugin/1.2.2/jinstall-1_2_2-win.cab#Version=1,2,2,0">
<PARAM name="java_code" value="coreservlets.ShadowedTextApplet.class">
<PARAM name="type" value="application/x-java-applet;">
<PARAM name="MESSAGE" value="Tutaj podaj komunikat">
<COMMENT>
<EMBED type="application/x-java-applet;"  width="475"  height="350"
pluginspage="http://java.sun.com/products/plugin/"
java_code="coreservlets.ShadowedTextApplet.class" MESSAGE=Tutaj podaj komunikat>
<NOEMBED>
</COMMENT>
</NOEMBED></EMBED>
</OBJECT>

</CENTER>

</BODY>
</HTML>
```

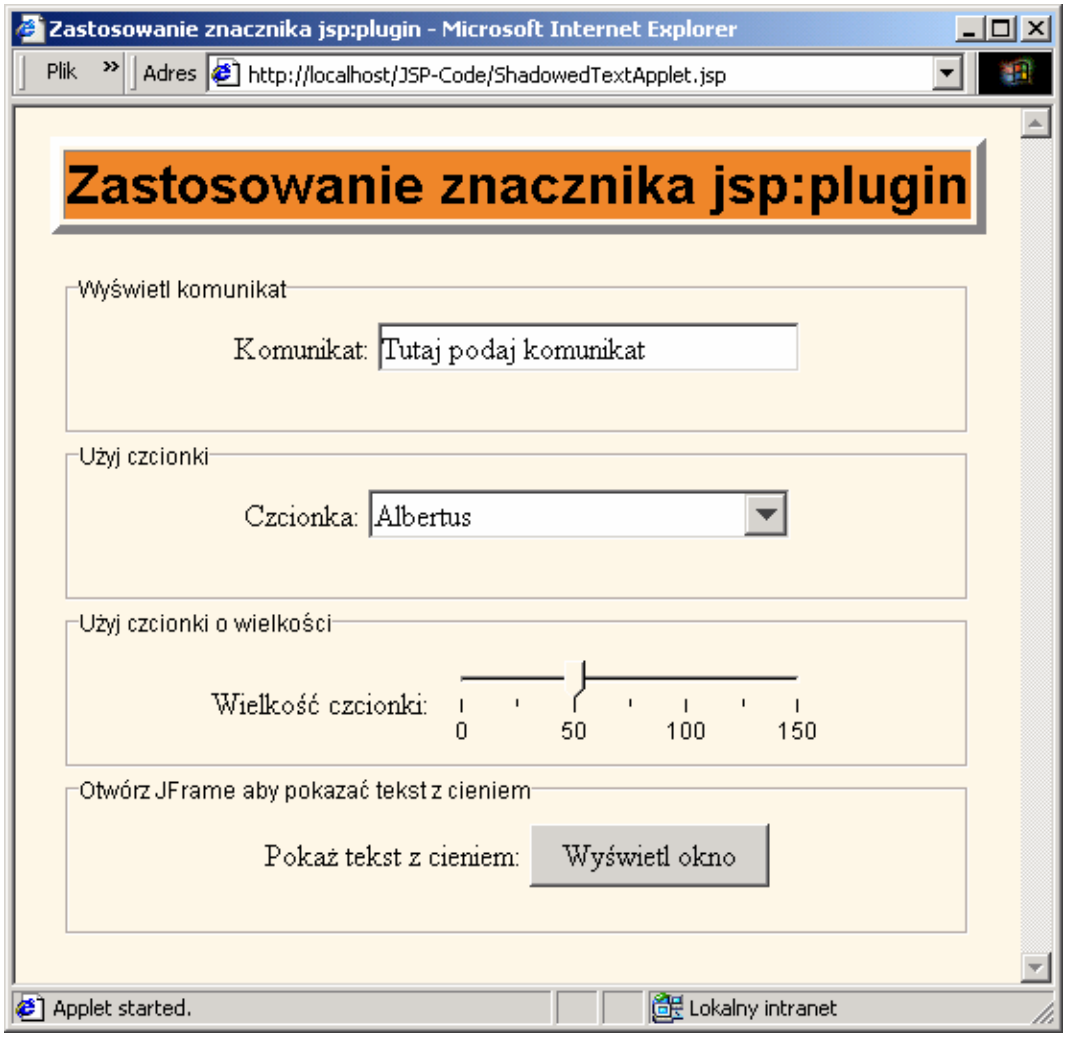
Listing 12.10 ShadowedTextApplet.java

```
package coreservlets;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class ShadowedTextApplet extends JApplet
                                implements ActionListener {
  private JTextField messageField;
  private JComboBox fontBox;
  private JSlider fontSizeSlider;
```

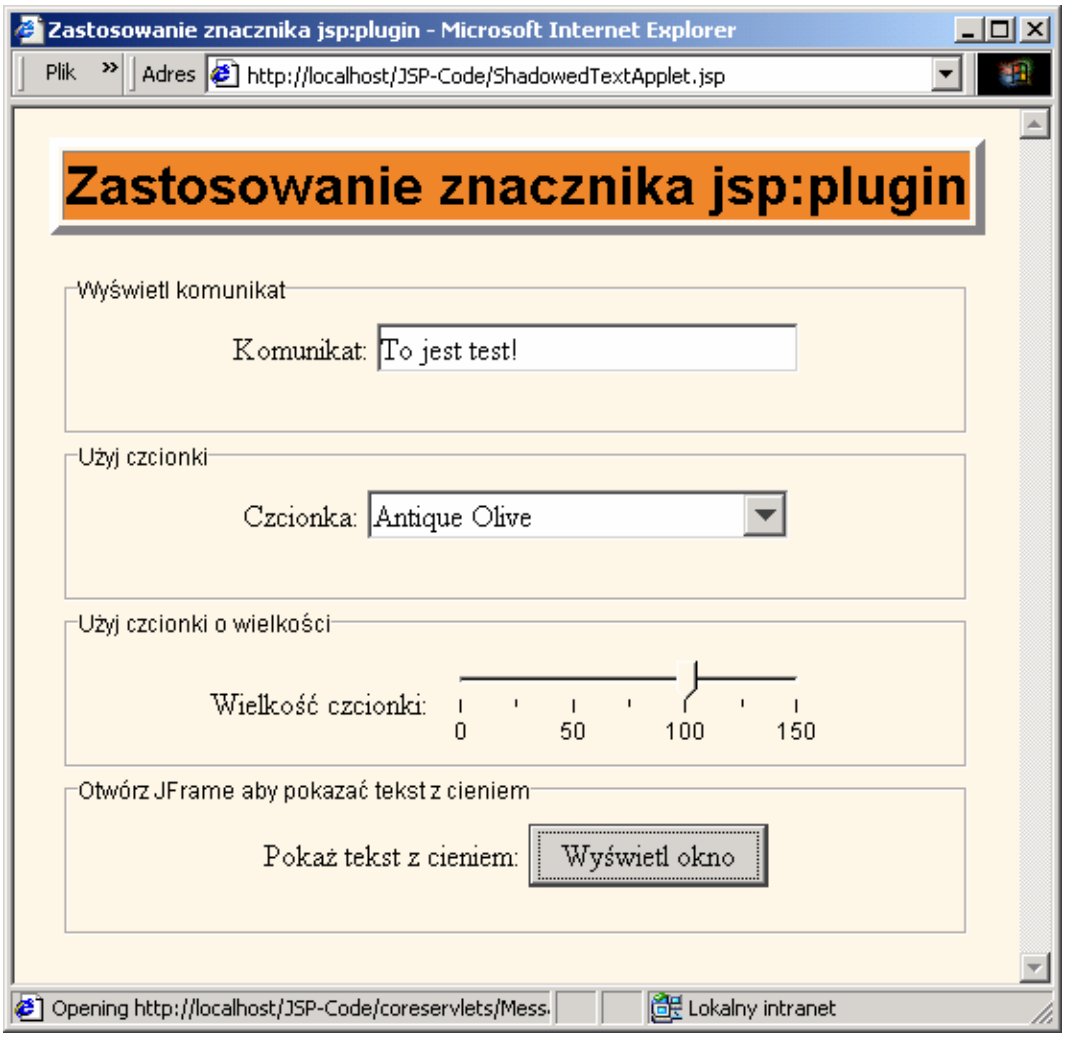
```
  private JButton showFrameButton;

  public void init() {
    WindowUtilities.setNativeLookAndFeel();
    Color bgColor = new Color(0xFD, 0xF5, 0xE6);
    Font font = new Font("Serif", Font.PLAIN, 16);
    Container contentPane = getContentPane();
    contentPane.setLayout(new GridLayout(4, 1));
    contentPane.setBackground(bgColor);

    // Użyj pola JTextField do pobrania tekstu komunikatu.
    // Jeśli w stronie JSP został zdefiniowany parametr
    // MESSAGE to użyj jego wartości jako domyślnej zawartości
    // pola.
    messageField = new JTextField(20);
    String message = getParameter("MESSAGE");
    if (message != null) {
      messageField.setText(message);
    }
    JPanel messagePanel =
      new LabelPanel("Komunikat:", "Wyświetl komunikat",
                     bgColor, font, messageField);
    contentPane.add(messagePanel);

    // Użyj pola JComboBox aby umożliwić użytkownikom
    // wybór jednej z czcionek zainstalowanych w ich systemie.
    GraphicsEnvironment env =
      GraphicsEnvironment.getLocalGraphicsEnvironment();
    String[] fontNames = env.getAvailableFontFamilyNames();
    fontBox = new JComboBox(fontNames);
    fontBox.setEditable(false);
    JPanel fontPanel =
      new LabelPanel("Czcionka:", "Użyj czcionki",
                     bgColor, font, fontBox);
    contentPane.add(fontPanel);

    // Użyj suwaka (JSlider) do określenia wielkości czcionki.
    fontSizeSlider = new JSlider(0, 150);
    fontSizeSlider.setBackground(bgColor);
    fontSizeSlider.setMajorTickSpacing(50);
    fontSizeSlider.setMinorTickSpacing(25);
    fontSizeSlider.setPaintTicks(true);
    fontSizeSlider.setPaintLabels(true);
    JPanel fontSizePanel =
      new LabelPanel("Wielkość czcionki:", "Użyj czcionki o wielkości",
                     bgColor, font, fontSizeSlider);
    contentPane.add(fontSizePanel);

    // Naciśnięcie tego przycisku spowoduje otworzenie okna
    // i wyświetlenie w nim tekstu z cieniem.
    showFrameButton = new JButton("Wyświetl okno");
    showFrameButton.addActionListener(this);
    JPanel buttonPanel =
      new LabelPanel("Pokaż tekst z cieniem:",
                     "Otwórz JFrame aby pokazać tekst z cieniem",
                     bgColor, font, showFrameButton);
    contentPane.add(buttonPanel);
  }

  public void actionPerformed(ActionEvent event) {
    String message = messageField.getText();
    if (message.length() == 0) {
      message = "Brak komunikatu!";
    }
    String fontName = (String)fontBox.getSelectedItem();
    int fontSize = fontSizeSlider.getValue();
    JFrame frame = new JFrame("Tekst z cieniem");
    JPanel panel =
      new ShadowedTextFrame(message, fontName, fontSize);
    frame.setContentPane(panel);
    frame.pack();
    frame.setVisible(true);
  }
}
```

Listing 12.11 LabelPane.java

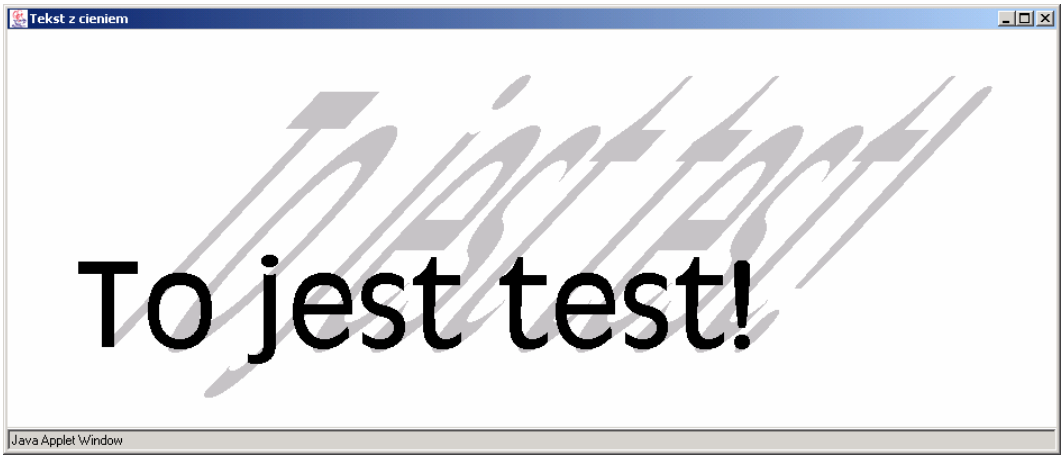
```
package coreservlets;

import java.awt.*;
import javax.swing.*;

public class LabelPanel extends JPanel {
  public LabelPanel(String labelMessage, String title,
                    Color bgColor, Font font,
                    JComponent component) {
    setBackground(bgColor);
    setFont(font);
    setBorder(BorderFactory.createTitledBorder(title));
    JLabel label = new JLabel(labelMessage);
    label.setFont(font);
    add(label);
    component.setFont(font);
    add(component);
  }
}
```



Rysunek 12.3 Początkowe wyniki wyświetlenia strony ShadowedTextApplet.jsp w przeglądarce z zainstalowanym apletem JDK 1.2



Rysunek 12.4 Aplet wyświetlony na stronie ShadowedTextApplet.jsp po zmianie czcionki, jej wielkości oraz treści komunikatu



Rysunek 12.5 Obraz wygenerowany po kliknięciu przycisku Wyświetl okno, przy ustawieniach przedstawionych na rysunku 12.4

Rysunek 12.6 Inny obraz wygenerowany przy użyciu strony ShadowedTextApplet.jsp

Rozdział 13.
Wykorzystanie komponentów JavaBeans w dokumentach JSP

Interfejs programistyczny (API) JavaBeans określa standardowy format klas tworzonych w języku Java. Wizualne narzędzia programistyczne oraz inne narzędzia mogą automatycznie pobierać informacje o takich klasach, tworzyć je i manipulować nimi, przy czym użytkownik nie musi w tym celu pisać żadnego kodu.

Wyczerpujące omówienie zagadnień związanych z JavaBeans wykracza poza ramy niniejszej książki. Jeśli chcesz zdobyć szczegółowe informacje na ten temat, możesz sięgnąć po jedną z wielu książek poświęconych zagadnieniom tworzenia i wykorzystania komponentów JavaBeans lub przejrzeć dokumentacje i podręczniki udostępnione na witrynie firmy Sun, pod adresem *http://java.sun.com/beans/docs/*. Abyś mógł przeczytać niniejszy rozdział, zrozumieć i wykorzystać zawarte w nim informacje, wystarczy że będziesz wiedział trzy podstawowe rzeczy na temat komponentów JavaBeans:

1. **W klasie komponentu musi być zdefiniowany konstruktor, który nie pobiera żadnych argumentów.** Wymóg ten można spełnić na dwa sposoby. Pierwszym z nich jest jawne zdefiniowanie takiego konstruktora, a drugim — pominięcie definicji jakichkolwiek konstruktorów (bowiem w takim przypadku konstruktor, który nie pobiera żadnych argumentów jest tworzony domyślnie).
2. **Klasa komponentu nie powinna mieć żadnych publicznych zmiennych instancyjnych (pól).** Mam nadzieję, że już postępujesz według tej zasady i zamiast bezpośredniego dostępu do zmiennych instancyjnych klasy tworzysz, tak zwane, metody dostępowe — czyli metody pozwalające na pobieranie i podawanie wartości zmiennych instancyjnych. Użycie metod dostępowych umożliwia narzucenie ograniczeń na wartości przypisywane zmiennym instancyjnym (na przykład, metoda `setSzybkosc` klasy `Samochod` może zapewnić, że wartość aktualnie wybranego biegu nie będzie mniejsza o zera) oraz pozwala zmieniać wewnętrzną strukturę danych klasy bez konieczności modyfikowania jej interfejsu (na przykład, wewnętrznie zmieniać jednostki z angielskich na metryczne, przy ciągłej możliwości korzystania z metod `getSzybkoscWMPH` oraz `getSzybkoscWKPH`). Zastosowanie metod dostępowych pozwala także na wykonywanie czynności ubocznych w momencie określania wartości zmiennej instancyjnej (na przykład, aktualizować interfejs graficzny w momencie wywołania metody `setPosition`).
3. **Wszelkie operacje na wartościach trwałych powinne być wykonywane przy użyciu metod `getXxxx` oraz `setXxxx`.** Na przykład, jeśli klasa `Samochod` przechowuje informacje o bieżącej liczbie pasażerów, to można by w niej zaimplementować metody `getIloscPasazerow` (która nie pobiera żadnych argumentów i zwraca wartość typu `int`) oraz `setIloscPasazerow` (która pobiera argument typu `int` i nie zwraca żadnej wartości). W takim przypadku mówi się, że klasa `Samochod` posiada *właściwość* o nazwie `iloscPasazerow` (zwróć uwagę, iż nazwa właściwości zaczyna się od małej litery „i”, choć w nazwach metod

została użyta duża litera). Jeśli klasa udostępnia wyłącznie metodę `getXxxx`, to mówi się, że posiada właściwość *xxxx* przeznaczoną wyłącznie do odczytu.

Istnieje jeden wyjątek od tej konwencji określania nazw metod dostępowych — dotyczy on zmiennych instancyjnych przechowujących wartości logiczne (typu `boolean`). Otóż w tym przypadku, metody zwracające wartość takich zmiennych instancyjnych noszą nazwy `isXxxx`. A zatem, nasza przykładowa klasa `Samochod` może mieć metodę `isPozyczony` (która nie wymaga podania żadnych argumentów i zwraca wartość typu `boolean`) oraz metodę `setPozyczony` (która wymaga podania wartości typu `boolean` i niczego nie zwraca). W takim przypadku, nasza klasa `Samochod` miałaby właściwość logiczną (typu `boolean`) o nazwie `pozyczony` (także w tym przypadku nazwa właściwości rozpoczyna się z małej litery).

Choć w skryptletach i wyrażeniach JSP można korzystać z dowolnych metod używanych klas, to jednak standardowe znaczniki akcji JSP służące do operowania na komponentach JavaBeans mogą posługiwać się wyłącznie metodami stworzonymi zgodnie z konwencją `getXxxx`/`setXxxx` oraz `isXxxx`/`setXxxx`.

13.1 Podstawowe sposoby użycia komponentów

Do załadowania komponentu JavaBean, który ma być użyty w dokumencie JSP służy znacznik akcji `jsp:useBean`. Komponenty JavaBeans są niezwykle przydatne, gdyż zapewniają możliwość wielokrotnego używania tego samego kodu udostępnianą klasy języka Java, a jednocześnie nie ograniczają wygody, jaką daje zastosowanie technologii JSP w porównaniu z wykorzystaniem samych serwletów.

Poniżej przedstawiłem najprostszą postać znacznika akcji informującego o użyciu komponentu JavaBean:

```
<jsp:useBean id="nazwa" class="pakiet.Klasa" />
```

Powyższy znacznik informuje, że należy „utworzyć kopię obiektu wskazanej *Klasy* i skojarzyć go ze zmienną, której nazwę określa atrybut `id`". A zatem, poniższy znacznik JSP

```
<jsp:useBean id="ksiazka1" class="coreservlets.Ksiazka" />
```

jest odpowiednikiem następującego skryptletu:

```
<% coreservlets.Ksiazka ksiazka1 = new coreservlets.Ksiazka(); %>
```

Takie utożsamienie znacznika `jsp:useBean` z kodem tworzącym kopię obiektu jest bardzo wygodne; jednak znacznik ten dysponuje dodatkowymi atrybutami, które sprawiają, że jego możliwości są znacznie większe. W podrozdziale 13.4. (pt.: „Wspólne wykorzystywanie komponentów") poznasz atrybut `scope`, dzięki któremu komponent może być dostępny nie tylko w obrębie bieżącej strony JSP. Jeśli można wspólnie korzystać z komponentów, można także pobrać odwołania do wszystkich istniejących komponentów. To z kolei oznacza, że użycie znacznika `jsp:useBean` spowoduje utworzenie nowego egzemplarza komponentu wyłącznie w sytuacji, gdy nie istnieje żaden inny komponent o identycznych wartościach atrybutów `id` oraz `scope`.

Zamiast atrybutu `class` można używać atrybutu `beanName`. Różnica polega na tym, iż atrybut `beanName` może się odwoływać nie tylko do klasy lecz także do pliku zawierającego zapisany obiekt. Wartość tego atrybutu jest przekazywana jako argument wywołania metody `instantiate` klasy `java.beans.Bean`.

W większości przypadków będziesz chciał, aby zmienna lokalna była tego samego typu co tworzony obiekt. Może się jednak zdarzyć, że będziesz chciał, aby zmienna została zadeklarowana jako zmienna typu będącego jedną z klas bazowych klasy tworzonego komponentu lub jako interfejs, który komponent implementuje. Do tego celu służy atrybut `type`, którego zastosowanie przedstawiłem na poniższym przykładzie:

```
<jsp:useBean id="watek1" class="MojaKlasa" type="Runnable" />
```

Użycie powyższego znacznika spowoduje wstawienie do metody `_jspService` wygenerowanego serwletu, kodu podobnego do:

```
Runnable watek1 = new MojaKlasa();
```

Zwróć uwagę, iż znacznik akcji `jsp:useBean` jest zapisywany zgodnie z zasadami języka XML, przez co różni się od znaczników HTML z trzech powodów — uwzględniana jest wielkość liter jakimi zapisywane są nazwy atrybutów, wartości atrybutów muszą być zapisane pomiędzy znakami cudzysłowu lub apostrofu, koniec znacznika oznaczany jest przy użyciu znaków `/>` a samego znaku `>`. Pierwsze dwie różnice dotyczą wszystkich elementów JSP o postaci `jsp:`*`xxx`*; natomiast ostatnia z nich dotyczy wyłącznie tych, które nie mają odrębnych znaczników otwierających i zamykających.

Ostrzeżenie
Składnia zapisu elementów `jsp:`*`xxx`* różni się od składni zapisu elementów HTML z trzech powodów — uwzględniana jest wielkość liter w nazwach atrybutów, wartości atrybutów muszą być zapisane pomiędzy znakami cudzysłowu bądź pomiędzy apostrofami oraz jeśli dany element nie jest „pojemnikiem", to musi się być zakończony przy użyciu kombinacji znaków `/>` a nie pojedynczego znaku `>`.

Istnieje także kilka znaków i kombinacji znaków, które wymagają specjalnego potraktowania w przypadku umieszczania ich w wartościach atrybutów:

- aby umieścić apostrof (`'`) w wartości atrybutu, należy go zapisać jako `\'`,
- aby umieścić apostrof (`"`) w wartości atrybutu, należy go zapisać jako `\"`,
- aby umieścić apostrof (`\`) w wartości atrybutu, należy go zapisać jako `\\`,
- aby umieścić kombinację znaków `%>` w wartości atrybutu, należy zapisać ją w postaci `%\>`,
- aby umieścić apostrof (`<%`) w wartości atrybutu, należy zapisać ją w postaci `<\%`.

Dostęp do właściwości komponentów

Po utworzeniu komponentu można uzyskać dostęp do jego właściwości przy użyciu znacznika akcji `jsp:getProperty`. W znaczniku tym należy podać wartości dwóch atrybutów — `name` oraz `property`, przy czym wartość pierwszego z nich (`name`) powinna odpowiadać wartości atrybutu `id` użytego w znaczniku `jsp:useBean`, natomiast drugi (`property`) określa nazwę właściwości. Alternatywnym rozwiązaniem jest użycie wyrażenia JSP, które jawnie wywołuje odpowiednią metodę obiektu; w tym przypadku nazwa użytej zmiennej musi odpowiadać nazwie podanej jako wartość atrybutu `id` znacznika `jsp:useBean`. Załóżmy, że klasa `Ksiazka` ma właściwość `tytul` typu `String` oraz że utworzyłeś kopię komponentu tej klasy o nazwie `ksiazka1` (posługując się znacznikiem akcji `jsp:useBean`, w sposób przedstawiony we wcześniejszej części rozdziału). W takiej sytuacji wartość właściwości `tytul` komponentu można wyświetlić w następujący sposób:

```
<jsp:getProperty name="ksiazka1" property="tytul" />
```

lub

```
<%= ksiazka1.getTytul() %>
```

W naszym przypadku zalecane jest wykorzystanie pierwszego sposobu, gdyż jest bardziej zrozumiały dla twórców stron WWW, którzy nie znają języka Java. Niemniej jednak możliwość bezpośredniego dostępu do zmiennej także jest przydatna, szczególnie w przypadkach gdy trzeba jej używać w pętlach, instrukcjach warunkowych oraz w wywołaniach metod, które nie są dostępne jako właściwości.

Jeśli nie spotkałeś się jeszcze z pojęciem właściwości stosowanym w komponentach JavaBeans, to spieszę wyjaśnić, iż wyrażenie „komponent ma właściwość typu `T` o nazwie `cos`" należy rozumieć że „jest to klasa definiująca metodę `getCos` zwracającą obiekt typu `T` oraz metodę `setCos` pobierającą argument typu `T` i zapisującą jego wartość, aby później można ją było pobrać przy użyciu metody `getCos`".

Określanie właściwości komponentów — prosty przypadek

Do określania wartości właściwości komponentów JavaBeans jest zazwyczaj stosowany znacznik akcji `jsp:setProperty`. Znacznik ten można zapisywać na wiele różnych sposobów, lecz w najprostszej formie wymaga podania trzech atrybutów — `name`, `property` oraz `value`. Wartość atrybutu `name` powinna odpowiadać wartości atrybutu `id` znacznika `jsp:useBean`; atrybut `property` określa nazwę modyfikowanej właściwości, a atrybut `value` — wartość jaką należy jej przypisać. W podrozdziale 13.3. — „Określanie wartości właściwości komponentów" przedstawię kilka alternatywnych sposobów zapisu znacznika `jsp:setProperty`, które pozwalają automatycznie skojarzyć właściwość z parametrem żądania HTTP. Wyjaśnię tam również, w jaki sposób można podawać wartości obliczane w czasie przetwarzania żądania (a nie z góry określone łańcuchy znaków) oraz opiszę konwencje konwersji typów, dzięki którym, używając łańcuchów znaków, można określać wartości właściwości przechowujących liczby, znaki oraz wartości logiczne.

Zamiast wykorzystania znacznika `jsp:setProperty` można także użyć skryptletu zawierającego bezpośrednie wywołanie metody komponentu. Kontynuując nasz przykład komponentu `ksiazka1`, którym posłużyłem się już we wcześniejszej części rozdziału, przedstawię teraz dwa sposoby określenia wartości właściwości `tytul` tego komponentu:

```
<jsp:setProperty name="ksiazka1"
                 property="tytul"
                 value="Java Servlet i Java Server Pages" />
<% ksiazka1.setTytul("Java Servlet i Java Server Pages" ); %>
```

Zastosowanie znacznika `jsp:setProperty` ma tę zaletę, iż jest łatwiejsze dla osób, które nie znają języka Java. Z drugiej strony bezpośrednie odwołanie do metody obiektu pozwala na realizację bardziej złożonych czynności, takich jak warunkowe określanie wartości, bądź też wywoływanie metod innych niż `set`*`Xxxx`* i `get`*`Xxxx`*.

Instalacja klas komponentów

Klasy używane przy tworzeniu komponentów muszą się znajdować w zwyczajnych katalogach serwera, a nie w katalogach służących do przechowywania klas, które są automatycznie przeładowywane w razie modyfikacji. Na przykład, na serwerze Java Web Server wszystkie pliki klasowe komponentów JavaBeans oraz klas pomocniczych powinne być przechowywane w katalogu *katalog_instalacyjny/classes*, można je także zapisać w pliku JAR i umieścić w katalogu *katalog_instalacyjny/lib*; jednak nie powinno się ich umieszczać w katalogu *katalog_instalacyjny/servlets*. Serwery Tomcat 3.0 oraz JSWDK nie udostępniają możliwości automatycznego przeładowywania serwletów, a zatem w ich przypadku pliki klasowe komponentów mogą być umieszczane w dowolnych katalogach służących do przechowania serwletów. Na serwerze Tomcat 3.0 głównym katalogiem przeznaczonych do przechowywania plików klasowych serwletów (zakładając, że nie zdefiniowałeś własnej aplikacji) jest *katalog_instalacyjny/webpages/WEB-INF/classes*; na serwerze JSWDK jest to katalog *katalog_instalacyjny/webpages/WEB-INF/servlets*. Niezależnie którego z tych trzech serwerów używasz, musisz pamiętać iż nazwa pakietu zawsze odpowiada katalogowi. A zatem, plik klasowy komponentu klasy `Fordhook` należącej do pakietu `lima`, powinien zostać umieszczony w następujących katalogach:

- **na serwerze Tomcat 3.0**: *katalog_instalacyjny/webpages/WEB-INF/classes/lima/Fordhook.class*;
- **na serwerze JSWDK 1.0.1**: *katalog_instalacyjny/webpages/WEB-INF/servlets/lima/Fordhook.class*,
- **na serwerze Java Web Server 2.0**: *katalog_instalacyjny/classes/lima/Fordhook.class*.

Dokument JSP, w którym są używane komponenty pewnej klasy nie musi być instalowany w żadnym konkretnym miejscu — zazwyczaj pliki JSP przechowywane na serwerach obsługujących tę technologię mogą być umieszczane wszędzie tam, gdzie normalne dokumenty HTML.

13.2 Przykład: StringBean

Na listingu 13.1 przedstawiłem prostą klasę o nazwie `StringBean`, należącą do pakietu `coreservlets`. Klasa ta nie ma żadnych publicznie dostępnych zmiennych instancyjnych (pól) i dysponuje konstruktorem nie pobierających żadnych argumentów (który jest generowany automatycznie, gdyż klasa nie deklaruje żadnych konstruktorów). A zatem, klasa ta spełnia podstawowe kryteria konieczne do tego aby można ją było uznać za komponent. Klasa `StringBean` posiada także metodę `getMessage` zwracającą wartość typu `String` oraz metodę `setMessage` wymagającą podania argumentu typu `String`, co oznacza, że w terminologii JavaBeans możemy powiedzieć, iż klasa ta posiada właściwość o nazwie `message`.

Listing 13.2 przedstawia stronę JSP, która korzysta z klasy `StringBean`. W pierwszej kolejności, przy użyciu znacznika `jsp:useBean` tworzona jest kopia obiektu tej klasy:

```
<jsp:useBean id="stringBean" class="coreservlets.StringBean" />
```

Następnie w treści strony można wyświetlić wartość właściwości `message` tego komponentu; można to zrobić na dwa sposoby:

```
<jsp:getProperty name="stringBean" property="message" />
<%= stringBean.getMessage() %>
```

Wartość właściwości `message` można również zmodyfikować; także tę czynność można wykonać na dwa sposoby:

```
<jsp:setProperty name="stringBean"
                 property="message"
                 value="jakiś komunikat" />
<% stringBean.setMessage( "jakiś komunikat" ); %>
```

Wyniki wykonania strony *StringBean.jsp* przedstawiłem na rysunku 13.1.

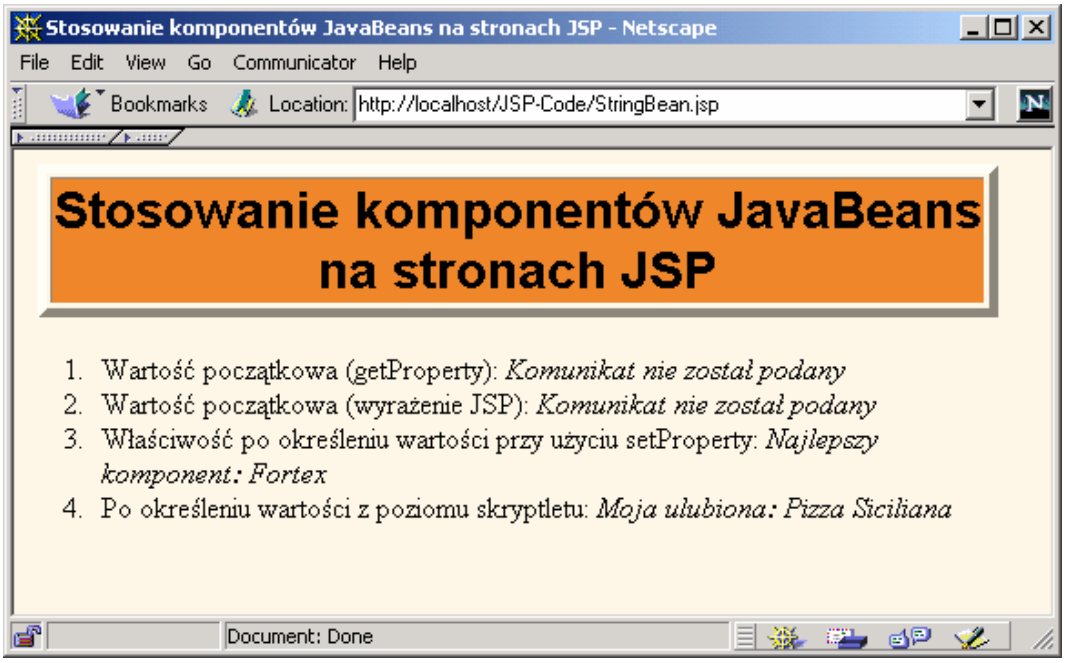
Listing 13.1 StringBean.java

```
package coreservlets;

public class StringBean {
  private String message = "Komunikat nie został podany";

  public String getMessage() {
    return(message);
  }

  public void setMessage(String message) {
    this.message = message;
  }
}
```

Listing 13.2 StringBean.jsp

```
<%@ page contentType="text/html; charset=ISO-8859-2" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<HTML>
<HEAD>
<TITLE>Stosowanie komponentów JavaBeans na stronach JSP</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
```

```
<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      Stosowanie komponentów JavaBeans na stronach JSP</TABLE>

<jsp:useBean id="stringBean" class="coreservlets.StringBean" />

<OL>
<LI>Wartość początkowa (getProperty):
    <I><jsp:getProperty name="stringBean"
                        property="message" /></I>

<LI>Wartość początkowa (wyrażenie JSP):
    <I><%= stringBean.getMessage() %></I>
<LI><jsp:setProperty name="stringBean"
                     property="message"
                     value="Najlepszy komponent: Fortex" />
    Właściwość po określeniu wartości przy użyciu setProperty:
    <I><jsp:getProperty name="stringBean"
                        property="message" /></I>

<LI><% stringBean.setMessage("Moja ulubiona: Pizza Siciliana"); %>
    Po określeniu wartości z poziomu skryptletu:
    <I><%= stringBean.getMessage() %></I>
</OL>

</BODY>
</HTML>
```



Rysunek 13.1 Wyniki wykonania strony StringBean.jsp

13.3 Określanie wartości właściwości komponentów

Do określania wartości właściwości komponentów JavaBeans służy zazwyczaj znacznik `jsp:setProperty`. Jego najprostsza forma wymaga podania trzech atrybutów — `name` (którego wartość powinna odpowiadać wartości atrybutu `id` znacznika `jsp:useBean`), `property` (zawierającego nazwę właściwości) oraz `value` (określającego nową wartość jaka powinna zostać przypisana właściwości).

Na przykład, klasa `SaleEntry` przedstawiona na listingu 13.3 zawiera właściwości `itemID` (typu `String`), `numItems` (typu `int`) oraz dwie właściwość typu `double` przeznaczone tylko do odczytu — `itemCost` oraz `totalCost`. Listing 13.4 przedstawia stronę JSP, która tworzy egzemplarz obiektu klasy `SaleEntry` w poniższy sposób:

```
<jsp:useBean id="entry" class="coreservlets.SaleEntry" />
```

Listing 13.3 SaleEntry.java

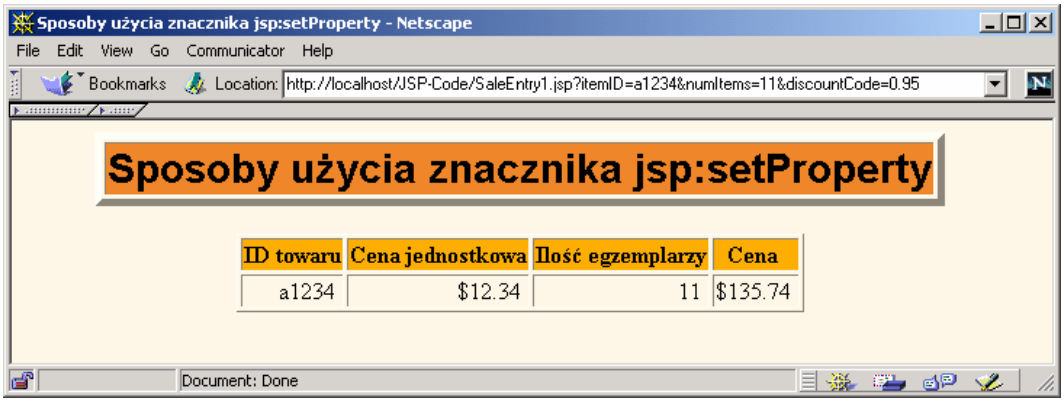
```
package coreservlets;

public class SaleEntry {
  private String itemID = "nieznany";
  private double discountCode = 1.0;
  private int numItems = 0;

  public String getItemID() {
    return(itemID);
  }

  public void setItemID(String itemID) {
    if (itemID != null) {
      this.itemID = itemID;
    } else {
      this.itemID = "nieznany";
    }
  }

  public double getDiscountCode() {
    return(discountCode);
  }

  public void setDiscountCode(double discountCode) {
    this.discountCode = discountCode;
  }

  public int getNumItems() {
    return(numItems);
  }

  public void setNumItems(int numItems) {
    this.numItems = numItems;
  }

  // Zastąp to prawdziwym wyszukaniem informacji w bazie danych

  public double getItemCost() {
    double cost;
    if (itemID.equals("a1234")) {
      cost = 12.99*getDiscountCode();
    } else {
      cost = -9999;
    }
    return(roundToPennies(cost));
  }

  private double roundToPennies(double cost) {
    return(Math.floor(cost*100)/100.0);
  }

  public double getTotalCost() {
    return(getItemCost() * getNumItems());
  }
}
```

Listing 13.4 SaleEntry1.jsp

```
<%@ page contentType="text/html; charset=ISO-8859-2" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Sposoby użycia znacznika jsp:setProperty</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>

<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      Sposoby użycia znacznika jsp:setProperty</TABLE>
```

```
<jsp:useBean id="entry" class="coreservlets.SaleEntry" />

<jsp:setProperty
    name="entry"
    property="itemID"
    value='<%= request.getParameter("itemID") %>' />

<%
int numItemsOrdered = 1;
try {
  numItemsOrdered =
    Integer.parseInt(request.getParameter("numItems"));
} catch(NumberFormatException nfe) {}
%>
<jsp:setProperty
    name="entry"
    property="numItems"
    value="<%= numItemsOrdered %>" />

<%
double discountCode = 1.0;
try {
  String discountString =
    request.getParameter("discountCode");
  // metoda Double.parseDouble nie jest dostępna w JDK 1.1.
  discountCode =
    Double.valueOf(discountString).doubleValue();
} catch(NumberFormatException nfe) {}
%>
<jsp:setProperty
    name="entry"
    property="discountCode"
    value="<%= discountCode %>" />

<BR>
<TABLE ALIGN="CENTER" BORDER=1>
<TR CLASS="COLORED">
  <TH>ID towaru<TH>Cena jednostkowa<TH>Ilość egzemplarzy<TH>Cena
<TR ALIGN="RIGHT">
  <TD><jsp:getProperty name="entry" property="itemID" />
  <TD>$<jsp:getProperty name="entry" property="itemCost" />
  <TD><jsp:getProperty name="entry" property="numItems" />
  <TD>$<jsp:getProperty name="entry" property="totalCost" />
</TABLE>

</BODY>
</HTML>
```

Wyniki wykonania strony z listingu 13.4 przedstawiłem na rysunku 13.2.



Rysunek 13.2 Wyniki wykonania strony SaleEntry1.jsp

Po utworzeniu kopii komponentu, użycie parametru wejściowego podanego w żądaniu, do określenia wartości właściwości `itemID` jest bardzo proste:

```
<jsp:setProperty
```

```
    name="entry"
    property="itemID"
    value='<%= request.getParameter("itemID") %>' />
```

Zwróć uwagę, iż wartość parametru `value` określiłem przy wykorzystaniu wyrażenia JSP. Wartości niemal wszystkich atrybutów znaczników JSP muszą być konkretnymi łańcuchami znaków. Wyjątkiem są tu wartości atrybutów `name` oraz `value` znacznika akcji `jsp:setProperty`, które można określać przy użyciu wyrażeń obliczanych w czasie obsługi żądania. Jeśli w samym wyrażeniu są używane znaki cudzysłowu, to warto pamiętać, iż wartości atrybutów można także zapisywać w apostrofach oraz że w wartościach atrybutów cudzysłowy i apostrofy mogą być reprezentowane przez kombinacje znaków `\"` i `\'`.

Kojarzenie właściwości z parametrami wejściowymi

Określenie wartości właściwości `itemID` było proste, gdyż był to łańcuch znaków (`String`). Nieco więcej problemów nastręczyło przypisanie wartości właściwościom `numItems` oraz `discountCode`. Problem polegał na tym, iż wartości tych właściwości muszą być liczbami, natomiast metoda `getParameter` zwraca łańcuch znaków — czyli wartość typu `String`. Poniżej przedstawiłem nieco rozbudowany i nieelegancki kod konieczny do określenia wartości właściwości `numItems`:

```
<%
int numItemsOrdered = 1;
try {
  numItemsOrdered =
    Integer.parseInt(request.getParameter("numItems"));
} catch(NumberFormatException nfe) {}
%>
<jsp:setProperty
    name="entry"
    property="numItems"
    value="<%= numItemsOrdered %>" />
```

Na szczęście technologia JSP udostępnia rozwiązanie tego problemu. Polega ono na skojarzeniu właściwości z parametrem wejściowym i automatycznym przeprowadzeniu konwersji typów z łańcucha znaków (typu `String`) do liczby, znaku lub wartości logicznej. Aby użyć tej metody, w znaczniku `jsp:setProperty` należy zamiast atrybutu `value` umieścić atrybut `param` i przypisać mu nazwę parametru wejściowego. Wartość wskazanego parametru wejściowego zostanie przypisana właściwości, a przy tym automatycznie zostanie przeprowadzona odpowiednia konwersja typów. Jeśli podanego parametru wejściowego nie będzie w żądaniu, nie zostaną wykonane żadne czynności (czyli system nie przypisze właściwości wartości `null`). A zatem, określenie wartości właściwości `numItems` można uprościć do następującej postaci:

```
<jsp:setProperty
     name="entry"
     property="numItems"
     parameter="numItems" />
```

Na listingu 13.5 przedstawiłem drugą wersję strony *SaleEntry1.jsp*, w której wykorzystany został prostszy sposób kojarzenia właściwości z wartościami parametrów wejściowych.

Listing 13.5 SaleEntry2.jsp

```
<%@ page contentType="text/html; charset=ISO-8859-2" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Sposoby użycia znacznika jsp:setProperty</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>

<TABLE BORDER=5 ALIGN="CENTER">
```

```
  <TR><TH CLASS="TITLE">
      Sposoby użycia znacznika jsp:setProperty</TABLE>

<jsp:useBean id="entry" class="coreservlets.SaleEntry" />

<jsp:setProperty
    name="entry"
    property="itemID"
    param="itemID" />

<jsp:setProperty
    name="entry"
    property="numItems"
    param="numItems" />

<%-- OSTRZEŻENIE!
              Zarówno w JSWDK 1.0.1 jak i w Java Web Serverze
              jest błąd, który powoduje pojawianie się błędów
              przy przeprowadzaniu takiej konwersji do liczby
              typu dobule.
--%>
<jsp:setProperty
    name="entry"
    property="discountCode"
    param="discountCode" />

<BR>
<TABLE ALIGN="CENTER" BORDER=1>
<TR CLASS="COLORED">
  <TH>ID towaru<TH>Cena jednostkowa<TH>Ilość egzemplarzy<TH>Cena
<TR ALIGN="RIGHT">
  <TD><jsp:getProperty name="entry" property="itemID" />
  <TD>$<jsp:getProperty name="entry" property="itemCost" />
  <TD><jsp:getProperty name="entry" property="numItems" />
  <TD>$<jsp:getProperty name="entry" property="totalCost" />
</TABLE>

</BODY>
</HTML>
```

Automatyczna konwersja typów

W tabeli 13.1 przedstawiłem automatyczne konwersje typów jakie mogą być wykonane gdy skojarzymy właściwość z parametrem wejściowym. Koniecznie należy jednak zwrócić uwagę na to, iż zarówno w JSWDK 1.0.1 jak i w Java Web Serverze jest błąd, który powoduje awarię serwera w czasie przekształcania strony, jeśli jest na niej wykonywana automatyczna konwersja wartości parametru wejściowego do wartości typu `double`. Zarówno Tomcat jak i większość serwerów komercyjnych działa zgodnie z oczekiwaniami.

Ostrzeżenie
W serwerach JSWDK oraz Java Web Server nie możesz kojarzyć parametrów wejściowych z właściwości wymagających podania liczby zmiennoprzecinkowej o podwójnej precyzji.

Tabela 13.1. Konwersje typów wykonywane jeśli właściwości komponentów są kojarzone bezpośrednio z parametrami wejściowymi.

Typ właściwości	Sposób konwersji
`boolean`	`Boolean.valueOf(parametrString).booleanValue()`
`Boolean`	`Boolean.valueOf(parametrString)`
`byte`	`Byte.valueOf(parametrString).byteValue()`
`Byte`	`Byte.valueOf(parametrString)`
`char`	`Character.valueOf(parametrString).charValue()`
`Character`	`Character.valueOf(parametrString)`
`double`	`Double.valueOf(parametrString).doubleValue()`
`Double`	`Double.valueOf(parametrString)`
`int`	`Integer.valueOf(parametrString).intValue()`

Integer	Integer.valueOf(parametrString)
float	Float.valueOf(parametrString).floatValue()
Float	Float.valueOf(parametrString)
long	Long.valueOf(parametrString).longValue()
Long	Long.valueOf(parametrString)

Kojarzenie wszystkich właściwości z parametrami wejściowymi

Skojarzenie właściwości komponentów z parametrami wejściowymi oszczędza Ci problemów związanych z koniecznością przeprowadzania konwersji wielu podstawowych typów danych. JSP pozwala jednak posunąć cały proces o jeden krok dalej i skojarzyć *wszystkie* właściwości z parametrami wejściowymi o identycznych nazwach. Jedyną rzeczą jaką należy w tym celu zrobić, jest przypisanie atrybutowi `property` znacznika `jsp:setProperty` łańcucha znaków `"*"`. A zatem, wszystkie trzy znaczniki `jsp:setProperty` z listingu 13.5 można zastąpić jednym (zmodyfikowana wersja strony została przedstawiona na listingu 13.6):

```
<jsp:setProperty name="entry" property="*" />
```

Listing 13.6 SaleEntry3.jsp

```
<%@ page contentType="text/html; charset=ISO-8859-2" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Sposoby użycia znacznika jsp:setProperty</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>

<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      Sposoby użycia znacznika jsp:setProperty</TABLE>

<jsp:useBean id="entry" class="coreservlets.SaleEntry" />
<%-- OSTRZEŻENIE! Zarówno JSWDK 1.0.1 jak i Java Web Server
              mają błąd, który powoduje awarie serwera podczas
              próby automatycznej konwersji liczb typu double.
--%>
<jsp:setProperty name="entry" property="*" />

<BR>
<TABLE ALIGN="CENTER" BORDER=1>
<TR CLASS="COLORED">
  <TH>ID towaru<TH>Cena jednostkowa<TH>Ilość egzemplarzy<TH>Cena
<TR ALIGN="RIGHT">
  <TD><jsp:getProperty name="entry" property="itemID" />
  <TD>$<jsp:getProperty name="entry" property="itemCost" />
  <TD><jsp:getProperty name="entry" property="numItems" />
  <TD>$<jsp:getProperty name="entry" property="totalCost" />
</TABLE>

</BODY>
</HTML>
```

Choć ten sposób określania wartości właściwości na podstawie parametrów wejściowych jest bardzo prosty, to należy jednak zwrócić uwagę na cztery potencjalne źródła problemów. Po pierwsze, podobnie jak w przypadku indywidualnego określania wartości właściwości, także i teraz nie są wykonywane żadne czynności w przypadku gdy parametr wejściowy nie zostanie podany. W szczególności, system nie użyje wartości `null` jako domyślnej wartości właściwości. Po drugie, na serwerach JSWDK 1.0.1 oraz Java Web Server występują błędy podczas konwersji i przypisywania wartości właściwościom oczekującym liczb całkowitych o podwójnej precyzji (czyli liczb typu `double`). Poza tym automatyczna konwersja typów nie zabezpiecza programu przed różnymi

błędami jakie mogą się pojawić podczas konwersji danych, w równie wysokim stopniu co konwersja wykonywana ręcznie. A zatem, w przypadku stosowania automatycznej konwersji typów możesz także wykorzystać własne strony obsługi błędów (patrz listingi 11.9 oraz 11.10). I w końcu ostatnia sprawa. Zarówno w nazwach właściwości jak i parametrów uwzględniana jest wielkość liter, dlatego musisz zwrócić baczną uwagę, aby nazwy te były identyczne.

Ostrzeżenie
Aby właściwości zostały poprawnie skojarzone z parametrami wejściowymi ich nazwy muszą być identyczne (dotyczy to także wielości liter).

13.4 Wspólne wykorzystywanie komponentów

Jak na razie traktowałem obiekty tworzone przy użyciu znacznika akcji `jsp:useBean` jak gdyby były one kojarzone ze zwyczajnymi zmiennymi zadeklarowanymi w metodzie `_jspService` (która jest wywoływana przez metodę service serwletu wygenerowanego na podstawie strony JSP). Choć komponenty są faktycznie kojarzone ze zmiennymi, to jednak ich możliwości nie ograniczają się wyłącznie do tego. Otóż komponenty mogą być przechowywane w jednym z czterech miejsc, a to, w którym z nich zostaną umieszczone zależy od wartości opcjonalnego atrybutu `scope` znacznika `jsp:useBean`. Atrybut ten może przybierać cztery wartości:

- **`page`**

  `Page` to domyślna wartość atrybutu `scope`. Oznacza ona, że komponent będzie skojarzony ze zmienną lokalną, a poza tym, na czas obsługi bieżącego żądania, zastanie zapisany w obiekcie `PageContext`. Ogólnie rzecz biorąc, zapisanie komponentu oznacza, że kod serwletu będzie mógł odwołać się do niego przy użyciu metody `getAttribute` predefiniowanej zmiennej `pageContext`. W praktyce, niemal wszystkie operacje na komponentach tworzonych przy wykorzystaniu atrybutu `scope` o wartości `page` są wykonywane na tej samej stronie i realizowane przy użyciu znaczników akcji `jsp:getProperty`, `jsp:setProperty`, serwletów bądź wyrażeń JSP.

- **`application`**

  Ta bardzo przydatna wartość oznacza, że komponent nie tylko będzie skojarzony ze zmienną lokalną, lecz dodatkowo zostanie zapisany w obiekcie `ServletContext`, dostępnym jako predefiniowana zmienna `application` lub za pośrednictwem metody `getServletContext`. Obiekt `ServletContext` jest wspólnie wykorzystywany przez wszystkie serwlety należące do tej samej aplikacji WWW (lub wszystkie serwlety wykonywane na danym serwerze lub mechanizmie obsługi serwletów, jeśli nie została jawnie zdefiniowana żadna aplikacja). Dane przechowywane w obiekcie `ServletContext` można pobierać przy użyciu metody `getAttribute`. To wspólne korzystanie z obiektów wiąże się z dwoma dodatkowymi zagadnieniami.

  Po pierwsze, dostępny jest prosty mechanizm zapewniający serwletom i stronom JSP możliwość dostępu do tego samego obiektu. Szczegółowe informacje na ten temat oraz przykład znajdziesz w następnej części rozdziału, pod tytułem „Warunkowe tworzenie komponentów”.

  A po drugie, serwlety mogą *tworzyć* nowe komponenty które będą wykorzystywane w dokumentach JSP, a nie tylko korzystać z komponentów, które zostały wcześniej utworzone. Dzięki temu, serwlety mogą obsługiwać złożone żądania tworząc w tym celu komponenty, zapisując je w obiekcie `ServletContext` i przekazując żądanie do jeden z kilku stron JSP, która je obsłuży i zwróci wyniki. Więcej informacji na ten temat znajdziesz w rozdziale 15., pt.: „Integracja serwletów i dokumentów JSP”.

- **`session`**

Ta wartość oznacza, iż komponent będzie skojarzony ze zmienną lokalną, a oprócz tego, podczas obsługi bieżącego żądania, zostanie umieszczony w obiekcie `HttpSession`. Obiekt ten jest skojarzony z predefiniowaną zmienną `session`. Dane przechowywane w obiekcie `HttpSession` można pobierać przy użyciu metody `getValue`. Jeśli w dyrektywie `page` określono, że dana strona nie będzie należała do sesji, to przypisanie atrybutowi `scope` wartości `session` spowoduje zgłoszenie błędów podczas przekształcania strony do postaci serwletu (patrz podrozdział 11.4. — „Atrybut session").

- **`request`**

  Ta wartość oznacza, że komponent nie tylko będzie skojarzony ze zmienną lokalną, lecz dodatkowo, podczas obsługi bieżącego żądania, zostanie zapisany w obiekcie `ServletRequest`. Komponent będzie można pobrać przy użyciu metody `getAttribute`. Użycie tej wartości bardzo nieznacznie różni się od przypisania atrybutowi `scope` wartości `page` (lub wykorzystania jego domyślnej wartości, stosowanej gdy wartość atrybutu nie zostanie jawnie określona).

Warunkowe tworzenie komponentów

Istnieją dwie sytuacje, w których znaczniki związane z wykorzystaniem komponentów są przetwarzane warunkowo; zostały one wprowadzone po to, by wspólne korzystanie z komponentów było bardziej wygodne.

Przede wszystkim znacznik `jsp:useBean` powoduje stworzenie nowej kopii komponentu wyłącznie wtedy, gdy nie istnieje jeszcze żaden inny komponent o tym samym identyfikatorze (atrybucie `id`) oraz zasięgu (atrybucie `scope`). Jeśli uda się odnaleźć komponent o tych samych wartościach atrybutów `id` i `scope`, to zostanie on skojarzony ze zmienną lokalną. Jeśli klasa odnalezionego komponentu jest klasą potomną klasy deklarowanej w znaczniku `jsp:useBean`, to zostanie przeprowadzone odpowiednie rzutowanie typów. Jeśli nie będzie ono poprawne, zostanie zgłoszony wyjątek `ClassCastException`.

Poza tym zamiast znacznika

```
<jsp:useBean ... />
```

można użyć znacznika o postaci

```
<jsp:useBean ...>
  instrukcje
</jsp:useBean>
```

Ta druga forma zapisu daje dodatkową możliwość, iż wszystkie *instrukcje* umieszczone pomiędzy otwierającym i zamykającym znacznikiem `jsp:useBean` będą wykonywane wyłącznie w razie tworzenia nowej kopii komponentu; jeśli zostanie odnaleziony i użyty istniejąca kopia, to nie zostaną one wykonane. Ta możliwość warunkowego wykonania pewnych czynności jest bardzo wygodna przy określaniu początkowych wartości właściwości komponentów wykorzystywanych na wielu stronach JSP. Ponieważ nie wiadomo, która strona zostanie wykonana jako pierwsza, nie sposób określić w której z nich należy umieścić kod inicjalizujący komponent. Ale nie ma żadnego problemu — wszystkie strony mogą zawierać kod inicjalizujący, lecz zostanie on wykonany wyłącznie na pierwszej zażądanej stronie. Listing 13.7 przedstawia przykład prostego komponentu, który można użyć do rejestracji łącznej ilości odwiedzin grup powiązanych ze sobą strony. Komponent ten zapamiętuje także nazwę pierwszej strony, która została wyświetlona. Ponieważ w żaden sposób nie można określić, która ze stron danej grupy zostanie wyświetlona jako pierwsza, a zatem każda ze stron używająca naszego „licznika grupowego" będzie zawierać następujący kod:

```
<jsp:useBean id="counter"
             class="coreservlets.AccessCountBean"
             scope="application">
  <jsp:setProperty name="counter"
                   property="firstPage"
                   value="SharedCounts1.jsp" />
</jsp:useBean>
```

```
Łącznie zarejestrowano
<jsp:getProperty name="counter" property="accessCount" />
odwołań do grupy tych trzech stron.
```

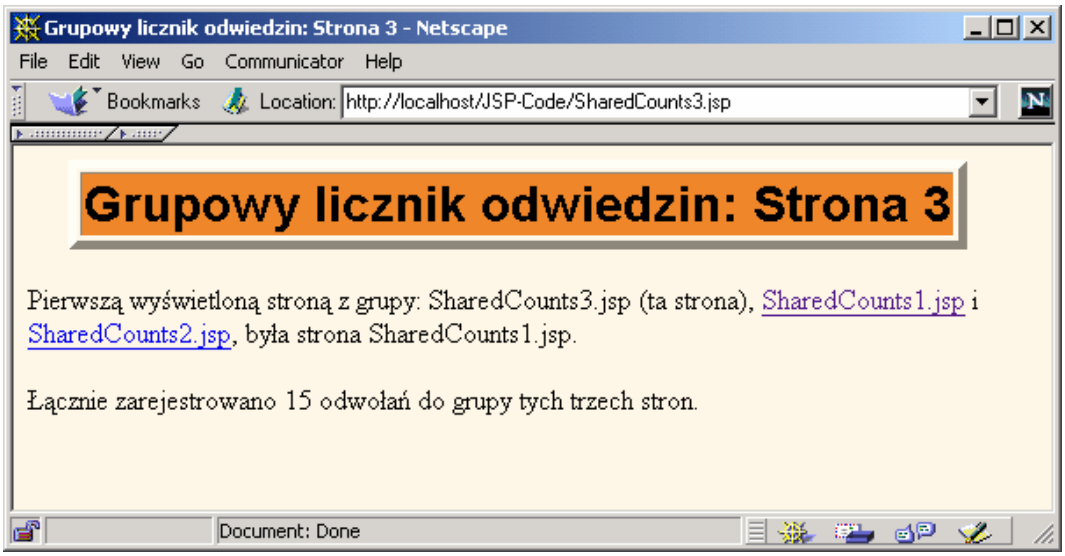
Listing 13.8 przedstawia pierwszą ze stron JSP wykorzystujących nasz „grupowy licznik" odwiedzin. Pozostałe dwie strony znajdziesz w pliku archiwalnym zawierającym kody źródłowe wszystkich przykładów podanych w niniejszej książce (*ftp://ftp.helion.pl/przyklady/jsjsp.zip*). Różnice pomiędzy wszystkimi trzema stronami są minimalne. Przykładowe wyniki wykonania jednej z tych trzech stron, zostały przedstawione na rysunku 13.3.

Listing 13.7 AccessCountBean.java

```
package coreservlets;

public class AccessCountBean {
  private String firstPage;
  private int accessCount = 1;

  public String getFirstPage() {
    return(firstPage);
  }

  public void setFirstPage(String firstPage) {
    this.firstPage = firstPage;
  }

  public int getAccessCount() {
    return(accessCount++);
  }
}
```

Listing 13.8 SharedCounts1.jsp

```
<%@ page contentType="text/html; charset=ISO-8859-2" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Grupowy licznik odwiedzin: Strona 1</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>

<TABLE BORDER=5 ALIGN="CENTER">
  <TR><TH CLASS="TITLE">
      Grupowy licznik odwiedzin: Strona 1</TABLE>
<P>
<jsp:useBean id="counter"
             class="coreservlets.AccessCountBean"
             scope="application">
  <jsp:setProperty name="counter"
                   property="firstPage"
                   value="SharedCounts1.jsp" />
</jsp:useBean>

Pierwszą wyświetloną stroną z grupy: SharedCounts1.jsp (ta strona),
<A HREF="SharedCounts2.jsp">SharedCounts2.jsp</A> i
<A HREF="SharedCounts3.jsp">SharedCounts3.jsp</A>,
była strona <jsp:getProperty name="counter" property="firstPage" />.
<P>
Łącznie zarejestrowano
<jsp:getProperty name="counter" property="accessCount" />
odwołań do grupy tych trzech stron.

</BODY>
</HTML>
```



Rysunek 13.3 Wyniki wyświetlone w przeglądarce po wykonaniu strony SharedCounts3.jsp. Pierwszą wyświetloną stroną z grupy była strona SharedCounts1.jsp. Strony SharecCounts1.jsp, SharedCounts2.jsp oraz SharedCounts3.jsp zostały w sumie wyświetlone 14 razy od czasu uruchomienia serwera lecz przed wyświetleniem strony pokazanej na tym rysunku

Rozdział 14.
Tworzenie bibliotek znaczników

Specyfikacja JSP 1.1 wprowadza niezwykle cenną innowację — możliwość definiowania własnych znaczników JSP. Pozwala ona na określenie sposobu interpretacji znacznika, jego atrybutów oraz zawartości oraz grupowania własnych znaczników w kolekcje nazywane *bibliotekami znaczników*. Znaczniki wchodzące w skład takich bibliotek mogą być używane na dowolnych stronach JSP. Możliwość definiowania bibliotek znaczników pozwala programistom na implementację skomplikowanych zadań realizowanych na serwerze, w postaci prostych elementów, które autorzy stron będą mogli bez trudu umieszczać w swoich dokumentach JSP.

Własne znaczniki JSP, częściowo realizują te same zadania co komponenty JavaBeans, tworzone przy użyciu znacznika akcji `jsp:useBean` (przedstawiłem go w rozdziale 13., pt.: „Wykorzystanie komponentów JavaBeans w dokumentach JSP") — czyli w prostej formie, łatwej do zastosowania zawierają realizację skomplikowanych czynności. Istnieje jednak kilka różnic pomiędzy zastosowaniem komponentów JavaBeans i bibliotek znaczników. Po pierwsze, komponenty JavaBeans nie są w stanie manipulować zawartością stron JSP, natomiast znaczniki mogą to robić. Po drugie, znaczniki umożliwiają zredukowanie skomplikowanych czynności do znacznie prostszej postaci niż komponenty. Po trzecie, stworzenie własnych znaczników wymaga znacznie więcej pracy niż stworzenie komponentów. Po czwarte, komponenty są bardzo często definiowane w jednym serwlecie lub stronie JSP, a następnie wykorzystywane przez inne serwlety lub strony (patrz rozdział 15. — „Integracja serwletów i dokumentów JSP"), natomiast znaczniki definiują zazwyczaj bardziej niezależne czynności. I w końcu ostatnia sprawa — znaczniki mogą być stosowane wyłącznie w narzędziach zgodnych ze specyfikacją JSP 1.1, natomiast komponenty JavaBeans także w starszych narzędziach zgodnych ze specyfikacją JSP 1.0.

W czasie gdy niniejsza książka była oddawana do druku, nie istniała jeszcze oficjalna wersja serwera Tomcat 3.0, która byłaby w stanie poprawnie obsługiwać biblioteki znaczników. Z tego względu przykłady przedstawione w tym rozdziale były wykonywane na testowej wersji serwera Tomcat 3.1. Oprócz możliwości obsługi bibliotek znaczników, kilku usprawnień efektywności działania oraz poprawek niewielkich błędów, obie wersje serwera Tomcat nie różnią się od siebie. Warto jeszcze wspomnieć, że serwer Tomcat 3.1 używa nieco innej struktury katalogów; najistotniejsze zmiany przedstawiłem w tabeli 14.1.

Tabela 14.1 Standardowe katalogi różnych wersji serwera Tomcat.

	Tomcat 3.0	Tomcat 3.1
Położenie skryptów służących do uruchamiania i zatrzymywania serwera	*katalog_instalacyjny*	*katalog_instalacyjny/bin*
Standardowy, główny katalog służący do	*katalog_instalacyjny/webpages/WEB-INF/classes*	*katalog_instalacyjny/webapps/ROOT/WEB-INF/classes*

przechowywania serwletów i innych klas pomocniczych		
Standardowy, główny katalog służący do przechowywania dokumentów HTML i JSP	*katalog_instalacyjny/webpages*	*katalog_instalacyjny/webapps/ROOT*

14.1 Elementy tworzące bibliotekę znaczników

Aby móc używać własnych znaczników JSP, należy zdefiniować trzy odrębne elementy — klasę obsługującą znacznik (która będzie definiować jego działanie), plik deskryptora biblioteki znaczników (który kojarzy nazwy elementów XML z implementacją znaczników) oraz dokument JSP wykorzystujący znaczniki. W tej części rozdziału bardziej szczegółowo omówię każdy z tych elementów, natomiast w kolejnych podrozdziałach pokaże jak tworzyć te elementy dla różnych rodzajów znaczników.

Klasa obsługi znacznika

Definiując nowy znacznik, w pierwszej kolejności należy zdefiniować klasę języka Java, która poinformuje system co należy zrobić po jego napotkaniu. Taka klasa musi implementować `interfejs javax.servlet.jsp.tagext.Tag`. W tym celu są zazwyczaj tworzone klasy, będące klasami potomnymi klas `TagSupport` lub `BodyTagSupport`. Na listingu 14.1 przedstawiłem prosty znacznik, którego użycie spowoduje wyświetlenie na stronie JSP łańcucha znaków „Przykład znacznika (coreservlets.tags.ExampleTag)”. Nie staraj się zrozumieć działania tej klasy — wszystko dokładnie wyjaśnię w dalszej części rozdziału. Jak na razie, powinieneś jedynie zwrócić uwagę, iż klasa ta nosi nazwę `ExampleTag` i jest umieszczona w pakiecie `coreservlets.tags`. A zatem, jeśli do testowania przykładów używasz serwera Tomcat 3.1, to plik klasowy *ExampleTag.class* powinien zostać zapisany jako *katalog_instalacyjny/webapps/ROOT/WEB-INF/classes/coreservlets/tags/ExampleTag.class*.

Listing 14.1 ExampleTag.java

```
package coreservlets.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

/** Bardzo prosty znacznik JSP który powoduje wyświetlenie
 *  na stronie łańcucha znaków "Przykład znacznika...".
 *  Nazwa znacznika nie jest definiowana w tym miejscu,
 *  jest ona podawana w pliku Deskryptora Biblioteki
 *  znaczników (TLD), do którego odwołuje się dyrektywa
 *  taglib umieszczona na stronie JSP.
*/

public class ExampleTag extends TagSupport {
  public int doStartTag() {
    try {
      JspWriter out = pageContext.getOut();
      out.print("Przykład znacznika " +
                "(coreservlets.tags.ExampleTag)");
    } catch(IOException ioe) {
      System.out.println("Błąd w ExampleTag: " + ioe);
    }
    return(SKIP_BODY);
```

```
  }
}
```

Plik deskryptora biblioteki znaczników

Po zdefiniowaniu klasy obsługi znacznika, kolejnym zadaniem jakie należy wykonać jest poinformowanie serwera o istnieniu tej klasy i skojarzenie jej z konkretną nazwą znacznika XML. Zadanie to jest realizowane przy wykorzystaniu pliku deskryptora biblioteki znaczników. Plik ten zapisywany jest w formacie XML, a jego przykładową postać przedstawiłem na listingu 14.2. Plik deskryptora biblioteki znaczników zawiera pewne ściśle określone informacje, dowolną, krótką nazwę biblioteki, jej opis oraz serię opisów poszczególnych znaczników. Fragment poniższego listingu, który nie został wydrukowany pogrubioną czcionką, jest taki same niemal we wszystkich plikach deskryptora biblioteki znaczników i można go bezpośrednio i bez żadnych modyfikacji skopiować z przykładów dołączonych do tej książki lub dostarczanych wraz z serwerem Tomcat 3.1 (znajdują się one w katalogu *katalog_instalacyjny/webapps/examples/WEB-INF/jsp*).

Listing 14.2 csajsp-tablib.tld

```
<?xml version="1.0" encoding="ISO-8859-2" ?>
<!DOCTYPE taglib
 PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<!-- deskryptor biblioteki znaczników -->

<taglib>
  <!-- teraz domyślną przestrzenią jest
        "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
   -->

  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>csajsp</shortname>
  <urn></urn>
  <info>
    Biblioteka znaczników książki Core Servlets and JavaServer Pages,
    http://www.coreservlets.com/.
  </info>

  <tag>
    <name>example</name>
    <tagclass coreservlets.tags.ExampleTag</tagclass>
    <info>Najprostszy przykład: wyświetla wiersz tekstu</info>
    <bodycontent>EMPTY</bodycontent>
  </tag>

   <!-- Inne znaczniki zdefiniowane w dalszej części rozdziału ... -->

</taglib>
```

Format zapisu pliku deskryptora opiszę szczegółowo w dalszej części rozdziału. Jak na razie wystarczy, abyś zwrócił uwagę, że element `tag` definiuje główną nazwę znacznika (a w zasadzie, jak się niebawem przekonasz, jego końcową część) oraz określa klasę obsługującą ten znacznik. Klasa obsługująca znacznik należy do pakietu `coreservlets.tags` i dlatego w elemencie `tag` podana została jej pełna nazwa — `coreservlets.tags.ExampleTag`. Zwróć uwagę, iż jest to nazwa klasy, a nie URL bądź względna ścieżka dostępu. Klasa ta może być umieszczona na serwerze w dowolnym miejscu, w katalogach przeznaczonych do przechowywania serwletów oraz innych, pomocniczych klas. W przypadku serwera Tomcat 3.1 standardowym położeniem jest katalog *katalog_instalacyjny/webapps/ROOT/WEB-INF/classes*, a zatem plik klasowy *ExampleTag.class* powinien zostać umieszczony w katalogu *katalog_instalacyjny/webapps/ROOT/WEB-INF/classes/coreservlets/tags*. Choć umieszczanie klas serwletów w pakietach zawsze jest dobrym

pomysłem, to jednak zaskakującą cechą serwera Tomcat 3.1 jest to, iż *wymaga* on aby klasy obsługi znaczników były umieszczane w pakietach.

Plik JSP

Po stworzeniu implementacji obsługi znacznika oraz pliku deskryptora biblioteki znaczników, można przystąpić do pisania dokumentu JSP wykorzystującego utworzony znacznik. Przykład takiego pliku przedstawiłem na listingu 14.3. Gdzieś przed pierwszym użyciem znacznika należy umieścić dyrektywę `taglib`. Poniżej przedstawiłem jej składnię:

```
<%@ taglib uri="..." prefix="..." %>
```

Listing 14.3 SimpleExample.jsp

```
<%@ page contentType="text/html; charset=ISO-8859-2" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>

<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>

<TITLE><csajsp:example /></TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<H1><csajsp:example /></H1>
<csajsp:example />

</BODY>
</HTML>
```

Wymagany atrybut `uri` może zawierać bezwzględny lub względny adres URL odwołujący się do pliku deskryptora biblioteki znaczników (takiego jak ten, który przedstawiłem na listingu 14.2). Jednak aby wszystko dodatkowo utrudnić, serwer Tomcat 3.1 korzysta z pliku *web.xml* który odwzorowuje bezwzględne adresy URL plików deskryptora na ścieżki w lokalnym systemie plików. Nie zalecam jednak stosowania tej metody; warto jednak abyś pamiętał o jej istnieniu, na wypadek gdybyś przeglądając przykłady dostarczane wraz z serwerem, zastanawiał się dlaczego one działają skoro odwołują się do nieistniejących adresów URL podawanych jako wartości atrybutu `uri` dyrektywy `taglib`.

Kolejny atrybut dyrektywy `taglib` — `prefix` — określa prefiks, który będzie umieszczany przed wszystkimi nazwami znaczników, zdefiniowanych w danym pliku deskryptora. Na przykład, jeśli w pliku TLD (pliku deskryptora biblioteki znaczników) został zdefiniowany znacznik o nazwie `tag1`, a atrybutowi `prefix` dyrektywy `taglib` przypisano wartość `test`, to faktyczna nazwa znacznika będzie miała postać `test:tag1`. Sam znacznik można umieszczać w pliku JSP na dwa sposoby; wybór jednego z nich zależy od tego, czy znacznik został zdefiniowany jako „pojemnik", w którym wykorzystywana jest zawartość umieszczona wewnątrz znacznika:

```
<test:tag1>
dowolny kod JSP
</test:tag1>
```

lub w formie uproszczonej

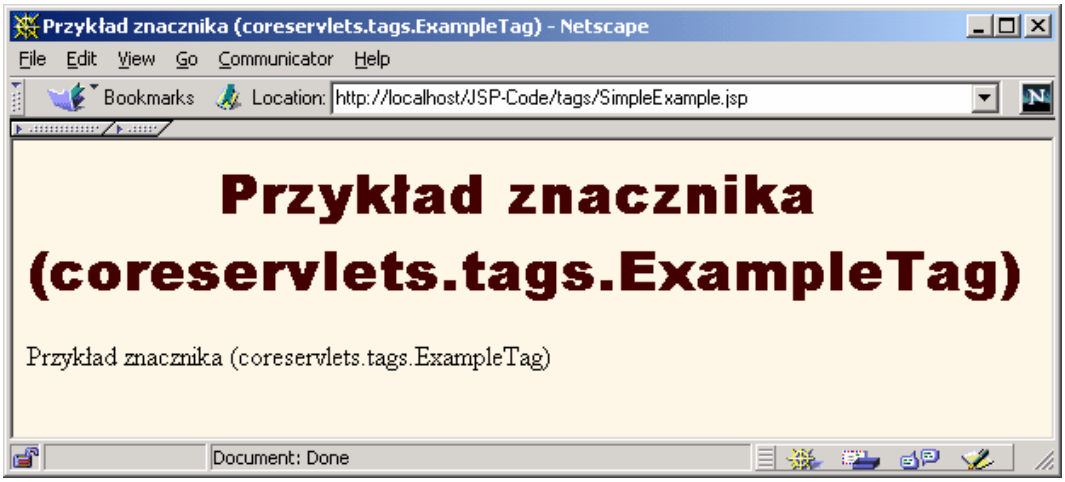
```
<test:tag1 />
```

Aby zilustrować to na konkretnym przykładzie przyjrzymy się plikowi deskryptora biblioteki znaczników przedstawionemu na listingu 14.2. Plik ten nosi nazwę *csajsp-taglib.tld* i jest przechowywany w tym samym katalogu, w którym znajduje się strona JSP z listingu 14.3. Dzięki

temu, dyrektywa `taglib` umieszczona w pliku JSP może zawierać prosty, względny adres URL określający wyłącznie nazwę pliku TLD:

```
<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>
```

Co więcej, atrybut `prefix` powyższej dyrektywy ma wartość `csajsp` (od tytułu oryginału książki — „JavaServer and JavaServer Pages"), a zatem w całej stronie JSP można się odwoływać do znacznika zdefiniowanego we wskazanym pliku deskryptora za pomocą nazwy `csajsp:example`. Wyniki wykonania strony używającej tego znacznika przedstawiłem na rysunku 14.1.



Rysunek 14.1 Wyniki wykonania strony SimpleExample.jsp

14.2 Definiowanie prostych znaczników

W tej części rozdziału podałem szczegółowe informacje dotyczące sposobów definiowania prostych znaczników, które nie mają ani atrybutów ani zawartości. A zatem, znaczniki o jakich będziemy tu mówić, można umieszczać na stronie przy użyciu zapisu o postaci `<prefiks:nazwa />`.

Klasa obsługi znacznika

Znaczniki, które nie mają żadnej zawartości oraz te, których zawartość jest wstawiana dosłownie, powinne być tworzone jako klasy potomne klasy `TagSupport`. Jest to wbudowana klasa należąca do pakietu `javax.servlet.jsp.tagext`, która implementuje interfejs `Tag` i zawiera niemal wszystkie możliwości funkcjonalne, które będą potrzebne przy tworzeniu prostych znaczników. Ze względu na inne klasy których będziesz używać, zazwyczaj należy zaimportować wszystkie klasy z pakietów `javax.servlet.jsp` oraz `java.io`. A zatem, przeważająca większość klas implementujących znaczniki JSP, po deklaracji pakietu, zawiera następujące instrukcje `import`:

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
```

Zachęcam Cię, abyś skopiował kody źródłowe przykładów przedstawionych w niniejszej książce (*ftp://ftp.helion.pl/przyklady/jsjsp.zip*) i wykorzystał je jako punkt startowy przy tworzeniu własnych znaczników.

W przypadku znaczników, które nie mają ani atrybutów ani zawartości, jedyną rzeczą jaką należy zrobić jest przesłonięcie metody `doStartTag`; metoda ta definiuje kod, który jest wywoływany *w podczas obsługi żądania*, w momencie odnalezienia znacznika otwierającego. Aby wygenerować dane wyjściowe w tej metodzie, należy pobrać obiekt `JspWriter` (jest to

specjalizowana wersja klasy `PrintWriter`; obiekt klasy `JspWriter` jest dostępny w dokumentach JSP jako predefiniowana zmienna `out`). W tym celu należy wywołać metodę `getOut` zmiennej instancyjnej `pageContext`. Zmienna ta (typu `PageContext`) udostępnia także inne metody, służące do pobierania różnych struktur danych związanych z żądaniem. Najważniejsze z tych metod to: `getRequest`, `getResponse`, `getServletContext` oraz `getSession`.

Ponieważ metoda `print` klasy `JspWriter` zgłasza wyjątek `IOException`, to jej wywołanie powinno zostać zapisane w bloku `try`/`catch`. Aby przekazać do przeglądarki użytkownika informacje o błędach innych typów, możesz zadeklarować by metoda `doStartTag` zgłaszała wyjątek `JspException`, a następnie, w przypadku jakiegoś błędu zgłosić ten wyjątek.

Jeśli tworzony znacznik JSP nie ma żadnej zawartości, to metoda `doStartTag` powinna zwracać wartość `SKIP_BODY`. Przekazanie tej wartości informuje system, iż należy pominąć całą zawartość umieszczoną pomiędzy znacznikiem otwierającym i zamykającym. W podrozdziale 14.5 (pt.: „Opcjonalne dołączanie zawartości znacznika") przekonasz się, że wartość `SKIP_BODY` jest czasami przydatna nawet w przypadkach gdy znacznik ma zawartość. Niemniej jednak prosty znacznik, który tworzymy w tym przykładzie nie będzie używany ze znacznikiem zamykającym (będziemy go zapisywali w formie `<prefiks:nazwa />`), a zatem nie będzie miał żadnej zawartości.

Listing 14.4 przedstawia implementację naszego przykładowego znacznika, stworzonego według przedstawionej wcześniej metody. Znacznik ten generuje listę losowych, 50-cio cyfrowych liczb pierwszych, wykorzystując do tego klasę `Prime` stworzoną w rozdziale 7 (pt.: „Generacja odpowiedzi: Nagłówki odpowiedzi HTTP", patrz listing 7.4).

Listing 14.4 SimplePrimeTag.java

```
package coreservlets.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.math.*;
import coreservlets.*;

/** Generuje liczby pierwsze o długości około 50 cyfr
 *  (50 cyfr ma długość losowo wygenerowanej liczby,
 *  zwrócona liczba pierwsza będzie większa od wygenerowanej
 *  liczy losowej.)
*/

public class SimplePrimeTag extends TagSupport {
  protected int len = 50;

  public int doStartTag() {
    try {
      JspWriter out = pageContext.getOut();
      BigInteger prime = Primes.nextPrime(Primes.random(len));
      out.print(prime);
    } catch(IOException ioe) {
      System.out.println("Błąd przy generowaniu liczby pierwszej: " + ioe);
    }
    return(SKIP_BODY);
  }
}
```

Plik deskryptora biblioteki znaczników

Ogólny format zapisu pliku deskryptora biblioteki znaczników niemal zawsze jest taki sam. Powinien się on zaczynać od identyfikatora wersji XML, po którym podawana jest deklaracja `DOCTYPE` oraz element `taglib`. Aby ułatwić sobie pracę możesz skopiować przykładowy plik z serwera FTP Wydawnictwa HELION (*ftp://ftp.helion.pl/przyklady/jsjsp.zip*) i użyć go jako szablonu. Najważniejsze informacje w pliku deskryptora umieszczane są *wewnątrz* elementu `taglib`

— a są to elementy `tag`. W przypadku prostych znaczników, które nie mają atrybutów, element `tag` powinien zawierać cztery kolejne elementy zapisywane pomiędzy znacznikami `<tag>` i `</tag>`:

1. **`name`** — definiuje on podstawową nazwę znacznika, do której będzie dodawany prefiks określony w dyrektywnie `taglib`. W tym przykładzie znacznik ten ma postać
   ```
   <name>simplePrime</name>
   ```
   co oznacza, że główną nazwą znacznika będzie `simplePrime`.
2. **`tagclass`** — ten element określa pełną nazwę pliku klasowego klasy obsługi znacznika; w tym przykładzie ma on następującą postać:
   ```
   <tagclass>coreservlets.tags.SimplePrimeTag</tagclass>
   ```
3. **`info`** — zawiera krótki opis, w tym przykładzie ma on następującą postać:
   ```
   <info>Wyświetla losową, 50-cio cyfrową liczbę pierwszą.</info>
   ```
4. **`bodycontent`** — w przypadku znaczników, które nie posiadają zawartości, element ten powinien zawierać wartość `EMPTY`. Dla znaczników posiadających zawartość, która może być interpretowana jako zwyczajny kod JSP, element ten powinien zawierać wartość `JSP`; natomiast w przypadku rzadko stosowanych znaczników, które samodzielnie przetwarzają swoją zawartość, w elemencie tym umieszczana jest wartość `TAGDEPENDENT`. Jeśli chodzi o omawiany przykład znacznika `SimplePrimeTag`, użyta zostanie wartość `EMPTY`:
   ```
   <bodycontent>EMPTY</bodycontent>
   ```

Pełny kod pliku TLD przedstawiony został na listingu 14.5.

Listing 14.5 csajsp-taglib.tld

```
<?xml version="1.0" encoding="ISO-8859-2" ?>
<!DOCTYPE taglib
 PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<!-- deskryptor biblioteki znaczników -->

<taglib>
  <!-- teraz domyślną przestrzenią jest
        "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
   -->

  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>csajsp</shortname>
  <urn></urn>
  <info>
    Biblioteka znaczników książki Core Servlets and JavaServer Pages,
    http://www.coreservlets.com/.
  </info>

  <!-- ... inne znaczniki zdefiniowane wcześniej ... -->

  <tag>
    <name>simplePrime</name>
    <tagclass>coreservlets.tags.SimplePrimeTag</tagclass>
    <info>Wyświetla losową, 50-cio cyfrową liczbę pierwszą.</info>
    <bodycontent>EMPTY</bodycontent>
  </tag>

  <!-- Inne znaczniki zdefiniowane w dalszej części rozdziału ... -->

</taglib>
```

Plik JSP

W dokumencie JSP, w którym będzie używany nasz przykładowy znacznik, należy umieścić dyrektywę `taglib`. Dyrektywa ta musi zawierać atrybut `uri` określający położenie pliku deskryptora biblioteki znaczników oraz atrybut `prefix` określający krótki łańcuch znaków (tak zwany *prefiks*),

który będzie umieszczany (wraz z dwukropkiem) przed główną nazwą znacznika. Listing 14.6 przedstawia dokument JSP, w którym została umieszczona dyrektywa `taglib` o następującej postaci:

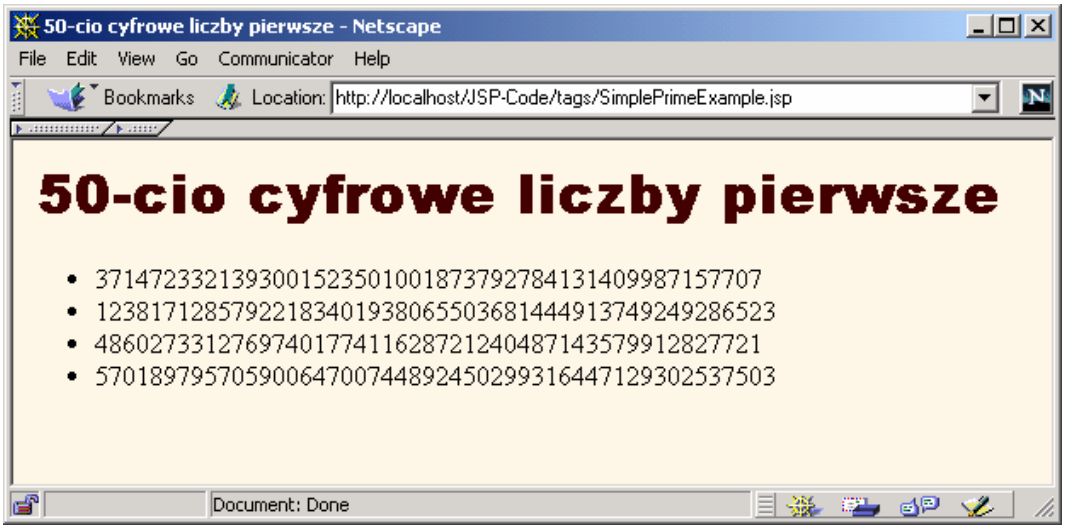
```
<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>
```

Oznacza ona, że zostanie użyty plik TLD przedstawiony na listingu 14.5, a nazwy znaczników będą poprzedzane prefiksem `csajsp`.

Wyniki wykonania strony *SimplePrimeExample.jsp* przedstawiłem na rysunku 14.2

Listing 14.6 SimplePrimeExample.jsp

```
<%@ page contentType="text/html; charset=ISO-8859-2" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>50-cio cyfrowe liczby pierwsze</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<H1>50-cio cyfrowe liczby pierwsze</H1>

<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>

<UL>
  <LI><csajsp:simplePrime />
  <LI><csajsp:simplePrime />
  <LI><csajsp:simplePrime />
  <LI><csajsp:simplePrime />
</UL>

</BODY>
</HTML>
```



Rysunek 14.2 Wyniki wykonania strony SimplePrimeExample.jsp

14.3 Przypisywanie atrybutów znacznikom

Umożliwienie zastosowania znaczników o ogólnej postaci

```
<prefiks:nazwa atrybut1="wartość1" atrybut2="wartość2" ... />
```

znacznie poprawia elastyczność biblioteki znaczników. W tej części rozdziału opiszę w jaki sposób można wzbogacić tworzone znaczniki o obsługę atrybutów.

Klasa obsługi znacznika

Implementacja obsługi atrybutów jest wyjątkowo prosta. Użycie atrybutu o nazwie `atrybut1` powoduje (w klasie potomnej klasy `TagSupport` lub w klasie, która w jakikolwiek inny sposób implementuje interfejs `Tag`) wywołanie metody o nazwie `setAtrybut1`. Wartość atrybutu jest przekazywana do metody jako łańcuch znaków (wartość typu `String`). A zatem, dodanie obsługi atrybutu o nazwie `atrybut1`, sprowadza się do zaimplementowania poniższej metody:

```
public void setAtrybut1(String wartosc1) {
  zrobCosZ(wartosc1);
}
```

Zwróć uwagę, iż atrybut o nazwie `nazwaAtrybutu` (pisanej z małej litery) odpowiada metodzie `setNazwaAtrybutu` (gdzie słowo `Nazwa` rozpoczyna się z dużej litery).

Metody służące do obsługi atrybutów najczęściej zapisują ich wartości w zmiennych instancyjnych (polach), tak aby później można z nich było skorzystać z metodach takich jak `doStartTag`. Na przykład, poniżej przedstawiłem fragment implementacji znacznika zawierającego obsługę atrybutu `komunikat`:

```
private String komunikat = "Domyślna treść komunikatu";

public void setKomunikat(String komunikat) {
  this.komunikat = komunikat;
}
```

Jeśli inne klasy będą mogły odwoływać się do klasy obsługi znacznika, to oprócz metody `setNazwaAtrubutu` warto w niej także zaimplementować metodę `getNazwaAtrybutu`. Wymagana jest jednak wyłącznie metoda `setNazwaAtrybutu`.

Na listingu 14.7 przedstawiłem klasę potomną klasy `SimplePrimeTag`, która została wyposażona w możliwość obsługi atrybutu `length`. Gdy atrybut ten zostanie podany w kodzie dokumentu, spowoduje to wywołanie metody `setLength`, która zamieni przekazany łańcuch znaków na liczbę typu `int` i zapisze jej wartość w zmiennej instancyjnej `len`, która jest już wykorzystywana w metodzie `doStartTag` klasy bazowej.

Listing 14.7 PrimeTag.java

```
package coreservlets.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.math.*;
import coreservlets.*;

/** Generuje N-cyfrową, losową liczbę pierwszą (domyślnie
 *  N = 50). Jest to klasa potomna klasy SimplePrimeTag,
 *  w której została dodana obsługa atrybutu umożliwiającego
 *  określanie długości generowanych liczb pierwszych.
 *  Metoda doStartTag klasy bazowej już używa zmiennej
 *  instancyjnej len, na podstawie której określana jest
 *  przybliżona długość liczby pierwszej.
 */

public class PrimeTag extends SimplePrimeTag {
  public void setLength(String length) {
    try {
      len = Integer.parseInt(length);
    } catch(NumberFormatException nfe) {
      len = 50;
    }
  }
}
```

Plik deskryptora biblioteki znaczników

Atrybuty znacznika muszą zostać zadeklarowane wewnątrz znacznika `tag` w pliku TLD. Do deklarowania atrybutów służą elementy `attribute`. Każdy z nich zawiera w sobie trzy kolejne elementy, które mogą się pojawić pomiędzy znacznikami `<attribute>` i `</attribute>`:

1. **`name`** — jest to wymagany element określający nazwę atrybutu (przy czym, przy określaniu tej nazwy uwzględniana jest wielkość liter). W naszym przykładzie element ten będzie miał następującą postać:
   ```
   <name>length</name>
   ```

2. **`required`** — ten wymagany element określa czy atrybut zawsze musi zostać umieszczony w znaczniku (w takim przypadku element `required` musi zawierać wartość `true`), czy też jest on opcjonalny (element `required` zawiera wartość `false`). W naszym przykładzie atrybut `length` jest opcjonalny, a zatem element `required` będzie miał postać:
   ```
   <required>false</required>
   ```

   Jeśli w kodzie dokumentu JSP pominiesz atrybut, to nie zostanie wywołana metoda `setNazwaAtrybut`; pamiętaj zatem, aby określić wartość domyślną zmiennej instancyjnej, w której jest przechowywana wartość atrybutu.
3. **`rtexprvalue`** — ten opcjonalny element określa czy wartością atrybutu może być wyrażenie JSP o ogólnej postaci `<%= wyrażenie %>` (jeśli tak, to element ten powinien zawierać wartość `true`), czy też ma to być z góry określony łańcuch znaków (w takim przypadku element `rtexprvalue` powinien zawierać wartość `false`). Domyślną wartością tego elementu jest `false`, a zatem jest on zazwyczaj pomijany, chyba że chcesz, aby wartości atrybutu mogły być określane w czasie obsługi żądania.

Na listingu 14.8 przedstawiłem pełny element `tag` umieszczony w pliku deskryptora biblioteki znaczników. Oprócz elementu `attribute` definiującego atrybut `length`, element ten zawiera także cztery standardowe elementy — `name` (o wartości `prime`), `tagclass` (o wartości `coreservlets.tags.PrimeTag`), `info` (zawierający krótki opis znacznika) oraz `bodycontent` (o wartości `EMPTY`).

Listing 14.8 csajsp-taglib.tld

```
<?xml version="1.0" encoding="ISO-8859-2" ?>
<!DOCTYPE taglib
 PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<!-- deskryptor biblioteki znaczników -->

<taglib>
  <!-- teraz domyślną przestrzenią jest
        "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
   -->

  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>csajsp</shortname>
  <urn></urn>
  <info>
    Biblioteka znaczników książki Core Servlets and JavaServer Pages,
    http://www.coreservlets.com/.
  </info>

  <!-- ... inne znaczniki zdefiniowane wcześniej ... -->

  <tag>
    <name>prime</name>
    <tagclass>coreservlets.tags.PrimeTag</tagclass>
    <info>Wyświetla N-cyfrową, losową liczbę pierwszą.</info>
    <bodycontent>EMPTY</bodycontent>
    <attribute>
      <name>length</name>
      <required>false</required>
```

```
    </attribute>
  </tag>

  <!-- Inne znaczniki zdefiniowane w dalszej części rozdziału ... -->

</taglib>
```

Plik JSP

Na listingu 14.9 przedstawiłem dokument JSP zawierający dyrektywę `taglib`, która powoduje załadowanie pliku deskryptora biblioteki znaczników i określa, że prefiks znaczników będzie miał postać `csajsp`. Używany w tym przykładzie znacznik `prime` został zadeklarowany w taki sposób, iż może zawierać atrybut `length`; a zatem, w dokumencie przedstawionym na listingu 14.9, jest on zapisywany w następującej postaci:

```
<csajsp:prime length="xxx" />
```

Pamiętaj, że znaczniki definiowane przez programistów są zapisywane zgodnie z zasadami składni języka XML, która wymaga by wartości atrybutów były umieszczane pomiędzy znakami cudzysłowu lub apostrofu. Atrybut `length` jest opcjonalny, a zatem można go także zapisać w następującej postaci:

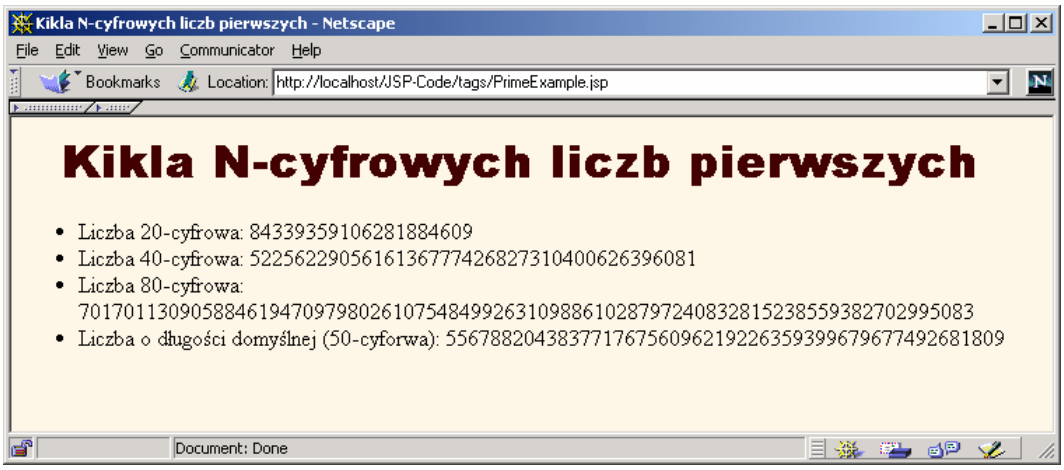
```
<csajsp:prime />
```

W takim przypadku klasa obsługi znacznika odpowiada za określenie sensownej, domyślnej wartości atrybutu.

Wyniki wykonania strony z listingu 14.9 przedstawiłem na rysunku 14.3.

Listing 14.9 PrimeExample.jsp

```
<%@ page contentType="text/html; charset=ISO-8859-2" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Kikla N-cyfrowych liczb pierwszych</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<H1>Kikla N-cyfrowych liczb pierwszych</H1>

<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>

<UL>
  <LI>Liczba 20-cyfrowa: <csajsp:prime length="20" />
  <LI>Liczba 40-cyfrowa: <csajsp:prime length="40" />
  <LI>Liczba 80-cyfrowa: <csajsp:prime length="80" />
  <LI>Liczba o długości domyślnej (50-cyforwa): <csajsp:prime />
</UL>

</BODY>
</HTML>
```



Rysunek 14.3 Wyniki wykonania strony PrimeExample.jsp

14.4 Dołączanie zawartości znacznika

Wszystkie znaczniki które stworzyliśmy do tej pory ignorowały wszelką zawartość umieszczaną wewnątrz nich i z tego względu mogły być zapisywane w skróconej postaci:

```
<prefiks:nazwa />
```

W tej części rozdziału pokażę jak należy definiować znaczniki, które wykorzystują umieszczone wewnątrz nich informacje, i są zapisywane w następującej formie:

```
<prefiks:nazwa>zawartość</prefiks:nazwa>
```

Klasa obsługi znacznika

Klasy obsługi znaczników przedstawione w poprzednich przykładach definiowały metody `doStartTag`, które zwracały wartość `SKIP_BODY`. Aby poinformować system, iż należy wykorzystać treść podaną pomiędzy znacznikiem otwierającym i zamykającym, metoda `doStartTag` powinna zwrócić wartość `EVAL_BODY_INCLUDE`. Treść znacznika może zawierać elementy skryptowe JSP, dyrektywy oraz znaczniki akcji, podobnie jak cała reszta dokumentu. Wszelkie konstrukcje JSP w czasie przekształcania strony są zamieniane na kod serwletu, który jest następnie wywoływany w momencie obsługi żądania.

Jeśli masz zamiar korzystać z zawartości znacznika, to być może będziesz chciał wykonać jakieś czynności także *po* jej przetworzeniu. Czynności te są wykonywane w metodzie `doEndTag`. Niemal zawsze, po zakończeniu obsługi własnego znacznika należy kontynuować przetwarzanie dalszej części dokumentu i dlatego metoda `doEndTag` powinna zwracać wartość `EVAL_PAGE`. Jeśli jednak chcesz przerwać przetwarzanie dalszej części strony, możesz to zrobić zwracając z metody `doEndTag` wartość `SKIP_PAGE`.

Na listingu 14.10 przedstawiłem klasę obsługującą elementy, które są znacznie bardziej elastyczne od standardowych znaczników od `<H1>` do `<H6>` języka HTML. Nasz przykładowy element pozwala na precyzyjne określenie wielkości czcionki, nazwy preferowanej czcionki (zostanie użyta pierwsza pozycja z listy czcionek zainstalowanych na komputerze użytkownika), koloru tekstu, koloru tła, obramowania oraz wyrównania (dostępne wartości to: `LEFT`, `CENTER` oraz `RIGHT`). Zwyczajne znaczniki `<Hx>` udostępniają wyłącznie możliwość określenia wyrównania na stronie. Nasz nagłówek zostanie zaimplementowany przy użyciu tabeli zawierającej jedną komórkę, w której zostanie umieszczony znacznik `SPAN` określający wygląd tekstu za pomocą arkusza stylów. Metoda `doStartTag` wygeneruje otwierające znaczniki `<TABLE>` oraz `<SPAN>` i po czym zwróci wartość `EVAL_BODY_INCLUDE` informując tym samym system, że należy dołączyć zawartość znacznika. Z kolei metoda `doEndTag` wygeneruje zamykające znaczniki `</SPAN>` oraz `</TABLE>` i

zwróci wartość `EVAL_PAGE` nakazując dalsze przetwarzanie dokumentu. Nasz znacznik posiada także kilka metod `setNazwaAtrybutu`, służących do obsługi atrybutów, takich jak `bgColor` lub `fontSize`.

Listing 14.10 HeadingTag.java

```
package coreservlets.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

/** Generuje nagłówek HTML o określonym kolorze tła,
 *  kolorze tekstu, wyrównaniu, kroju i wielkości czcionki.
 *  Można także wyświetlić obramowanie nagłówka, które zazwyczaj
 *  dokładnie obejmuje nagłówek, lecz może być szersze.
 *  Wszystkie atrybuty za wyjątkiem bgColor są opcjonalne.
*/

public class HeadingTag extends TagSupport {
  private String bgColor; // Jedyny wymagany atrybut
  private String color = null;
  private String align="CENTER";
  private String fontSize="36";
  private String fontList="Arial, Helvetica, sans-serif";
  private String border="0";
  private String width=null;

  public void setBgColor(String bgColor) {
    this.bgColor = bgColor;
  }

  public void setColor(String color) {
    this.color = color;
  }

  public void setAlign(String align) {
    this.align = align;
  }

  public void setFontSize(String fontSize) {
    this.fontSize = fontSize;
  }

  public void setFontList(String fontList) {
    this.fontList = fontList;
  }

  public void setBorder(String border) {
    this.border = border;
  }

  public void setWidth(String width) {
    this.width = width;
  }

  public int doStartTag() {
    try {
      JspWriter out = pageContext.getOut();
      out.print("<TABLE BORDER=" + border +
                " BGCOLOR=\"" + bgColor + "\"" +
                " ALIGN=\"" + align + "\"");
      if (width != null) {
        out.print(" WIDTH=\"" + width + "\"");
      }
      out.print("><TR><TH>");
      out.print("<SPAN STYLE=\"" +
                "font-size: " + fontSize + "px; " +
                "font-family: " + fontList + "; ");
      if (color != null) {
        out.println("color: " + color + ";");
      }
      out.print("\"> "); // koniec znacznika <SPAN ...>
    } catch(IOException ioe) {
      System.out.println("Błąd w znaczniku HeadingTag: " + ioe);
    }
    return(EVAL_BODY_INCLUDE); // Dołączamy zawartość
```

```
  }

  public int doEndTag() {
    try {
      JspWriter out = pageContext.getOut();
      out.print("</SPAN></TABLE>");
    } catch(IOException ioe) {
      System.out.println("Błąd w znaczniku HeadingTag: " + ioe);
    }
    return(EVAL_PAGE); // Przetwórz dalszą część dokumentu JPS
  }
}
```

Plik deskryptora biblioteki znaczników

W przypadku definiowania znaczników wykorzystujących własną zawartość, w elemencie `tag` wprowadzana jest tylko jedna modyfikacja — element `bodycontent` powinien zawierać wartość `JSP`:

```
<bodycontent>JSP</bodycontent>
```

Pozostałe elementy — `name`, `tagclass`, `info` oraz `attribute` — są stosowane tak samo jak w poprzednich przypadkach. Kod pliku deskryptora biblioteki znaczników wraz z definicją naszego nowego znacznika przedstawiłem na listingu 14.11.

Listing 14.11 csajsp-taglib.tld

```
<?xml version="1.0" encoding="ISO-8859-2" ?>
<!DOCTYPE taglib
 PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<!-- deskryptor biblioteki znaczników -->

<taglib>
  <!-- teraz domyślną przestrzenią jest
        "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
   -->

  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>csajsp</shortname>
  <urn></urn>
  <info>
    Biblioteka znaczników książki Core Servlets and JavaServer Pages,
    http://www.coreservlets.com/.
  </info>

  <!-- ... inne znaczniki zdefiniowane wcześniej ... -->

  <tag>
    <name>heading</name>
    <tagclass>coreservlets.tags.HeadingTag</tagclass>
    <info>Wyświetla nagłówek o postaci tabeli z jedną komórką.</info>
    <bodycontent>JSP</bodycontent>
    <attribute>
      <name>bgColor</name>
      <required>true</required>  <!-- bgColor jest wymagany -->
    </attribute>
    <attribute>
      <name>color</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>align</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>fontSize</name>
      <required>false</required>
    </attribute>
    <attribute>
```

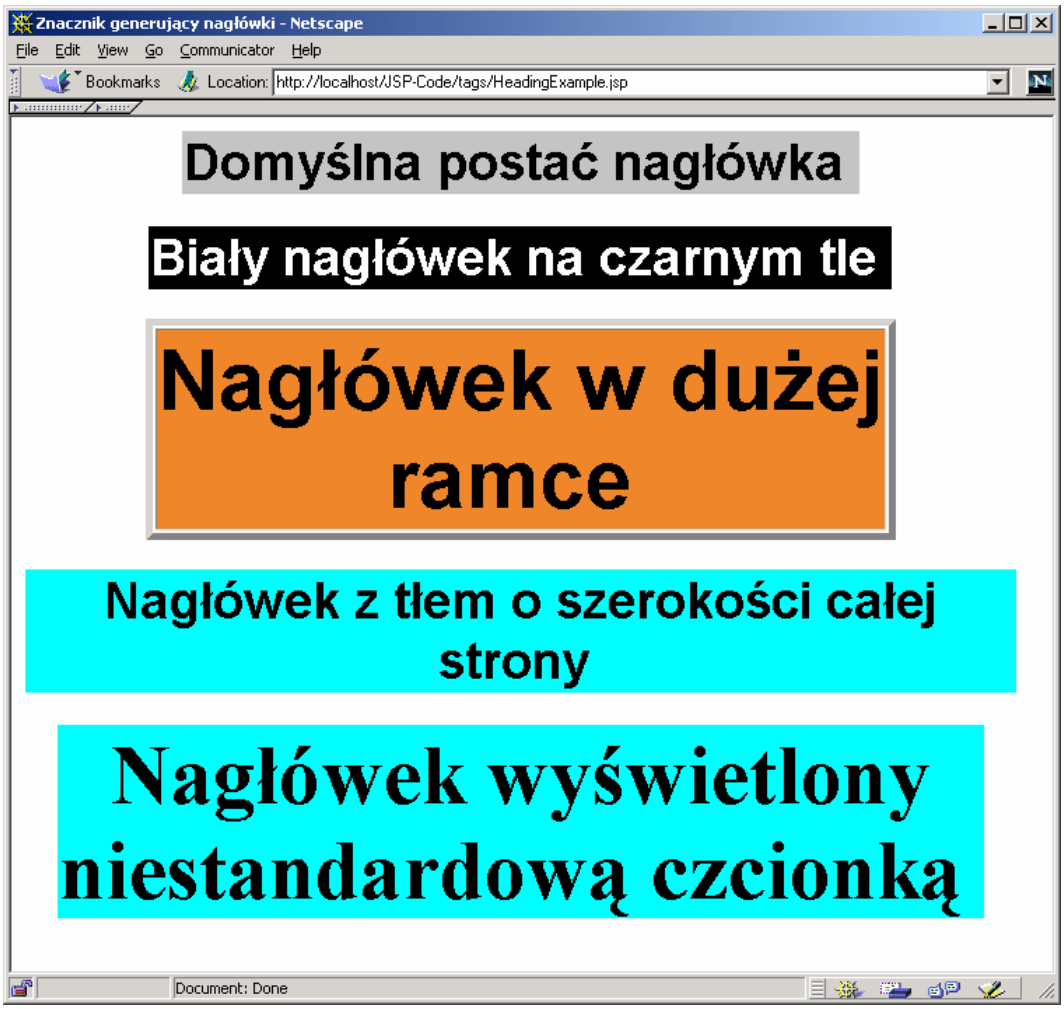
```
      <name>fontList</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>border</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>width</name>
      <required>false</required>
    </attribute>
  </tag>

  <!-- Inne znaczniki zdefiniowane w dalszej części rozdziału ... -->

</taglib>
```

Plik JSP

Listing 14.12 przedstawia kod źródłowy dokumentu JSP wykorzystującego zdefiniowany przed chwilą znacznik `heading`. Ponieważ atrybut `bgColor` został zdefiniowany jako wymagany, umieściłem go we wszystkich przykładach użycia znacznika. Wyniki wykonania tej strony JSP zostały przedstawione na rysunku 14.4.

Listing 14.12 HeadingExample.jsp

```
<%@ page contentType="text/html; encoding=ISO-8859-2" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Znacznik generujący nagłówki</TITLE>
</HEAD>

<BODY>
<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>

<csajsp:heading bgColor="#C0C0C0">
Domyślna postać nagłówka
</csajsp:heading>
<P>
<csajsp:heading bgColor="BLACK" color="WHITE">
Biały nagłówek na czarnym tle
</csajsp:heading>
<P>
<csajsp:heading bgColor="#EF8429" fontSize="60" border="5">
Nagłówek w dużej ramce
</csajsp:heading>
<P>
<csajsp:heading bgColor="CYAN" width="100%">
Nagłówek z tłem o szerokości całej strony
</csajsp:heading>
<P>
<csajsp:heading bgColor="CYAN" fontSize="60"
                fontList="Brush Script MT, Times, serif">
Nagłówek wyświetlony niestandardową czcionką
</csajsp:heading>
<P>
</BODY>
</HTML>
```



Rysunek 14.4 Znacznik csajsp:heading daje znacznie większą kontrolę nad wyglądem nagłówka niż standardowe znaczniki Hx języka HTML

14.5 Opcjonalne dołączanie zawartości znacznika

Większość znaczników albo *nigdy* nie wykorzystuje swojej zawartości lub robi to *zawsze*. W tej części rozdziału pokaże jak można wykorzystać informacje dostępne podczas obsługi żądania, aby zdecydować, czy zawartość znacznika ma zostać dołączona czy nie. Choć treść znacznika może zawierać kod JSP interpretowany w czasie przetwarzania strony, to jednak kod ten jest przekształcany do postaci serwletu i podczas obsługi żądania można go wywołać lub zignorować.

Klasa obsługi znacznika

Opcjonalne dołączania zawartości znacznika jest trywialne — otóż wystarczy, w zależności od wartości jakiegoś wyrażenia obliczanego w czasie obsługi żądania, zwrócić wartość `EVAL_BODY_INCLUDE` bądź `SKIP_BODY`. Należy jednak wiedzieć w jaki sposób można pobrać potrzebne informacje, gdyż w metodzie `doStartTag` nie ma dostępnych obiektów `HttpServletRequest` ani `HttpServletResponse` przekazywanych do metod `service`, `_jspService`, `doGet` oraz `doPost`. Rozwiązaniem tego problemu jest wykorzystanie metody `getRequest`, która zwraca obiekt `HttpServletRequest` przechowywany w automatycznie definiowanej zmiennej instancyjnej (polu) `pageContext` klasy `TagSupport`. Gwoli ścisłości należało by powiedzieć, iż metoda `getRequest` zwraca obiekt `ServletRequest`. Oznacza to, że aby korzystać z metod które nie zostały zdefiniowane w interfejsie `ServletRequest` konieczne będzie rzutowanie wartości zwróconej prze tę

metodę, do typu `HttpServletRequest`. Jednak w naszym przypadku będziemy używali wyłącznie metody `getParameter`, a zatem żadne rzutowanie typów nie będzie konieczne.

Listing 14.13 zawiera kod definiujący znacznik, który ignoruje umieszczoną wewnątrz niego zawartość, chyba że w czasie obsługi żądania okaże się, iż został zdefiniowany parametr wejściowy o nazwie `debug`. Taki znacznik udostępnia przydatną możliwość polegającą na umieszczaniu informacji testowych wewnątrz znacznika na etapie tworzenia kodu, a jednocześnie gwarantuje że informacje te będą wyświetlane wyłącznie w razie pojawienia się problemów.

Listing 14.13 DebugTag.java

```
package coreservlets.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import javax.servlet.*;

public class DebugTag extends TagSupport {
  public int doStartTag() {
    ServletRequest request = pageContext.getRequest();
    String debugFlag = request.getParameter("debug");
    if ((debugFlag != null) &&
        (!debugFlag.equalsIgnoreCase("false"))) {
      return(EVAL_BODY_INCLUDE);
    } else {
      return(SKIP_BODY);
    }
  }
}
```

Plik deskryptora biblioteki znaczników

Jeśli tworzony znacznik kiedykolwiek ma zamiar korzystać z umieszczonej wewnątrz niego zawartości, to w elemencie `bodycontent` musisz umieścić wyrażenie `JSP`. Poza tym, wszystkie elementy umieszczane wewnątrz elementu `tag` są stosowane w taki sam sposób jak w poprzednich przykładach. Listing 14.14 przedstawia definicję konieczną do użycia znacznika `DebugTag`.

Listing 14.14 csajsp-taglib.tld

```
<?xml version="1.0" encoding="ISO-8859-2" ?>
<!DOCTYPE taglib
 PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<!-- deskryptor biblioteki znaczników -->

<taglib>
  <!-- teraz domyślną przestrzenią jest
        "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
   -->

  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>csajsp</shortname>
  <urn></urn>
  <info>
    Biblioteka znaczników książki Core Servlets and JavaServer Pages,
    http://www.coreservlets.com/.
  </info>

  <!-- ... inne znaczniki zdefiniowane wcześniej ... -->

  <tag>
    <name>debug</name>
    <tagclass>coreservlets.tags.DebugTag</tagclass>
    <info>Dołącza zawartość wyłącznie jeśli określony jest parametr debug.</info>
    <bodycontent>JSP</bodycontent>
  </tag>
```

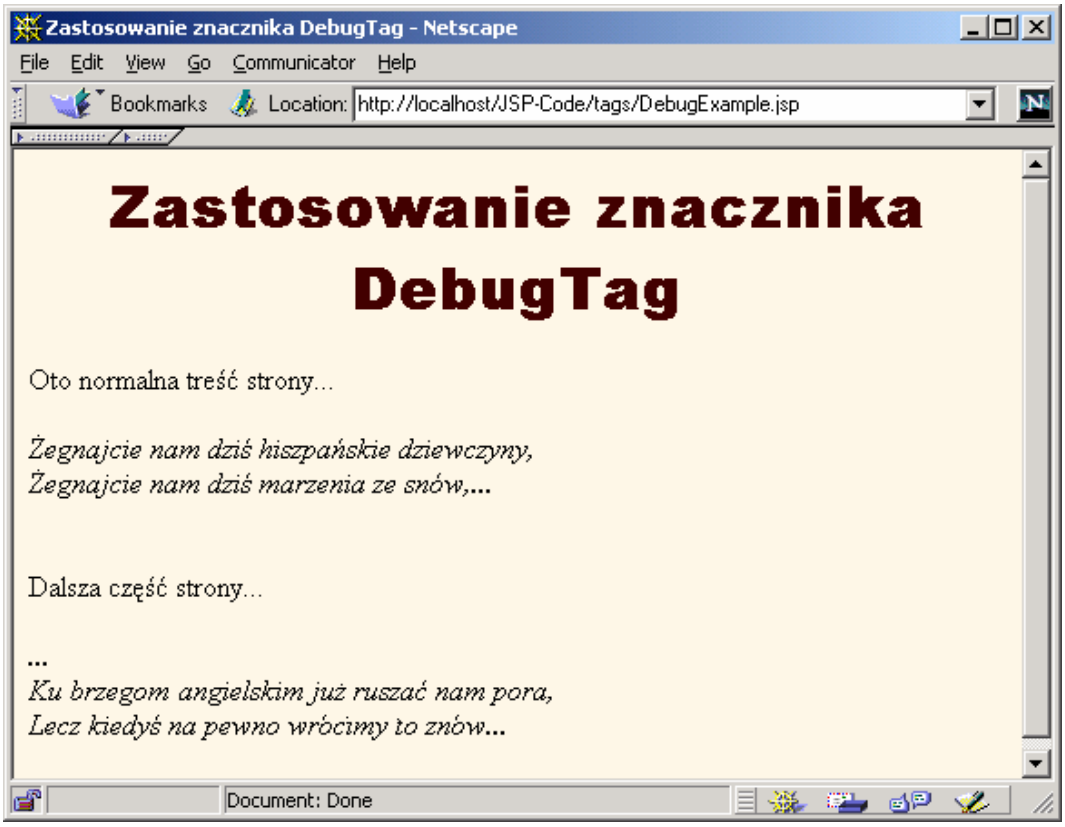
```
  <!-- Inne znaczniki zdefiniowane w dalszej części rozdziału ... -->

</taglib>
```

Plik JSP

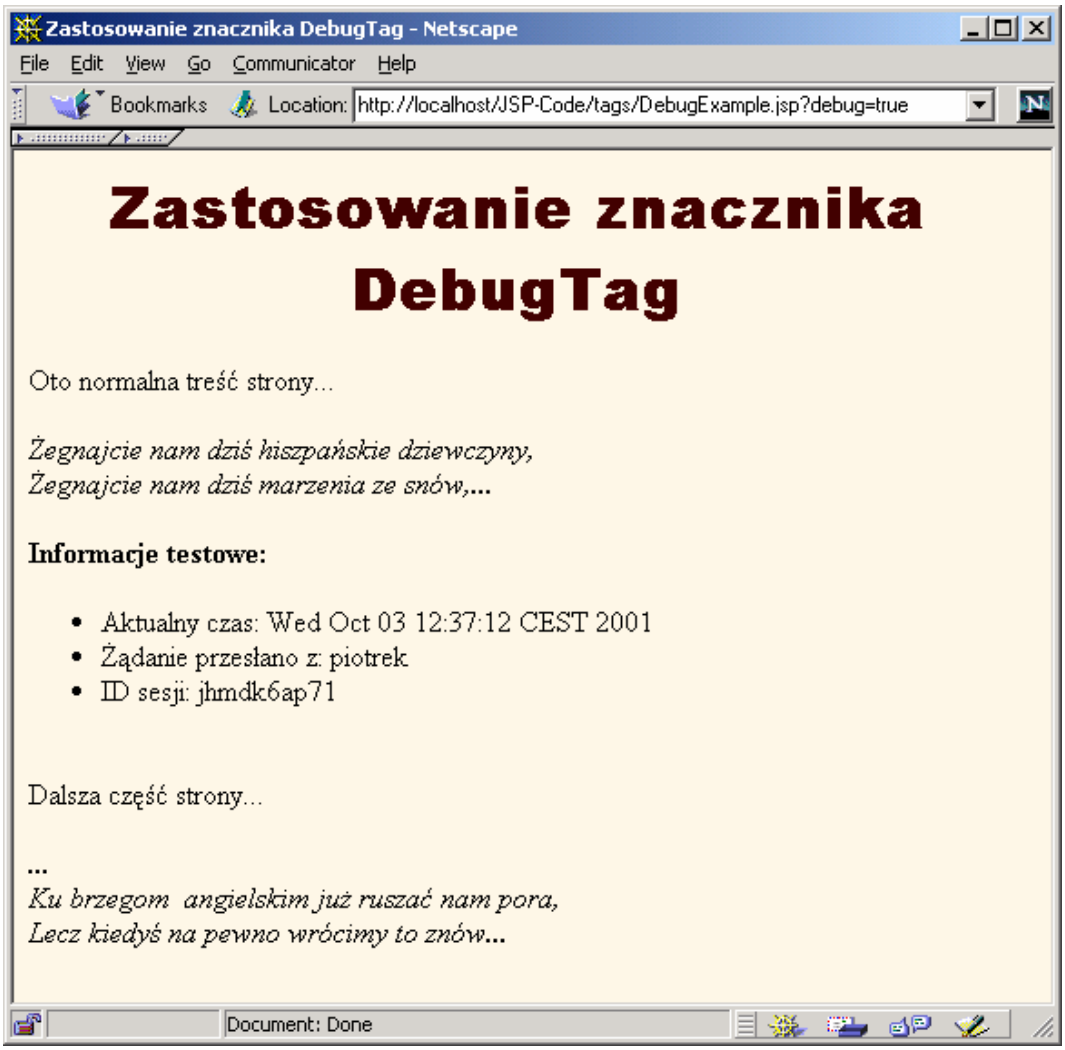
Na listingu 14.15 przedstawiłem dokument JSP zawierający informacje testowe, umieszczone pomiędzy znacznikami `<csajsp:debug>` oraz `</csajsp:debug>`. Na rysunku 14.5 można zobaczyć standardowe wyniki wygenerowane przez tę stronę, a na rysunku 14.6 wyniki uzyskane w sytuacji gdy został podany parametr `debug`.

Listing 14.15 DebugExample.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Zastosowanie znacznika DebugTag</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<H1>Zastosowanie znacznika DebugTag</H1>

<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>

Oto normalna treść strony...
<P>
<I>Żegnajcie nam dziś hiszpańskie dziewczyny,<BR>
Żegnajcie nam dziś marzenia ze snów,...</I>
</P>

<csajsp:debug>
<B>Informacje testowe:</B>
<UL>
  <LI>Aktualny czas: <%= new java.util.Date() %>
  <LI>Żądanie przesłano z: <%= request.getRemoteHost() %>
  <LI>ID sesji: <%= session.getId() %>
</UL>
</csajsp:debug>

<BR>
Dalsza część strony...

<P>
<I>...<BR>
Ku brzegom angielskim już ruszać nam pora,<BR>
Lecz kiedyś na pewno wrócimy to znów...</I>
</P>

</BODY>
</HTML>
```



Rysunek 14.5 Zazwyczaj zawartość znacznika csajsp:debug jest ignorowana



Rysunek 14.6 W razie podania parametru wejściowego debug, zawartość znacznika csajsp:debug jest wyświetlana

14.6 Manipulowanie zawartością znacznika

Znacznik `csajsp:prime` (patrz podrozdział 14.3) całkowicie ignorował swą zawartość, kolejny znacznik — `csajsp:heading` (patrz podrozdział 14.4) wykorzystywał ją, a ostatni z przedstawionych znaczników — `csajsp:debug` — wykorzystywał ją lub ignorował w zależności od wartości parametru wejściowego. Jednak wszystkie te znaczniki miały jedną cechę wspólną — ich zawartość nigdy nie była modyfikowana. Owszem, była ignorowana lub dołączana w oryginalnej postaci (po przekształceniu kodu JSP). W tej części rozdziału jak można przetwarzać zawartość znacznika.

Klasa obsługi znacznika

Jak na razie, wszystkie tworzone klasy były klasami potomnymi klasy `TagSupport`. Klasa ta stanowi doskonały punkt początkowy do tworzenia własnych znaczników, gdyż implementuje interfejs `Tag` i wykonuje wiele przydatnych czynności wstępnych, takich jak zapisanie odwołania do obiektu `PageContext` w zmiennej instancyjnej `pageContext`. Jednak możliwości udostępniane przez tę klasę nie są wystarczające w przypadkach tworzenia znaczników, które muszą manipulować swoją zawartością. Takie klasy, muszą być tworzone na bazie klasy `BodyTagSupport`.

`BodyTagSupport` jest klasą potomną klasy `TagSupport`, a zatem metody `doStartTag` oraz `doEndTag` są używane tak samo jak wcześniej. Jednak klasa ta udostępnia dwie nowe, ważne metody:

1. **`doAfterBody`** — tę metodę należy przesłonić w celu wykonania jakiś operacji na zawartości znacznika. Zazwyczaj powinna ona zwracać wartość `SKIP_BODY` oznaczającą, że w żaden inny sposób nie należy przetwarzać zawartości znacznika.
2. **`getBodyContent`** — ta metoda zwraca obiekt klasy `BodyContent`, w którym zostały zgromadzone informacje o zawartości znacznika.

Klasa `BodyContent` udostępnia trzy, bardzo ważne metody:

1. **`getEnclosingWriter`** — metoda ta zwraca obiekt `JspWriter` używany także przez metody `doStartTag` oraz `doEndTag`.
2. **`getReader`** — metoda zwraca obiekt `Reader`, przy użyciu którego można odczytać zawartość znacznika.
3. **`getString`** — metoda zwraca łańcuch znaków zawierający całą treść znacznika.

W podrozdziale 3.4 (pt.: „Przykład: Odczyt wszystkich parametrów") przedstawiłem statyczną metodę `filter`, która pobierała łańcuch znaków i zastępowała wszystkie odszukane w nim znaki `<`, `>`, `"` oraz `&` odpowiednimi symbolami HTML — `<`, `>`, `&qout;` oraz `&`. Metoda ta jest przydatna w sytuacjach gdy serwlet generuje łańcuchy znaków, które mogą zaburzyć strukturę i postać strony na jakiej są umieszczane. Na listingu 14.16 przedstawiłem implementację znacznika, który został wzbogacony o możliwości filtrowania zawartości.

Listing 14.16 FilterTag.java

```
package coreservlets.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import coreservlets.*;

/** Znacznik zamieniający znaki <, >, " oraz & na odpowiednie
 *  symbole HTML (&lt;, &gt;, &quot;, and &amp;).
 *  Po przefiltrowaniu, dowolne łańcuchy znaków mogą zostać
 *  osadzone bądź to w treści generowanej strony WWW, lub
 *  też można ich użyć jako wartości atrybutów znaczników.
*/

public class FilterTag extends BodyTagSupport {
  public int doAfterBody() {
    BodyContent body = getBodyContent();
    String filteredBody =
      ServletUtilities.filter(body.getString());
    try {
      JspWriter out = body.getEnclosingWriter();
      out.print(filteredBody);
    } catch(IOException ioe) {
      System.out.println("Błąd w znaczniku FilterTag: " + ioe);
    }
    // SKIP_BODY oznacza że praca została zakończona. Aby
    // jeszcze raz przetworzyć i obsłużyć zawartość znacznika,
    // należało by zwrócić wartość EVAL_BODY_TAG.
    return(SKIP_BODY);
  }
}
```

Plik deskryptora biblioteki znaczników

Znaczniki operujące na swojej zawartości powinne używać elementu `bodycontent` tak samo jak znaczniki, które wyświetlają swoją zawartość w oryginalnej postaci — powinny stosować wartość `JSP`. Poza tym metody korzystania z pliku deskryptora biblioteki znaczników nie ulegają zmianie, o czym możesz się przekonać analizując plik przedstawiony na listingu 14.17.

Listing 14.17 csajsp-taglib.tld

```
<?xml version="1.0" encoding="ISO-8859-2" ?>
<!DOCTYPE taglib
 PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<!-- deskryptor biblioteki znaczników -->

<taglib>
  <!-- teraz domyślną przestrzenią jest
        "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
   -->

  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>csajsp</shortname>
  <urn></urn>
  <info>
    Biblioteka znaczników książki Core Servlets and JavaServer Pages,
    http://www.coreservlets.com/.
  </info>

  <!-- ... inne znaczniki zdefiniowane wcześniej ... -->

  <tag>
    <name>filter</name>
    <tagclass>coreservlets.tags.FilterTag</tagclass>
    <info>Zastępuje znaki specjalne HTML umieszczone w zawartości.</info>
    <bodycontent>JSP</bodycontent>
  </tag>

  <!-- Inne znaczniki zdefiniowane w dalszej części rozdziału ... -->

</taglib>
```

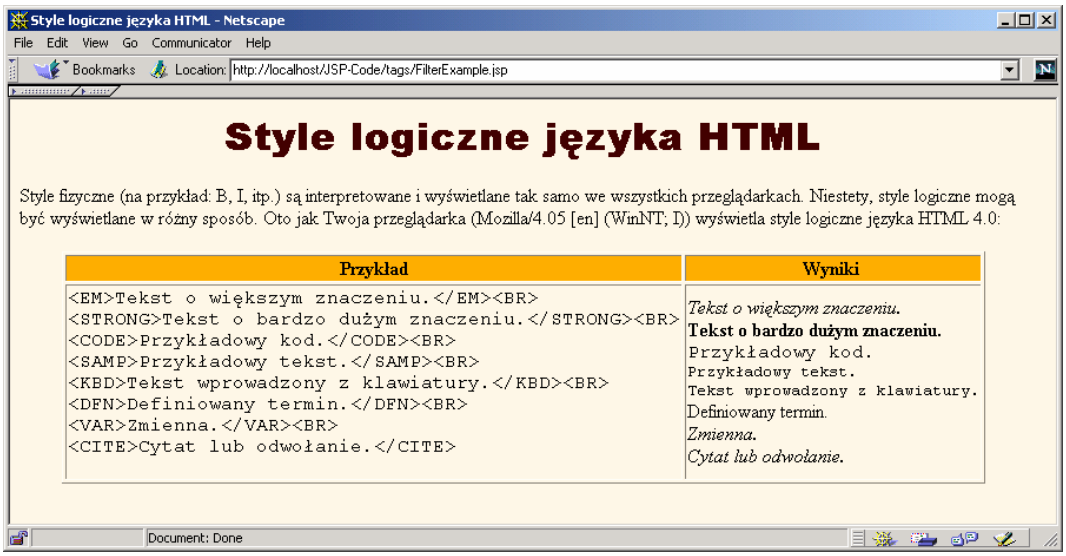
Plik JSP

Listing 14.18 przedstawia stronę JSP która wyświetla w tabeli przykładowy kod HTML oraz jego wyniki. Stworzenie takiej tabeli w języku HTML byłoby dosyć uciążliwe, gdyż w całej zawartości komórki prezentującej kod HTML należy zamienić znaki < oraz > na odpowiednie symbole HTML — `<` oraz `>`. Wykonywanie tego zadania jest szczególnie pracochłonne na etapie tworzenia strony, gdyż przykładowy kod HTML może się bardzo często zmieniać. Na szczęście, cały proces można znacznie uprościć dzięki wykorzystaniu znacznika `csajsp:filter` (patrz listing 14.18). Wyniki wykonania strony z listingu 14.18 przedstawiłem na rysunku 14.6.

Listing 14.18 FilterExample.jsp

```
<%@ page contentType="text/html encoding=ISO-8859-2" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<!--
Przykład użycia znacznika FilterTag.
-->
<HTML>
<HEAD>
<TITLE>Style logiczne języka HTML</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<H1>Style logiczne języka HTML</H1>
Style fizyczne (na przykład: B, I, itp.) są interpretowane i
wyświetlane tak samo we wszystkich przeglądarkach. Niestety,
style logiczne mogą być wyświetlane w różny sposób. Oto jak
Twoja przeglądarka
(<%= request.getHeader("User-Agent") %>)
wyświetla style logiczne języka HTML 4.0:
<P>
```

```
<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>

<TABLE BORDER=1 ALIGN="CENTER">
<TR CLASS="COLORED"><TH>Przykład<TH>Wyniki
<TR>

<TD><PRE>
<csajsp:filter>
<EM>Tekst o większym znaczeniu.</EM><BR>
<STRONG>Tekst o bardzo dużym znaczeniu.</STRONG><BR>
<CODE>Przykładowy kod.</CODE><BR>
<SAMP>Przykładowy tekst.</SAMP><BR>
<KBD>Tekst wprowadzony z klawiatury.</KBD><BR>
<DFN>Definiowany termin.</DFN><BR>
<VAR>Zmienna.</VAR><BR>
<CITE>Cytat lub odwołanie.</CITE>
</csajsp:filter>
</PRE>

<TD>
<EM>Tekst o większym znaczeniu.</EM><BR>
<STRONG>Tekst o bardzo dużym znaczeniu.</STRONG><BR>
<CODE>Przykładowy kod.</CODE><BR>
<SAMP>Przykładowy tekst.</SAMP><BR>
<KBD>Tekst wprowadzony z klawiatury.</KBD><BR>
<DFN>Definiowany termin.</DFN><BR>
<VAR>Zmienna.</VAR><BR>
<CITE>Cytat lub odwołanie.</CITE>

</TABLE>
</BODY>
</HTML>
```



Rysunek 14.7 Znacznik csajsp:filter pozwala wyświetlać na stronach WWW tekst bez zwracania uwagi na to czy zawiera on znaki specjalne HTML czy nie

14.7 Wielokrotne dołączanie lub obsługa zawartością znacznika

Może się zdarzyć, że zamiast jednokrotnego wyświetlenia lub przetworzenia zawartości znacznika, będziesz chciał zrobić to kilka razy. Możliwość wielokrotnego dołączania zawartości znacznika umożliwia definiowanie wielu różnych znaczników iteracyjnych — czyli takich, które powtarzają określony fragment kodu JSP, na przykład, podaną ilość razy lub do momentu gdy

zostanie spełniony podany warunek logiczny. W tej części rozdziału dowiesz się, jak można tworzyć takie znaczniki.

Klasa obsługi znacznika

Znaczniki przetwarzające swą zawartość wiele razy powinny być klasami potomnymi klasy `BodyTagSupport` i implementować metody `doStartTag`, `doEndTag` oraz, przed wszystkim, metodę `doAfterBody` (podobnie jak znaczniki wykorzystujące swą zawartość jeden raz, patrz poprzedni podrozdział). Różnica polega na wartości zwracanej przez metodę `doAfterBody`. Jeśli metoda ta zwróci wartość `EVAL_BODY_TAG`, to zawartość znacznika zostanie przetworzona jeszcze raz, co spowoduje kolejne wywołanie metody `doAfterBody`. Proces ten będzie powtarzany do momentu, gdy metoda `doAfterBody` zwróci wartość `SKIP_BODY`.

Listing 14.19 definiuje znacznik, który powtarza swoją zawartość podaną ilość razy. Liczba powtórzeń określana jest przy użyciu atrybut `reps`. Ponieważ wewnątrz znacznika umieszczony jest kod JSP (który w momencie przetwarzania strony jest przekształcany do postaci serwletu, lecz nie wykonywany) to każda iteracja może spowodować wygenerowanie różnych wyników.

Listing 14.19 RepeatTag.java

```
package coreservlets.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

/** Znacznik który kilkukrotnie wyświetla swoją
 *  zawartość.
*/

public class RepeatTag extends BodyTagSupport {
  private int reps;

  public void setReps(String repeats) {
    try {
      reps = Integer.parseInt(repeats);
    } catch(NumberFormatException nfe) {
      reps = 1;
    }
  }

  public int doAfterBody() {
    if (reps-- >= 1) {
      BodyContent body = getBodyContent();
      try {
        JspWriter out = body.getEnclosingWriter();
        out.println(body.getString());
        // czyścimy aby przygotować do następnego wykonania
        body.clearBody();
      } catch(IOException ioe) {
        System.out.println("Błąd w znaczniku RepeatTag: " + ioe);
      }
      return(EVAL_BODY_TAG);
    } else {
      return(SKIP_BODY);
    }
  }
}
```

Plik deskryptora biblioteki znaczników

Listing 14.20 przedstawia plik TLD nadający naszemu nowemu znacznikowi nazwę `csajsp:repeat`. Aby możliwe było określenie wartości atrybuty `reps` podczas obsługi żądani, w definiującym go elemencie `tag` umieściłem element `rtexprvalue` o wartości `true`.

Listing 14.20 csajsp-taglib.tld

```
<?xml version="1.0" encoding="ISO-8859-2" ?>
<!DOCTYPE taglib
 PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<!-- deskryptor biblioteki znaczników -->

<taglib>
  <!-- teraz domyślną przestrzenią jest
        "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
   -->

  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>csajsp</shortname>
  <urn></urn>
  <info>
    Biblioteka znaczników książki Core Servlets and JavaServer Pages,
    http://www.coreservlets.com/.
  </info>

  <!-- ... inne znaczniki zdefiniowane wcześniej ... -->

  <tag>
    <name>repeat</name>
    <tagclass>coreservlets.tags.RepeatTag</tagclass>
    <info>Powtarza zawartość określoną ilość razy.</info>
    <bodycontent>JSP</bodycontent>
    <attribute>
      <name>reps</name>
      <required>true</required>
      <!-- element rtexprvalue określa czy atrybut
           może być wyrażeniem JSP. -->
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>

  <!-- Inne znaczniki zdefiniowane w dalszej części rozdziału ... -->

</taglib>
```

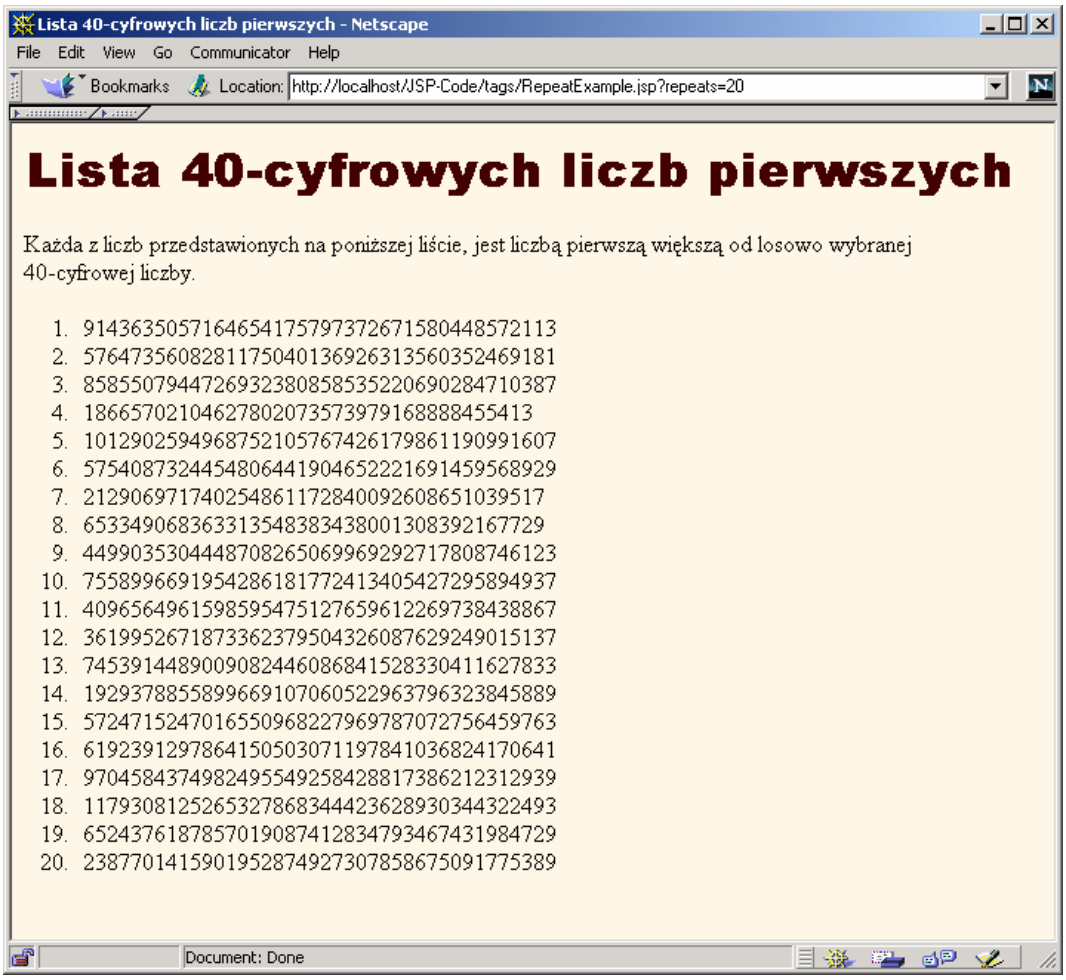
Plik JSP

Listing 14.21 przedstawia dokument JSP tworzący ponumerowaną listę liczb pierwszych. Ilość liczb wyświetlanych na liście określana jest w czasie obsługi żądania, na podstawie parametru o nazwie `repeats`. Przykładowe wyniki wykonania tej strony zostały przedstawione na rysunku 14.8.

Listing 14.21 RepeatExample.jsp

```
<%@ page contentType="text/html; encoding=ISO-8859-2" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Lista 40-cyfrowych liczb pierwszych</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<H1>Lista 40-cyfrowych liczb pierwszych</H1>
Każda z liczb przedstawionych na poniższej liście, jest
liczbą pierwszą większą od losowo wybranej 40-cyfrowej liczby.

<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>

<OL>
<!-- Powtóż to N razy. Pominięcie atrybutu reps oznacza, że
     zawartość znacznika ma być wyświetlona tylko raz. -->
```

```
<csajsp:repeat reps='<%= request.getParameter("repeats") %>'>
  <LI><csajsp:prime length="40" />
</csajsp:repeat>
</OL>

</BODY>
</HTML>
```



Rysunek 14.8 Wyniki wykonania strony RepeatExample.jsp w przypadku gdy parametr repeats miał wartość 20

14.8 Stosowanie znaczników zagnieżdżonych

Choć w przykładzie przedstawionym na listingu 14.21 znacznik `csajsp:prime` został umieszczony wewnątrz znacznika `csajsp:repeat`, to jednak oba te znaczniki są od siebie całkowicie niezależne. Pierwszy z nich generuje liczbę pierwszą zupełnie niezależnie od tego w jakim miejscu zostanie użyty; natomiast drugi znacznik powtarza swą zawartość określoną ilość razy, zupełnie niezależnie od tego czy znajduje się w niej znacznik `csajsp:prime` czy też nie.

Jednak niektóre znaczniki zależą od kontekstu, czyli od tego wewnątrz jakiego znacznika się znajdują. Przykładem mogą tu być elementy `TD` oraz `TH`, które muszą być umieszczone wewnątrz elementów `TR`; a z kolei elementy `TR` mogą być umieszczane wyłącznie wewnątrz elementów `TABLE`. Ustawienia dotyczące kolorów i wyrównania określone w elemencie `TABLE` są także wykorzystywane przez elementy `TR`, a z kolei wartości elementów `TR` wpływają na postać i działanie elementów `TD` i `TH`. A zatem, takie „zagnieżdżane” elementy nie działają niezależnie, nawet jeśli zostaną poprawnie zagnieżdżone. Na podobnej zasadzie plik deskryptora biblioteki znaczników

wykorzystuje wiele elementów — takich jak `taglib`, `tag`, `attribute` oraz `required` — które muszą być zapisywane w ściśle określonej hierarchii.

W tej części rozdziału pokażę jak można definiować znaczniki, których zachowanie zależy od kolejności zagnieżdżania, a działanie — od wartości określonych w innych znacznikach.

Klasy obsługi znaczników

Klasy definiujące znaczniki zagnieżdżane mogą być klasami potomnymi klas `TagSupport` bądź `BodyTagSupport`. Wybór użytej klasy bazowej zależy od tego czy definiowany znacznik będzie przetwarzać swoją zawartość (w takich przypadku klasą bazową powinna być klasa `BodyTagSupport`) czy też będzie ją ignorować bądź wyświetlać w oryginalnej postaci (takie znaczniki, znacznie bardziej popularne, są klasami potomnymi klasy `TagSupport`).

Jednak podczas implementacji znaczników zagnieżdżanych wykorzystywane są dwa, całkowicie nowe rozwiązania. Po pierwsze, klasy zagnieżdżane używają metody `findAncestorWithClass`, aby określić znacznik, a jakim zostały umieszczone. Argumentami wywołania tej metody są: odwołanie do bieżącej klasy (na przykład: `this`) oraz obiekt `Class` określający klasę zewnętrznego znacznika (na przykład: `ZewnetrznyZnacznik.class`). Jeśli znacznik zewnętrzny nie zostanie odnaleziony, to metoda zgłasza wyjątek `JspTagException`, który informuje o zaistniałym problemie. Po drugie, jeśli pewna klasa chce przechować informacje, z których inna klasa będzie mogła później skorzystać, to będzie mogła je umieścić w kopii obiektu reprezentującego zewnętrzny znacznik. Definicja takiego zewnętrznego znacznika, powinna zawierać metody pozwalające na zapisanie i pobranie takich informacji. Listing 14.22 przedstawia przykład ilustrujący sposób implementacji znaczników zagnieżdżanych.

Listing 14.22 Szablon znaczników zagnieżdżanych

```
public class ZnacznikZewn extends TagSupport {
  public void setJakasWartosc(JakasKlasa arg) { ... }
  public JakasKlasa getJakasWartosc() { ... }
}

public class PierwszyZnacznikWewn extends BodyTagSupport {
  public int doStartTag() throws JspTagException {
    ZnacznikZewn zewnetrzny =
      (ZnacznikZewn)findAncestorWithClass(this, ZnacznikZewn.class);
    if (zewnetrzny == null) {
      throw new JspTagException("nieprawidłowe zagnieżdżenie znaczników");
    } else {
      zewnetrzny.setJakasWartosc(...);
    }
    return(EVAL_BODY_TAG);
  }
  ...
}

public class DrugiZnacznikWewn extends BodyTagSupport {
  public int doStartTag() throws JspTagException {
    ZnacznikZewn zewnetrzny =
      (ZnacznikZewn)findAncestorWithClass(this, ZnacznikZewn.class);
    if (parent == null) {
      throw new JspTagException("nieprawidłowe zagnieżdżenie znaczników");
    } else {
      JakasKlasa wartosc = parent.getSomeValue();
      zrobCosZ(wartosc);
    }
    return(EVAL_BODY_TAG);
  }
  ...
}
```

Załóżmy teraz, że chcemy zdefiniować grupę znaczników, które będą używane w następujący sposób:

```
<csajsp:if>
```

```
  <csajsp:condition><%= jakieśWyrażenie %><csajsp:condition>
  <csajsp:then>Kod JSP dołączany gdy warunek jest spełniony</csajsp:then>
  <csajsp:else>Kod dołączany gdy warunek nie jest spełniony </csajsp:else>
</csajsp:if>
```

Pierwszym krokiem jaki należy wykonać aby zrealizować to zadanie, jest zdefiniowanie klasy `IfTag`, która będzie obsługiwać znacznik `csajsp:if`. Klasa ta powinna dysponować metodami pozwalającymi na podanie i sprawdzenie wartości warunku (będą to metody `setCondition` oraz `getCondition`) jak również metodami umożliwiającymi określenie i sprawdzenie czy warunek kiedykolwiek został jawnie określony (będą to metody `setHasCondition` oraz `getHasCondition`). Ta druga para metod będzie mam potrzebna gdyż, chcemy zabronić przetwarzania znaczników `csajsp:if`, w których nie ma podanego znacznika `csajsp:condition`. Kod klasy `IfTag` przedstawiłem na listingu 14.23.

Listing 14.23 IfTag.java

```
package coreservlets.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import javax.servlet.*;

/** Znacznik działający jak instrukcja if/then/else.
 */

public class IfTag extends TagSupport {
  private boolean condition;
  private boolean hasCondition = false;

  public void setCondition(boolean condition) {
    this.condition = condition;
    hasCondition = true;
  }

  public boolean getCondition() {
    return(condition);
  }

  public void setHasCondition(boolean flag) {
    this.hasCondition = flag;
  }

  /** Czy wartość pola warunku została jawnie określona? */

  public boolean hasCondition() {
    return(hasCondition);
  }

  public int doStartTag() {
    return(EVAL_BODY_INCLUDE);
  }
}
```

Kolejnym etapem jest zdefiniowanie klasy implementującej znacznik `csajsp:condition`. Klasa ta, o nazwie `IfConditionTag`, definiuje metodę `doStartTag`, która tylko i wyłącznie sprawdza czy znacznik jest umieszczony wewnątrz znacznika `csajsp:if`. Jeśli znacznik został poprawnie zagnieżdżony, to metoda zwraca wartość `EVAL_BODY_TAG`, a w przeciwnym razie — zgłasza wyjątek. Metoda `doAfterBody` klasy `IfConditionTag` sprawdza zawartość znacznika (posługując się metodą `getBodyContent`), zamieni ją do postaci łańcucha znaków (przy użyciu metody `getString`) i porównuje z łańcuchem znaków `"true"`. Ten sposób działania oznacza, że zamiast wyrażenia o postaci `<%= wyrażenie %>`, wewnątrz znacznika można umieścić wartość `"true"`; rozwiązanie takie może być przydatne w początkowych etapach tworzenia strony, gdy będziesz chciał, aby zawsze była wyświetlana zawartość znacznika `csajsp:then`. Wykorzystanie porównania z wartością `"true"` oznacza także, iż wszystkie pozostałe wartości będą traktowane jako `"false"` (czyli logiczna

nieprawda). Po przeprowadzeniu tego porównania, jego wynik jest zapisywany w zewnętrznym znaczniku `IfTag`, przy użyciu metody `setCondition`. Kod klasy `IfConditionTag` został przedstawiony na listingu 12.24.

Listing 14.24 IfConditionTag.java

```
package coreservlets.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import javax.servlet.*;

/** Część określająca warunek w znaczniku if.
*/

public class IfConditionTag extends BodyTagSupport {
  public int doStartTag() throws JspTagException {
    IfTag parent =
      (IfTag)findAncestorWithClass(this, IfTag.class);
    if (parent == null) {
      throw new JspTagException("Warunek nie jest umieszczony wewnątrz znacznika if");
    }
    return(EVAL_BODY_TAG);
  }

  public int doAfterBody() {
    IfTag parent =
      (IfTag)findAncestorWithClass(this, IfTag.class);
    String bodyString = getBodyContent().getString();
    if (bodyString.trim().equals("true")) {
      parent.setCondition(true);
    } else {
      parent.setCondition(false);
    }
    return(SKIP_BODY);
  }
}
```

Kolejnym — trzecim — krokiem jest zdefiniowanie klasy, która będzie obsługiwać znacznik `csajsp:then`. Metoda `doStartTag` tej klasy sprawdza czy znacznik został umieszczony wewnątrz znacznika `csajsp:if` oraz czy został określony warunek (co z kolei pozwala nam sprawdzić czy wewnątrz znacznika `csajsp:if` został umieszczony znacznik `csajsp:condition`). Metoda `doAfterBody` tej klasy pobiera wartość warunku przechowywaną w klasie `IfTag` i jeśli warunek został spełniony (ma wartość `true`), to pobiera i wyświetla zawartość znacznika `csajsp:then`. Kod tej klasy został przedstawiony na listingu 14.25.

Listing 14.25 IfThenTag.java

```
package coreservlets.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import javax.servlet.*;

/** Część then znacznika if
 */

public class IfThenTag extends BodyTagSupport {
  public int doStartTag() throws JspTagException {
    IfTag parent =
      (IfTag)findAncestorWithClass(this, IfTag.class);
    if (parent == null) {
      throw new JspTagException("Then poza znacznikiem If");
    } else if (!parent.hasCondition()) {
      String warning =
        "Przed znacznikiem Then należy podać warunek (Condition)";
      throw new JspTagException(warning);
    }
    return(EVAL_BODY_TAG);
```

```
  }

  public int doAfterBody() {
    IfTag parent =
      (IfTag)findAncestorWithClass(this, IfTag.class);
    if (parent.getCondition()) {
      try {
        BodyContent body = getBodyContent();
        JspWriter out = body.getEnclosingWriter();
        out.print(body.getString());
      } catch(IOException ioe) {
        System.out.println("Błąd w znaczniku IfThenTag: " + ioe);
      }
    }
    return(SKIP_BODY);
  }
}
```

Ostatnim krokiem naszego przykładu jest zdefiniowanie klasy obsługującej znacznik `csajsp:else`. Klasa ta nazwa się `IfElseTag` i jest bardzo podobna do klasy obsługującej element `then` naszego przykładowego znacznika. Jedyna różnica pomiędzy nimi polega na tym, iż metoda `doAfterBody` klasy `IfElseTag` wyświetla zawartość znacznika `csajsp:else` wyłącznie w przypadku, gdy wartość warunku przechowywanego w obiekcie klasy `IfTag` (reprezentującym zewnętrzny znacznik `csajsp:if`) wynosi `false`. Kod klasy `IfElseTag` przedstawiłem na listingu 14.26.

Listing 14.26 IfElseTag.java

```
package coreservlets.tags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import javax.servlet.*;

/** Część else znacznika if
 */

public class IfElseTag extends BodyTagSupport {
  public int doStartTag() throws JspTagException {
    IfTag parent =
      (IfTag)findAncestorWithClass(this, IfTag.class);
    if (parent == null) {
      throw new JspTagException("Else poza znacznikiem If");
    } else if (!parent.hasCondition()) {
      String warning =
        "Przed znacznikiem Else należy podać warunek (Condition)";
      throw new JspTagException(warning);
    }
    return(EVAL_BODY_TAG);
  }

  public int doAfterBody() {
    IfTag parent =
      (IfTag)findAncestorWithClass(this, IfTag.class);
    if (!parent.getCondition()) {
      try {
        BodyContent body = getBodyContent();
        JspWriter out = body.getEnclosingWriter();
        out.print(body.getString());
      } catch(IOException ioe) {
        System.out.println("Błąd w znaczniku IfElseTag: " + ioe);
      }
    }
    return(SKIP_BODY);
  }
}
```

Plik deskryptora biblioteki znaczników

Choć zdefiniowane przed chwilą znaczniki muszą być zapisane w ściśle określonej kolejności, to w pliku TLD, każdy z nich musi zostać zadeklarowany niezależnie. Oznacza to, że

sprawdzanie poprawności zapisu (zagnieżdżania) znaczników odbywa się wyłącznie w czasie obsługi żądania, a nie na etapie przekształcania strony JSP do postaci serwletu. W zasadzie istnieje możliwość zmuszenia systemu do przeprowadzenia wstępnego sprawdzenia poprawności zapisu znaczników już na etapie przekształcania strony — służy do tego klasa `TagExtraInfo`. Klasa ta udostępnia metodę `getVariableInfo`, której można użyć do sprawdzenia czy podane atrybuty istnieją oraz gdzie są używane. Kiedy już zdefiniujesz klasę potomną klasy `TagExtraInfo` należy ją skojarzyć z klasą znacznika. Służy do tego element `teiclass` umieszczany w pliku TLD i stosowany tak samo jak element `tagclass`. Jednak w praktyce klasa `TagExtraInfo` nie jest dobrze udokumentowana, a używanie jej jest trudne i uciążliwe.

Listing 14.27 csajsp-taglib.tld

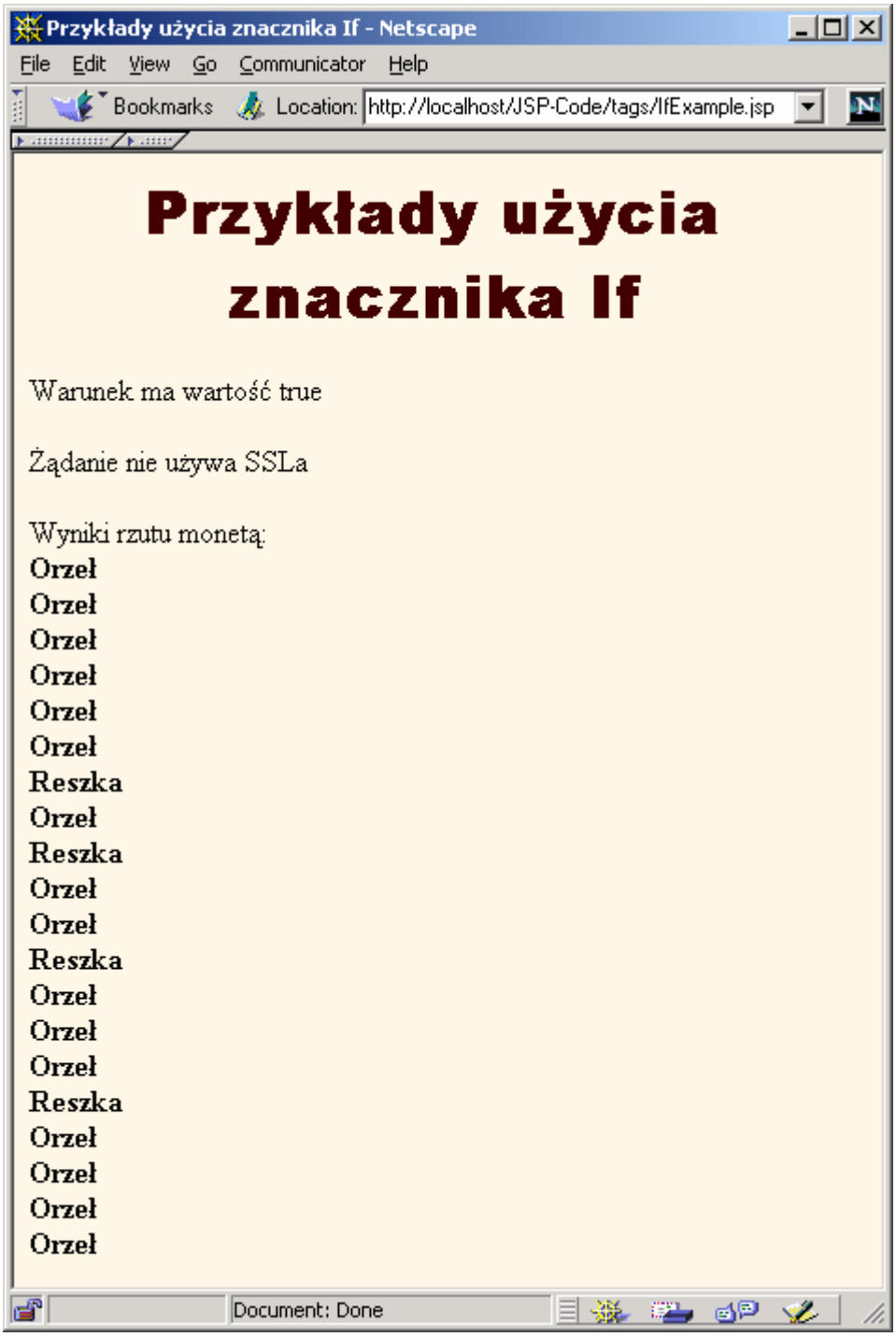
```
<?xml version="1.0" encoding="ISO-8859-2" ?>
<!DOCTYPE taglib
 PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
 "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<!-- deskryptor biblioteki znaczników -->

<taglib>
  <!-- teraz domyślną przestrzenią jest
        "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
   -->

  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>csajsp</shortname>
  <urn></urn>
  <info>
    Biblioteka znaczników książki Core Servlets and JavaServer Pages,
    http://www.coreservlets.com/.
  </info>

  <!-- ... inne znaczniki zdefiniowane wcześniej ... -->

  <tag>
    <name>if</name>
    <tagclass>coreservlets.tags.IfTag</tagclass>
    <info>Znacznik warunkowy: if/condition/then/else.</info>
    <bodycontent>JSP</bodycontent>
  </tag>

  <tag>
    <name>condition</name>
    <tagclass>coreservlets.tags.IfConditionTag</tagclass>
    <info>Część warunku (condition) znacznika if/condition/then/else.</info>
    <bodycontent>JSP</bodycontent>
  </tag>

  <tag>
    <name>then</name>
    <tagclass>coreservlets.tags.IfThenTag</tagclass>
    <info>Część "then" znacznika if/condition/then/else.</info>
    <bodycontent>JSP</bodycontent>
  </tag>

  <tag>
    <name>else</name>
    <tagclass>coreservlets.tags.IfElseTag</tagclass>
    <info>Część "else" znacznika if/condition/then/else.</info>
    <bodycontent>JSP</bodycontent>
  </tag>

</taglib>
```

Plik JSP

Listing 14.28 przedstawia stronę JSP w której znacznik `csajsp:if` jest wykorzystany na trzy różne sposoby. W pierwszym przykładzie użycia znacznika, w jego warunku na stałe została podana wartość `true`. W drugim przykładzie, wartość warunku określana jest na podstawie wartości parametru żądania HTTP. I w końcu w trzecim przykładzie znacznika `csajsp:if`, wartość jego warunku określana jest na podstawie porównania liczby pseudolosowej i pewnej stałej wartości. Przykładowe wyniki wykonania strony *IfExample.jsp* przedstawiłem na rysunku 14.9.

Listing 14.28 IfExample.jsp

```
<%@ page contentType="text/html; encoding=ISO-8859-2" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Przykłady użycia znacznika If</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<H1>Przykłady użycia znacznika If</H1>

<%@ taglib uri="csajsp-taglib.tld" prefix="csajsp" %>

<csajsp:if>
  <csajsp:condition>true</csajsp:condition>
  <csajsp:then>Warunek ma wartość true</csajsp:then>
  <csajsp:else>Warunek ma wartość false</csajsp:else>
</csajsp:if>
<P>
<csajsp:if>
  <csajsp:condition><%= request.isSecure() %></csajsp:condition>
  <csajsp:then>Żądanie używa SSLa (https)</csajsp:then>
  <csajsp:else>Żądanie nie używa SSLa</csajsp:else>
</csajsp:if>
<P>
Wyniki rzutu monetą:<BR>
<csajsp:repeat reps="20">
  <csajsp:if>
    <csajsp:condition>
      <%= Math.random() > 0.5 %>
    </csajsp:condition>
    <csajsp:then><B>Orzeł</B><BR></csajsp:then>
    <csajsp:else><B>Reszka</B><BR></csajsp:else>
  </csajsp:if>
</csajsp:repeat>

</BODY>
</HTML>
```



Rysunek 14.9 Wyniki wykonania strony IfExample.jsp

Rozdział 15.
Integracja serwletów i dokumentów JSP

Serwlety doskonale nadają do realizacji zadań wymagających zwyczajnej pracy programistycznej. Jak się przekonałeś czytając tę książkę, serwlety mogą określać kod statusu oraz nagłówki odpowiedzi HTTP, wykorzystywać cookies, przechowywać informacje pomiędzy kolejnymi żądaniami, kompresować tworzone strony WWW, generować obrazy GIF oraz efektywnie i elastycznie wykonywać wiele innych czynności. Jednak generacja kodu HTML z poziomu serwletów może być nieco uciążliwa, a uzyskane w ten sposób wyniki są trudne do modyfikacji. I właśnie w tym miejscu na arenę wkracza technologia JSP, która w znacznym stopniu pozwala oddzielić część prezentacyjną dokumentów od informacji generowanych dynamicznie. Dzięki temu można tworzyć kod HTML w standardowy sposób, używając do tego nawet specyficznych narzędzi służących do tego celu, a następnie przekazać stronę programistom JSP. Wyrażenia JSP, skryptlety oraz deklaracje pozwalają na umieszczanie prostego kodu napisanego w języku Java, w kodzie serwletu tworzonego na podstawie strony JSP; natomiast dyrektywy JSP pozwalają na określanie ogólnej postaci strony. Aby zaspokoić bardziej złożone wymagania możesz umieścić swój kod w komponentach JavaBeans lub zdefiniować swoje własne znaczniki.

Wspaniale! A zatem wiemy już wszystko co będzie nam potrzebne, prawda? Otóż… jeszcze nie. Do tej pory tworząc dokumenty JSP zakładaliśmy, że będą one określały jedyny, ogólny sposób prezentacji. A co zrobić jeśli chcemy wygenerować zupełnie inne wyniki w zależności od trzymanych danych? Komponenty oraz własne znaczniki, choć ich możliwości są bardzo duże i elastyczne, nie są w stanie przezwyciężyć ograniczenia, polegającego na tym, iż dokumenty JSP definiują względnie niezmienny, ogólny wygląd strony. Rozwiązaniem tego problemu jest *wspólne* wykorzystanie zarówno serwletów jak i stron Java Server Pages. Jeśli musisz stworzyć skomplikowaną aplikację, wymagającą zastosowania kilku sposobów prezentacji znacząco różniących się od siebie, to możesz zastosować rozwiązanie polegające na wstępnym przetworzeniu żądania przez serwlet, który przetworzy dane, zainicjalizuje komponenty, a następnie, w zależności od okoliczności, przekaże wyniki do jednego z kilku różnych dokumentów JSP. We wczesnych wersjach specyfikacji technologii Java Server Pages, metoda ta była określana jako *podejście do programowania JSP poziomu drugiego*. Zamiast całkowicie przekazywać żądanie, serwlet może samodzielnie wygenerować część wyników, a następnie dołączyć do nich wyniki wygenerowane przez jedną bądź kilka strony JSP.

15.1 Przekazywanie żądań

Kluczowym narzędziem umożliwiającym serwletom na przekazywanie żądań lub dołączanie do wyników zewnętrznych informacji, jest obiekt `RequestDispatcher`. Obiekt ten można uzyskać wywołując metodę `getRequestDispatcher` obiektu `ServletContext` i podając w jej wywołaniu

względny adres URL. na przykład, aby uzyskać obiekt `RequestDispatcher` skojarzony z dokumentem *http://host/prezentacje/prezentacja1.jsp* należałoby użyć następującego fragmentu kodu:

```
String url = "/prezentacje/prezentacja1.jsp";
RequestDispatcher dipatcher =
  getServletContext().getRequestDispatcher(url);
```

Gdy już będziesz dysponował obiektem `RequestDispatcher`, możesz wywołać metodę `forward`, aby całkowicie przekazać obsługę żądania pod skojarzony z nim adres, bądź metodę `include` — aby wyświetlić wyniki wykonania wskazanego zasobu. Obie metody wymagają podania dwóch argumentów — obiektów `HttpServletRequest` oraz `HttpServletResponse`. Obie metody zgłaszają także te same, dwa wyjątki — `ServletException` oraz `IOException`. Na listingu 15.1 przedstawiłem fragment serwletu, który, w zależności od wartości parametru `operacja`, przekazuje żądanie do jednej z trzech stron JSP. Aby uniknąć kilkukrotnego powtarzania wywołań metod `getRequestDispatcher` oraz `forward` posłużyłem się metodą `gotoPage`, która wymaga przekazania trzech argumentów — adresu URL, obiektu `HttpServletRequest` oraz `HttpServletResponse`. Metoda ta pobiera obiekt `RequestDispatcher`, a następnie wywołuje jego metodę `forward`.

Listing 15.1 Przykład przekazywania żądań.

```
// ForwardSnippet.java

public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
  String operation = request.getParameter("operacja");
  if (operation == null) {
    operation = "unknown";
  }
  if (operation.equals("operacja1")) {
    gotoPage("/operacje/prezentacja1.jsp",
             request, response);
  } else if (operation.equals("operacja2")) {
    gotoPage("/operacje/prezentacja2.jsp",
             request, response);
  } else {
    gotoPage("/operacje/nieznaneZadanie.jsp",
             request, response);
  }
}

private void gotoPage(String address,
                      HttpServletRequest request,
                      HttpServletResponse response)
    throws ServletException, IOException {
  RequestDispatcher dispatcher =
    getServletContext().getRequestDispatcher(address);
  dispatcher.forward(request, response);
}
```

Użycie zasobów statycznych

W większości przypadków żądania będą przekazywane do stron JSP lub innych serwletów. Jednak może się zdarzyć, że będziesz chciał przekazać je do zwyczajnego, statycznego dokumentu HTML. Na przykład, na witrynie zajmującej się handlem elektronicznym, żądania w których nie będzie informacji o poprawnym koncie użytkownika mogą by przekazywane do dokumentu HTML zawierającego formularz służący do pobrania koniecznych informacji. W przypadku żądań `GET` przekazywanie ich do statycznych dokumentów HTML jest dozwolone i całkowicie poprawne, nie wymaga także zastosowania żadnej specjalnej składni — wystarczy po prostu podać adres strony jako argument wywołania metody `getRequestDispatcher`. Jednak żądania `POST` nie mogą być przekazywane do statycznych stron HTML, gdyż przekazywane żądanie jest tego samego typu co żądanie oryginalne. Rozwiązanie tego problemu jest bardzo proste — wystarczy zmienić nazwę

statycznego dokumentu HTML i przypisać mu rozszerzenie *.jsp*. Zmiana nazwy, na przykład z *plik.html* na *plik.jsp*, w żaden sposób nie zmieni wyników generowanych przez tę stronę w przypadku otrzymania żądania `GET`; jednak dokument *plik.html* nie jest w stanie obsługiwać żądań `POST`, a strona *plik.jsp* będzie generować te same wyniki zarówno w przypadku otrzymania żądania `GET` jak i `POST`.

Przekazywanie informacji do strony docelowej

Aby przekazać żądanie do strony JSP, serwlet musi jedynie pobrać obiekt `RequestDispatcher` posługując się w tym celu metodą `getRequestDispatcher`, a następnie wywołać jego metodę `forward` podając w jej wywołaniu obiekty `HttpServletRequest` oraz `HttpServletResponse`. To rozwiązanie jest stosunkowo dobre, lecz wymaga, aby strona docelowa samodzielnie pobrała potrzebne informacje z obiektu `HttpServletRequest`. Można podać co najmniej dwa powody, dla których pobieranie i przetwarzanie informacji wejściowych przez stronę docelową nie jest dobrym rozwiązaniem. Po pierwsze, znacznie łatwiej jest wykonywać skomplikowane zadania programistyczne w serwletach a nie w stronach JSP. A po drugie, wiele różnych stron JSP może korzystać z tych samych informacji, a zatem niezależne przetwarzanie ich przez każdą ze stron jest stratą czasu i zasobów komputera. Znacznie lepszym rozwiązaniem jest przetworzenie informacji przez serwlet, który jako pierwszy otrzymał żądanie i zapisanie ich w takim miejscu, z którego strona docelowa będzie w stanie je pobrać.

Istnieją dwa podstawowe miejsca, w których serwlet może przechowywać dane, z których będą potem korzystać strony JSP — w obiekcie `HttpServletRequest` oraz w komponentach JavaBeans umieszczanych w położeniu zależnym od wartości atrybutu `scope` znacznika akcji `jsp:useBean` (patrz podrozdział 13.4. — „Wspólne wykorzystywanie komponentów").

Serwlet, który jako pierwszy otrzyma żądanie, może zapisać dowolną wartość w obiekcie `HttpServletRequest` w następujący sposób:

```
request.setAttribute("klucz", wartość);
```

Strona docelowa może pobrać tę wartość w kodzie JSP, za pomocą wywołania:

```
Typ wartosc = (Typ) request.getAttribute("klucz");
```

W przypadku złożonych wartości jeszcze lepszym rozwiązaniem jest przedstawienie ich w postaci komponentu i zapisanie go w miejscu używanym przez znacznik akcji `jsp:useBean` do przechowywania komponentów wykorzystywanych wspólnie przez wiele serwletów i stron JSP. Na przykład, użycie znacznika akcji `jsp:useBean` z atrybutem `scope` o wartości `application` sprawi, że komponent zostanie umieszczony w obiekcie `ServletContext`, a w obiektach tych do zapisywania wartości używana jest metoda `setAttribute`. A zatem, aby komponent był dostępny dla wszystkich serwletów oraz stron JSP wykonywanych na serwerze lub wchodzących w skład danej aplikacji WWW, serwlet który pierwszy otrzyma żądanie powinien wykonać następujące czynności:

```
Typ wartosc = ObliczWartoscNaPodstawieZadania(request);
getServletContext().setAttribute("klucz", wartosc );
```

Na docelowej stronie JSP typowym sposobem uzyskania dostępu do takiego komponentu będzie użycie znacznika akcji `jsp:useBean` o następującej postaci:

```
<jsp:useBean id="klucz" class="Typ" scope="application" />
```

Alternatywnym rozwiązaniem może być użycie (na stronie docelowej) elementu skryptowego zawierającego jawne wywołanie `application.getAttribute("klucz")` oraz rzutowanie wyniku do typu `Typ`.

Istnieje także możliwość, aby serwlet skojarzył informacje z sesją użytkownik a nie globalnie z całą aplikacją. W takim przypadku serwlet powinien zapisać je w zwyczajny sposób w obiekcie `HttpSession`:

```
Typ wartosc = ObliczWartoscNaPodstawieZadania(request);
HttpSession sesja = request.getSession(true);
sesja.putValue("klucz", wartosc );
```

Typowym sposobem uzyskania dostępu do takiej wartości na docelowej stronie, będzie użycie znacznika akcji `jsp:useBean` o następującej postaci:

```
<jsp:useBean id="klucz" class="Typ" scope="session" />
```

Specyfikacja Java Servlet 2.2 udostępnia trzeci sposób przesyłania informacji do strony docelowej, pod warunkiem, że zostanie wykorzystane żądanie `GET`; polega on na dopisaniu tych informacji do adresu URL. Oto przykład:

```
String adres = "/sciezka/strona.jsp?nowyParametr=wartość";
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher(adres);
dispatcher.forward(request, response);
```

Zastosowanie tej metody powoduje dodanie nowego parametru żądania o nazwie `nowyParametr` (i wartości `wartość`) do pozostałych parametrów. Ten nowy parametr zostanie dodany na samym początku łańcucha zapytania, a zatem zastąpi istniejące wartości jeśli strona docelowa będzie pobierać wartości parametrów przy użyciu metody `getParameter` (która zwraca wyłącznie wartość pierwszego parametru o podanej nazwie) a nie metody `getParameterValues` (która zwraca wartości wszystkich wystąpień parametru o podanej nazwie).

Interpretacja względnych adresów URL przez stronę docelową

Choć serwlet może przekazać żądanie pod dowolny adres na tym samym serwerze, to jednak sposób realizacji całego procesu znacznie się różni od sposobu działania metody `sendRedirect` interfejsu `HttpServletResponse` (patrz podrozdział 6.1). Przede wszystkim metoda `sendRedirect` wymaga, aby klient ponownie nawiązał połączenie z serwerem i zażądał nowego zasobu. W odróżnieniu od niej, metoda `forward` interfejsu `RequestDispatcher` jest w całości realizowana na serwerze. Poza tym metoda `sendRedirect` nie zachowuje automatycznie wszelkich danych przesyłanych w żądaniu, a metoda `forward` to robi. I w końcu, użycie metody `sendRedirect` powoduje zmianę adresu żądanego zasobu, natomiast w przypadku użycia metody `forward` adres żądanego serwletu zostaje zachowany.

Ta ostatnia informacja oznacza, że jeśli na docelowej stronie obrazy oraz arkusze stylów podawane są przy użyciu względnych adresów URL, to muszą one zostać podane względem głównego katalogu serwera, a nie bieżącego położenia strony docelowej. Przeanalizujmy następujący przykład:

```
<LINK REL="STYLESHEET"
      HREF="my-style.css"
      TYPE="text/css">
```

Jeśli strona JSP, w której powyższy znacznik zostanie umieszczony, zostanie wykonana na skutek przekierowanego żądania, to adres *my-style.css* zostanie zinterpretowany jako adres podany względem bieżącego położenia serwletu, który jako pierwszy odebrał żądanie, a nie względem samej strony JSP. Niemal na pewno spowoduje to pojawienie się błędów. Rozwiązaniem jest podanie pełnej ścieżki, określonej względem katalogu głównego serwera, jak to pokazałem na poniższym przykładzie:

```
<LINK REL="STYLESHEET"
      HREF="/ścieżka/my-style.css"
      TYPE="text/css">
```

Tą samą metodę należy zastosować podczas podawania adresów w znacznikach `<IMG SRC=…>` oraz `<A HREF=…>`.

Inne sposoby pobierania obiektu RequestDispatcher

Serwlety tworzone zgodnie ze specyfikacją Java Servlet 2.2 dysponują jeszcze dwiema innymi metodami uzyskiwania obiektu `RequestDispatcher` (oprócz wspominanej wcześniej metody `getRequestDispatcher` interfejsu `ServletContext`).

Po pierwsze, większość serwerów pozwala na nadawanie konkretnych nazw serwletom oraz stronom JSP. Z tego względu, sensownym rozwiązaniem jest odwoływanie się do nich za pośrednictwem nazwy a nie ścieżki dostępu. Do tego celu służy metoda `getNamedDispatcher` interfejsu `ServletContext`.

Po drugie, może się zdarzyć, że będziesz chciał odwoływać się do zasobów przy użyciu adresu URL określanego względem bieżącego położenie serwletu, a nie względem głównego katalogu serwera. Ta metoda nie jest stosowana często jeśli odwołania do serwletów przybierają standardową postać (*http://host/servlet/NazwaSerwletu*). Wynika to z prostego faktu, iż do stron JSP nie można się odwoływać przy użyciu adresu o postaci *http://host/servlet/...*, gdyż jest on zarezerwowany specjalnie dla serwletów. Jednak bardzo często serwlety są rejestrowane pod inną nazwą i w takiej sytuacji można użyć metody `getRequestDispatcher` interfejsu `HttpServletRequest` a nie `ServletContext`. Na przykład, jeśli serwlet, do którego jest kierowane żądanie ma adres *http://host/travel/PoziomGlowny*, to wywołanie

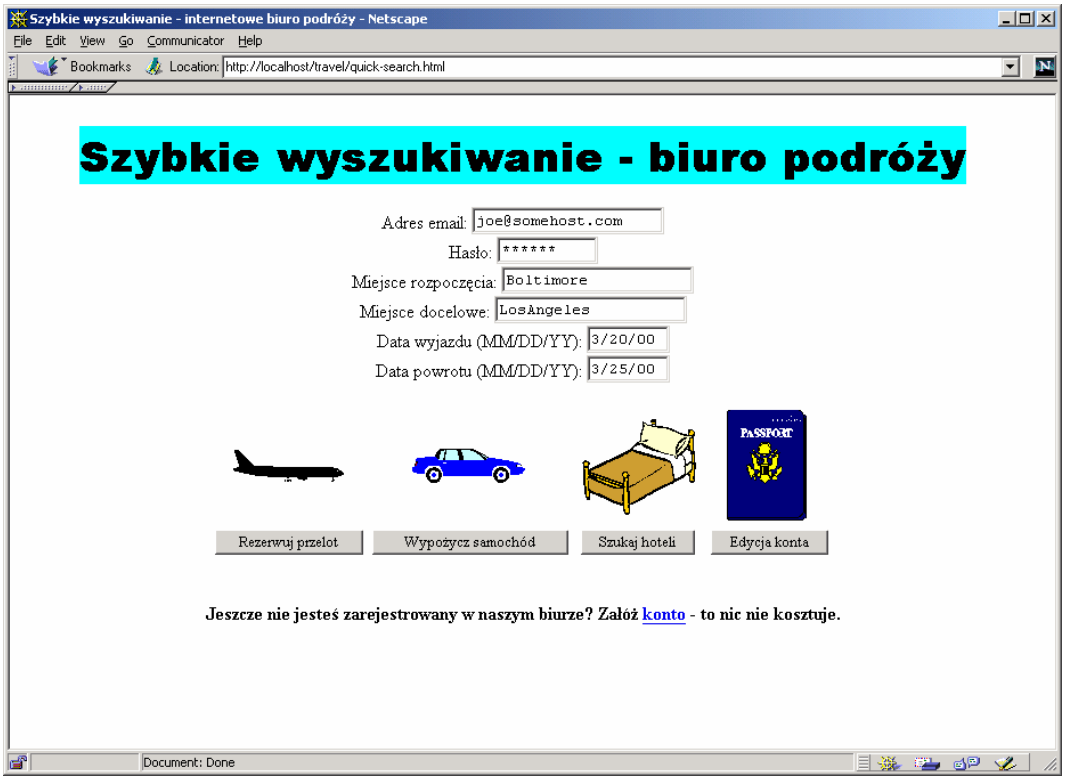
```
getServletContext().getRequestDispatcher("/travel/wycieczki.jsp");
```

można zastąpić wywołaniem

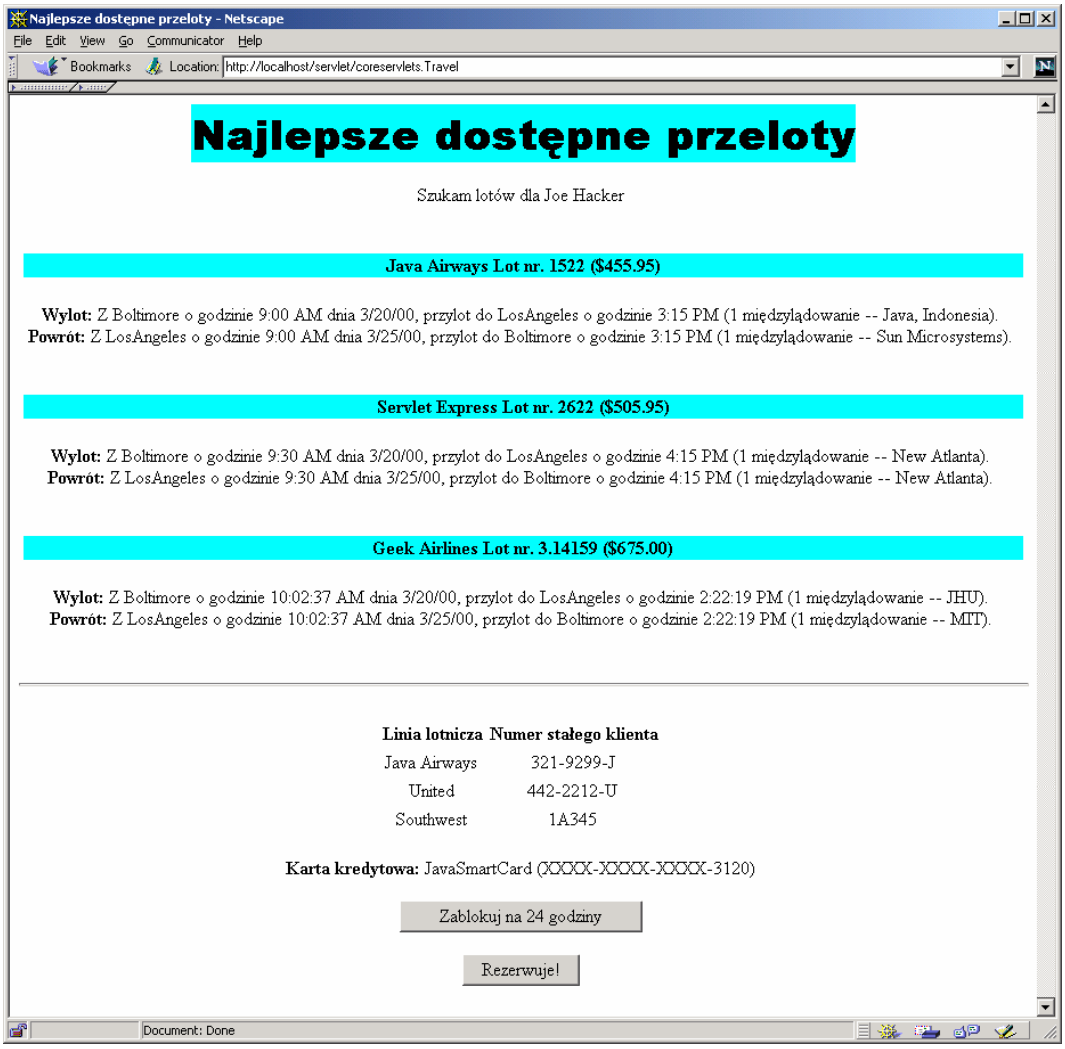
```
request.getRequestDispatcher("wycieczki.jsp");
```

15.2 Przykład: Internetowe biuro podróży

Rozważmy przykład internetowego biura podróży, które na swojej witrynie ma stronę wyszukiwawczą przedstawioną na rysunkach 15.1 oraz 15.2. Użytkownicy muszą podać na niej swój adres poczty elektronicznej oraz hasło, aby można było skojarzyć ich żądania z utworzonymi wcześniej kontami. Każde żądanie zawiera także miejsce rozpoczęcia wycieczki, miejsce jej zakończenia oraz daty początku i końca. Jednak czynności jakie zostaną wykonane po przesłaniu żądania będą całkowicie zależne od informacji poszukiwanych przez użytkownika i określonych w żądaniu. Na przykład, kliknięcie przycisku „Rezerwuj przelot” powinno spowodować wyświetlenie listy połączeń lotniczych dostępnych w określonych dniach i posortowanych według ceny (patrz rysunek 15.1). Do wygenerowania wynikowej strony powinne zostać wykorzystane prawdziwe dane personalne użytkownika, informacje czy jest on stałym klientem linii lotniczych oraz numer karty kredytowej. Z drugiej strony, kliknięcie przycisku „Edycja konta” powinno spowodować wyświetlenie podanych wcześniej informacji o użytkowniku, które będzie można dowolnie zmienić. Podobnie kliknięcie przycisków „Wypożycz samochód” oraz „Szukaj hoteli” spowoduje wykorzystanie tych samych danych personalnych użytkownika, lecz jednocześnie wygenerowane wyniki będą całkowicie różne.



Rysunek 15.1 Interfejs użytkownika serwletu obsługującego internetowe biuro podróży (patrz listing 15.2)



Rysunek 15.2 Wynik wykonania serwletu obsługującego internetowe biuro podróży (patrz listing 15.3), który przekazał żądanie do dokumentu BookFlights.jsp (patrz listing 15.4).

Aby umożliwić taki sposób działania aplikacji, jej interfejs użytkownika (przedstawiony na listingu 15.2) przesyła żądanie do głównego serwletu, którego kod przedstawiłem na listingu 15.3. Serwlet ten pobiera informacje o użytkowniku (patrz listingi od 15.5 do 15.9), zapisuje je w obiekcie klasy `coreservlets.TravelCustomer`, który kojarzy z kluczem `customer` i umieszcza w obiekcie `HttpSession`, a następnie przekazuje żądanie do strony JSP odpowiadającej czynności jakiej zażądał użytkownik. Strona docelowa (patrz listing 15.4 oraz rysunek 15.2) pobiera informacje o użytkowniku posługując znacznikiem akcji `jsp:useBean` o następującej postaci:

```
<jsp:useBean id="customer"
             class="coreservlets.TravelCustomer"
             scope="session" />
```

a następnie wyświetla w różnych miejscach wynikowej strony posługując się znacznikami akcji `jsp:getParameter`. W klasie `TravelCustomer` powinieneś zwrócić uwagę na dwie rzeczy.

Po pierwsze, klasa ta wkłada wiele wysiłku, aby udostępnić informacje o użytkowniku w formie łańcuchów znaków zawierających zwyczajny tekst lub nawet tekst zawierający znaczniki HTML, które można pobrać przy użyciu prostych właściwości. Niemal wszystkie zadania, których realizacja wymaga znaczniejszej pracy programistycznej, zostały zaimplementowane w postaci komponentów JavaBeans — ich realizacja nie została zakodowana w stronach JSP. Takie rozwiązanie jest typowe dla integracji serwletów i dokumentów JSP — zastosowanie JSP nie zapobiega całkowicie konieczności przedstawiania danych w formie tekstu lub kodu HTML i formatowania ich bezpośrednio w kodzie programów. Znaczący wysiłek jaki należy włożyć w

przygotowanie informacji i udostępnienie ich dokumentom JSP zwraca się z nawiązką, jeśli z informacji tego samego typu będzie korzystać większa ilość dokumentów.

Po drugie, należy pamiętać, że wiele serwerów dysponujących możliwością automatycznego przeładowywania serwletów w przypadku modyfikacji ich pliku klasowego, nie pozwala aby pliki klasowe komponentów JavaBeans używanych w dokumentach JSP były umieszczane w katalogach umożliwiających takie automatyczne przeładowywanie. A zatem, w przypadku korzystania z Java Web Servera, pliki klasowe klasy `TravleCustomer` oraz jej klas pomocniczych, muszą być umieszczone w katalogu *katalog_instalacyjny/classes/coreservlets/* a nie w katalogu *katalog_instalacyjny/servlets/coreservlets/*. Serwery Tomcat 3.0 oraz JSWDK 1.0.1 nie udostępniają możliwości automatycznego przeładowywania serwletów, a zatem pliki klasowe komponentów mogą być umieszczane w tych samych katalogach co pliki klasowe serwletów.

Listing 15.2 /travel/quick-search.html[12]

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<!--
Interfejs użytkownika serwletu
obsługującego biuro podróży.
-->
<HTML>
<HEAD>
  <TITLE>Szybkie wyszukiwanie - internetowe biuro podróży</TITLE>
  <LINK REL=STYLESHEET
        HREF="travel-styles.css"
        TYPE="text/css">
</HEAD>

<BODY>
<BR>
<H1>Szybkie wyszukiwanie - biuro podróży</H1>

<FORM ACTION="/servlet/coreservlets.Travel" METHOD="POST">
<CENTER>
Adres email: <INPUT TYPE="TEXT" NAME="emailAddress"><BR>
Hasło: <INPUT TYPE="PASSWORD" NAME="password" SIZE=10><BR>
Miejsce rozpoczęcia: <INPUT TYPE="TEXT" NAME="origin"><BR>
Miejsce docelowe: <INPUT TYPE="TEXT" NAME="destination"><BR>
Data wyjazdu (MM/DD/YY):
  <INPUT TYPE="TEXT" NAME="startDate" SIZE=8><BR>
Data powrotu (MM/DD/YY):
  <INPUT TYPE="TEXT" NAME="endDate" SIZE=8><BR>
<P>
<TABLE CELLSPACING=1>
<TR>
  <TH> <IMG SRC="airplane.gif" WIDTH=100 HEIGHT=29
                 ALIGN="TOP" ALT="Rezerwuj przelot"> 
  <TH> <IMG SRC="car.gif" WIDTH=100 HEIGHT=31
                 ALIGN="MIDDLE" ALT="Wypożycz samochów"> 
  <TH> <IMG SRC="bed.gif" WIDTH=100 HEIGHT=85
                 ALIGN="MIDDLE" ALT="Szukaj hoteli"> 
  <TH> <IMG SRC="passport.gif" WIDTH=71 HEIGHT=100
                 ALIGN="MIDDLE" ALT="Edycja konta"> 
<TR>
  <TH><SMALL>
      <INPUT TYPE="SUBMIT" NAME="flights" VALUE="Rezerwuj przelot">
      </SMALL>
  <TH><SMALL>
      <INPUT TYPE="SUBMIT" NAME="cars" VALUE="Wypożycz samochód">
      </SMALL>
  <TH><SMALL>
      <INPUT TYPE="SUBMIT" NAME="hotels" VALUE="Szukaj hoteli">
      </SMALL>
  <TH><SMALL>
      <INPUT TYPE="SUBMIT" NAME="account" VALUE="Edycja konta">
```

[12] W przykładach dołączonych do niniejszej książki, dokumenty JSP używane w tej aplikacji umieszczone są w katalogu *travel* znajdującym się wewnątrz katalogu *JSP-Code*. Aby aplikacja działała poprawnie należy przenieść katalog *travel*, na główny poziom serwera (w przeciwnym razie odwołania */travel/xxx.jsp* umieszczone w pliku *Travel.java* oraz w poszczególnych dokumentach JSP nie będą poprawne). - przyp. tłum.

```
      </SMALL>
</TABLE>
</CENTER>
</FORM>
<BR>
<P ALIGN="CENTER">
<B>Jeszcze nie jesteś zarejestrowany w naszym biurze? Załóż
<A HREF="accounts.jsp">konto</A> - to nic nie kosztuje.</B></P>
</BODY>
</HTML>
```

Listing 15.3 Travel.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Główny serwlet obsługujący internetowe biuro podróży.
 *  Serwlet zapisuje dane klienta w formie komponentu JavaBean
 *  a następnie przekazuje żądanie do strony rezerwacji biletów
 *  lotniczych, wypożyczania samochodu, poszukiwania hoteli
 *  edycji konta istniejącego użytkownika lub tworzenia nowego
 *  konta.
 */

public class Travel extends HttpServlet {
  private TravelCustomer[] travelData;

  public void init() {
    travelData = TravelData.getTravelData();
  }

  /** Ponieważ przesyłamy hasło można użyć wyłącznie metody POST.
   *  Jednak użycie tej metody oznacza, że żądania nie mogą
   *  być przesyłane do statycznych dokumentów HTML (wynika to
   *  z faktu, że podczas przekazywania żądania używany jest
   *  ten sam typ żądania, a statyczne strony WWW nie są w stanie
   *  obsługiwać żądań POST). Rozwiązanie tego problemu jest
   *  proste: Niech statyczne strony HTML będą stronami JSP
   *  zawierającymi wyłącznie kod HTML. Tak jest w przypadku
   *  dokumentu accounts.jsp. Pozostałe pliki JSP faktycznie
   *  muszą być generowane dynamicznie, gdyż korzystają z
   *  danych o użytkowniku.
   */

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html; encoding=ISO-8859-2");
    String emailAddress = request.getParameter("emailAddress");
    String password = request.getParameter("password");
    TravelCustomer customer =
      TravelCustomer.findCustomer(emailAddress, travelData);
    if ((customer == null) || (password == null) ||
        (!password.equals(customer.getPassword()))) {
      gotoPage("/travel/accounts.jsp", request, response);
    }
    // Metody używające poniższych parametrów będą
    // same sprawdzały czy informacje są poprawne
    // i czy w ogóle zostały podane.
    customer.setStartDate(request.getParameter("startDate"));
    customer.setEndDate(request.getParameter("endDate"));
    customer.setOrigin(request.getParameter("origin"));
    customer.setDestination(request.getParameter
                            ("destination"));
    HttpSession session = request.getSession(true);
    session.putValue("customer", customer);
    if (request.getParameter("flights") != null) {
      gotoPage("/travel/BookFlights.jsp",
               request, response);
    } else if (request.getParameter("cars") != null) {
      gotoPage("/travel/RentCars.jsp",
               request, response);
    } else if (request.getParameter("hotels") != null) {
```

```
      gotoPage("/travel/FindHotels.jsp",
               request, response);
    } else if (request.getParameter("cars") != null) {
      gotoPage("/travel/EditAccounts.jsp",
               request, response);
    } else {
      gotoPage("/travel/IllegalRequest.jsp",
               request, response);
    }
  }

  private void gotoPage(String address,
                        HttpServletRequest request,
                        HttpServletResponse response)
      throws ServletException, IOException {
    RequestDispatcher dispatcher =
      getServletContext().getRequestDispatcher(address);
    dispatcher.forward(request, response);
  }
}
```

Listing 15.4 BookFlights.jsp

```
<%@ page contentType="text/html; encoding=ISO-8859-2" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<!--
Strona służąca do odnajdywania przelotów lotnicznych.
-->
<HTML>
<HEAD>
  <TITLE>Najlepsze dostępne przeloty</TITLE>
  <LINK REL=STYLESHEET
        HREF="/travel/travel-styles.css"
        TYPE="text/css">
</HEAD>

<BODY>
<H1>Najlepsze dostępne przeloty</H1>
<CENTER>
<jsp:useBean id="customer"
             class="coreservlets.TravelCustomer"
             scope="session" />
Szukam lotów dla
<jsp:getProperty name="customer" property="fullName" />
<P>
<jsp:getProperty name="customer" property="flights" />

<P>
<BR>
<HR>
<BR>
<FORM ACTION="/servlet/BookFlight">
<jsp:getProperty name="customer"
                 property="frequentFlyerTable" />
<P>
<B>Karta kredytowa:</B>
<jsp:getProperty name="customer" property="creditCard" />
<P>
<INPUT TYPE="SUBMIT" NAME="holdButton" VALUE="Zablokuj na 24 godziny">
<P>
<INPUT TYPE="SUBMIT" NAME="bookItButton" VALUE="Rezerwuje!">
</FORM>
</CENTER>

</BODY>
</HTML>
```

Listing 15.5 TravelCustomer.java

```
package coreservlets;

import java.util.*;
import java.text.*;

/** Klasa opisuje klienta biura podróży. Została ona
```

```
 *  zaimplementowana jako komponent JavaBean, dysponujący
 *  metodami zwracającymi dane w formacie HTML, dostosowanymi
 *  do wykorzystania w dokumentach JSP.
 */

public class TravelCustomer {
  private String emailAddress, password, firstName, lastName;
  private String creditCardName, creditCardNumber;
  private String phoneNumber, homeAddress;
  private String startDate, endDate;
  private String origin, destination;
  private FrequentFlyerInfo[] frequentFlyerData;
  private RentalCarInfo[] rentalCarData;
  private HotelInfo[] hotelData;

  public TravelCustomer(String emailAddress,
                        String password,
                        String firstName,
                        String lastName,
                        String creditCardName,
                        String creditCardNumber,
                        String phoneNumber,
                        String homeAddress,
                        FrequentFlyerInfo[] frequentFlyerData,
                        RentalCarInfo[] rentalCarData,
                        HotelInfo[] hotelData) {
    setEmailAddress(emailAddress);
    setPassword(password);
    setFirstName(firstName);
    setLastName(lastName);
    setCreditCardName(creditCardName);
    setCreditCardNumber(creditCardNumber);
    setPhoneNumber(phoneNumber);
    setHomeAddress(homeAddress);
    setStartDate(startDate);
    setEndDate(endDate);
    setFrequentFlyerData(frequentFlyerData);
    setRentalCarData(rentalCarData);
    setHotelData(hotelData);
  }

  public String getEmailAddress() {
    return(emailAddress);
  }

  public void setEmailAddress(String emailAddress) {
    this.emailAddress = emailAddress;
  }

  public String getPassword() {
    return(password);
  }

  public void setPassword(String password) {
    this.password = password;
  }

  public String getFirstName() {
    return(firstName);
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return(lastName);
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public String getFullName() {
    return(getFirstName() + " " + getLastName());
  }
```

```
public String getCreditCardName() {
  return(creditCardName);
}

public void setCreditCardName(String creditCardName) {
  this.creditCardName = creditCardName;
}

public String getCreditCardNumber() {
  return(creditCardNumber);
}

public void setCreditCardNumber(String creditCardNumber) {
  this.creditCardNumber = creditCardNumber;
}

public String getCreditCard() {
  String cardName = getCreditCardName();
  String cardNum = getCreditCardNumber();
  cardNum = cardNum.substring(cardNum.length() - 4);
  return(cardName + " (XXXX-XXXX-XXXX-" + cardNum + ")");
}

public String getPhoneNumber() {
  return(phoneNumber);
}

public void setPhoneNumber(String phoneNumber) {
  this.phoneNumber = phoneNumber;
}

public String getHomeAddress() {
  return(homeAddress);
}

public void setHomeAddress(String homeAddress) {
  this.homeAddress = homeAddress;
}

public String getStartDate() {
  return(startDate);
}

public void setStartDate(String startDate) {
  this.startDate = startDate;
}

public String getEndDate() {
  return(endDate);
}

public void setEndDate(String endDate) {
  this.endDate = endDate;
}

public String getOrigin() {
  return(origin);
}

public void setOrigin(String origin) {
  this.origin = origin;
}

public String getDestination() {
  return(destination);
}

public void setDestination(String destination) {
  this.destination = destination;
}

public FrequentFlyerInfo[] getFrequentFlyerData() {
  return(frequentFlyerData);
}

public void setFrequentFlyerData(FrequentFlyerInfo[]
                                 frequentFlyerData) {
```

```
    this.frequentFlyerData = frequentFlyerData;
  }

  public String getFrequentFlyerTable() {
    FrequentFlyerInfo[] frequentFlyerData =
      getFrequentFlyerData();
    if (frequentFlyerData.length == 0) {
      return("<I>Brak danych stałego klienta linii lotniczej.</I>");
    } else {
      String table =
        "<TABLE>\n" +
        "  <TR><TH>Linia lotnicza<TH>Numer stałego klienta\n";
      for(int i=0; i<frequentFlyerData.length; i++) {
        FrequentFlyerInfo info = frequentFlyerData[i];
        table = table +
                "<TR ALIGN=\"CENTER\">" +
                "<TD>" + info.getAirlineName() +
                "<TD>" + info.getFrequentFlyerNumber() + "\n";
      }
      table = table + "</TABLE>\n";
      return(table);
    }
  }

  public RentalCarInfo[] getRentalCarData() {
    return(rentalCarData);
  }

  public void setRentalCarData(RentalCarInfo[] rentalCarData) {
    this.rentalCarData = rentalCarData;
  }

  public HotelInfo[] getHotelData() {
    return(hotelData);
  }

  public void setHotelData(HotelInfo[] hotelData) {
    this.hotelData = hotelData;
  }

  // W realnie wykorzystywanej aplikacji WWW
  // czynności wykonywane w tej metodzie powinne zostać
  // zamienione na pobranie informacji z bazy danych

  public String getFlights() {
    String flightOrigin =
      replaceIfMissing(getOrigin(), "Nigdzie");
    String flightDestination =
      replaceIfMissing(getDestination(), "Nigdzie");
    Date today = new Date();
    DateFormat formatter =
      DateFormat.getDateInstance(DateFormat.MEDIUM);
    String dateString = formatter.format(today);
    String flightStartDate =
      replaceIfMissing(getStartDate(), dateString);
    String flightEndDate =
      replaceIfMissing(getEndDate(), dateString);
    String [][] flights =
      { { "Java Airways", "1522", "455.95", "Java, Indonesia",
          "Sun Microsystems", "9:00", "3:15" },
        { "Servlet Express", "2622", "505.95", "New Atlanta",
          "New Atlanta", "9:30", "4:15" },
        { "Geek Airlines", "3.14159", "675.00", "JHU",
          "MIT", "10:02:37", "2:22:19" } };
    String flightString = "";
    for(int i=0; i<flights.length; i++) {
      String[] flightInfo = flights[i];
      flightString =
        flightString + getFlightDescription(flightInfo[0],
                                            flightInfo[1],
                                            flightInfo[2],
                                            flightInfo[3],
                                            flightInfo[4],
                                            flightInfo[5],
                                            flightInfo[6],
                                            flightOrigin,
                                            flightDestination,
```

```
                                              flightStartDate,
                                              flightEndDate);
    }
    return(flightString);
  }

  private String getFlightDescription(String airline,
                                      String flightNum,
                                      String price,
                                      String stop1,
                                      String stop2,
                                      String time1,
                                      String time2,
                                      String flightOrigin,
                                      String flightDestination,
                                      String flightStartDate,
                                      String flightEndDate) {
    String flight =
      "<P><BR>\n" +
      "<TABLE WIDTH=\"100%\"><TR><TH CLASS=\"COLORED\">\n" +
      "<B>" + airline + " Lot nr. " + flightNum +
      " ($" + price + ")</B></TABLE><BR>\n" +
      "<B>Wylot:</B> Z " + flightOrigin +
      " o godzinie " + time1 + " AM dnia " + flightStartDate +
      ", przylot do " + flightDestination +
      " o godzinie " + time2 + " PM (1 międzylądowanie -- " + stop1 + ").\n" +
      "<BR>\n" +
      "<B>Powrót:</B> Z " + flightDestination +
      " o godzinie " + time1 + " AM dnia " + flightEndDate +
      ", przylot do " + flightOrigin +
      " o godzinie " + time2 + " PM (1 międzylądowanie -- " + stop2 + ").\n";
    return(flight);
  }

  private String replaceIfMissing(String value,
                                  String defaultValue) {
    if ((value != null) && (value.length() > 0)) {
      return(value);
    } else {
      return(defaultValue);
    }
  }

  public static TravelCustomer findCustomer
                                 (String emailAddress,
                                  TravelCustomer[] customers) {
    if (emailAddress == null) {
      return(null);
    }
    for(int i=0; i<customers.length; i++) {
      String custEmail = customers[i].getEmailAddress();
      if (emailAddress.equalsIgnoreCase(custEmail)) {
        return(customers[i]);
      }
    }
    return(null);
  }
}
```

Listing 15.6 TravelData.java

```
package coreservlets;

/** Ta klasa tworzy pewne statyczne dane, opisujące
 *  przykładowych klientów.
 *  W prawdziwej aplikacji należy wykorzystać bazę
 *  danych. Przykłady wykorzystania JDBC z poziomu
 *  serwletów znajdziesz w rozdziale 18 książki
 *  Java Servlet i Java Server Pages
 */

public class TravelData {
  private static FrequentFlyerInfo[] janeFrequentFlyerData =
    { new FrequentFlyerInfo("Java Airways", "123-4567-J"),
      new FrequentFlyerInfo("Delta", "234-6578-D") };
  private static RentalCarInfo[] janeRentalCarData =
```

```
    { new RentalCarInfo("Alamo", "345-AA"),
      new RentalCarInfo("Hertz", "456-QQ-H"),
      new RentalCarInfo("Avis", "V84-N8699") };
  private static HotelInfo[] janeHotelData =
   { new HotelInfo("Marriot", "MAR-666B"),
     new HotelInfo("Holiday Inn", "HI-228-555") };
  private static FrequentFlyerInfo[] joeFrequentFlyerData =
    { new FrequentFlyerInfo("Java Airways", "321-9299-J"),
      new FrequentFlyerInfo("United", "442-2212-U"),
      new FrequentFlyerInfo("Southwest", "1A345") };
  private static RentalCarInfo[] joeRentalCarData =
    { new RentalCarInfo("National", "NAT00067822") };
  private static HotelInfo[] joeHotelData =
   { new HotelInfo("Red Roof Inn", "RRI-PREF-236B"),
     new HotelInfo("Ritz Carlton", "AA0012") };
  private static TravelCustomer[] travelData =
    { new TravelCustomer("jane@somehost.com",
                         "tarzan52",
                         "Jane",
                         "Programmer",
                         "Visa",
                         "1111-2222-3333-6755",
                         "(123) 555-1212",
                         "6 Cherry Tree Lane\n" +
                           "Sometown, CA 22118",
                         janeFrequentFlyerData,
                         janeRentalCarData,
                         janeHotelData),
      new TravelCustomer("joe@somehost.com",
                         "qWeRtY",
                         "Joe",
                         "Hacker",
                         "JavaSmartCard",
                         "000-1111-2222-3120",
                         "(999) 555-1212",
                         "55 25th St., Apt 2J\n" +
                           "New York, NY 12345",
                         joeFrequentFlyerData,
                         joeRentalCarData,
                         joeHotelData)
    };

  public static TravelCustomer[] getTravelData() {
    return(travelData);
  }
}
```

Listing 15.7 FrequentFlayerInfo.java

```
package coreservlets;

/** Prosta klasa opisująca linię lotniczą i numer
 *  jej stałego klienta; używana w klasie TravelData
 *  (gdzie została zdefiniowana tablic obiektów klasy
 *  FrequentFlayerInfo, skojarzona z każdym klientem).
 */

public class FrequentFlyerInfo {
  private String airlineName, frequentFlyerNumber;

  public FrequentFlyerInfo(String airlineName,
                           String frequentFlyerNumber) {
    this.airlineName = airlineName;
    this.frequentFlyerNumber = frequentFlyerNumber;
  }

  public String getAirlineName() {
    return(airlineName);
  }

  public String getFrequentFlyerNumber() {
    return(frequentFlyerNumber);
  }
}
```

Listing 15.8 RentalCarInfo.java

```
package coreservlets;

/** Prosta klasa opisująca firmę wynajmującą samochody
 *  i kojarzącą numer stałego klienta. Jest ona stosowana
 *  w klasie TravelData (gdzie tablica obiektów klasy
 *  RentalCarInfo jest skojarzona z każdym klientem).
 */

public class RentalCarInfo {
  private String rentalCarCompany, rentalCarNumber;

  public RentalCarInfo(String rentalCarCompany,
                       String rentalCarNumber) {
    this.rentalCarCompany = rentalCarCompany;
    this.rentalCarNumber = rentalCarNumber;
  }

  public String getRentalCarCompany() {
    return(rentalCarCompany);
  }

  public String getRentalCarNumber() {
    return(rentalCarNumber);
  }
}
```

Listing 15.9 HotelInfo.java

```
package coreservlets;

/** Prosta klasa zawierająca nazwę hotelu i numer
 *  stałego gościa, wykorzystywana w klasie TravelData
 *  (gdzie z każdym klientem kojarzona jest tablica obiektów
 *  klasy HotelInfo)
 */

public class HotelInfo {
  private String hotelName, frequentGuestNumber;

  public HotelInfo(String hotelName,
                   String frequentGuestNumber) {
    this.hotelName = hotelName;
    this.frequentGuestNumber = frequentGuestNumber;
  }

  public String getHotelName() {
    return(hotelName);
  }

  public String getfrequentGuestNumber() {
    return(frequentGuestNumber);
  }
}
```

15.3 Dołączanie danych statycznych bądź dynamicznych

Jeśli serwlet używa metody `forward` interfejsu `RequestDispatcher`, to w rzeczywistości nie może przesłać do klienta jakichkolwiek danych wyjściowych — całość wyników musi zostać wygenerowana przez stronę docelową. Jeśli serwlet chce samodzielnie wygenerować część wyników, a jako pozostałej części użyć statycznego dokumentu HTML lub wyników zwróconych przez stronę JSP, to powinien użyć metody `include` interfejsu `RequestDispatcher`. Sposób wykorzystania tej metody przypomina przekazywanie żądań do innych stron JSP lub serwletów — należy wywołać metodę `getRequestDispatcher` obiektu `ServletContext` podając w jej wywołaniu

adres URL określony względem głównego katalogu serwera, a następnie wywołać metodę `include` przekazując do niej obiekty `HttpServletRequest` oraz `HttpServletResponse`. Dwie podstawowe różnice pomiędzy stosowaniem metody `include` i `forward` polegają na tym, iż przed wywołaniem metody `include` można przesyłać zawartość strony wynikowej do klienta, a po jej wykonaniu sterowanie jest z powrotem przekazywane do serwletu. Choć dołączane strony (serwlety, strony JSP, a nawet statyczne dokumenty HTML) mogą przesyłać wyniki do klienta, to jednak nie powinne generować nagłówków odpowiedzi HTTP. Oto przykład:

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("...");
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher("/sciezka/zasob");
dispatcher.include(request, response);
out.println("...");
```

Metoda `include` ma wiele cech wspólnych z metodą `forward`. Jeśli oryginalne żądanie wykorzystywało metodę `POST`, ta sama metoda zostanie użyta do dalszego przekazania żądania. Jakiekolwiek dane były skojarzone z oryginalnym żądaniem, będą także dostępne w żądaniu pomocniczym; co więcej, w serwletach tworzonych zgodnie ze specyfikacją Java Servlet 2.2 można dodawać nowe parametry, dopisując je do adresu URL przekazanego w wywołaniu metody `getRequestDispatcher`. Specyfikacja Java Servlet 2.2 udostępnia także możliwość pobrania obiektu `RequestDispatcher` na podstawie nazwy (służy do tego metod `getNamedDispatcher`) lub użycia względnego adresu URL (w tym celu należy użyć metody `getRequestDispatcher` obiektu `HttpServletRequest`); więcej informacji na ten temat znajdziesz w podrozdziale 15.1. — „Przekazywanie żądań". Jednak metoda `include` robi jedną rzecz, której nie robi metoda forward — automatycznie określa w obiekcie `HttpServletRequest` atrybuty opisujące oryginalną ścieżkę żądania; oczywiście jeśli dołączany serwlet lub strona JSP potrzebuje tych informacji. Atrybuty te można pobrać w dołączanej stronie przy użyciu metody `getAttribute` interfejsu `HttpServletRequest`; poniżej podałem listę tych atrybutów:

- `javax.servlet.include.request_uri`,
- `javax.servlet.include.context_path`,
- `javax.servlet.include.servlet_path`,
- `javax.servlet.include.path_info`,
- `javax.servlet.include.query_string`.

Zwróć uwagę, że takie dołączanie plików nie jest tym samym co niestandardowa metoda *łączenia serwletów w łańcuch* udostępniana jako rozszerzenie przez kilka mechanizmów obsługi serwletów. Metoda ta pozwala, aby każdy z grupy serwletów obsługujących żądania mógł „zobaczyć" (i zmodyfikować) wyniki wygenerowane przez poprzedni serwlet. Podczas stosowania metody `include` interfejsu `RequestDispatcher`, dołączany zasób nie ma dostępu do wyników wygenerowanych przez serwlet, do którego było skierowane żądanie. W rzeczywistości, w specyfikacji serwletów nie ma żadnego standardowego mechanizmu przypominającego łączenie serwletów w łańcuch.

Zwróć także uwagę, że ten typ dołączania plików różni się od możliwości funkcjonalnych jakie udostępnia dyrektywa `include` JSP przedstawiona w podrozdziale 12.1. — „Dołączanie plików w czasie przekształcania strony". Dyrektywa ta powodowała bowiem umieszczenie w stronie kodu źródłowego dołączanego pliku, natomiast metoda `include` interfejsu `RequestDispatcher` powoduje dołączenie *wyników* wykonania wskazanego zasobu. Z drugiej strony, znacznik akcji `jsp:include` omawiany w podrozdziale 12.2. (pt.: „Dołączanie plików podczas obsługi żądań") działa podobnie do omawianej tu metody `include`, z tą różnicą, iż można go stosować wyłącznie w stronach JSP (w serwletach jest niedostępny).

15.4 Przykład: Prezentacja nieprzetworzonych wyników zwracanych przez serwlety lub strony JSP

Podczas testowania serwletów oraz dokumentów JSP warto czasami mieć możliwość wyświetlenia wygenerowanych przez nie wyników w postaci nieprzetworzonej. Oczywiście, można to zrobić wybierając w przeglądarce opcje *Źródło* (lub *View Source*). Ewentualnie, aby móc określać nagłówki żądania oraz przeanalizować zwrócone nagłówki odpowiedzi oraz wygenerowany kod HTML, można posłużyć się programem *WebClient* przedstawionym w podrozdziale 2.10. — „WebClient: Interaktywna wymiana informacji z serwerem WWW". Jednak dostępna jest jeszcze inna możliwość przydatna przy szybkim testowaniu serwletów i stron JSP. Polega ona na stworzeniu serwletu, do którego będzie przekazywany adres URL i który będzie wyświetlał stronę zawierającą wygenerowany kod HTML. Wykonanie tego zadania jest możliwe dzięki temu, iż element `TEXTAREA` ignoruje wszelkie znaczniki HTML oprócz znacznika `</TEXTAREA>`. A zatem „testowy" serwlet będzie generował początek wynikowej strony WWW, włącznie ze znacznikiem `<TEXTAREA>`. Następnie, serwlet dołączy dowolny zasób określony przy użyciu adresu URL przekazanego w żądaniu, po czym wygeneruje dalszą część strony rozpoczynając od zamykającego znacznika `</TEXTAREA>`. Oczywiście, serwlet nie będzie działał poprawnie jeśli dołączany zasób będzie zawierał znacznik `</TEXTAREA>`, jednak najważniejsze w tym przypadku jest proces dołączania plików.

Listing 15.10 przedstawia serwlet wykonujący przedstawione wcześniej zadanie. Na listingu 15.11 przedstawiłem formularz służący do pobierania informacji i przesyłania ich do serwletu. Wygląd tego formularza pokazałem na rysunku 15.3, natomiast rysunek 15.4 przedstawia wyniki wykonania serwletu.

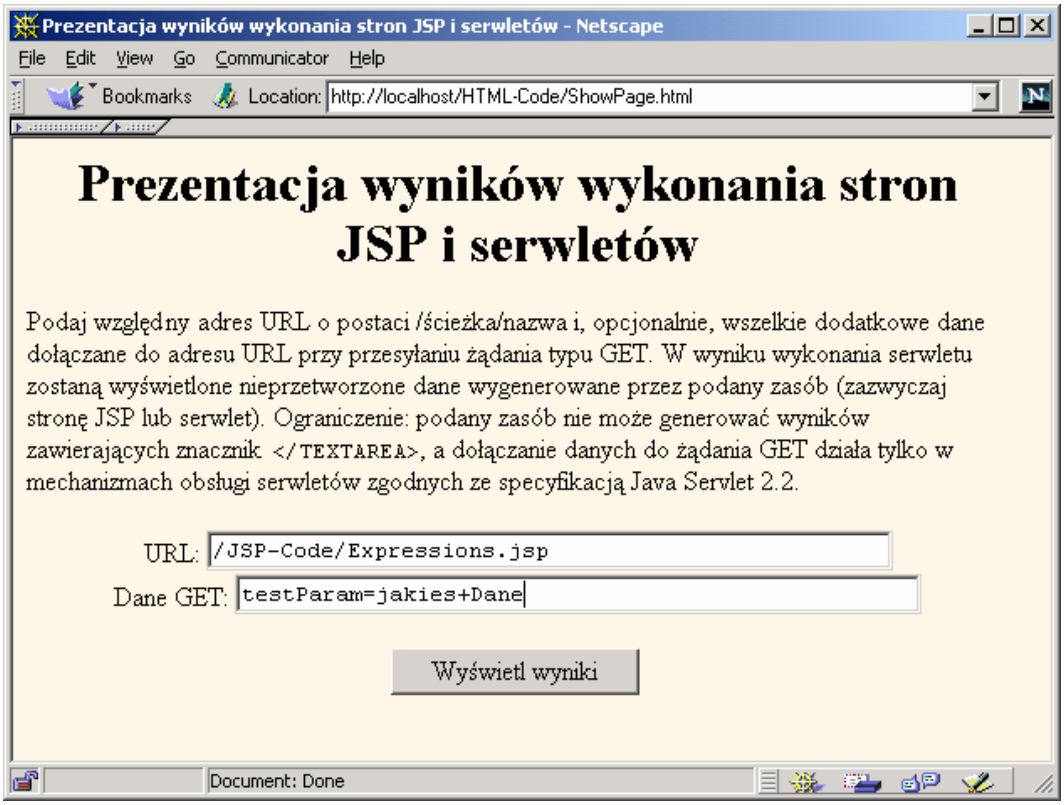
Listing 15.10 ShowPage.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Przykład zastosowania metody include interfejsu
 *  RequestDispatcher. Na podstawie podanego URI
 *  odnoszącego się do zasobu na tym samym serwerze co
 *  serwlet, serwlet wyświetla nieprzetworzone dane
 *  zwrócone przez wskazany zasób.
 */

public class ShowPage extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String url = request.getParameter("url");
    out.println(ServletUtilities.headWithTitle(url) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + url + "</H1>\n" +
                "<FORM><CENTER>\n" +
                "<TEXTAREA ROWS=30 COLS=70>");
    if ((url == null) || (url.length() == 0)) {
      out.println("Nie podano adresu.");
    } else {
      // Dołączanie działa tylko w specyfikacji Java Servlet 2.2
      String data = request.getParameter("data");
      if ((data != null) && (data.length() > 0)) {
        url = url + "?" + data;
      }
      RequestDispatcher dispatcher =
        getServletContext().getRequestDispatcher(url);
      dispatcher.include(request, response);
    }
    out.println("</TEXTAREA>\n" +
```

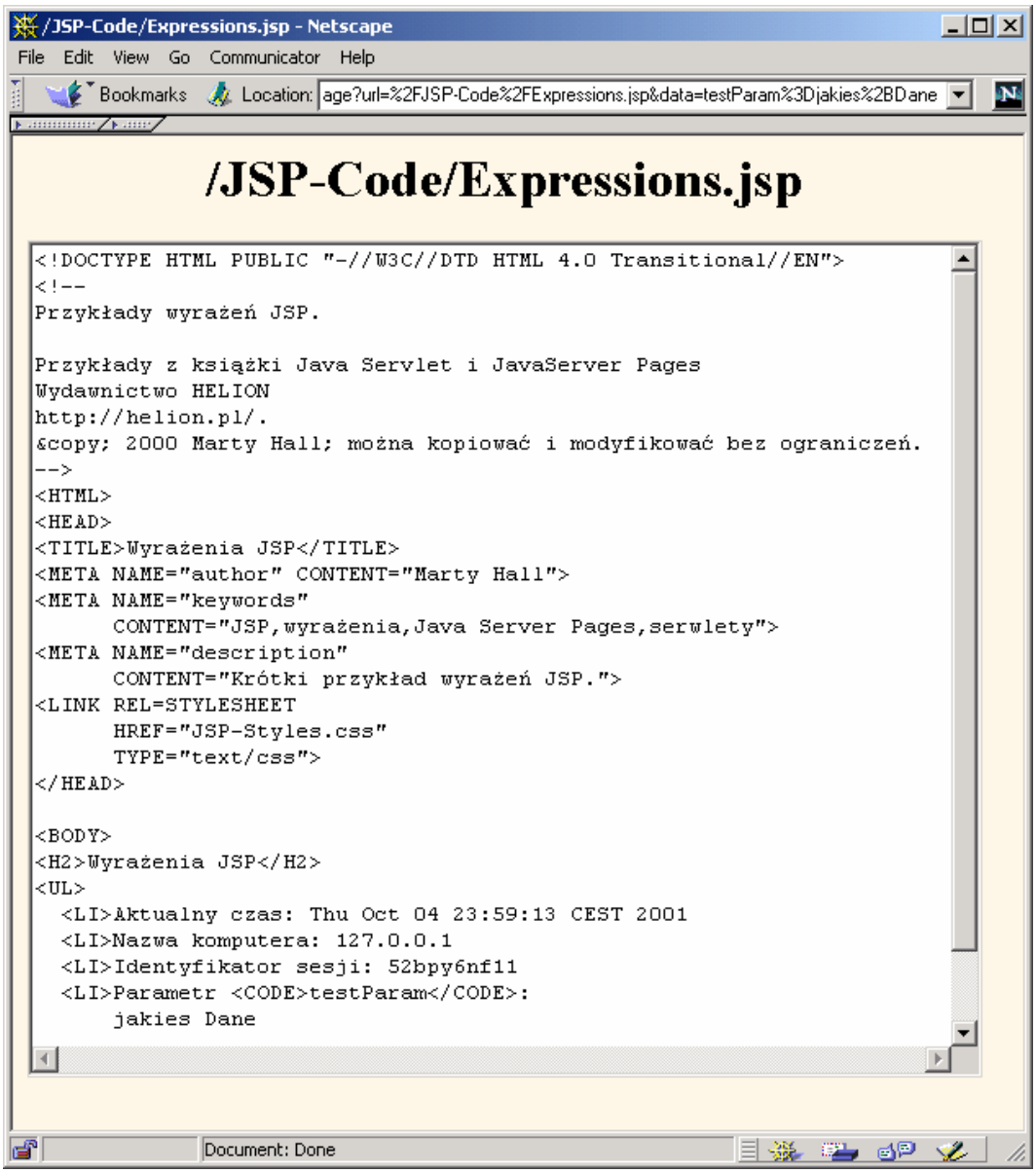
```
                "</CENTER></FORM>\n" +
                "</BODY></HTML>");
  }

  /** Żądania GET and POST obsługiwane tak samo. */

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
}
```

Listing 15.11 ShowPage.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<!--
Interfejs użytkownika do obsługi serwletu wyświetlającego
nieprzetworzone dane zwrócone przez inny serwlet lub
stronę JSP.
-->
<HTML>
<HEAD>
  <TITLE>Prezentacja wyników wykonania stron JSP i serwletów</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">Prezentacja wyników wykonania stron JSP i serwletów</H1>
Podaj względny adres URL o postaci /ścieżka/nazwa i, opcjonalnie,
wszelkie dodatkowe dane dołączane do adresu URL przy przesyłaniu
żądania typu GET. W wyniku wykonania serwletu zostaną wyświetlone
nieprzetworzone dane wygenerowane przez podany zasób (zazwyczaj
stronę JSP lub serwlet). Ograniczenie: podany zasób nie może generować
wyników zawierających znacznik <CODE>&lt;/TEXTAREA&gt;</CODE>,
a dołączanie danych do żądania GET działa tylko w mechanizmach
obsługi serwletów zgodnych ze specyfikacją Java Servlet 2.2.

<FORM ACTION="/servlet/coreservlets.ShowPage">
  <CENTER>
    URL:
    <INPUT TYPE="TEXT" NAME="url" SIZE=50 VALUE="/"><BR>
    Dane GET:
    <INPUT TYPE="TEXT" NAME="data" SIZE=50><BR><BR>
    <Input TYPE="SUBMIT" VALUE="Wyświetl wyniki">
  </CENTER>
</FORM>

</BODY>
</HTML>
```



Rysunek 15.3 Interfejs użytkownika służący do obsługi serwletu ShowPage.java. Kod źródłowy tego formularza został przedstawiony na listingu 15.11



Rysunek 15.4 Wyniki wykonania serwletu ShowPage.java, po przekazaniu do niego adresu URL odwołującego się do strony Expressions.jsp (patrz listing 10.1 w podrozdziale 10.2).

15.5 Przekazywanie żądań ze stron JSP

Najczęściej spotykany scenariusz przekazywania żądań wygląda w następujący sposób: na serwer dociera żądanie skierowane do serwletu, które następnie jest przekazywane do dokumentu JSP. To właśnie serwlety zazwyczaj obsługują żądania w pierwszej kolejności, gdyż sprawdzenie parametrów żądania, stworzenie i zapisanie komponentów wymaga zazwyczaj poważnej pracy programistycznej, którą jest znacznie łatwiej wykonać w serwlecie niż w dokumencie JSP. Natomiast strona docelowa jest zazwyczaj dokumentem JSP, gdyż technologia ta znacząco upraszcza proces generacji kodu HTML.

Oczywiście fakt, że tak wygląda *typowy* scenariusz wcale nie oznacza, że jest to jedyna metoda, którą możemy wykorzystać. Nie ma żadnych powodów, aby strona docelowa nie mogła być serwletem. Możliwe jest także przekazywanie żądań ze stron JSP do innych zasobów. Na przykład, żądanie może być skierowane do strony JSP, która normalnie je przetwarza i zwraca

wyniki pewnego typu, a przekazuje je dalej wyłącznie w sytuacji gdy otrzyma nieprawidłowe lub nieoczekiwane dane.

Przesłanie żądania do serwletu a nie do strony JSP nie wymaga jakichkolwiek modyfikacji w sposobie wykorzystania metod interfejsu `RequestDispatcher`. Jednak istnieje specjalne narzędzie ułatwiające przekazywanie żądań ze stron JSP. Podczas przekazywania żądań z jednej strony JSP na drugą, znacznie łatwiej jest posłużyć się znacznikiem akcji `jsp:forward` niż tworzyć skryptlet wykorzystujący metodę `forward` interfejsu `RequestDispatcher`. Znacznik `jsp:forward` ma następującą postać:

```
<jsp:forward page="względnyAdresURL" />
```

Atrybut `page` może zawierać wyrażenie JSP, dzięki czemu adres strony docelowej może być określany w momencie obsługi żądania. Przedstawiony poniżej, przykładowy fragment kodu kieruje około połowy użytkowników na stronę *http://host/przyklad/strona1.jsp*, a pozostałą część na stronę *http://host/przyklad/strona2.jsp*.

```
<% String adresDocelowy;
   if (Math.random() > 0.5) {
     adresDocelowy = "/przykład/strona1.jsp";
   } else {
     adresDocelowy = "/przykład/strona2.jsp";
   }
%>
<jsp:forward page="<%= adresDocelowy %>" />
```

Rozdział 16.
Formularze HTML

W tym rozdziale omówię zastosowanie formularzy HTML jako interfejsu użytkownika służącego do obsługi serwletów lub innych programów działających po stronie serwera. Formularze te udostępniają proste i niezawodne elementy sterujące przeznaczone do pobierania danych od użytkowników i przesyłanie ich na serwer. W rozdziale przedstawię także zagadnienia związane z wykorzystanie apletów jako interfejsu użytkownika służącego do posługiwania się serwletami. Wykorzystanie apletów w takim celu wymaga znacznie większego nakładu pracy, a co więcej, możliwości apletów są ograniczane zasadami bezpieczeństwa. Niemniej jednak, aplety pozwalają na tworzenie znacznie bogatszych interfejsów użytkownika i oferują efektywne i elastyczne możliwości komunikacji sieciowej.

Jeśli chcesz korzystać z formularzy, będziesz musiał wiedzieć gdzie należy umieścić dokumenty HTML, aby serwer WWW miał do nich dostęp. Konkretne nazwy katalogów zależą od używanego serwera; i tak, w przypadku serwerów Tomcat 3.0 oraz JSWDK dokumenty HTML umieszczane są w katalogu *katalog_instalacyjny/webpages/sciezka/plik.html*, a z poziomu WWW należy się do nich odwoływać przy użyciu adresu *http://localhost/sciezka/plik.html* (jeśli korzystasz z serwera, który nie działa na lokalnym komputerze, to *localhost* w powyższym adresie należ zastąpić poprawną nazwą komputera).

16.1 Jak przesyłane są dane z formularzy HTML

Formularze HTML pozwalają na umieszczanie na stronach WWW wielu różnych elementów kontrolnych służących do pobierania informacji. Każdy z tych elementów ma zazwyczaj nazwę oraz wartość. Nazwy elementów kontrolnych formularzy są zawsze określane w dokumentach HTML, a wartości mogą być podane bądź to w dokumencie, bądź bezpośrednio przez użytkownika. Z całym formularzem jest skojarzony adresem URL programu, który ma zostać użyty do przetworzenia informacji podanych w formularzu. Gdy użytkownik wysyła formularz (zazwyczaj naciskając odpowiedni przycisk), to nazwy i wartości wszystkich pól są przesyłane pod wskazany adres, przy czym, są one zapisywane w następującej postaci:

```
Nazwa1=Wartosc1&Nazwa2=Wartosc2&...&NazwaN=WartoscN
```

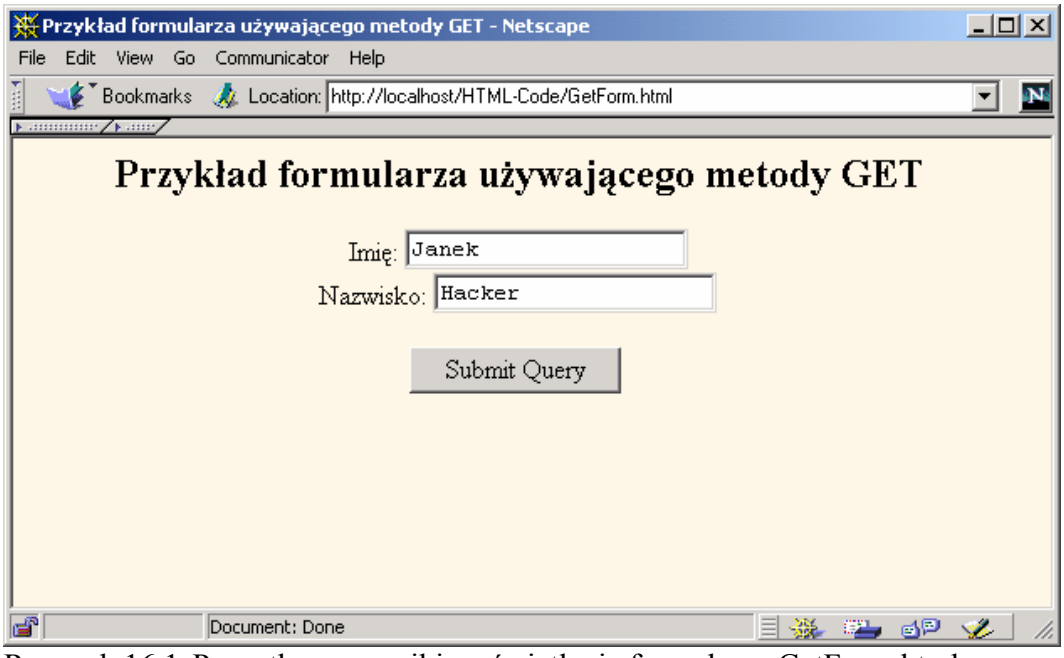
Łańcuch ten może zostać przesłany na serwer na dwa sposoby. Pierwszy z nich polega na wykorzystaniu metody `GET` protokołu HTTP. W tym przypadku informacje podane w formularzu są poprzedzane znakiem pytajnika i dopisywane na końcu podanego adresu URL. Drugim sposobem jest wykorzystanie metody `POST`. W tym przypadku, na serwer są kolejno przesyłane: wiersz żądania HTTP (`POST`), nagłówki żądania HTTP, pusty wiersz, a następnie łańcuch znaków zawierający informacje wpisane w formularzu.

Na listingu 16.1 przedstawiłem prosty, przykładowy formularz zawierający dwa pola tekstowe (patrz rysunek 16.1). Elementy HTML tworzące ten formularz zostaną szczegółowo omówione w dalszej części rozdziału. Jak na razie powinieneś jednak zwrócić uwagę na kilka spraw. Po pierwsze, jedno z pól ma nazwę `firstName`, a drugie `lastName`. Po drugie, elementy

sterujące formularzy są uważane za elementy wpisane (tekstowe) języka HTML, a zatem konieczne będzie zastosowanie specjalnych sposobów formatowania, aby zapewnić ich odpowiednie położenie względem opisującego je tekstu. I ostatnia sprawa — zwróć uwagę, iż nasz przykładowy formularza określa, iż program przeznaczony do obsługi danych ma adres *http://localhost:8088/Program*.

Listing 16.1 GetForm.html

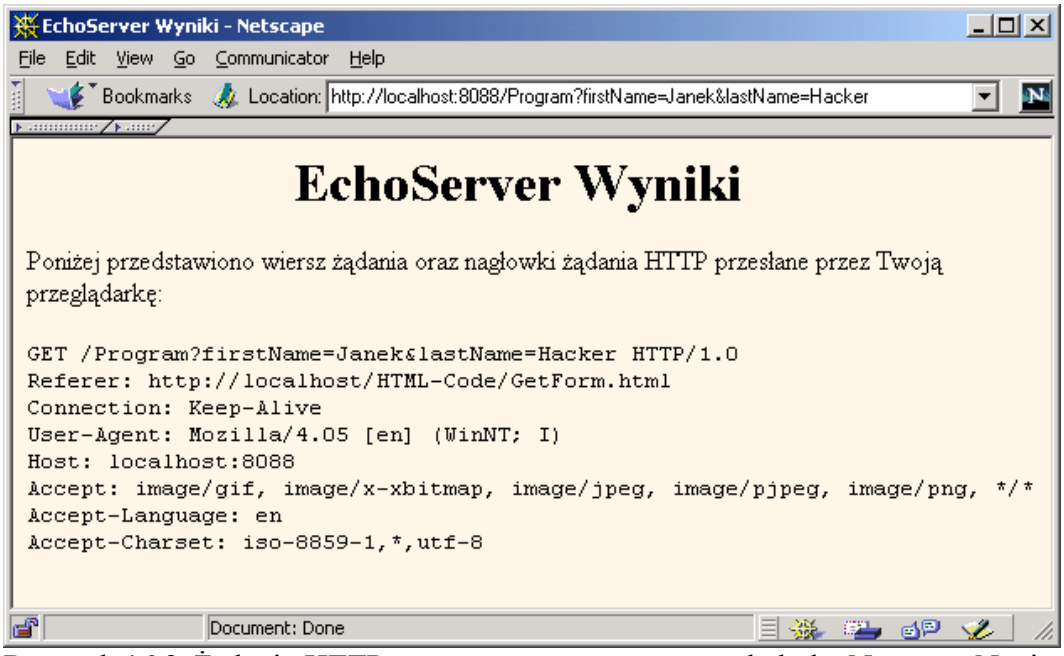
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Przykład formularza używającego metody GET</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H2 ALIGN="CENTER">Przykład formularza używającego metody GET</H2>

<FORM ACTION="http://localhost:8088/Program">
  <CENTER>
  Imię:
  <INPUT TYPE="TEXT" NAME="firstName" VALUE="Janek"><BR>
  Nazwisko:
  <INPUT TYPE="TEXT" NAME="lastName" VALUE="Hacker"><P>
  <INPUT TYPE="SUBMIT" VALUE="Prześlij formularz">
  <!-- Kliknij ten przycisk aby wysłać formularz -->
  </CENTER>
</FORM>

</BODY>
</HTML>
```



Rysunek 16.1 Początkowe wyniki wyświetlenia formularza GetForm.html

Przed przesłaniem informacji podanych w tym formularzu uruchomiłem na serwerze program o nazwie *EchoServer*; program ten działał na porcie 8088 mojego lokalnego komputera. Program *EchoServer* jest miniaturowym serwerem WWW używanym do celów testowych, przedstawiłem go dokładniej w podrozdziale 16.12. Niezależnie od podanego adresu URL oraz wpisanych informacji, wyświetla on stronę WWW prezentującą nadesłane dane. Jak pokazałem na rysunku 16.2, gdy w pierwszym polu formularza zostanie wpisane słowo `Janek`, a w drugim — `Hacker`, to przeglądarka prześle żądanie dotyczące adresu URL o postaci `http://localhost:8088/Program?firstName=Janek&lastName=Hacker`. Listing 16.2 (kod HTML)

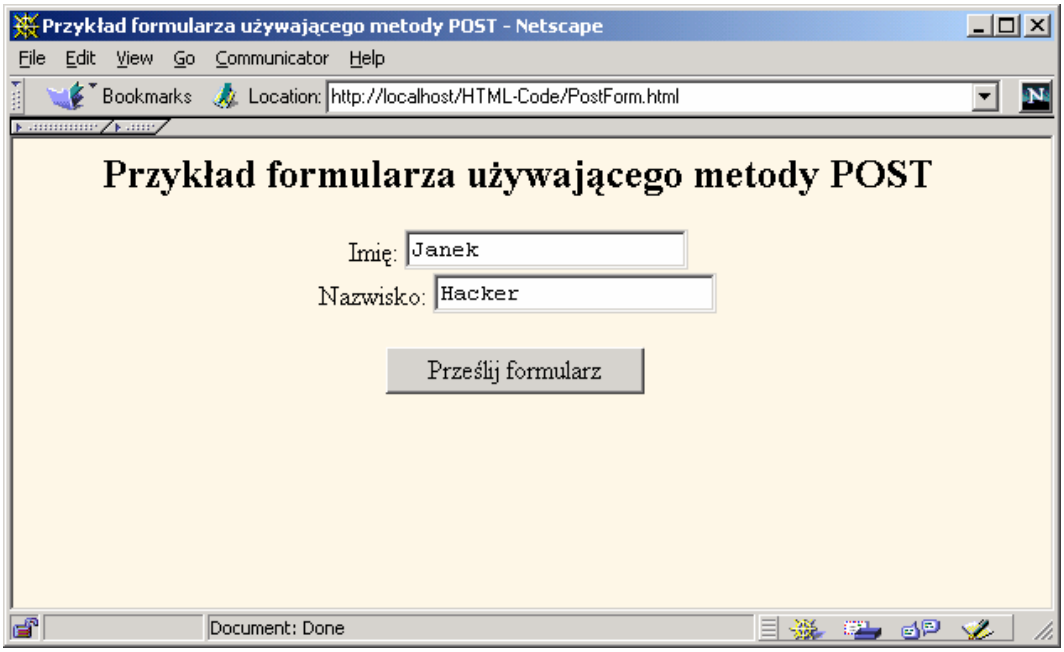
oraz 16.3 (przykładowe wyniki) prezentują inną wersję powyższego przykładu, wykorzystującą metodę `POST` zamiast `GET`. Jak widać, wpisanie w pierwszym polu tekstowym wartości `Janek`, a w drugim — `Hacker`, spowoduje, że wśród nagłówków żądania przesyłanych na serwer pojawi się dodatkowy wiersz zawierający łańcuch znaków `firstName=Janek&lastName=Hacker`.



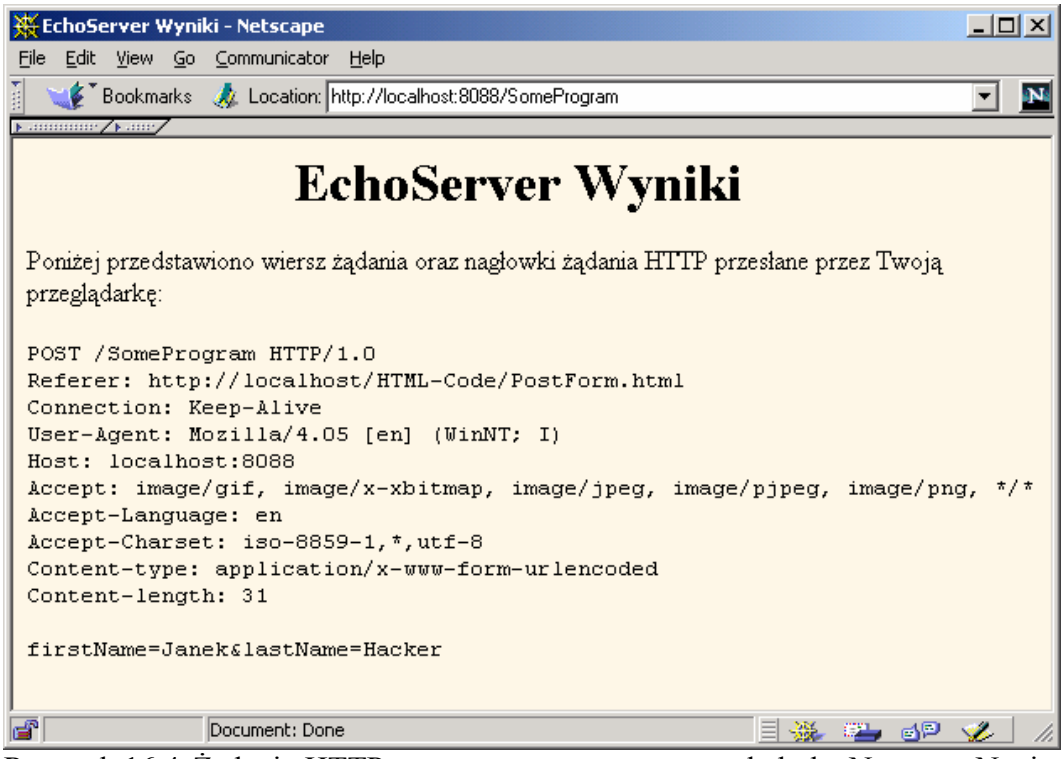
Rysunek 16.2 Żądanie HTTP wygenerowane przez przeglądarkę Netscape Navigator 4.7 po przesłaniu formularza zdefiniowanego na stronie GetForm.html

Listing 16.2 PostForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Przykład formularza używającego metody POST</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H2 ALIGN="CENTER">Przykład formularza używającego metody POST</H2>

<FORM ACTION="http://localhost:8088/SomeProgram"
      METHOD="POST">
  <CENTER>
  Imię:
  <INPUT TYPE="TEXT" NAME="firstName" VALUE="Janek"><BR>
  Nazwisko:
  <INPUT TYPE="TEXT" NAME="lastName" VALUE="Hacker"><P>
  <INPUT TYPE="SUBMIT" VALUE="Prześlij formularz">
  </CENTER>
</FORM>

</BODY>
</HTML>
```



Rysunek 16.3 Początkowy wygląd strony PostForm.html



Rysunek 16.4 Żądanie HTTP wygenerowane przez przeglądarkę Netscape Navigator 4.7 po przesłaniu formularza zdefiniowanego na stronie PostForm.html

Oto ogólne wyjaśnienie sposobu działania formularzy HTML — graficzne elementy kontrolne umożliwiają pobieranie informacji od użytkowników, każdy z tych elementów posiada nazwę oraz wartość, a po przesłaniu formularza, na serwer przekazywany jest łańcuch znaków zawierający pary nazwa-wartość. Jeśli do obsługi formularzy na serwerze wykorzystamy serwlety, to pobranie nazw elementów kontrolnych oraz odpowiadających im wartości, będzie bardzo proste. Zagadnienia te omówiłem w rozdziale 3. — „Obsługa żądań: Dane przesyłane z formularzy". W

dalszej części tego rozdziału przedstawiłem możliwości konfiguracji formularzy oraz wszelkie elementy kontrolne jaki można w nich umieszczać.

16.2 Element FORM

Formularze HTML pozwalają na tworzenie grup elementów kontrolnych służących do podawania informacji, oraz na kojarzenie ich z wybranym adresem URL. Zazwyczaj, każdemu z takich elementów kontrolnych nadaje się nazwę, a jego wartość określana jest bądź to w dokumencie HTML, bądź przez osobę oglądającą formularz. Gdy formularz jest wysyłany, nazwy oraz wartości wszystkich aktywnych elementów kontrolnych są łączone w jeden łańcuch znaków; przy czym nazwy elementów oraz ich wartości są oddzielane od siebie znakami równości (=), a poszczególne pary nazwa-wartość — znakami &. Tak otrzymany łańcuch znaków jest następnie przesyłany pod adres URL określony w elemencie FORM. W zależności od wybranego sposobu przesłania danych na serwer (GET lub POST), łańcuch ten może zostać poprzedzony znakiem zapytania (?) i dodany do wskazanego adresu URL lub przesłany po nagłówkach żądania HTTP, oddzielony od nich pustym wierszem. W tej części rozdziału omówię element FORM, używany przede wszystkich do określania adresu URL programu służącego do przetwarzania informacji podawanych w formularzu, oraz do określania sposobu przesyłania tych informacji na serwer. Pozostałe części rozdziału będą poświęcone poszczególnym elementom kontrolnym jakie można umieszczać na formularzach.

Element FORM: **<FORM ACTION="URL" ...> ... </FORM>**

Atrybuty: ACTION (wymagany), METHOD, ENCTYPE, TARGET, ONSUBMIT, ONRESET, ACCEPT, ACCEPT-CHARSET.

Element FORM tworzy na stronie WWW obszar przeznaczony do wyświetlenia elementów kontrolnych i określa adres URL, pod jaki zostaną przesłane wszelkie informacje podane w formularzu. Oto przykład tego elementu:

```
<FORM ACTION="http://jakis.serwer.com.pl/servlet/Program">
  Elementy kontrolne formularza oraz zwyczajny kod HTML
</FORM>
```

W dalszej części tego podrozdziału opiszę poszczególne atrybuty elementu FORM — ACTION, METHOD, ENCTYPE, TARGET, ONSUBMIT, ONRESET, ACCEPT oraz ACCEPT-CHARSET. Zwróć uwagę, iż nie będę opisywał atrybutów takich jak STYLE, CLASS oraz LANG, które można stosować we wszystkich elementach języka HTML, a jedynie atrybuty charakterystyczne dla elementu FORM.

ACTION

Atrybut ACTION określa adres URL serwletu lub programu CGI, który zostanie użyty do przetworzenia informacji wpisanych w formularzu (na przykład: *http://cgi.whitehouse.gov/bin/schedule-fund-raiser*) lub adres poczty elektronicznej na jaki zostaną one przesłane (na przykład: *mailto:audit@iris.gov*). Niektórzy dostawcy usług internetowych nie pozwalają zwyczajnym użytkownikom na tworzenie serwletów i programów CGI lub pobierają za ten przywilej dodatkowe opłaty. W takich przypadkach, jeśli musisz gromadzić informacje podawane w formularzu lecz nie chcesz zwracać żadnych wyników (na przykład potwierdzenia przyjęcia zamówienia), przesyłanie danych pocztą elektroniczną jest wygodnym rozwiązaniem. Jeśli informacje wpisane w formularzu mają być przesłane pod wskazany adres poczty elektronicznej, konieczne jest użycie metody POST (patrz kolejny punkt, poświęcony atrybutowi METHOD).

METHOD

Atrybut METHOD określa w jaki sposób informacje zostaną przesłane na serwer. W przypadku użycia metody GET są one poprzedzane znakiem zapytani i dopisywane do adresu URL podanego w

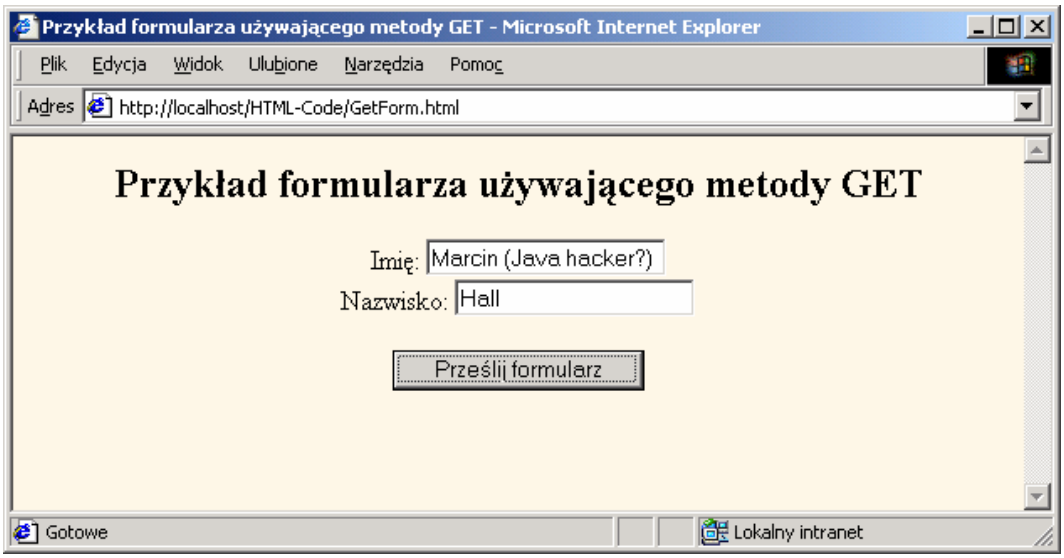
atrybucie ACTION. Przykład wykorzystania tej metody przedstawiłem w podrozdziale 16.1. — „Jak przesyłane są dane z formularzy HTML". Jest to standardowa metoda przesyłania informacji podawanych w formularzach HTML, wykorzystywana przez przeglądarki także podczas pobierania zwyczajnych stron WWW. W przypadku wykorzystania metody POST, informacje z formularza są przesyłane w osobnym wierszu.

Wykorzystanie metody GET ma dwie zalety. Po pierwsze, jest ona prostsza. A po drugie, a w przypadku pisania serwletów pobierających dane tą metodą, użytkownicy mogą je testować bez konieczności tworzenia formularzy — wystarczy podać URL serwletu i dopisać do niego dane. Jednak z drugiej strony, ze względu na ograniczenia długości adresu URL jakie narzucają niektóre przeglądarki, użycie metody GET ogranicza wielkość danych jakie mogą być przesyłane. W przypadku stosowania metody POST, wielkość przesyłanych informacji nie jest niczym ograniczona. Kolejną wadą metody GET jest to, iż przeważająca większość przeglądarek wyświetla adres URL — w tym także dołączone do niego dane pochodzące z formularza — w polu adresowym umieszczonym u góry okna programu. Z tego względu metoda GET zupełnie nie nadaje się do przesyłania ważnych i poufnych informacji, zwłaszcza jeśli Twój komputer znajduje się w ogólnie dostępnym miejscu.

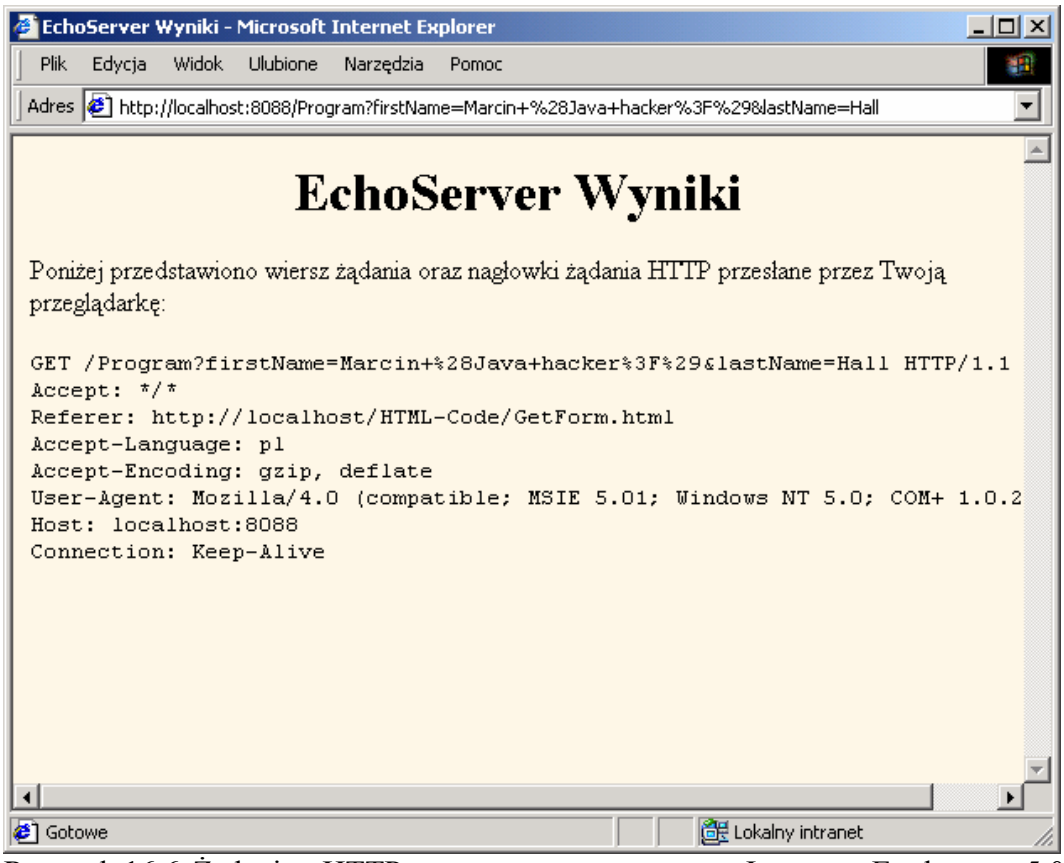
ENCTYP

Ten atrybut określa sposób w jaki informacje podane w formularzu zostaną zakodowane przed ich przesłaniem na serwer. Domyślnie używanym sposobem kodowania jest `application/x-www-form-urlencoded`. Metoda ta polega na zamianie znaków odstępu na znaki plusa (+) oraz wszelkich znaków, które nie są literą bądź cyfrą, na dwie cyfry szesnastkowe reprezentujące wartość znaku (w kodzie ASCII lub ISO Latin-1) poprzedzone znakiem procentu. Kodowanie to odbywa się niezależnie od rozdzielenia nazw pól formularza oraz ich wartości znakami równości (=), a poszczególnych par nazwa-wartość — znakami &.

Na przykład, na rysunku 16.5 przedstawiłem formularz *GetForm.html* (patrz listing 16.1), w którym, w pierwszym polu tekstowym został wpisany łańcuch znaków „`Marcin (Java hacker?)`". Analizując wyniki przedstawione na rysunku 16.6., można się przekonać, że łańcuch ten został przesłany w postaci: „`Marcin+%28Java+hacker%3F%29`". Dlaczego akurat tak? Ponieważ odstępy zostały zamienione na znaki plusa, 28 to zapisana szesnastkowo wartość kodu ASCII nawiasu otwierającego, 3F to wartość kod ASCII znaku zapytania, a 29 to wartość kodu ASCII nawiasu zamykającego.



Rysunek 16.5 Zmodyfikowana zawartość formularza GetForm.html



Rysunek 16.6 Żądania HTTP wygenerowane przez Internet Explorera 5.0 podczas przesyłania formularza GetForm.html z danymi przedstawionymi na rysunku 16.5.

Większość nowych wersji przeglądarek udostępnia dodatkowy sposób kodowania — `multipart/form-data`. Jego użycie spowoduje, że każde z pól formularza zostanie przesłane jako niezależna część dokumentu zgodnego ze specyfikacją MIME, a przeglądarka automatycznie wyśle je przy użyciu metody `POST`. Ten sposób kodowania czasami ułatwia programom działającym na serwerze obsługę złożonych typów danych; poza tym jest on wymagany w przypadku używania elementów kontrolnych umożliwiających przesyłanie na serwer całych plików (patrz podrozdział 16.7). Formularz przedstawiony na listingu 16.3 różni się od formularza *GetForm.html* (patrz listing 16.1) tylko tym, iż znacznik

```
<FORM ACTION="http://localhost:8088/Program>
```

został zamieniony na znacznik

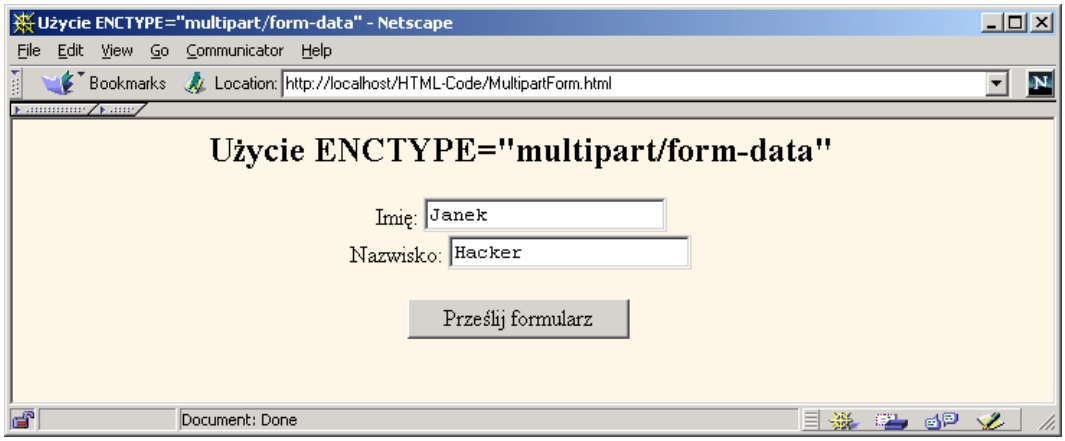
```
<FORM ACTION="http://localhost:8088/Program"
      ENCTYPE="multipart/form-data">
```

Listing 16.3 MultipartForm.html

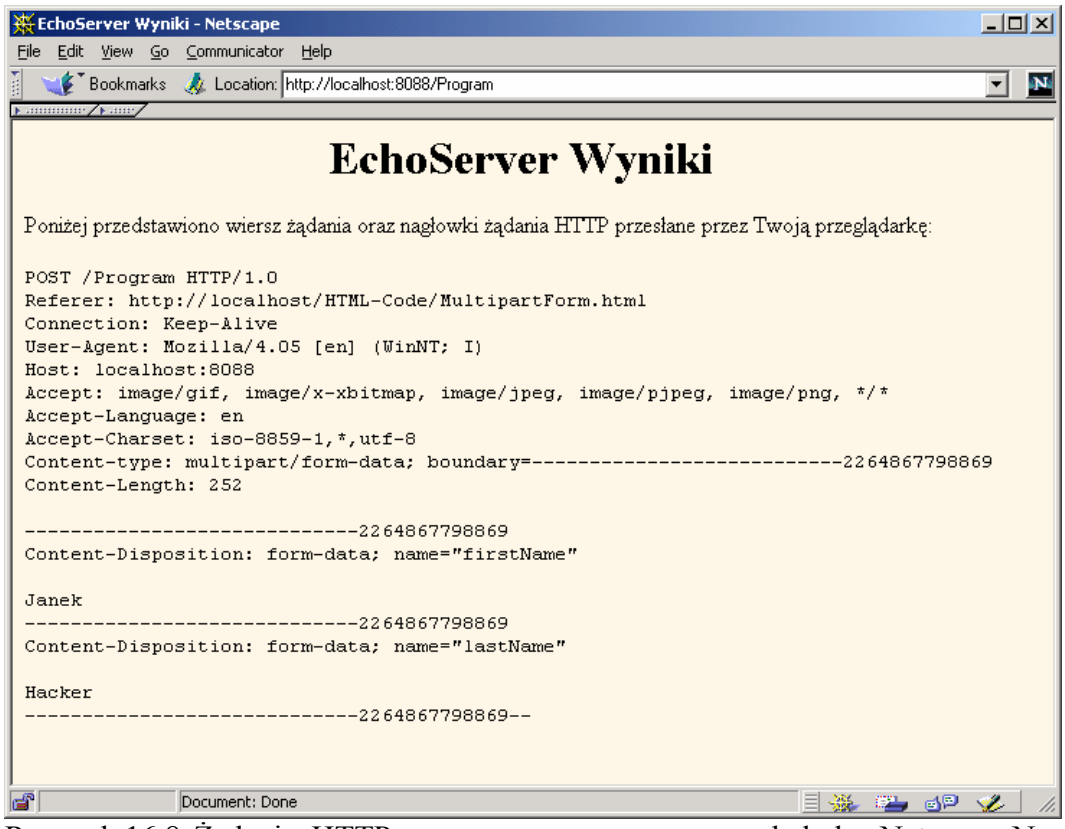
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Użycie ENCTYPE="multipart/form-data"</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H2 ALIGN="CENTER">Użycie ENCTYPE="multipart/form-data"</H2>

<FORM ACTION="http://localhost:8088/Program"
      ENCTYPE="multipart/form-data">
  <CENTER>
  Imię:
  <INPUT TYPE="TEXT" NAME="firstName" VALUE="Janek"><BR>
  Nazwisko:
  <INPUT TYPE="TEXT" NAME="lastName" VALUE="Hacker"><P>
```

```
    <INPUT TYPE="SUBMIT" VALUE="Prześlij formularz">
    </CENTER>
  </FORM>

  </BODY>
  </HTML>
```

Nowa wersja formularza oraz wyniki jej przesłania na serwer zostały przedstawione odpowiednio na rysunkach 16.7 oraz 16.8.



Rysunek 16.7 Wygląd formularza MultipartForm.html



Rysunek 16.8 Żądania HTTP wygenerowane przez przeglądarkę Netscape Navigator 4.7 podczas przesyłania formularza MultipartForm.html

TARGET

Atrybut `TARGET` jest wykorzystywany przez przeglądarki obsługujące ramki, w celu określenia w jakiej ramce mają zostać wyświetlone wyniki wygenerowane przez serwlet lub inny

program przeznaczony do obsługi informacji podanych w formularzu. Domyślna wartość tego atrybutu powoduje, że wyniki zostaną wyświetlone w tej samej ramce, w której był wyświetlony formularz.

ONSUBMIT ORAZ **ONRESET**

Te dwa atrybuty są wykorzystywane w języku JavaScript, w celu określenia kodu, który należy wykonać w momencie wysyłania oraz czyszczenia formularza. W przypadku atrybutu `ONSUBMIT`, jeśli wartość zwrócona przez przetworzony fragment kodu będzie równa `false`, to formularz nie zostanie wysłany. Możliwość ta pozwala na wykonanie w przeglądarce programu napisanego w języku JavaScript, który sprawdzi poprawność formatu informacji podanych w polach formularza, a w przypadku ich braku lub nieprawidłowej postaci zażąda skorygowania błędów.

ACCEPT ORAZ **ACCEPT-CHARSET**

Te dwa atrybuty zostały wprowadzone w języku HTML 4.0 i określają typy MIME (atrybut `ACCEPT`) oraz sposoby kodowania znaków (`ACCEPT-CHARSET`), które muszą być akceptowane przez serwlet bądź inny program używany do przetwarzania danych z formularza. Lista typów MIME podawana w atrybucie `ACCEPT` może być także wykorzystana w przeglądarce do określenia jakie typy plików będą wyświetlane w elementach kontrolnych umożliwiających przesyłanie plików na serwer.

16.3 Tekstowe elementy kontrolne

Język HTML udostępnia trzy typy tekstowych elementów kontrolnych — pola tekstowe, pola hasła oraz wielowierszowe pola tekstowe (nazywane także obszarami tekstowymi). Każdemu z takich elementów należy przypisać nazwę, natomiast wartość jest określana na podstawie zawartości elementu. W momencie przesyłania formularza na serwer — co zazwyczaj następuje po kliknięciu przycisku SUBMIT (patrz podrozdział 16.4) — wysyłana jest nazwa elementu oraz jego wartość.

Pola tekstowe

Element HTML: `<INPUT TYPE="TEXT" NAME="..." ...>`
(brak znacznika zamykającego)

Atrybuty: `NAME` (wymagany), `VALUE`, `SIZE`, `MAXLENGTH`, `ONCHANGE`, `ONSELECT`, `ONFOCUS`, `ONBLUR`, `ONKEYDOWN`, `ONKEYPRESS`, `ONKEYUP`

Ten element tworzy pole tekstowe składające się z jednego wiersza, w którym użytkownicy mogą wpisywać dowolne łańcuchy znaków (patrz rysunki 16.1, 16.2 oraz 16.3). Aby stworzyć wielowierszowe pole tekstowe, należy się posłużyć elementem `TEXTAREA` opisanym w dalszej części rozdziału. `TEXT` jest domyślną wartością atrybutu `TYPE` elementów `INPUT`, choć zaleca się, aby wartość tego atrybutu określać jawnie. Należy pamiętać, iż wewnątrz elementu `FORM` przeglądarki wykorzystują standardowe sposoby przenoszenia wyrazów; a zatem należ uważać, aby tekst opisu nie został oddzielony od opisywanego pola tekstowego.

Metoda
Użyj jawnych konstrukcji języka HTML, aby zgrupować pola tekstowe z ich opisami.

Niektóre przeglądarki wysyłają formularz w momencie gdy kursor znajduje się w polu tekstowym a użytkownik naciśnie klawisz *Enter*. Nie należy jednak polegać na tym sposobie

działania, gdyż nie jest on standardowy. Na przykład, przeglądarka Netscape Navigator przesyła formularza gdy użytkownik naciśnie klawisz *Enter*, wyłącznie jeśli formularz wypełniany w danej chwili zawiera jedno pole tekstowe i niezależnie od ilości formularzy umieszczonych na danej stronie WWW. Internet Explorer przesyła formularz tylko wtedy, gdy na stronie jest jeden formularz, lecz niezależnie od ilości umieszczonych w nim pól tekstowych. Z kolei przeglądarka Mosaic wysyła formularz wyłącznie gdy kursor znajduje się w ostatnim polu tekstowym na całej stronie.

Ostrzeżenie
Nie polegaj na możliwości wysyłania formularzy po naciśnięciu klawisza Enter, gdy kursor znajduje się w polu tekstowym. Zawsze powinieneś stosować przycisk lub mapę odnośników, której kliknięcie spowoduje jawne wysłanie formularza.

W dalszej części tego podrozdziału opiszę atrybuty charakterystyczne dla pól tekstowych; nie będę tu przedstawiał atrybutów stosowanych we wszystkich elementach HTML (takich jak `STYLE`, `CLASS`, czy też `ID`). Atrybut `TABINDEX`, stosowany we wszystkich elementach formularzy, przedstawię w podrozdziale 16.11. — „Określanie kolejności poruszania się pomiędzy elementami formularzy".

NAME

Atrybut `NAME` identyfikuje dane pole tekstowe podczas przesyłania informacji wprowadzonych w formularzu na serwer. W standardowym języku HTML atrybut ten jest wymagany. Ponieważ informacje z formularzy są przesyłane na serwer w formie par nazwa-wartość, jeśli nazwa elementu kontrolnego nie zostanie podana, to żadne informacje z tego pola nie będą przesłane.

VALUE

Jeśli atrybut `VALUE` zostanie podany, to będzie on określać *początkową* zawartość pola tekstowego. W momencie wysyłania formularza, wysyłana jest zawsze *bieżąca* zawartość pola; mogą to być informacje wpisane przez użytkownika. Jeśli w momencie wysyłania formularza pole tekstowe będzie puste, to para nazwa-wartość przybierze postać samej nazwy pola oraz znaku równości (na przykład: `inne-dane&`**`nazwaPolaTekstowego=`**`&inne-dane`).

SIZE

Ten atrybut określa szerokość pola tekstowego, obliczaną na podstawie średniej szerokości znaków aktualnie używanej czcionki. Jeśli w polu zostanie wpisanych więcej liter niż można wyświetlić, tekst zostanie odpowiednio przesunięty. Sytuacja taka może zaistnieć gdy użytkownik wpisze w polu więcej liter niż wynosi wartość atrybut `SIZE` lub, w przypadku stosowania czcionki proporcjonalnej, gdy wpisze tyle liter ile wynosi wartość tego atrybutu lecz są to szerokie litery (takie jak duża litera „W"). Przeglądarka Netscape Navigator automatycznie używa w polach tekstowych czcionki proporcjonalnej. Niestety Internet Explorer tego nie robi, a czcionkę jaka zostanie użyta w polu tekstowym można określić umieszczając element `INPUT` wewnątrz elementów `FONT` lub `CODE`.

MAXLENGTH

Atrybut `MAXLENGTH` określa maksymalną ilość znaków jaką można wpisać w polu tekstowym. Wartość ta nie ma nic wspólnego z ilością znaków *wyświetlanych* w polu, określaną przy użyciu atrybutu `SIZE`.

ONCHANGE, **ONSELECT**, **ONFOCUS**, **ONBLUR**, **ONDBLDOWN**, **ONKEYPRESS** oraz **ONKEYUP**

Te atrybuty są używane wyłącznie w przeglądarkach, które są w stanie obsługiwać język JavaScript. Określają one czynności wykonywane w momencie gdy miejsce wprowadzania opuści pole tekstowe a jego zawartość została wcześniej zmieniona, gdy użytkownik zaznaczy fragment zawartości pola, gdy miejsce wprowadzania zostanie umieszczone w danym polu lub gdy zostanie z niego usunięte oraz kiedy są naciskane klawisze.

Pola hasła

Element HTML: **<INPUT TYPE="PASSWORD" NAME="..." ...>**
(brak znacznika zamykającego)

Atrybuty: NAME (wymagany), VALUE, SIZE, MAXLENGTH, ONCHANGE, ONSELECT, ONFOCUS, ONBLUR, ONKEYDOWN, ONKEYPRESS, ONKEYUP

Pola haseł są tworzone i stosowane tak jak pola tekstowe, jednak różnią się od nich pod tym względem, iż w momencie wpisywania tekstu zamiast podawanych znaków wyświetlany jest jakiś znak „maskujący" — zazwyczaj gwiazdka („*", patrz rysunek 16.9). Takie maskowanie informacji jest przydatne podczas podawania takich informacji jak numery kart kredytowych bądź hasła, których użytkownik nie chciałby pokazywać osobom mogącym przebywać blisko niego. W momencie wysyłania formularza, wartości pól haseł są przesyłane w postaci zwyczajnego tekstu. Jak wiadomo informacje przesyłane metodą GET są dodawane do adresu URL, z tego względu przesyłając dane z formularzy zawierających pola haseł, należy stosować metodę POST, gdyż dzięki temu osoby przebywające koło komputera nie będą w stanie przeczytać jawnie zapisanego hasła, dopisanego do adresu URL i wyświetlonego w polu adresu u góry okna przeglądarki.

Rysunek 16.9 Pole hasła stworzone przy użyciu elementu <INPUT TYPE="PASSWORD" ...>

Metoda
Aby chronić prywatność użytkownika, formularze zawierające pola haseł zawsze należy przesyłać przy użyciu metody POST.

ONCHANGE, **ONSELECT**, **ONFOCUS**, **ONBLUR**, **ONDBLDOWN**, **ONKEYPRESS** oraz **ONKEYUP**

Te atrybuty pól haseł są stosowane dokładnie tak samo jak analogiczne atrybuty zwyczajnych pól tekstowych.

Wielowierszowe pola tekstowe

Element HTML: **<TEXTAREA NAME="..." ROWS=*XXX* COLS=*yyy*>**
...
</TEXTAREA>

Atrybuty: NAME (wymagany), ROWS (wymagany), COLS (wymagany), WRAP (niestandardowy), ONCHANGE, ONSELECT, ONFOCUS, ONBLUR, ONKEYDOWN, ONKEYPRESS, ONKEYUP

Element TEXTAREA tworzy wielowierszowe pole tekstowe, którego przykład przedstawiłem na rysunku 16.10. Element ten nie posiada atrybutu VALUE, a początkową wartością pola staje się dowolny tekst umieszczony pomiędzy otwierającym znacznikiem (<TEXTAREA>) i zamykającym (</TEXTAREA>). Tekst umieszczony pomiędzy tymi znacznikami jest traktowany podobnie do tekstu zapisywanego wewnątrz elementu XMP języka HTML (aktualnie, element ten został już uznany za przestarzały). A zatem, żadne odstępy umieszczane w tym tekście nie są pomijane, a znaczniki

HTML są wyświetlane dosłownie; interpretowane są wyłącznie symbole HTML, takie jak <, >, itd. Wszystkie znaki umieszczane w wielowierszowych polach tekstowych są przed przesłaniem na serwer kodowane według zasad kodowania URL, chyba że w formularzu zostanie wykorzystany inny sposób kodowania, określony przy użyciu atrybutu ENCTYPE (patrz podrozdział 16.2. — „Element FORM"). A zatem, odstępy są zamieniane na znaki plus (+), a wszystkie znaki poza literami i cyframi na kombinacje %*XX*, gdzie *XX* to numeryczna wartość kodowanego znaku zapisana szesnastkowo.

NAME

Ten atrybut określa nazwę pola, która zostanie przesłana na serwer.

ROWS

Atrybut ROWS określa ilość widocznych wiersz tekstu. Jeśli użytkownik wpisze w polu więcej wierszy tekstu, to zostanie w nim wyświetlony pionowy pasek przewijania.

COLS

Atrybut COLS określa widoczną szerokość pola, obliczaną na podstawie średniej szerokości znaków aktualnie używanej czcionki. Zachowanie pola w przypadku wpisania w nim (w jednym wierszu) większej ilości znaków niż wynosi zawartość atrybutu COLS jest zależne od przeglądarki. I tak, w Netscape Navigatorze, zostaje dodany poziomy pasek przewijania (choć ten sposób działania może się zmienić w przypadku użycia atrybutu WRAP opisanego poniżej), natomiast w Internet Explorerze całe słowo zostaje przeniesione do kolejnego wiersza.

WRAP

Atrybut ten wprowadzony przez firmę Netscape i uwzględniany wyłącznie przez jej przeglądarki, określa co należy zrobić gdy długość jednego wiersza tekstu przekroczy dopuszczalną wartość określoną przy użyciu atrybutu COLS. Zastosowanie domyślnej wartości tego atrybutu — OFF — powoduje wyłącznie przenoszenia słów do kolejnego wiersza. W takim przypadku użytkownik może jawnie umieścić w tekście znaki nowego wiersza. Użycie wartości HARD sprawi, że słowa będą przenoszone, a dodane znaki nowego wiersza zostaną przekazane na serwer podczas przesyłania formularza. I w końcu, użycie wartości SOFT sprawi, że wyrazy będą przenoszone do następnego wiersza w polu tekstowym, lecz dodatkowe znaki końca wiersza nie będą przesyłane na serwer.

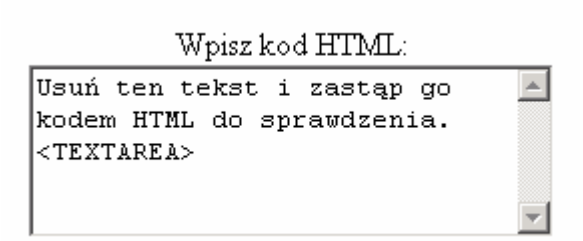
ONCHANGE, **ONSELECT**, **ONFOCUS**, **ONBLUR**, **ONKEYDOWN**, **ONKEYPRESS** oraz **ONKEYUP**

Te atrybuty są wykorzystywane wyłącznie w przeglądarkach obsługujących skrypty pisane w języku JavaScript. Określają one kod, jaki ma być wykonywany gdy zajdą pewne, ściśle określone warunki. Atrybut ONCHANGE określa kod, który będzie wykonany gdy miejsce wprowadzania zostanie usunięte z pola, którego zawartość została wcześniej zmieniona. Atrybut ONSELECT określa co należy zrobić gdy zawartość pola zostanie zaznaczona przez użytkownika. Atrybuty ONFOCUS oraz ONBLUR określają co należy zrobić gdy miejsce wprowadzania zostanie przeniesione do danego pola lub gdy zostanie z niego usunięte. Pozostałe atrybuty określają co należy zrobić w przypadku naciśnięcia klawiszy.

Przedstawiony poniżej przykład prezentuje wielowierszowe pole tekstowe zawierające 5 widocznych wierszy, z których każdy ma szerokość 30 znaków. Wygląd tego pola przedstawiłem na rysunku 16.10.

```
<CENTER>
<P>
Wpisz kod HTML:<BR>
```

```
<TEXTAREA NAME="HTML" ROWS=5 COLS=30>
Usuń ten tekst i zastąp go
kodem HTML do sprawdzenia.
<TEXTAREA>
```



Rysunek 16.10 Wielowierszowe pole tekstowe (obszar tekstowy)

16.4 Przyciski

W formularzach HTML przyciski są stosowane w dwóch podstawowych celach — przesyłania formularza oraz przywracania oryginalnych wartości pól podanych w dokumencie HTML. Przeglądarki, które są w stanie wykonywać programy pisane w języku JavaScript, mogą także używać przycisków w jeszcze jednym celu — aby wykonywać określony skrypt.

Tradycyjnie przyciski były tworzone przy wykorzystaniu elementu `INPUT` z atrybutem `TYPE` o wartościach `SUBMIT`, `RESET` lub `BUTTON`. W języku HTML 4.0 został wprowadzony dodatkowy element — `BUTTON`, jak na razie jest on jednak obsługiwany wyłącznie w Internet Explorerze. Ten nowy element pozwala na tworzenie przycisków, których etykiety są wyświetlane w kilku wierszach oraz mogą zawierać obrazy, różnego rodzaju czcionki, itp. Ze względu na swoje możliwości, stosowanie tego znacznika jest zalecane w sytuacjach gdy można mieć pewność, że użytkownicy będą korzystać z przeglądarek, które są w stanie go poprawnie obsłużyć (na przykład w korporacyjnych intranetach). Jak na razie Netscape Navigator nie obsługuje tego znacznika — przynajmniej wersja 4.7 — a zatem jego zastosowanie należy ograniczyć wyłącznie do witryn intranetowych, których użytkownicy korzystają z Internet Explorera.

Ostrzeżenie
Przeglądarka Netscape Navigator nie obsługuje elementu `BUTTON`.

Przycisk SUBMIT

Element HTML: **`<INPUT TYPE="SUBMIT" ...>`**
(brak znacznika zamykającego)

Atrybuty: `NAME`, `VALUE`, `ONCLICK`, `ONDBLCLICK`, `ONFOCUS`, `ONBLUR`

Po kliknięciu przycisku tego typu formularz jest przesyłany do serwletu lub innego programu działającego na serwerze i określonego przy użyciu atrybutu `ACTION` elementu `FORM`. Choć przesłanie formularza może zostać wyzwolone także w inny sposób (na przykład poprzez kliknięcie mapy odnośników), to jednak przeważająca większość formularzy zostaje wyposażonych w przynajmniej jeden przycisk `SUBMIT`. Przyciski te, podobnie jak i inne elementy kontrolne formularzy, są prezentowane w sposób zależny od używanego systemu operacyjnego. Oznacza to, że na różnych platformach komputerowych, będą wyglądały nieco inaczej. Rysunek 16.11 przedstawia przycisk `SUBMIT` utworzony w systemie Windows 98 przy użyciu znacznika o następującej postaci:

```
<INPUT TYPE="SUBMIT">
```

Rysunek 16.11 Przycisk SUBMIT z domyślną etykietą

NAME oraz **VALUE**

Większość elementów kontrolnych służących do wprowadzania danych posiada nazwę oraz skojarzoną z nią wartość. W momencie przesyłania formularza, nazwy oraz wartości aktywnych elementów kontrolnych są łączone ze sobą w jeden łańcuch znaków, zawierający wszystkie informacje wpisane w formularzu. Jeśli przycisk `SUBMIT` jest używany wyłącznie do zainicjowania procesu przesyłania formularza, to jego nazwę można pominąć, co sprawi, że żadne informacje o przycisku nie zostaną przekazane na serwer. Jeśli jednak nazwa przycisku *zostanie* podana, to na serwer będzie przesłana wyłącznie nazwa i wartość przycisku, który użytkownik klikną. W takim przypadku wartością przycisku przesyłaną na serwer, będzie wyświetlona na nim etykieta. Na przykład, poniższy fragment kodu tworzy pole tekstowe oraz dwa przyciski przedstawione na rysunku 16.12. Jeśli użytkownik kliknąłby, dajmy na to pierwszy przycisk, to dane przesłane z formularza na serwer miałyby następującą postać: `Towar=256MB+SIMM&Dodaj=Dodaj+do+koszyka`

```
<CENTER>
Towar:
<INPUT TYPE="TEXT" NAME="Towar" VALUE="256MB SIMM"><BR>
<INPUT TYPE="SUBMIT" NAME="Dodaj" VALUE="Dodaj do koszyka">
<INPUT TYPE="SUBMIT" NAME="Usun"  VALUE="Usuń z koszyka">
</CENTER>
```

Rysunek 16.12 Przyciski SUBMIT o etykietach określonych przez użytkownika

ONCLICK, **ONDBLCLICK**, **ONFOCUS** oraz **ONBLUR**

Te niestandardowe atrybuty są wykorzystywane przez przeglądarki, które są w stanie obsługiwać skrypty pisane w języku JavaScript. Służą one do kojarzenia skryptów z przyciskami. Kody określane przy użyciu atrybutów `ONCLICK` oraz `ONDBLCLICK` są wykonywane w momencie kliknięcia przycisku. Kod określony przy użyciu atrybutu `ONFOCUS` jest wykonywany gdy miejsce wprowadzania zostanie przeniesione do danego przycisku, a kod określony przy użyciu atrybutu `ONBLUR` — gdy miejsce wprowadzania zostanie usunięte z przycisku. Jeśli wykonanie fragmentu kodu skojarzonego z przyciskiem zwróci wartość `false`, to formularz nie zostanie przesłany na serwer. Przy podawaniu nazw atrybutów HTML wielkość liter nie jest brana pod uwagę, a programiści JavaScript zapisują zazwyczaj powyższe cztery atrybuty w postaci: `onClick`, `onDblClick`, `onFocus` oraz `onBlur`.

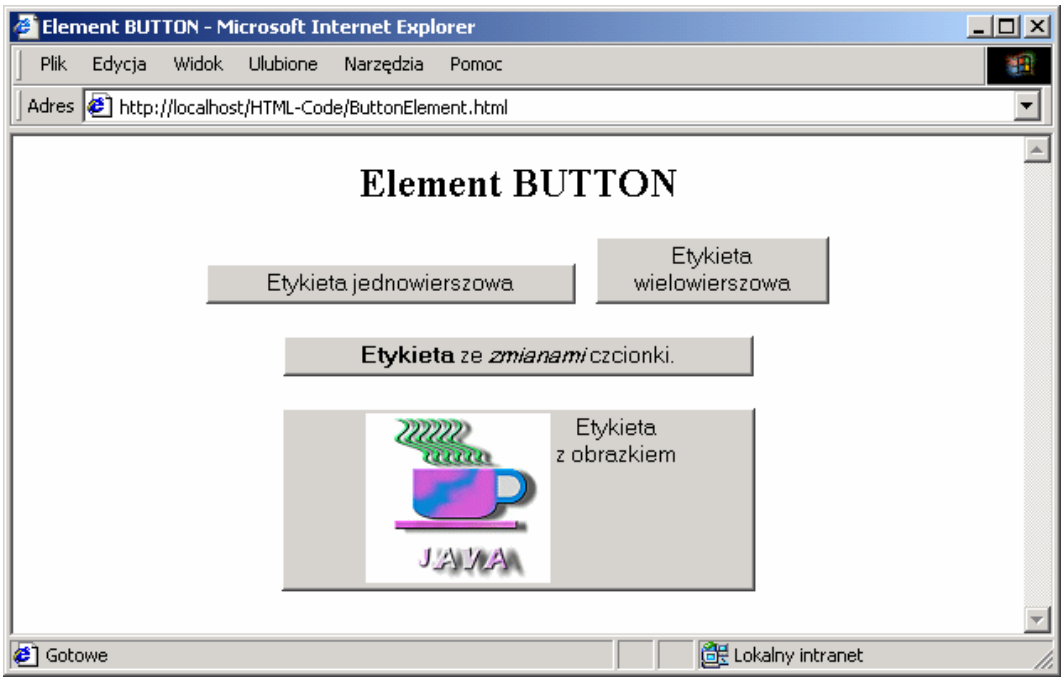
Element HTML: **`<BUTTON TYPE="SUBMIT" ...>`** ***kod*** **`</BUTTON>`** ***HTML***

Atrybuty: `NAME`, `VALUE`, `ONCLICK`, `ONDBLCLICK`, `ONFOCUS`, `ONBLUR`

Alternatywny sposób tworzenia przycisków (obsługiwany wyłącznie w przeglądarce Internet Explorer) pozwala na określanie etykiet przycisków przy użyciu zwyczajnego kodu HTML. Elementy tego typu umożliwiają tworzenie etykiet wyświetlanych w wielu wierszach, zmienianie czcionki, wyświetlanie obrazów na przyciskach, itd. Kilka przykładów użycia tego znacznika przedstawiłem na listingu 16.4 oraz rysunku 16.13.

Listing 16.4 ButtonElement.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Element BUTTON</TITLE>
</HEAD>
<BODY BGCOLOR="WHITE">
<H2 ALIGN="CENTER">Element BUTTON</H2>

<FORM ACTION="http://localhost:8088/SomeProgram">
<CENTER>
<BUTTON TYPE="SUBMIT">Etykieta jednowierszowa</BUTTON>
  
<BUTTON TYPE="SUBMIT">Etykieta<BR>wielowierszowa</BUTTON>
<P>
<BUTTON TYPE="SUBMIT">
<B>Etykieta</B> ze <I>zmianami</I> czcionki.
</BUTTON>
<P>
<BUTTON TYPE="SUBMIT">
<IMG SRC="images/Java-Logo.gif" WIDTH=110 HEIGHT=101
     ALIGN="LEFT" ALT="Java Cup Logo">
Etykieta<BR>z obrazkiem
</BUTTON>
</CENTER>
</FORM>

</BODY>
</HTML>
```



Rysunek 16.13 Przyciski SUBMIT tworzone przy użyciu elementu BUTTON

Przyciski RESET

Element HTML: `<INPUT TYPE="RESET" ...>`
(brak znacznika zamykającego)

Atrybuty: NAME, VALUE, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR

Przyciski RESET służą do przywracana oryginalnych wartości (czyli wartości określonych przy użyciu atrybutu VALUE) wszystkich elementów kontrolnych formularza. Wartości tych przycisków nigdy nie są przesyłane na serwer wraz z wartościami pozostałych pól formularzy.

VALUE

Atrybut VALUE określa etykietę jaka zostanie wyświetlona na przycisku. Domyślną etykietą przycisków tego typu jest słowo „Resetuj".

NAME

Ponieważ wartości przycisków typu RESET nie są umieszczane wśród danych przekazywanych z formularza na serwer, to w standardzie języka HTML nie trzeba określać nazw tych przycisków. Niemniej jednak w przypadku stosowania skryptów pisanych w języku JavaScript można stosować atrybutu NAME, aby uprościć sposób odwoływania się do tych elementów.

ONCLICK, **ONDBLCLICK**, **ONFOCUS** oraz **ONBLUR**

Te niestandardowe atrybuty są wykorzystywane przez przeglądarki, które są w stanie obsługiwać skrypty pisane w języku JavaScript. Służą one do kojarzenia skryptów z przyciskami. Kody określane przy użyciu atrybutów ONCLICK oraz ONDBLCLICK są wykonywane w momencie kliknięcia przycisku, kod określony przy użyciu atrybutu ONFOCUS — gdy miejsce wprowadzania zostanie przeniesione do danego przycisku, a kod określony przy użyciu atrybutu ONBLUR — gdy miejsce wprowadzania zostanie usunięte z przycisku. Przy podawaniu nazw atrybutów HTML wielkość liter nie jest brana pod uwagę, a programiści JavaScript zapisują zazwyczaj powyższe cztery atrybuty w postaci: onClick, onDblClick, onFocus oraz onBlur.

Element HTML: **<BUTTON TYPE="RESET" ...>**
kod HTML
</BUTTON>

Atrybuty: NAME, VALUE, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR

Ten alternatywny sposób tworzenia przycisków RESET można stosować wyłącznie w Internet Explorerze. Pozwala on na określenie treści etykiety przycisku, przy użyciu kodu HTML. Wszystkie atrybuty tego elementu są używane tak samo, jak analogiczne atrybuty elementu <INPUT TYPE="RESET" ...>.

Przyciski JavaScript

Element HTML: **<INPUT TYPE="BUTTON" ...>**
(brak znacznika zamykającego)

Atrybuty: NAME, VALUE, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR

Element BUTTON jest rozpoznawany wyłącznie przez przeglądarki, które są w stanie wykonywać skrypty pisane w języku JavaScript. Element ten tworzy przyciski wyglądające tak samo jak przyciski SUBMIT i RESET i jednocześnie pozwala kojarzyć skrypty z atrybutami ONCLICK, ONDBLCLICK, ONFOCUS oraz ONBLUR. Para nazwa-wartość skojarzona z takim przyciskiem nie jest przesyłana na serwer wraz z pozostałymi informacjami podanymi w formularzu. Choć z przyciskami tego typu można kojarzyć całkowicie dowolny kod, to jednak są one najczęściej wykorzystywane w celu sprawdzania czy wartości pozostałych pól formularza zostały zapisane poprawnie, jeszcze zanim zostaną one przesłane na serwer. Na przykład, przedstawiony poniżej fragment kodu tworzy przycisk, którego kliknięcie spowoduje wykonanie zdefiniowanej przez użytkownika funkcji sprawdzFormularz:

```
<INPUT TYPE="BUTTON" VALUE="Sprawdź dane"
       onClick="sprawdzFormularz()">
```

Element HTML: **<BUTTON TYPE="BUTTON" ...>**
kod HTML
</BUTTON>

Atrybuty: NAME, VALUE, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR

Tego alternatywnego sposobu tworzenia przycisków JavaScript można używać wyłącznie w przeglądarce Internet Explorer. Pozwala on na tworzenie przycisków, których etykiety określane są przy użyciu kodu HTML. Wszystkie atrybuty tego elementu są używane tak samo, jak analogiczne atrybuty elementu `<INPUT TYPE="BUTTONT" ...>`.

16.5 Pola wyboru i przyciski opcji

Pola wyboru oraz przyciski opcji są bardzo przydatnymi elementami kontrolnymi, które pozwalają użytkownikom na wybór jednej lub kilku wartości z grupy predefiniowanych opcji. W odróżnieniu od pól wyboru, z których każde można zaznaczać niezależnie od pozostałych, przyciski opcji mogą tworzyć grupy, w których, w danej chwili, można zaznaczyć tylko jedną opcję.

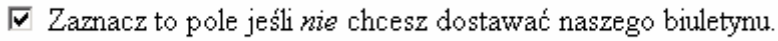
Pola wyboru

Element HTML: **`<INPUT TYPE="CHECKBOX" NAME="..." ...>`**
(brak znacznika zamykającego)

Atrybuty: `NAME` (wymagany), `VALUE`, `CHECKED`, `ONCLICK`, `ONFOCUS`, `ONBLUR`

Ten element tworzy parę nazwa-wartość, która zostanie przesłana na serwer *wyłącznie* wtedy, gdy podczas przesyłania formularza pole wyboru będzie zaznaczone. Na przykład, poniższy fragment kodu tworzy pole wyboru przedstawione na rysunku 16.14.

```
<P>
<INPUT TYPE="CHECKBOX" NAME="bezPoczty" CHECKED>
Zaznacz to pole jeśli <I>nie</I> chcesz dostawać
naszego biuletynu.
```



Rysunek 16.14 Pole wyboru

Warto zauważyć, iż tekst opisujący pole wyboru jest zwyczajnym kodem HTML i należy zwrócić dużą uwagę, aby zapewnić, że zostanie on wyświetlony obok pola. Z tego względu, na samym początku powyższego fragmentu kodu został umieszczony znacznik `<P>`, który zapewnia, że pole wyboru nie będzie należało do wcześniejszego akapitu.

Metoda
Akapity tekstu wewnątrz elementów `FORM` są wypełniane i łamane tak samo jak wszystkie pozostałe akapity umieszczone na stronie. Nie zapomnij użyć odpowiednich znaczników HTML, aby zapewnić, że elementy kontrolne formularzy będą wyświetlane wraz z opisującym je tekstem.

NAME

Ten atrybut określa nazwę, która zostanie przesłana na serwer. W standardzie języka HTML atrybut ten jest wymagany, jednak w przypadku używania pól wyboru wraz ze skryptami pisanymi w języku JavaScript, staje się on opcjonalny.

VALUE

Atrybut `VALUE` jest opcjonalny, a jego domyślaną wartością jest `on`. Przypominasz sobie zapewne, że para nazwa-wartość jest przesyłana na serwer wyłącznie wtedy, gdy w momencie przesyłania dane pole wyboru będzie zaznaczone. W powyższym przykładzie, do danych przesyłanych z formularza zostałby dopisany łańcuch znaków `bezPoczy=on`, gdyż pole jest zaznaczone; gdyby jednak nie było, to do pozostałych danych nie zostałyby dodane żadne

informacje. Z tego powodu, serwlety oraz wszelkie inne programy CGI bardzo często sprawdzają samą obecność nazwy pola wyboru, zupełnie ignorując jego wartość.

CHECKED

Jeśli atrybut `CHECKED` zostanie podany w dokumencie HTML, to bezpośrednio po jego wyświetleniu w przeglądarce, pole wyboru będzie zaznaczone. Jeśli atrybut nie zostanie podany, to bezpośrednio po wyświetleniu strony, pole wyboru będzie puste.

ONCLICK, **ONFOCUS** oraz **ONBLUR**

Te atrybuty umożliwiają podanie kodu napisanego w języku JavaScript, który zostanie wykonany w momencie kliknięcia przycisku, gdy do elementu kontrolnego zostanie przeniesione miejsce wprowadzania bądź też gdy miejsce wprowadzania zostanie z niego usunięte.

Przyciski opcji

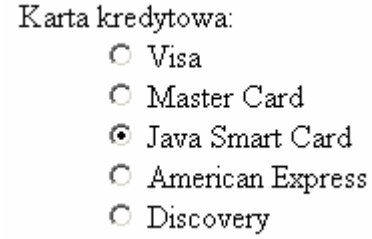
Element HTML: **`<INPUT TYPE="RADIO" NAME="..." VALUE="..." ...>`**
(brak znacznika zamykającego)

Atrybuty: `NAME` (wymagany), `VALUE` (wymagany), `CHECKED`, `ONCLICK`, `ONFOCUS`, `ONBLUR`

Przyciski opcji różnią się od pól wyboru tym, iż tylko jeden przycisk należący do jakiejś grupy może być w danej chwili zaznaczony. Grupę tworzą przyciski opcji o identycznej wartości atrybutu `NAME`. W danej chwili, tylko jeden przycisk w grupie może być „wciśnięty" — czyli zaznaczony. Wybór jednego z przycisków powoduje usunięcie zaznaczenia przycisku, który był wcześniej zaznaczony. Wartość zaznaczonego przycisku jest przesyłana na serwer w raz z pozostałymi informacjami podanymi w formularzu. Choć z technicznego punktu widzenia poszczególne przyciski opcji należące do jednej grupy nie muszą być wyświetlane blisko siebie, to jednak niemal zawsze zaleca się, aby na stronie były one umieszczone tuż obok siebie.

Poniżej przedstawiłem przykład grupy przycisków opcji. Ponieważ elementy kontrolne stanowią część zwyczajnych akapitów tekstu, użyłem elementu `DL`, dzięki któremu poszczególne przyciski opcji zostaną wyświetlone na wynikowej stronie WWW jeden poniżej drugiego, a dodatkowo będą przesunięte względem umieszczonego nad nimi nagłówka. Wygląd poniższego fragmentu strony przedstawiłem na rysunku 16.15. W przypadku przesłania formularza przedstawionego na tym rysunku, oprócz pozostałych podanych w nim informacji, na serwer zostałyby przekazane dane o postaci `kartaKredytowa=java`.

```
<DL>
  <DT>Karta kredytowa:
  <DD><INPUT TYPE="RADIO" NAME="kartaKredytowa" VALUE="visa">
      Visa
  <DD><INPUT TYPE="RADIO" NAME="kartaKredytowa" VALUE="mastercard">
      Master Card
  <DD><INPUT TYPE="RADIO" NAME="kartaKredytowa" VALUE="java" CHECKED>
      Java Smart Card
  <DD><INPUT TYPE="RADIO" NAME="kartaKredytowa" VALUE="amex">
      American Express
  <DD><INPUT TYPE="RADIO" NAME="kartaKredytowa" VALUE="discover">
      Discovery
</DL>
```



Rysunek 16.15 Przyciski opcji

NAME

W odróżnieniu od atrybutu NAME większości elementów kontrolnych formularzy, w przypadku przycisków opcji atrybut ten nie musi przybierać unikalnych wartości. Wszystkie przyciski opcji które mają tę samą wartość atrybutu NAME są grupowane logicznie, tak iż w danej chwili tylko jeden z nich może być zaznaczony. Należy zwrócić uwagę, iż w wartościach tego atrybutu uwzględniana jest wielkość liter, a zatem, w poniższym fragmencie kodu zostały zdefiniowane dwa przyciski opcji, które nie są ze sobą logicznie powiązane:

```
<INPUT TYPE="RADIO" NAME="Grupa1" VALUE="wartosc1">
<INPUT TYPE="RADIO" NAME="GRUPA1" VALUE="wartosc1">
```

Ostrzeżenie
Upewnij się, że wartości atrybutu NAME we wszystkich przyciskach opcji należących do tej samej grupy logicznej, są identyczne (także pod względem wielkości liter).

VALUE

Atrybut VALUE określa wartość przycisku opcji, która podczas przesyłania formularza zostanie przekazana na serwer wraz z jego nazwą (wartością atrybutu NAME). Wartość tego atrybutu nie ma żadnego wpływu na sposób prezentacji przycisku opcji — przyciski opcji, podobnie jak pola wyboru, są umieszczane wewnątrz zwyczajnego tekstu lub kodu HTML.

CHECKED

Jeśli ten atrybut zostanie podany, to bezpośrednio po wyświetleniu strony WWW, dany przycisk opcji będzie zaznaczony. W przeciwnym przypadku, po wyświetleniu strony przycisk opcji nie będzie zaznaczony.

ONCLICK, **ONFOCUS** oraz **ONBLUR**

Te atrybuty umożliwiają podanie kodu napisanego w języku JavaScript, który zostanie wykonany w momencie kliknięcia przycisku, gdy do przycisku zostanie przeniesione miejsce wprowadzania bądź też gdy miejsce wprowadzania zostanie usunięte z przycisku.

16.6 Listy i listy rozwijane

Element SELECT wyświetla grupę opcji, z których użytkownik może wybrać jedną lub kilka. Jeśli tylko jedna spośród tych opcji może zostać wybrana lub jeśli nie została określona ilość opcji jakie będą jednocześnie widoczne, to element SELECT zostanie przedstawiony w formie listy rozwijanej. Elementy SELECT są przedstawiane w formie zwyczajnej listy jeśli można zaznaczyć większą ilość opcji lub jeśli określono ilość opcji jakie będą jednocześnie widoczne na liście. Same opcje listy są określane przy użyciu elementów OPTION umieszczanych wewnątrz elementu SELECT. Poniżej przedstawiłem typową postać list:

```
<SELECT NAME="nazwa" ...>
  <OPTION VALUE="wartosc1">tekst opcji 1
  <OPTION VALUE="wartosc2">tekst opcji 2
  ...
  <OPTION VALUE="wartoscN">tekst opcji N
</SELECT>
```

Specyfikacja języka HTML 4.0 sugeruje, aby stosować element OPTGROUP (posiadający jeden atrybut — LABEL). Element ten służy do grupowania elementów OPTION i pozwala na tworzenie list hierarchicznych. Jak na razie jednak ani Netscape Navigator ani Internet Explorer nie obsługują tego elementu.

Element HTML: `<SELECT NAME="..." ...> ... </SELECT>`
Atrybuty: NAME (wymagany), SIZE, MULTIPLE, ONCLICK, ONFOCUS, ONBLUR, ONCHANGE

Element SELECT tworzy listę bądź listę rozwijaną z której użytkownik może wybrać jedną lub kilka spośród przedstawionych opcji. Każda z opcji definiowana jest przy użyciu elementu OPTION zapisywanego pomiędzy znacznikami <SELECT> i </SELECT>.

NAME

Atrybut NAME określa nazwę listy, która zostanie przekazana do serwletu bądź innego programu CGI.

SIZE

Ten atrybut określa ilość wierszy jakie będą widoczne na danej liście. Jeśli wartość tego atrybutu zostanie podana, to element SELECT będzie zazwyczaj przedstawiany w formie listy a nie listy rozwijanej. Elementy SELECT są przedstawiane w formie list rozwijanych gdy nie został podany atrybut MULTIPLE ani wartości atrybutu SIZE.

MULTIPLE

Atrybut MULTIPLE określa, że jednocześnie można zaznaczyć wiele opcji danej listy. Jeśli atrybut ten nie zostanie podany, to na liście będzie można zaznaczyć wyłącznie jedną opcję.

ONCLICK, **ONFOCUS** oraz **ONBLUR**

Te atrybuty umożliwiają podanie kodu napisanego w języku JavaScript, który zostanie wykonany w momencie kliknięcia przycisku, gdy do elementu kontrolnego zostanie przeniesione miejsce wprowadzania bądź też gdy miejsce wprowadzania zostanie z niego usunięte.

Element HTML: `<OPTION ...>`
(brak znacznika zamykającego)
Atrybuty: SELECTED, VALUE

Elementy OPTION można umieszczać wyłącznie wewnątrz elementów SELECT; określają one opcje list.

VALUE

Atrybut VALUE określa wartość jaka zostanie przesłana wraz z nazwą listy (wartością atrybuty NAME elementu SELECT), jeśli dana opcja zostanie zaznaczona. Atrybut ten *nie* określa tekstu danej opcji, który będzie wyświetlony na liście; tekst ten jest podawany za znacznikiem OPTION.

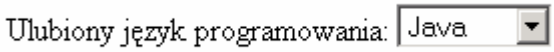
SELECTED

Jeśli atrybut ten zostanie podany, to bezpośrednio po wyświetleniu strony odpowiednia opcja list będzie zaznaczona.

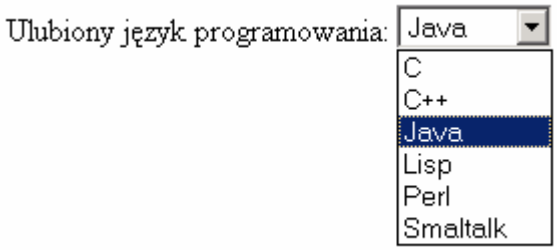
Przedstawiony poniżej przykład przedstawia listę języków programowania. Zaprezentowany element SELECT będzie przedstawiony w formie listy rozwijanej, gdyż można w nim wybrać tylko jedną opcję i nie została określona wartość atrybutu SIZE. Rysunek 16.16 przedstawia początkowy wygląd listy, natomiast rysunek 16.17 — wygląd listy po jej aktywacji (czyli kliknięciu). Jeśli w momencie przesyłania formularza będzie zaznaczona opcja Java, to na serwer zostanie przesłany łańcuch znaków language=java. Zwróć uwagę, że przesyłana jest wartość atrybutu VALUE, a nie tekst opisujący opcję, wyświetlany na liście.

```
Ulubiony język programowania:
<SELECT NAME="language">
```

```
  <OPTION VALUE="c">C
  <OPTION VALUE="c++">C++
  <OPTION VALUE="java" SELECTED>Java
  <OPTION VALUE="lisp">Lisp
  <OPTION VALUE="perl">Perl
  <OPTION VALUE="smalltalk">Smaltalk
</SELECT>
```



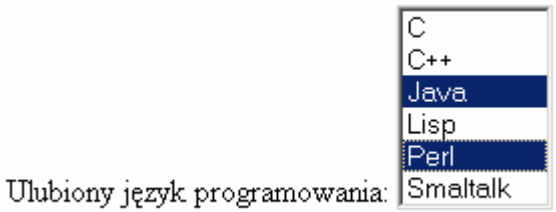
Rysunek 16.16 Element SELECT wyświetlony w formie listy rozwijanej



Rysunek 16.17 Wybieranie opcji listy rozwijanej

Kolejny przykład przedstawia element `SELECT` wyświetlony w formie zwyczajnej listy. Jeśli podczas wysyłania formularza na liście będzie zaznaczony więcej niż jeden element, to przekazanych zostanie więcej niż jedna wartość (przy czym każda z tych wartości zostanie podana w odrębnej parze nazwa-wartość, a w każdej z tych para *nazwa* będzie taka sama). Na przykład, w przykładzie przedstawionym na rysunku 16.18 do informacji przesyłanych na serwer zostanie dodany łańcuch znaków `language=java&language=perl`. To właśnie ze względu na możliwość przesyłania wielu par nazwa-wartość o tej samej *nazwie*, autorzy serwletów muszą znać nie tylko popularną metodę `getParameter`, lecz także rzadziej stosowaną metodę `getParameterValues` interfejsu `HttpServletRequest`. Szczegółowe informacje na temat tych metod znajdziesz w rozdziale 3. — „Obsługa żądań: Dane przesyłane z formularzy”.

```
Ulubiony język programowania:
<SELECT NAME="language" MULTIPLE>
  <OPTION VALUE="c">C
  <OPTION VALUE="c++">C++
  <OPTION VALUE="java" SELECTED>Java
  <OPTION VALUE="lisp">Lisp
  <OPTION VALUE="perl">Perl
  <OPTION VALUE="smalltalk">Smaltalk
</SELECT>
```



Rysunek 16.18 Elementy SELECT w których określono wartość atrybutu SIZE lub użyto atrybutu MULTIPLE są prezentowane w formie zwyczajnych list

16.7 Element kontrolny służący do przesyłania plików

Element HTML: `<INPUT TYPE="FILE" ...>`
(brak znacznika zamykającego)
Atrybuty: `NAME` (wymagany), `VALUE` (pomijany), `SIZE`, `MAXLENGTH`, `ACCEPT`, `CHECKED`, `ONSELECT`, `ONFOCUS`, `ONBLUR` (niestandardowy)

Użycie tego elementu kontrolnego spowoduje wyświetlenie na stronie pola tekstowego oraz przycisku *Przeglądaj*. Użytkownicy mogą podać pełną ścieżkę dostępu do przesyłanego pliku bezpośrednio w polu tekstowym, lub kliknąć przycisk *Przeglądaj*, aby wyświetlić okno dialogowe pozwalające interaktywnie określić położenie pliku. Jeśli w elemencie `FORM` został umieszczony atrybut `ENCTYPE` o wartości `multipart/form-data`, to w momencie wysyłania formularza, na serwer zostanie przesłana *zawartość* wybranego pliku. Element ten można z powodzeniem stosować przy tworzeniu stron służących do pomocy użytkownikom, na których mogą oni podać opis napotkanych problemów a wraz z nim przesłać dodatkowe dane lub pliki konfiguracyjne.

Podpowiedź
W formularzach wykorzystujących elementy kontrolne umożliwiające przesyłanie plików na serwer, zawsze należy stosować atrybut `ENCTYPE` o wartości `multipart/form-data`.

NAME

Atrybut `NAME` identyfikuje element kontrolny podczas przesyłania informacji z formularza na serwer.

VALUE

Ze względów bezpieczeństwa ten atrybut jest ignorowany. Wyłącznie końcowi użytkownicy mogą podawać nazwy plików.

SIZE oraz **MAXLENGTH**

Atrybuty `SIZE` oraz `MAXLENGTH` są stosowane w taki sam sposób jak w przypadku pól tekstowych — czyli atrybut `SIZE` określa widoczną szerokość pola, a atrybut `MAXLENGTH` maksymalną ilość znaków jaką można w nim wpisać.

ACCEPT

Ten atrybut umożliwia podanie listy typów MIME służącej do ograniczenia typów plików wyświetlanych w oknie dialogowym; poszczególne typy MIME podawane na tej liście są od siebie oddzielane przecinkami. Większość przeglądarek nie obsługuje tego atrybutu.

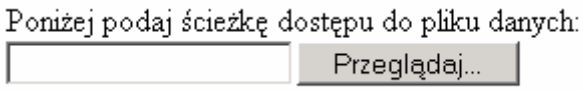
ONCHANGE, **ONSELECT**, **ONFOCUS** oraz **ONBLUR**

Te atrybuty umożliwiają podanie kodu napisanego w języku JavaScript, który będzie wykonany w momencie gdy miejsce wprowadzania zostanie usunięte z pola tekstowego, którego wartość została uprzednio zmieniona, gdy użytkownik zaznaczy zawartość pola tekstowego, gdy do elementu zostanie przeniesione miejsce wprowadzania bądź też gdy miejsce wprowadzania zostanie z niego usunięte.

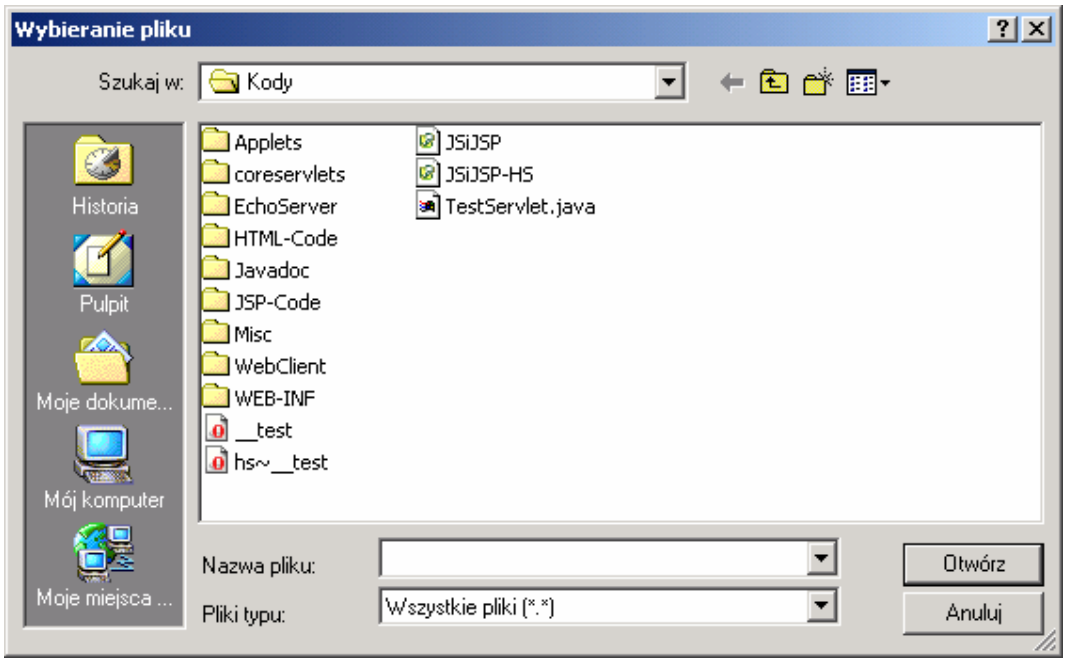
Przedstawiony poniżej fragment kodu tworzy element kontrolny służący do przesyłania plików na serwer. Jego początkowy wygląd przedstawiłem na rysunku 16.19, natomiast rysunek 16.20 przedstawia typowe okno dialogowe wyświetlane po kliknięciu przycisku *Przeglądaj*.

```
<FORM ACTION="http://localhost:8080/Program"
      ENCTYPE="multipart/form-data">
```

```
Poniżej podaj ścieżkę dostępu do pliku danych:<BR>
<INPUT TYPE="FILE" NAME="nazwaPliku">
</FORM>
```



Rysunek 16.19 Początkowy wygląd elementu kontrolnego służącego do przesyłania plików na serwer



Rysunek 16.20 Okno dialogowe służące do wyboru przesyłanego pliku, wyświetlane po kliknięciu przycisku Przeglądaj

16.8 Mapy odnośników obsługiwane na serwerze

W języku HTML dostępny jest element `MAP` umożliwia skojarzenie adresów URL z różnymi rejonami obrazka, a następnie, gdy użytkownik kliknie jeden z tych rejonów, sprawia że przeglądarka pobierze stronę o odpowiednim adresie. Ten typ map odnośników nazywany jest *mapami obsługiwanymi po stronie przeglądarki*, gdyż określenie adresu URL jakiego należy użyć jest dokonywane w przeglądarce, a w proces ten nie jest angażowany żaden program działający na serwerze. Jednak język HTML umożliwia także tworzenie *map odnośników obsługiwanych na serwerze*, które mogą być stosowane jako elementy kontrolne formularzy. W przypadku map tego typu, w przeglądarce jest wyświetlany obraz, a w momencie jego kliknięcia, współrzędne wybranego punktu zostają przesłane do programu wykonywanego na serwerze.

Mapy odnośników obsługiwane po stronie przeglądarki są prostsze i bardziej efektywne od map obsługiwanych po stronie serwera i należy ich używać w sytuacjach, gdy jedyną rzeczą jaką chcesz zrobić jest skojarzenie ściśle określonej grupy adresów URL z predefiniowanymi rejonami obrazu. Z drugiej strony, mapy odnośników obsługiwane po stronie serwera znacznie lepiej nadają się do wykorzystania w sytuacjach gdy adres URL należy określić dynamicznie (na przykład, w przypadku map meteorologicznych), regiony zmieniają się bardzo często lub gdy do żądania trzeba dołączyć informacje podane w formularzu. W tej części rozdziału przedstawię dwa sposoby tworzenia map odnośników obsługiwanych po stronie serwera.

IMAGE — standardowe mapy odnośników obsługiwane po stronie serwera

Standardowym sposobem tworzenia map odnośników obsługiwanych po stronie serwera jest umieszczenie znacznika <IMAGE TYPE="IMAGE" ...> wewnątrz elementu FORM.

Element HTML: **<INPUT TYPE="IMAGE" ...>**
(brak znacznika zamykającego)
Atrybuty: NAME (wymagany), SRC, ALIGN

Element ten wyświetla obraz, którego kliknięcie spowoduje przesłanie formularza do serwletu bądź innego programu wykonywanego na serwerze i określonego przy użyciu atrybutu ACTION elementu FORM. Sama nazwa elementu nie jest przesyłana, lecz zamiast niej przekazywane są dane o postaci *nazwa*.x=*xpoz* oraz *nazwa*.y=*ypoz*, gdzie *xpoz* oraz *ypoz* to współrzędne punktu kliknięcia, liczone względem lewego, górnego wierzchołka obrazu.

NAME
Atrybut NAME identyfikuje element podczas przesyłania na serwer informacji podanych w formularzu.

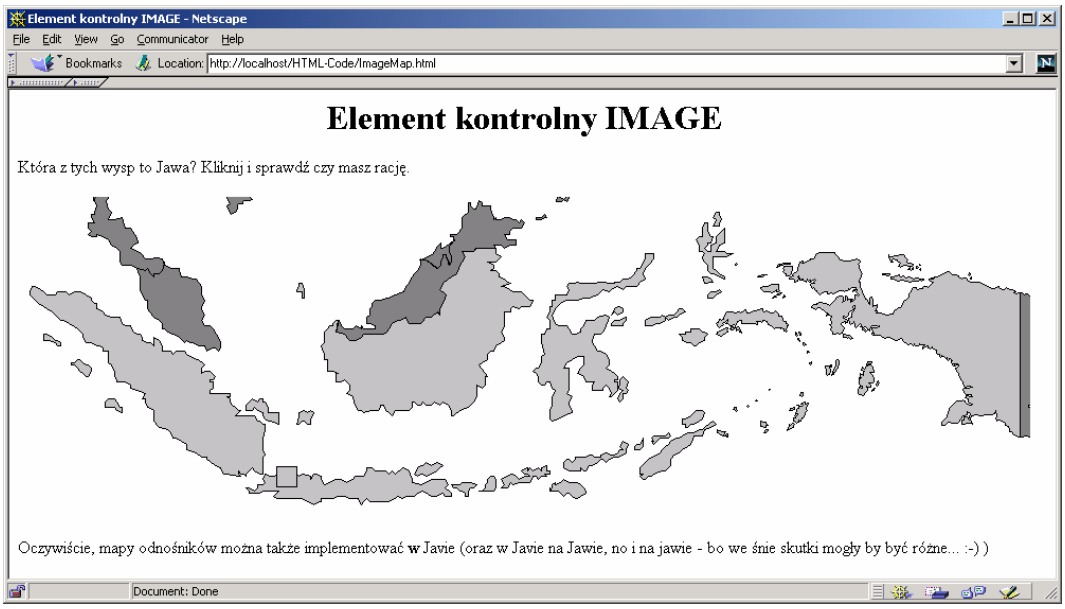
SRC
Atrybut SRC określa adres URL obrazu, który będzie wyświetlony jako mapa odnośników.

ALIGN
Atrybut ALIGN może przybierać te same wartości co analogiczny atrybut elementu IMG i jest stosowany w identyczny sposób (możliwe wartości tego atrybutu to: TOP, MIDDLE, BOTTOM, LEFT oraz RIGHT, przy czym wartością domyślną jest BOTTOM).

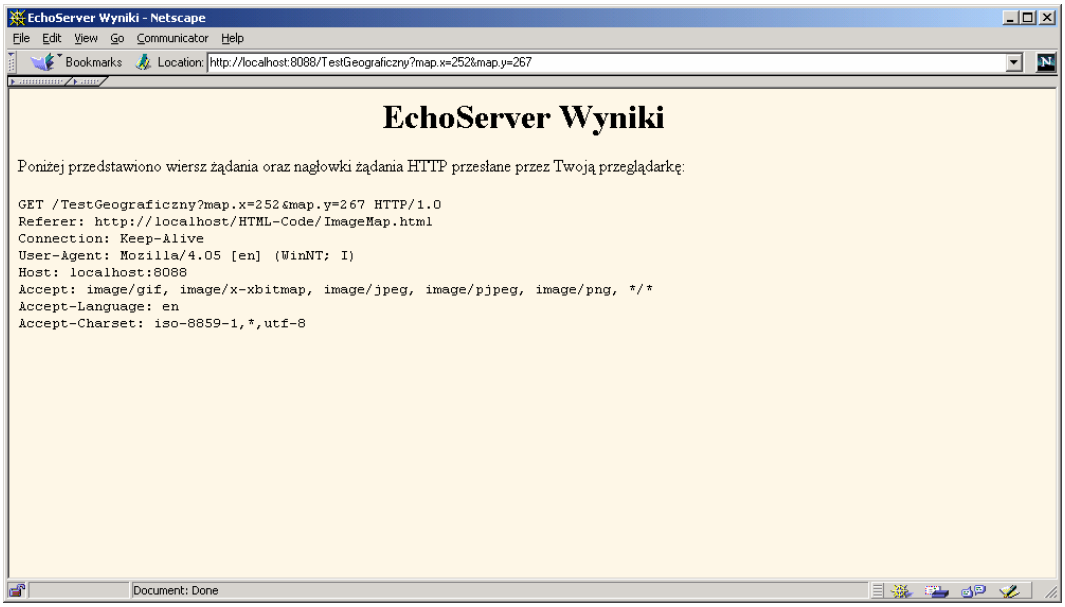
Listing 16.5 przedstawia prostą stronę WWW zawierającą formularz, którego atrybut ACTION (poprzez część adresu URL określającą komputer i numer portu) odwołuje się do programu *EchoServer*. Program ten przedstawię w podrozdziale 16.12. Rysunek 16.21 przedstawia wygląd strony WWW, a rysunek 16.22 wyniki wygenerowane po kliknięciu obrazka.

Listing 16.5 ImageMap.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Element kontrolny IMAGE</TITLE>
</HEAD>

<BODY>
<H1 ALIGN="CENTER">Element kontrolny IMAGE</H1>
Która z tych wysp to Jawa? Kliknij i sprawdź czy masz rację.

<FORM ACTION="http://localhost:8088/TestGeograficzny">
  <INPUT TYPE="IMAGE" NAME="map" SRC="images/indonesia.gif"
   BORDER="0">
</FORM>

Oczywiście, mapy odnośników można także implementować <B>w</B>
Javie (oraz w Javie na Jawie, no i na jawie - bo we śnie skutki
mogły by być różne... :-) )

</BODY>
</HTML>
```



Rysunek 16.21 Element kontrolny IMAGE którego atrybut NAME ma wartość "map"



Rysunek 16.22 Kliknięcie obrazu w punkcie o współrzędnych (252,267) powoduje przesłanie formularza i dodanie do pozostałych informacji danych o postaci map.x=252&map.y=267

ISMAP — alternatywny sposób tworzenia map odnośników obsługiwanych po stronie serwera

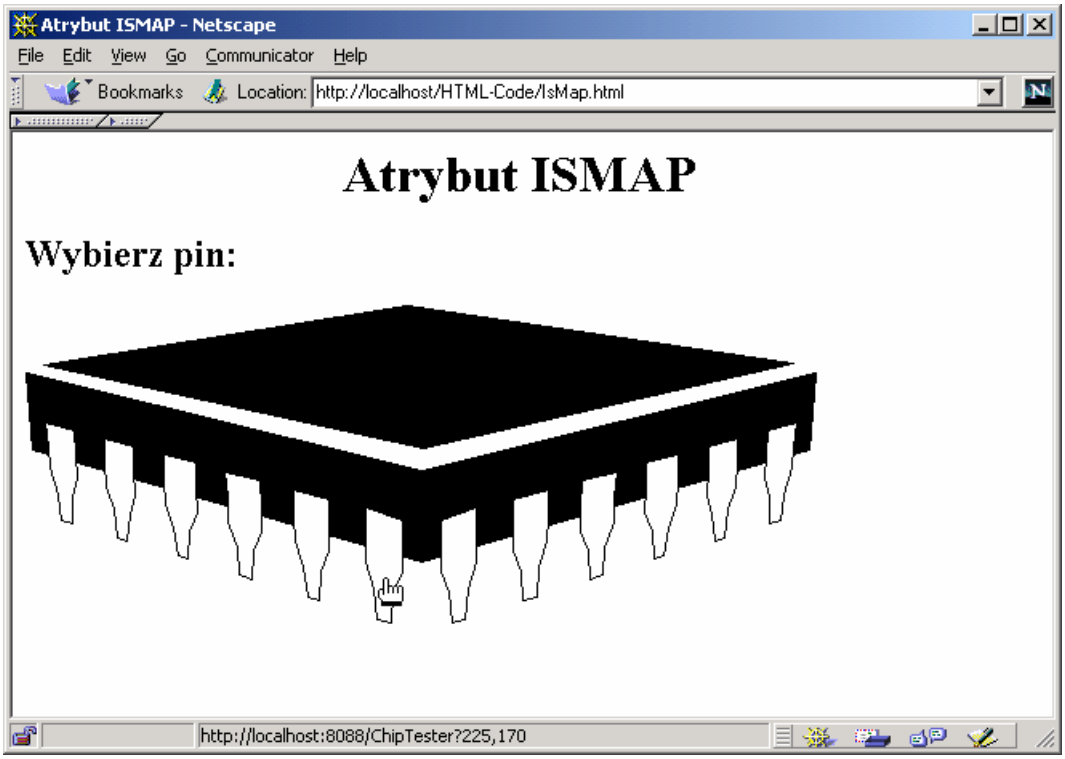
ISMAP to opcjonalny atrybut elementu `IMG`, którego można używać podobnie jak elementów `<INPUT TYPE="IMAGE" ...>` formularzy. `ISMAP` nie jest elementem formularzy, lecz mapy odnośników definiowane przy jego użyciu można wykorzystywać do tworzenia prostych odwołań do serwletów oraz programów CGI. Jeśli obraz z atrybutem `ISMAP` zostanie umieszczony wewnątrz hiperpołączenia, to jego kliknięcie spowoduj przesłanie współrzędnych klikniętego punktu pod wskazany adres URL. Współrzędne te są oddzielone od siebie przecinkiem i określają położenie punktu względem lewego, górnego wierzchołka obrazu.

Na przykład, strona przedstawiona na listingu 16.6 zawiera obraz ze zdefiniowanym atrybutem `ISMAP`, umieszczony wewnątrz połączenia odwołującego się pod adres

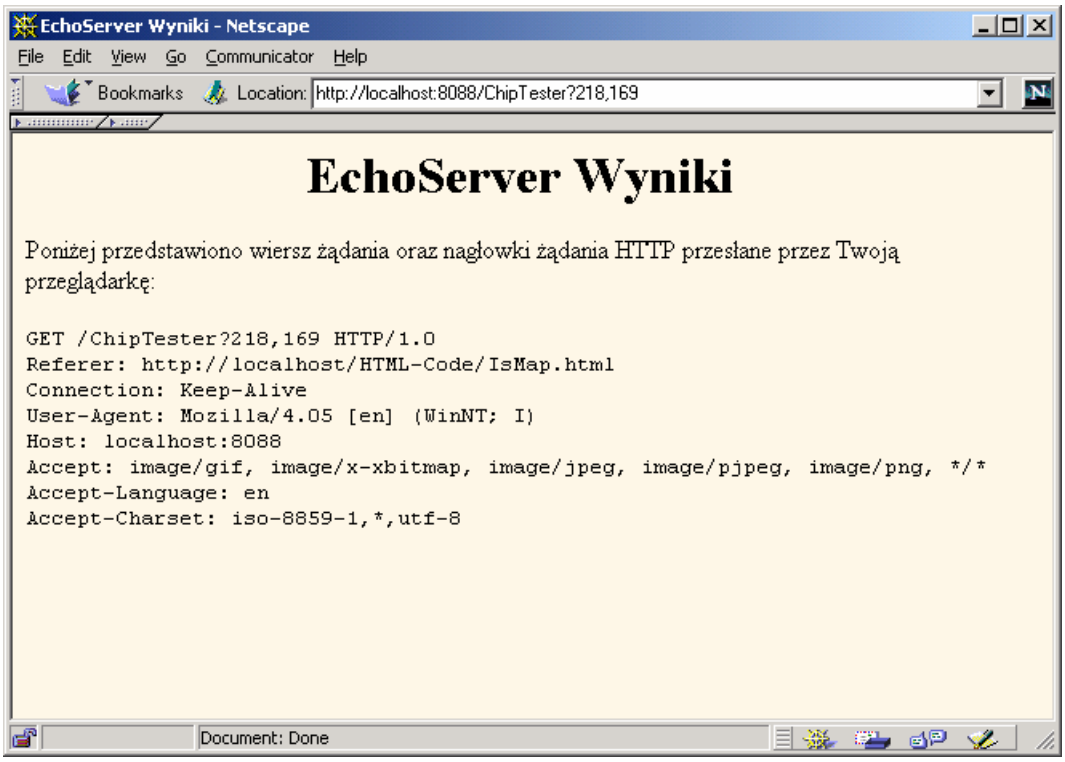
`http://localhost:8088/ChipTester`. Na żądania kierowane pod ten adres odpowiada mini serwer, który przedstawię w podrozdziale 16.12. Wygląd tej strony pokazałem na rysunku 16.23, wyglądała by ona dokładnie tak samo gdyby atrybut `ISMAP` nie został użyty. Jednak po umieszczeniu wskaźnika mysz 271 pikseli na prawo oraz 184 piksele poniżej lewego, górnego wierzchołka obrazu i kliknięciu, przeglądarka przesyła żądanie skierowane pod adres URL `http://localhost:8088/ChipTester?271,184` (co widać na rysunku 16.24).

Listing 16.6 IsMap.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Atrybut ISMAP</TITLE>
</HEAD>

<BODY>

<H1 ALIGN="CENTER">Atrybut ISMAP</H1>
<H2>Wybierz pin:</H2>
<A HREF="http://localhost:8088/ChipTester">
<IMG SRC="images/chip.gif" WIDTH=495 HEIGHT=200 ALT="Chip"
     BORDER=0 ISMAP></A>

</BODY>
</HTML>
```



Rysunek 16.23 Podanie atrybutu ISMAP w znaczniku <IMG> umieszczonym wewnątrz hiperpołączenia zmienia czynności wykonywane w momencie kliknięcia obrazu



Rysunek 16.24 Po kliknięciu obrazu zdefiniowanego przy użyciu znacznika <IMG> z atrybutem ISMAP, pod wskazany adres URL są przesyłane współrzędne punktu kliknięcia

16.9 Pola ukryte

Pola ukryte nie mają żadnego wpływu na wygląd stron WWW wyświetlanych w przeglądarkach. Służą one od przechowywania niezmiennych nazw oraz wartości, które zawsze są przesyłane na serwer w niezmienionej postaci, niezależnie od modyfikacji wprowadzonych przez użytkownika. Istnieją trzy podstawowe powody stosowania pól ukrytych.

Po pierwsze, pola ukryte stanowią jeden ze sposobów śledzenia poczynań użytkowników odwiedzających witrynę (patrz podrozdział 9.1. — „Potrzeba śledzenia sesji"). Autorzy serwletów zazwyczaj korzystają z wbudowanych narzędzi programistycznych służących do śledzenia sesji (patrz podrozdział 9.2.) i nie próbują implementować mechanizmów śledzenia na tak niskim poziomie.

Po drugie, pola ukryte są stosowane w celu przekazania predefiniowanych danych wejściowych do programów wykonywanych na serwerze, gdy wiele różnych dokumentów HTML stanowi interfejs użytkownika obsługiwany przez ten sam program. Na przykład, internetowy sklep może płacić osobom, które na swoich witrynach umieszczają odwołania do niego. W takim przypadku, autorzy witryny odwołującej się do sklepu mogą udostępnić swoim użytkownikom możliwość przeszukiwania katalogu towarów sklepu, zaimplementowaną przy użyciu formularza. Aby właściciele sklepu wiedzieli z jakiej witryny są przesyłane odwołania, jej autorzy mogą umieścić w formularzu pole ukryte zawierające unikalny identyfikator witryny.

W końcu, po trzecie, pola ukryte są wykorzystywane na dynamicznie generowanych stronach WWW do przechowywania informacji kontekstowych. Na przykład, na stronie potwierdzającej przyjęcie zamówienia w internetowym sklepie przedstawionym w podrozdziale 9.4, każdy wiersz wyświetlonej tabeli odpowiada konkretnemu zamówionemu towarowi (patrz rysunek 9.6). Użytkownik może zmienić ilość zamówionych produktów, lecz na formularzu nie ma żadnego widocznego elementu służącego do przechowywania identyfikatora danego towaru. A zatem, informacje te są przechowywane w polach ukrytych (patrz listing 9.5).

Element HTML: `<INPUT TYPE="HIDDEN" NAME="..." VALUE="...">`
(brak znacznika zamykającego)
Atrybuty: `NAME` (wymagany), `VALUE`

Ten element przechowuje nazwę oraz wartość, lecz w przeglądarce nie jest prezentowany w widocznej postaci. Para nazwa-wartość jest dodawana do pozostałych informacji podczas przesyłania formularza. W poniższym przykładzie, za każdym razem gdy zostanie wysłany formularz, do przesyłanych informacji zostanie dodany łańcuch znaków `IDtowaru=agd111`.

```
<INPUT TYPE="HIDDEN" NAME="IDtowaru" VALUE="agd111">
```

Zwróć uwagę, iż termin „hidden”[13], nie oznacza wcale, że użytkownik nie będzie mógł dowiedzieć się o istnieniu pola — będzie ono bowiem widoczne w kodzie źródłowym strony. Nie istnieje żaden bezpieczny sposób „ukrywania” kodu HTML strony i z tego względu nie zaleca się przechowywania w polach ukrytych ważnych informacji, takich jak hasła.

16.10 Grupowanie elementów kontrolnych

Język HTML 4.0 udostępnia element `FIELDSET`, który wraz z elementem `LEGEND` może posłużyć do wizualnego grupowania elementów kontrolnych formularzy. Możliwość ta jest bardzo przydatna, lecz jak na razie obsługuje ją jedynie Internet Explorer. Miejmy nadzieję, że piąta wersja przeglądarki Netscape Navigator także będzie ją obsługiwać. Jak na razie jednak powinieneś wykorzystywać te elementy wyłącznie w aplikacjach, których wszyscy użytkownicy posługują się Internet Explorerem.

Ostrzeżenie
Wersja 4.7 przeglądarki Netscape Navigator nie obsługuje elementu `FIELDSET`.

Element HTML: `<FIELDSET> ... </FIELDSET>`
Atrybuty: brak

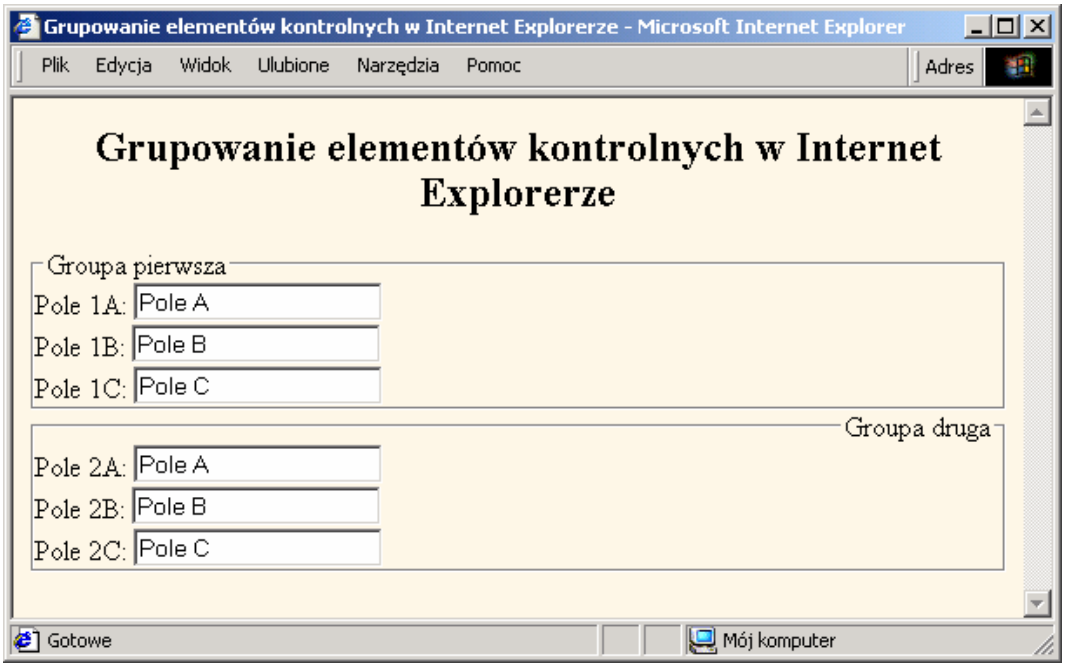
Ten element jest stosowany jako swoisty „pojemnik”, w którym można umieszczać elementy kontrolne formularzy oraz, ewentualnie, element `LEGEND`. Nie posiada on żadnych atrybutów, za wyjątkiem tych, które można stosować we wszystkich znacznikach HTML, czyli `STYLE`, `LANGUAGE`, itd. Kod strony wykorzystującej te elementy przedstawiłem na listingu 16.7, a jej wygląd w przeglądarce Internet Explorer — na rysunku 16.25.

Listing 16.7 Fieldset.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Grupowanie elementów kontrolnych w Internet Explorerze</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H2 ALIGN="CENTER">Grupowanie elementów kontrolnych w Internet Explorerze</H2>

<FORM ACTION="http://localhost:8088/SomeProgram">

<FIELDSET>
<LEGEND>Groupa pierwsza</LEGEND>
Pole 1A: <INPUT TYPE="TEXT" NAME="pole1A" VALUE="Pole A"><BR>
Pole 1B: <INPUT TYPE="TEXT" NAME="pole1B" VALUE="Pole B"><BR>
```

[13] ang.: ukryty

```
Pole 1C: <INPUT TYPE="TEXT" NAME="pole1C" VALUE="Pole C"><BR>
</FIELDSET>

<FIELDSET>
<LEGEND ALIGN="RIGHT">Groupa druga</LEGEND>
Pole 2A: <INPUT TYPE="TEXT" NAME="pole2A" VALUE="Pole A"><BR>
Pole 2B: <INPUT TYPE="TEXT" NAME="pole2B" VALUE="Pole B"><BR>
Pole 2C: <INPUT TYPE="TEXT" NAME="pole2C" VALUE="Pole C"><BR>
</FIELDSET>

</FORM>

</BODY>
</HTML>
```



Rysunek 16.25 Element FIELDSET pozwala na wizualne grupowanie elementów kontrolnych formularzy

Element HTML: `<LEGEND> ... </LEGEND>`
Atrybuty: `ALIGN`

Element `LEGEND` można umieszczać wyłącznie wewnątrz elementów `FIELDSET`, służy on do określania etykiety wyświetlanej na ramce, która jest rysowana wokół grupy elementów kontrolnych.

ALIGN

Ten atrybut określa położenie etykiety. Wartości, które może on przybierać to: `TOP`, `BOTTOM`, `LEFT` oraz `RIGTH`, przy czym wartością domyślną jest `TOP`. Na rysunku 16.25 etykieta pierwszej grupy elementów kontrolnych została wyświetlona w domyślnym położeniu, natomiast druga została przesunięta na prawo przy użyciu atrybutu `ALIGN="RIGHT"`. Język HTML udostępnia także inny sposób określania położenia etykiet, który często jest lepszy od atrybutu `ALIGN` — są nim arkusze stylów. Dzięki nim można w jednym miejscu dokumentu HTML określić opcje prezentacji, które będą dotyczyły wielu elementów strony.

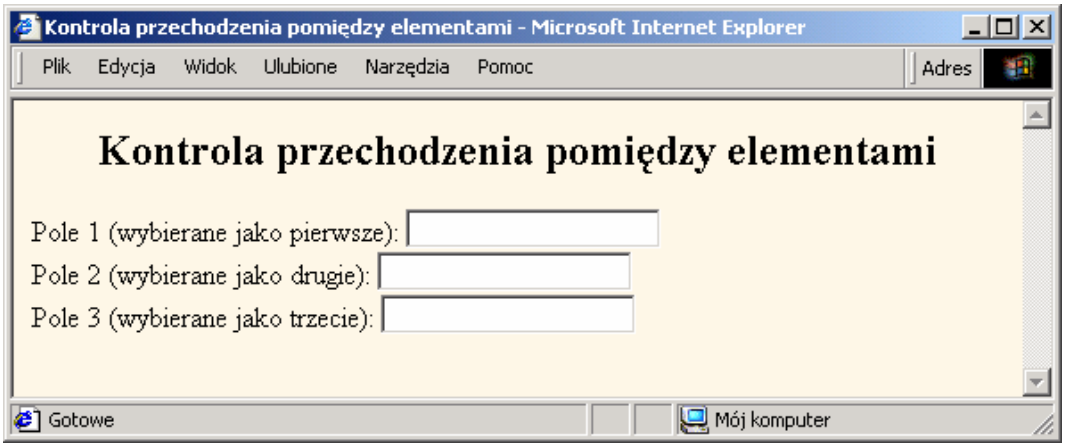
16.11 Określanie kolejności poruszania się pomiędzy elementami formularzy

Język HTML 4.0 wprowadza atrybut `TABINDEX`, którego można używać we wszystkich prezentowanych graficznie elementach HTML. Atrybut ten, którego wartością jest liczba całkowita, określa w jakiej kolejności miejsce wprowadzania będzie przenoszone pomiędzy elementami w momencie naciskania klawisza *Tab*. Niestety atrybut ten jest obsługiwany wyłącznie w przeglądarce Internet Explorer. Niemniej jednak można używać tego atrybutu także na stronach wyświetlanych we wszelkich przeglądarkach, o ile ma on jedynie ułatwiać życie użytkownikom a nie jest koniecznym elementem zapewniającym poprawne funkcjonowanie strony.

Ostrzeżenie
Przeglądarka Netscape Navigator 4.7 nie obsługuje atrybutu `TABINDEX`.

Listing 16.8 Tabindex.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Kontrola przechodzenia pomiędzy elementami</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H2 ALIGN="CENTER">Kontrola przechodzenia pomiędzy elementami</H2>

<FORM ACTION="http://localhost:8088/SomeProgram">
Pole 1 (wybierane jako pierwsze):
<INPUT TYPE="TEXT" NAME="field1" TABINDEX=1><BR>
Pole 2 (wybierane jako drugie):
<INPUT TYPE="TEXT" NAME="field2" TABINDEX=3><BR>
Pole 3 (wybierane jako trzecie):
<INPUT TYPE="TEXT" NAME="field3" TABINDEX=2><BR>
</FORM>

</BODY>
</HTML>
```



Rysunek 16.26 W przeglądarce Internet Explorer ciągłe naciskanie klawisza *Tab* powoduje cykliczne przenoszenie miejsca wprowadzania pomiędzy pierwszym, trzecim oraz drugim polem tekstowym (dokładnie w podanej kolejności, określonej przy użyciu atrybutu `TABINDEX`). W przeglądarce Netscape Navigator miejsce wprowadzania będzie przenoszone cyklicznie pomiędzy pierwszym, drugim i trzecim polem formularza, w takiej właśnie kolejności, określonej na podstawie położenia pól w dokumencie HTML.

16.12 Testowy serwer WWW

W tej części rozdziału przedstawię miniaturowy serwer WWW, bardzo przydatny w sytuacjach, gdy chcesz zrozumieć zachowanie formularzy HTML. Wykorzystywałem go w kilku przykładach przedstawionych we wcześniejszych częściach tego rozdziału. Serwer ten po prostu odczytuje wszystkie informacje przesłane w żądaniu i zwraca stronę WWW, na której informacje te zostały wyświetlone wewnątrz elementu `PRE`. Serwer ten jest także bardzo przydatny przy testowaniu serwletów. Gdy coś działa niezgodnie z oczekiwaniami, pierwszą rzeczą jaką należy zrobić jest określenie czy problem leży w sposobie zbierania danych czy też w sposobie ich przetwarzania. Uruchomienie programu *EchoServer* na lokalnym komputerze, dajmy na to na porcie 8088 i przesłanie danych z formularza pod adres *http://localhost:8088/* pozwoli sprawdzić czy zgromadzone informacje są przesyłane w oczekiwanej postaci.

EchoServer

Listing 16.9 przedstawia kod źródłowy głównej klasy programu *EchoServer*. Zazwyczaj program będzie uruchamiany z poziomu wiersza poleceń systemu, przy czym należy jawnie podać numer portu, na którym program ma działać, bądź użyć domyślnego portu 8088. Serwer może odczytywać powtarzane żądania przesyłane przez klientów i zwracać strony WWW zawierające wszystkie informacje przesłane w żądaniu HTTP. W większości przypadków serwer odczytuje informacje do momentu napotkania pustego wiersza, który oznacza koniec żądań `GET`, `POST` oraz większości innych typów żądań HTTP. Jednak w przypadku żądań `POST`, serwer określa wartość nagłówka żądania `Content-Length`, a następnie odczytuje określoną w ten sposób ilość bajtów po pustym wierszu.

Listing 16.9 EchoServer.java

```
import java.net.*;
import java.io.*;
import java.util.StringTokenizer;

/** Prosty serwer HTTP generujący stronę WWW
 *  przedstawiającą wszystkie informacje przesłane
 *  z klienta (zazwyczaj przeglądarki) w żądaniu HTTP.
 *  Aby użyć programu należy go uruchomić w wybranym
 *  komputerze, podając numer portu na którym
 *  ma działać (jeśli nie chcesz by działał na domyślnym
 *  porcie o numerze 8088). Następnie, na tym samym lub
 *  innym komputerze, uruchom przeglądarkę WWW
 *  i odwołaj się do adresu http://komputer:8088/jakasStrona.
 *  Wyświetlona, wynikowa strona będzie prezentować informacje
 *  przesłane przez przeglądarkę w żądaniu HTTP. W przypadku
 *  testowania serwletów lub programów CGI, należy podać
 *  adres http://komputer:8088/jakisProgram w atrybucie
 *  ACTION formularza. Można przesyłać dane zarówno metodą
 *  GET jak i POST; niezależnie od użytej metody
 *  wyniki będą przedstawiać wszystkie informacje
 *  przesłane przez przeglądarkę.
 */

public class EchoServer extends NetworkServer {
  protected int maxRequestLines = 50;
  protected String serverName = "EchoServer";

  /** Podaj numer porty jak argument wywołania programu.
   *  Jeśli numer portu nie zostanie podany, program
   *  użyje domyślnego portu o numerze 8088.
   */

  public static void main(String[] args) {
    int port = 8088;
    if (args.length > 0) {
      try {
        port = Integer.parseInt(args[0]);
      } catch(NumberFormatException nfe) {}
```

```
  }
  new EchoServer(port, 0);
}

public EchoServer(int port, int maxConnections) {
  super(port, maxConnections);
  listen();
}

/** Przesłania metodę handleConnection klasy
 *  NetworkServer. Metoda odczytuje każdy przesłany
 *  wiersz informacji, zapisuje go w tablicy łańcuchów
 *  znaków, a następnie umieszcza na stronie WWW
 *  wewnątrz elementu PRE. Tak stworzona strona WWW
 *  jest przesyłana do przeglądarki.
 */

public void handleConnection(Socket server)
    throws IOException{
  System.out.println
      (serverName + ": otrzymano połączenie z " +
       server.getInetAddress().getHostName());
  BufferedReader in = SocketUtil.getReader(server);
  PrintWriter out = SocketUtil.getWriter(server);
  String[] inputLines = new String[maxRequestLines];
  int i;
  for (i=0; i<maxRequestLines; i++) {
    inputLines[i] = in.readLine();
    if (inputLines[i] == null) // Klient zamknął połączenie
      break;
    if (inputLines[i].length() == 0) { // Pusty wiersz
      if (usingPost(inputLines)) {
        readPostData(inputLines, i, in);
        i = i + 2;
      }
      break;
    }
  }
  printHeader(out);
  for (int j=0; j<i; j++) {
    out.println(inputLines[j]);
  }
  printTrailer(out);
  server.close();
}

// Przesyła standardową odpowiedź HTTP i początek standardowej
// strony WWW. Dla uzyskania zgodności ze wszystkimi klientami
// wykorzystywany jest protokół HTTP 1.0.

private void printHeader(PrintWriter out) {
  out.println
    ("HTTP/1.0 200 OK\r\n" +
     "Server: " + serverName + "\r\n" +
     "Content-Type: text/html; encoding=ISO-8859-2\r\n" +
     "\r\n" +
     "<!DOCTYPE HTML PUBLIC " +
       "\"-//W3C//DTD HTML 4.0 Transitional//EN\">\n" +
     "<HTML>\n" +
     "<HEAD>\n" +
     "  <TITLE>" + serverName + " Wyniki</TITLE>\n" +
     "</HEAD>\n" +
     "\n" +
     "<BODY BGCOLOR=\"#FDF5E6\">\n" +
     "<H1 ALIGN=\"CENTER\">" + serverName +
       " Wyniki</H1>\n" +
     "Poniżej przedstawiono wiersz żądania oraz nagłowki żądania HTTP\n" +
     "przesłane przez Twoją przeglądarkę:\n" +
     "<PRE>");
}

// Generuje koniec standardowej strony WWW.

private void printTrailer(PrintWriter out) {
  out.println
    ("</PRE>\n" +
     "</BODY>\n" +
```

```
        "</HTML>\n");
  }

  // Normalne żądania dotyczące stron WWW wykorzystują
  // metodę GET, a zatem ten serwer może odczytywać
  // przesyłane informacje kolejno, po jednym wierszu.
  // Jednak formularze HTML mogą także używać metody POST.
  // W takim przypadku należy określić ilość przesyłanych
  // bajtów, aby było wiadomo ile dodatkowych bajtów
  // informacji należy odczytać po zakończeniu pobierania
  // nagłówków żądania HTTP.

  private boolean usingPost(String[] inputs) {
    return(inputs[0].toUpperCase().startsWith("POST"));
  }

  private void readPostData(String[] inputs, int i,
                            BufferedReader in)
      throws IOException {
    int contentLength = contentLength(inputs);
    char[] postData = new char[contentLength];
    in.read(postData, 0, contentLength);
    inputs[++i] = new String(postData, 0, contentLength);
  }

  // Dysponując wierszem rozpoczynającym się od łańcucha znaków
  // Content-Length, metoda zwraca zapisaną w tym wierszu
  // liczbę całkowitą.

  private int contentLength(String[] inputs) {
    String input;
    for (int i=0; i<inputs.length; i++) {
      if (inputs[i].length() == 0)
        break;
      input = inputs[i].toUpperCase();
      if (input.startsWith("CONTENT-LENGTH"))
        return(getLength(input));
    }
    return(0);
  }

  private int getLength(String length) {
    StringTokenizer tok = new StringTokenizer(length);
    tok.nextToken();
    return(Integer.parseInt(tok.nextToken()));
  }
}
```

ThreadedEchoServer

Listing 16.10 przedstawia wielowątkową wersję serwera *EchoServer*, przydatną w sytuacjach gdy serwer musi przyjmować i obsługiwać wiele, jednocześnie nadsyłanych żądań HTTP.

Listing 16.10 ThreadedEchoServer.java

```
import java.net.*;
import java.io.*;

/** Wielowątkowa wersja serwera EchoServer.
 */

public class ThreadedEchoServer extends EchoServer
                                 implements Runnable {
  public static void main(String[] args) {
    int port = 8088;
    if (args.length > 0) {
      try {
        port = Integer.parseInt(args[0]);
      } catch(NumberFormatException nfe) {}
    }
    ThreadedEchoServer echoServer =
      new ThreadedEchoServer(port, 0);
```

```
    echoServer.serverName = "Wielowątkowa wersja serwera EchoServer";
  }

  public ThreadedEchoServer(int port, int connections) {
    super(port, connections);
  }

  /** Nowa wersja metody handleConnection uruchamia
   *  nowy wątek. Ten nowy wątek odwoła się z powrotem
   *  do <i>starej</i> wersji metody handleConnection,
   *  przez co serwer będzie działać tak samo, tylko w
   *  sposób wielowątkowy. Wątek przechowuje egzemplarz
   *  obiektu klasy Socket, gdyż metoda run nie pobiera
   *  żadnych argumentów, oraz ponieważ przechowanie
   *  go w zmiennej instancyjnej może grozić utratą
   *  tego obiektu w przypadku gdyby inny wątek
   *  został uruchomiony zanim metoda run będzie miała
   *  okazję skopiować odwołanie do gniazda.
   */

  public void handleConnection(Socket server) {
    Connection connectionThread = new Connection(this, server);
    connectionThread.start();
  }

  public void run() {
    Connection currentThread =
      (Connection)Thread.currentThread();
    try {
      super.handleConnection(currentThread.serverSocket);
    } catch(IOException ioe) {
      System.out.println("IOException: " + ioe);
      ioe.printStackTrace();
    }
  }
}

/** To jest wyłącznie obiekt klasy Thread dysponujący
 *  polem umożliwiającym przechowanie egzemplarza
 *  obiektu klasy Socket. Obiekty tej klasy są używane
 *  do bezpiecznego przekazywania obiektów Socket z
 *  metody handleConnection do metody run.
 */

class Connection extends Thread {
  protected Socket serverSocket;

  public Connection(Runnable serverObject,
                    Socket serverSocket) {
    super(serverObject);
    this.serverSocket = serverSocket;
  }
}
```

NetworkServer

Listingi 16.11 oraz 16.12 przedstawiają kilka klas pomocniczych, ułatwiających komunikację sieciową. Klasy te wykorzystuje program *EchoServer*.

Listing 16.11 NetworkServer.java

```
import java.net.*;
import java.io.*;

/** Klasa bazowa używana przy tworzeniu serwerów sieciowych.
 *  Należy przesłonić metodę handleConnection, jednak w wielu
 *  przypadkach metoda listen może pozostać w niezmienionej
 *  postaci. Klasa NetworkServer używa klasy SocketUtil
 *  aby uprościć sobie zadanie tworzenia egzemplarzy obiektów
 *  PrintWriter oraz BufferedReader.
 *  <P>
 */
```

```
public class NetworkServer {
  private int port, maxConnections;

  /** Tworzy serwer pracujący na podanym porcie. Serwer będzie
   *  przyjmował połączenia, przekazując każde z nich do
   *  metody handleConnection, aż do momentu otrzymania
   *  jawnego polecenia przerwania pracy (na przykład:
   *  System.exit) lub przekroczenia ilości dopuszczalnych
   *  połączeń. Jeśli chcesz, aby serwer działał w nieskończoność
   *  to jako maxConnections podaj wartość 0.
   */

  public NetworkServer(int port, int maxConnections) {
    setPort(port);
    setMaxConnections(maxConnections);
  }

  /** Monitoruje port na który będą przesyłane prośby
   *  o połączenie z serwerem. Za każdym razem gdy
   *  połączenie zostanie nawiązane, uzyskany egzemplarz
   *  obiektu klasy Socket jest przekazywany do metody
   *  handleConnection.
   */

  public void listen() {
    int i=0;
    try {
      ServerSocket listener = new ServerSocket(port);
      Socket server;
      while((i++ < maxConnections) || (maxConnections == 0)) {
        server = listener.accept();
        handleConnection(server);
      }
    } catch (IOException ioe) {
      System.out.println("IOException: " + ioe);
      ioe.printStackTrace();
    }
  }

  /** To jest metoda definiująca sposób działania
   *  serwera, gdyż określa ona co się dzieje z wynikowym
   *  gniazdem (egzemplarzem obiektu klasy Socket).
   *  <B>W serwerach, które będziesz pisać, powinieneś
   *  przesłonić tę metodę</B>.
   *  <P>
   *  Ta ogólna wersja metody określa komputer, który
   *  nadesłał żądanie, wyświetla pierwszy wiersz żądania
   *  nadesłany przez klienta i generuje pierwszy wiersz
   *  odpowiedzi HTTP.
   */

  protected void handleConnection(Socket server)
      throws IOException{
    BufferedReader in = SocketUtil.getReader(server);
    PrintWriter out = SocketUtil.getWriter(server);
    System.out.println
      ("Ogólny serwer sieciowy: odebrano połączenie z " +
       server.getInetAddress().getHostName() + "\n" +
       "pierwszy wiersz żądania '" + in.readLine() + "'");
    out.println("Ogólny serwer sieciowy");
    server.close();
  }

  /** Zwraca maksymalną ilość połączeń, jaka zostanie
   *  obsłużona zanim serwer przestanie działać.
   *  Wartość 0 oznacza, że serwer powinien działać aż
   *  do momentu gdy jawnie zostanie zamknięty.
   */

  public int getMaxConnections() {
    return(maxConnections);
  }

  /** Określa maksymalną ilość połączeń. Wartość 0 oznacza
   *  że serwer powinien działać w nieskończoność (aż do
   *  momentu gdy zostanie jawnie zamknięty).
   */
```

```
  public void setMaxConnections(int maxConnections) {
    this.maxConnections = maxConnections;
  }

  /** Zwraca numer portu, na którym działa serwer. */

  public int getPort() {
    return(port);
  }

  /** Określa numer portu. <B>Port można określić wyłącznie
   *  przed wywołaniem metody "connect"</B>. Zazwyczaj port
   *  jest określany w konstruktorze.
   */

  protected void setPort(int port) {
    this.port = port;
  }
}
```

Listing 16.12 SocketUtil.java

```
import java.net.*;
import java.io.*;

/** Uproszczony sposób tworzenia egzemplarzy obiektów
 *  klas PrintWriter oraz BufferedReader skojarzonych
 *  z gniazdem (egzemplarzem obiektu klasy Socket).
 */

public class SocketUtil {
  /** Tworzy BufferedReader pobierający nadsyłane informacje. */

  public static BufferedReader getReader(Socket s)
      throws IOException {
    return(new BufferedReader(
       new InputStreamReader(s.getInputStream())));
  }

  /** Tworzy PrintWriter wysyłający wyjściowe informacje.
   *  Ten obiekt będzie automatycznie opróżniał strumień
   *  wyjściowy w momencie wywołania metody println.
   */

  public static PrintWriter getWriter(Socket s)
      throws IOException {
    // drugi argument o wartości true oznacza, że należy
    // stosować automatyczne opróżnianie strumienia wyjściowego.
    return(new PrintWriter(s.getOutputStream(), true));
  }
}
```

Rozdział 17.
Użycie apletów jako interfejsu użytkownika dla serwletów

Formularze HTML przedstawione w rozdziale 16. stanowią prostą lecz nieco ograniczoną metodę pobierania informacji od użytkowników i przesyłania ich do serwletów bądź programów CGI. Od czasu do czasu może się jednak zdarzyć, że konieczne będzie zastosowanie bardziej złożonego interfejsu użytkownika. Aplety dają znacznie większą kontrolę nad wielkością, kolorami oraz czcionką używaną w elementach graficznego interfejsu użytkownika, udostępniają także więcej elementów kontrolnych (suwaki, możliwość rysowania linii, wyświetlania okien, itp.), dają możliwość śledzenia czynności wykonywanych przy użyciu myszy i klawiatury, pozwalają na tworzenie własnych elementów kontrolnych (tarcz zegarowych, termometrów, ikon które można przeciągać, itp.), a co więcej, pozwalają przesyłać te same informacje podane przez użytkownika do wielu programów działających na serwerze. Te nowe możliwości wiążą się jednak z większymi kosztami, gdyż zaprojektowanie i stworzenie interfejsu użytkownika w języku Java wymaga znacznie więcej wysiłku niż stworzenie formularza HTML, zwłaszcza jeśli interfejs ten zawiera wiele, odpowiednio sformatowanego tekstu. A zatem wybór pomiędzy zastosowaniem formularzy HTML bądź apletów, będzie zależał od tworzonej aplikacji.

W przypadku formularzy HTML, żądania `POST` i `GET` są obsługiwane niemal identycznie — wszystkie elementy kontrolne służące do wprowadzania danych są takie same, a zmienia się wyłącznie wartość atrybutu `METHOD` elementu `FORM`. Jednak w przypadku apletów, proces przesyłania danych i obsługi wyników można realizować na trzy różne sposoby. Pierwszy z nich, przedstawiony w podrozdziale 17.1, polega na tym, iż aplet imituje działanie formularza używającego metody `GET` — czyli aplet przesyła dane a w przeglądarce jest wyświetlana wynikowa strona WWW. Przykład takiego rozwiązania został przedstawiony w podrozdziale 17.2. — „Narzędzie korzystające z wielu serwisów wyszukiwawczych”. W drugiej metodzie, przedstawionej w podrozdziale 17.3, aplet przesyła żądanie `GET` do serwletu i samemu przetwarza otrzymane wyniki. Przykład wykorzystania tej metody przedstawiłem w podrozdziale 17.4. — „Przeglądarka zapytań wykorzystująca serializację obiektów i tunelowanie HTTP”. Trzeci sposób, przedstawiony w podrozdziale 17.5, polega na tym, iż aplet przesyła do serwletu żądanie `POST`, a następnie samemu przetwarza otrzymane wyniki. Przykład wykorzystania tego sposobu przedstawiłem w podrozdziale 17.6. — „Aplet przesyłający dane metodą POST”. W końcu, w podrozdziale 17.7, pokażę, że aplet może w ogóle pominąć serwer HTTP i nawiązać bezpośrednią komunikację z programem działającym jako serwer, uruchamianym na tym samym komputerze na którym działa aplet.

Omawiając zagadnienia przedstawione w tym rozdziale zakładam, że Czytelnik dysponuje już podstawową wiedzą na temat apletów i koncentruję uwagę na technikach komunikacji z programami działającymi na serwerze. Czytelnicy, którzy nie znają zasad tworzenia apletów, powinni sięgnąć po ogólną książkę poświęconą językowi Java; taką jak *Java 2 dla każdego* wydaną przez Wydawnictwo HELION.

17.1 Przesyłanie danych metodą GET i wyświetlanie wynikowej strony WWW

Metoda `showDocument`[14] informuje przeglądarkę, że należy wyświetlić zasób o podanym adresie URL. Przypomnij sobie, że korzystając z metody `GET` można przesłać dane do serwletu lub programu CGI poprzedzając je znakiem zapytania (`?`) i dopisując na końcu adresu URL danego programu. A zatem, aby przesłać w ten sposób dane z apletu, należy dopisać je do łańcucha znaków określającego adres URL, następnie stworzyć kopię obiektu `URL` i, w zwyczajny sposób, wywołać metodę `showDocument`. Poniższy prosty przykład przedstawia czynności jakie należy wykonać, aby z poziomu apletu zażądać wyświetlenia w przeglądarce konkretnego zasobu. W przykładzie zakładam, iż `bazowyURL` to łańcuch znaków zawierający adres URL programu działającego na serwerze, a `dane` to informacje jakie chcemy przesłać w żądaniu.

```
try {
  URL programURL = new URL(bazowyURL + "?" + dane);
  getAppletContext().showDocument(programURL);
} catch (MalformedURLException mue) { ... }
```

Jednak gdy przeglądarka przesyła dane, są one *zakodowane w formacie URL*, co oznacza, że odstępy są zamieniane na znaki plusa (`+`), a wszystkie pozostałe znaki, za wyjątkiem liter i cyfr, na kombinacje znaku procenta (`%`) oraz dwucyfrowej liczby szesnastkowej określającej wartość danego znaku (więcej informacji na ten temat znajdziesz w podrozdziale 16.2. — „Element FORM"). Powyższy przykład zakłada, że dane zostały już poprawnie zakodowane, jeśli jednak nie zostały, to przedstawiony kod nie będzie działać poprawnie. JDK 1.1 udostępnia klasę `URLEncoder` definiującą statyczną metodę `encode`, która zapisuje podany łańcuch znaków w formacie URL. A zatem, jeśli aplet kontaktuje się z programem działającym na serwerze, który zazwyczaj otrzymuje dane przekazywane z formularzy HTML metodę `GET`, to będzie on musiał zakodować wartości wszystkich pól, za wyjątkiem znaków równości (`=`) oddzielającym nazwy pól od ich wartości, oraz znaków "`&`" oddzielających poszczególne pary nazwa-wartość. Oznacza to, że nie można zakodować wszystkich przesyłanych informacji przy użyciu jednego wywołania o postaci `URLEncoder.encode(dane)`, lecz należy zakodować wyłącznie wartości każdej z par nazwa-wartość. Czynności te można wykonać w następujący sposób:

```
String dane =
  nazwa1 + "=" + URLEncoder.encode(wartosc1) + "&" +
  nazwa2 + "=" + URLEncoder.encode(wartosc2) + "&" +
  ...
  nazwaN + "=" + URLEncoder.encode(wartoscN);
try {
  URL programURL = new URL(bazowyURL + "?" + dane);
  getAppletContext().showDocument(programURL);
} catch (MalformedURLException mue) { ... }
```

W następnym podrozdziale przedstawiłem pełny przykład prezentujący ten sposób przesyłania danych i prezentacji wyników.

17.2 Narzędzie korzystające z wielu serwisów wyszukiwawczych

W podrozdziale 6.3. (pt.: „Interfejs użytkownika obsługujący różne serwisy wyszukiwawcze") przedstawiłem klasę `SearchSpec` (patrz listing 6.2) używaną przez serwlet do generacji ściśle określonych adresów URL, koniecznych do przekierowania żądań do różnych

[14] Autor ma tu na myśli metodę `showDocument` interfejsu `AppletContext`. Obiekt `AppletContext` dla danego apletu można pobrać przy użyciu metody `getAppletContext` klasy `Applet`. (przyp. tłum.)

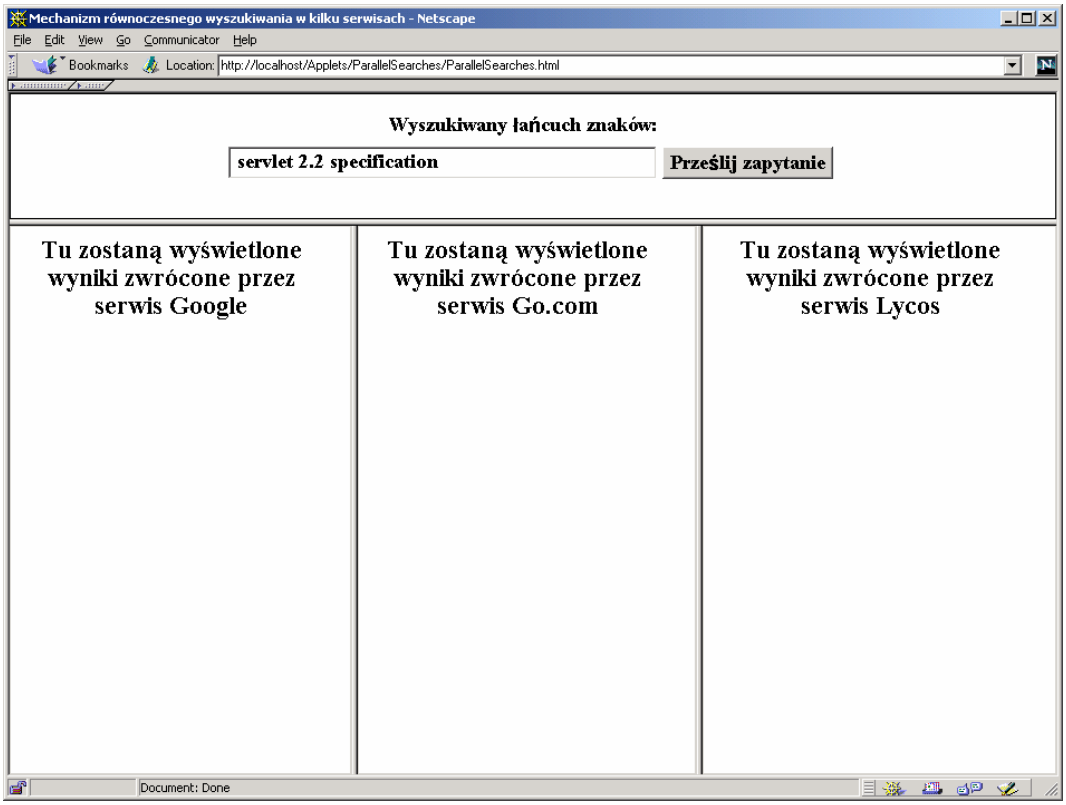
serwisów wyszukiwawczych. Klasy tej można także użyć przy tworzeniu apletów. Listing 17.1 przedstawia aplet wyświetlający pole tekstowe służące do pobierania informacji od użytkowników. Kiedy użytkownik zażąda przesłania danych, aplet koduje zawartość pola tekstowego w formacie URL a następnie generuje trzy różne adresy URL i dołącza do nich zakodowane informacje. Wygenerowane adresy URL odwołują się do trzech serwisów wyszukiwawczych — Google, Go.com[15] oraz Lycos. Następnie aplet używa metody `showDocument`, aby nakazać przeglądarce wyświetlenie tych trzech adresów URL w trzech różnych ramkach. Wygląd apletu oraz wyniki jego działania przedstawiłem na rysunkach 17.1 oraz 17.2. Przy tworzeniu takiej aplikacji nie można wykorzystać formularzy HTML, gdyż umożliwiają one przesłanie danych tylko pod jeden adresu URL.

Listing 17.1 SearchApplet.java

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import coreservlets.SearchSpec;

/** Aplet odczytuje wartość z pola TextField,
 *  a następnie używa jej do stworzenia trzech różnych
 *  adresów URL zawierających w sobie dane podane w
 *  formularzu. Adresy te odwołują się do mechanizmów
 *  wyszukiwawczych Google, Go.com, and Lycos.
 *  Przeglądarka pobiera zasoby o podanych adresach URL
 *  i wyświetla je w trzech umieszczonych obok siebie
 *  ramkach. Zwróć uwagę iż zwyczajne formularze HTML
 *  nie są w stanie wykonać takiego zadania, gdyż
 *  nie dysponują możliwością przesyłania kilku żądań
 *  jednocześnie.
 */

public class SearchApplet extends Applet
                          implements ActionListener {
  private TextField queryField;
  private Button submitButton;

  public void init() {
    setFont(new Font("Serif", Font.BOLD, 18));
    add(new Label("Wyszukiwany łańcuch znaków:"));
    queryField = new TextField(40);
    queryField.addActionListener(this);
    add(queryField);
    submitButton = new Button("Prześlij zapytanie");
    submitButton.addActionListener(this);
    add(submitButton);
  }

  /** Wyślij dane gdy zostanie kliknięty przycisk <B>lub</B>
   *  użytkownik naciśnie klawisz Enter w polu TextField.
   */

  public void actionPerformed(ActionEvent event) {
    String query = URLEncoder.encode(queryField.getText());
    SearchSpec[] commonSpecs = SearchSpec.getCommonSpecs();
    // Pomiń HotBot (ostatni wpis), gdyż ta wyszukiwarka używa
    // JavaScriptu do wyświetlenia wyników w ramce najwyższego poziomu.
    // Z tego względu poniżej używam wyrażenia length-1 .
    for(int i=0; i<commonSpecs.length-1; i++) {
      try {
        SearchSpec spec = commonSpecs[i];
        // Klasa SearchSpec tworzy adresy URL o postaci używanej
        // przez kilka popularnych mechanizmów wyszukiwawczych.
        URL searchURL = new URL(spec.makeURL(query, "10"));
        String frameName = "results" + i;
        getAppletContext().showDocument(searchURL, frameName);
      } catch(MalformedURLException mue) {}
    }
  }
```

[15] dawniej Infoseek

```
}
```



Rysunek 17.1 Aplet SearchApplet pozwala użytkownikom na podawanie wyszukiwanego wyrażenia

Rysunek 17.2 Przesłanie zapytania powoduje wyświetlenie obok siebie wyników zwróconych przez trzy różne mechanizmy wyszukiwawcze

Listing 17.2 przedstawia główny dokument HTML używany w omawianym przykładzie, natomiast listing 17.3 — kod źródłowy dokumentu HTML zawierającego aplet. Kody źródłowe trzech niewielkich dokumentów HTML wyświetlanych początkowo trzech dolnych ramkach układu (rysunek 17.1) znajdziesz w pliku archiwalnym zawierającym kody wszystkich przykładów przedstawionych w niniejszej książce; plik ten znajdziesz pod adresem *ftp://ftp.helion.pl/przyklady/jsjsp.zip*.

Listing 17.2 ParallelSearch.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Frameset//EN">
<HTML>
<HEAD>
  <TITLE>Mechanizm równoczesnego wyszukiwania w kilku serwisach</TITLE>
  <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=ISO-8859-2">
</HEAD>

<FRAMESET ROWS="120,*">
  <FRAME SRC="SearchAppletFrame.html" SCROLLING="NO">
  <FRAMESET COLS="*,*,*">
    <FRAME SRC="GoogleResultsFrame.html" NAME="results0">
    <FRAME SRC="InfoseekResultsFrame.html" NAME="results1">
    <FRAME SRC="LycosResultsFrame.html" NAME="results2">
  </FRAMESET>
</FRAMESET>
```

Listing 17.3 SearchAppletFrame.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Aplet obsługujący wyszukiwanie</TITLE>
  <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=ISO-8859-2">
</HEAD>

<BODY BGCOLOR="WHITE">
<CENTER>
<APPLET CODE="SearchApplet.class" WIDTH=600 HEIGHT=100>
  <B>Ten przykład wymaga przeglądarki obsługującej język Java.</B>
</APPLET>
</CENTER>
</BODY>
</HTML>
```

17.3 Przesyłanie danych metodą GET i bezpośrednie przetwarzanie wyników (tunelowanie HTTP)

W poprzednim przykładzie aplet zażądał od przeglądarki wyświetlenia wyników zwróconych przez program działający na serwerze, w konkretnej ramce. Wykorzystanie przeglądarki do wyświetlania wyników jest rozsądnym rozwiązaniem w przypadku korzystania z już istniejących usług. Wynika to z faktu, że większość programów CGI zwraca dokumenty HTML. Niemniej jednak, jeśli tworzysz zarówno klienta jak i serwer obsługujący jakiś proces, przesyłanie za każdym razem całego dokumentu HTML wydaje się nieoptymalnym rozwiązaniem. W niektórych przypadkach znacznie lepszym wyjściem z sytuacji byłoby przekazanie informacji do już działającego apletu, który mógłby je następnie przedstawić w formie grafu lub w jakikolwiek inny sposób. Takie rozwiązanie jest czasami określane jako *tunelowanie HTTP*, gdyż własny protokół komunikacyjny jest realizowany przy wykorzystaniu protokołu HTTP; w ten sposób działają serwery pośredniczące, szyfrowanie, przekierowania do innych serwerów, nawiązywanie połączeń przez zapory ogniowe, itd.

Tę metodę można implementować na dwa podstawowe sposoby. W obu, do utworzenia strumienia wejściowego i pobierania danych spod podanego adresu URL jest wykorzystywana klasa `URLConnection`. Oba sposoby różnią się od siebie typem używanego strumienia wejściowego. Pierwszy z nich wykorzystuje strumień `BufferedInputStream` bądź inny strumień niskiego poziomu, który pozwala na odczytywanie danych binarnych lub informacji zapisanych w kodzie ASCII, przesyłanych z dowolnego programu działającego na serwerze. Drugi sposób polega na wykorzystaniu strumienia `ObjectInputStream`, pozwalającego na bezpośrednie odczytywanie złożonych struktur danych. Ten drugi sposób przedstawiłem w drugim podrozdziale. Należy zauważyć, iż można go stosować wyłącznie w sytuacjach, gdy także program działający na serwerze został napisany w języku Java.

Odczyt danych binarnych lub danych ASCII

Aby aplet mógł odczytywać dane przesyłane przez serwlet, należy w pierwszej kolejności utworzyć kopię obiektu `URLConnection` bazując na podanym adresie URL programu działającego na serwerze, a następnie dołączyć do niego strumień `BufferedInputStream`. Poniżej opisałem siedem podstawowych czynności jakie należy wykonać, aby zaimplementować w kliencie tę metodę pobierania wyników. Prezentując tę metodę pomijam kod programu uruchamianego na serwerze, gdyż utworzony w ten sposób klient może współpracować zarówno z dowolnym programem tego typu, jak i ze statycznymi dokumentami HTML.

Należy zwrócić uwagę, iż wiele operacji wykonywanych na strumieniach zgłasza wyjątek `IOException`, a zatem czynności prezentowane poniżej muszą być umieszczone wewnątrz bloku `try/catch`.

1. **Utwórz kopię obiektu `URL` odwołującą się do komputera, z którego został pobrany aplet**. Do konstruktora klasy `URL` można przekazać bezwzględny adres URL (na przykład: *http://komputer/sciezka*), jednak ze względu na mechanizmy zabezpieczeń zezwalający apletom na nawiązywanie połączeń wyłącznie z komputerem z którego zostały one pobrane; najbardziej sensownym rozwiązaniem jest określenie adresu URL na podstawie nazwy komputera z którego aplet został pobrany.
```
URL aktualnaStrona = getCodeBase();
String protokol = aktualnaStrona.getProtocol();
String host = aktualnaStrona.getHost();
int port = aktualnaStrona.getPort();
String urlSuffix = "/servlet/jakisSerwlet";
URL daneURL = new URL(protokol, host, port, urlSuffix);
```

2. **Stwórz kopię obiektu `URLConnection`**. Obiekt ten zwraca metoda `openConnection` klasy `URL`, a użyjemy go do pobrania strumieni wejściowych.
```
URLConnection polaczenie = daneURL.openConnection();
```

3. **Poinformuj przeglądarkę, iż nie należy przechowywać danych żądania w pamięci podręcznej**. Po utworzeniu obiektu `URLConnection`, pierwszą czynnością jaką należy wykonać jest poinformowanie przeglądarki, iż obiektu tego nie można przechowywać w pamięci podręcznej. W ten sposób uzyskujemy pewność, że za każdym razem uzyskamy aktualne informacje.
```
polaczenie.setUseCache(false);
```

4. **Określ wszelkie dodatkowe nagłówki żądania HTTP, które chcesz wygenerować**. Jeśli chcesz określić nagłówki żądania HTTP (patrz rozdział 4), to możesz to zrobić przy użyciu metody `setRequestProperty`.
```
polaczenie.setRequestProperty("naglowek", "wartosc");
```

5. **Utwórz strumień wejściowy**. Istnieje wiele strumieni, których można użyć, jednak najczęściej jest stosowany strumień `BufferedReader`. To właśnie podczas tworzenia

strumienia wejściowego, w niewidoczny sposób, jest tworzone połączenie sieciowe z serwerem WWW.

```
BufferedReader in =
  new BufferedReader(new InputStreamReader(
                       polaczenie.getInputStream()));
```

6. **Odczytaj każdy wiersz dokumentu**. Specyfikacja protokołu HTTP wymusza zamknięcie połączenia przez serwer, gdy wszystkie informacje zostaną już przesłane. Gdy połączenie zostanie zamknięte, metoda `readLine` zwraca wartość `null`. A zatem, odczytuj dane wejściowe do momentu pobrania wartości `null`.

```
String wiersz;
while ((wiersz = in.readLine()) != null) {
  zrobCosZ(wiersz);
}
```

7. **Zamknij strumień wejściowy**.

```
in.close();
```

Odczyt serializowanych struktur danych

Wykorzystanie metody przedstawionej w poprzednim rozdziale ma sens jeśli aplet odczytuje wyniki zwracane przez dowolny program działający na serwerze lub odczytuje zawartość statycznych dokumentów HTML. Jednak jeśli aplet komunikuje się z serwletem, to można zastosować lepsze rozwiązanie. Zamiast przesyłania danych binarnych bądź informacji zapisanych w kodzie ASCII, serwlet może przesyłać dowolne struktury danych — jest to możliwe dzięki wykorzystaniu mechanizmu serializacji dostępnego w języku Java. Aplet może odczytać te dane przy użyciu pojedynczego wywołania metody `readObject` — nie trzeba będzie w tym celu wykonywać żadnej długiej i uciążliwej analizy danych. Poniżej przedstawiłem czynności jakie należy wykonać by zaimplementować ten sposób komunikacji. Zwróć uwagę, iż także tym razem, w tworzonym aplecie poniższy kod będzie musiał być zapisany wewnątrz bloku `try/catch`.

Po stronie klienta

Aby aplet mógł odczytywać serializowane dane przesyłane z serwera, będzie musiał wykonywać siedem, opisanych poniżej czynności. Jedynie piąty i szósty punkt poniższej procedury różni się od czynności wykonywanych podczas odczytywania danych tekstowych (zapisanych w kodzie ASCII). Przedstawione poniżej czynności zostały nieco uproszczone poprzez pominięcie bloków `try/catch`, w jakich powinne być zapisane.

1. **Stwórz kopię obiektu `URL` odwołującą się do komputera, z którego aplet został pobrany**. Ponieważ użyty adres URL musi się odwoływać do komputera z którego aplet został pobrany, a zatem, także tym razem, najbardziej sensownym rozwiązaniem jest podanie końcówki adresu i automatyczne określenie jego pozostałych elementów.

```
URL aktualnaStrona = getCodeBase();
String protokol = aktualnaStrona.getProtocol();
String host = aktualnaStrona.getHost();
int port = aktualnaStrona.getPort();
String urlSuffix = "/servlet/jakisSerwlet";
URL daneURL = new URL(protokol, host, port, urlSuffix);
```

2. **Stwórz kopię obiektu `URLConnection`**. Obiekt ten zwraca metoda `openConnection` klasy `URL`, wykorzystamy go do pobrania strumieni wejściowych.

```
URLConnection polaczenie = daneURL.openConnection();
```

3. **Poinformuj przeglądarkę, iż nie należy przechowywać danych żądania w pamięci podręcznej**. Po utworzeniu obiektu `URLConnection`, pierwszą czynnością jaką należy wykonać jest poinformowanie przeglądarki, iż obiektu tego nie można przechowywać w

pamięci podręcznej. W ten sposób uzyskujemy pewność, że za każdym razem uzyskamy aktualne informacje.

```
polaczenie.setUseCache(false);
```

4. **Określ wszelkie dodatkowe nagłówki żądania HTTP, które chcesz wygenerować**. Jeśli chcesz określić nagłówki żądania HTTP (patrz rozdział 4), to możesz to zrobić przy użyciu metody `setRequestProperty`.

```
polaczenie.setRequestProperty("naglowek", "wartosc");
```

5. **Utwórz kopię obiektu `ObjectInputStream`**. Konstruktor tej klasy wymaga przekazania obiektu nieprzetworzonego strumienia wejściowego, który można pobrać z obiektu `URLConnection`. To właśnie podczas tworzenia strumienia wejściowego, w niewidoczny sposób, jest tworzone połączenie sieciowe z serwerem.

```
ObjectInputStream in =
  new ObjectInputStream(polaczenie.getInputStream());
```

6. **Odczytaj strukturę danych przy użyciu metody `readObject`**. Metoda ta zwraca wartość typu `Object`, a zatem będziesz musiał wykonać rzutowanie typów, aby uzyskać obiekt klasy przesłanej przez serwer.

```
JakasKlasa wartosc = (JakasKlasa) in.readObject();
zrobCosZ(wartosc);
```

7. **Zamknij strumień wejściowy**.

```
in.close();
```

Po stronie serwera

W celu przesłania serializowanych informacji do apletu, serwlet musi wykonać cztery, opisane poniżej czynności. Zakładam, że zmienne `request` oraz `response` zawierają odpowiednio obiekty `HttpServletRequest` oraz `HttpServletResponse` przekazywane jako argumenty wywołania metod `doGet` oraz `doPost`. Także w tym przypadku, prezentowane czynności zostały nieco uproszczone poprzez pominięcie bloków `try/catch`, w jakich należy je zapisać w kodzie serwletu.

1. **Określ, że przesyłane są dane binarne**. Można to zrobić podając, że typem MIME odpowiedzi będzie `application/x-java-serialized-object`. To standardowy typ MIME obiektów kodowanych przez strumień `ObjectOutputStream`, jednak w naszym przypadku nie odgrywa on szczególnego znaczenia, gdyż to aplet, a nie przeglądarka, odczytuje wyniki. Więcej informacji na temat typów MIME znajdziesz w podrozdziale 7.2. — „Nagłówki odpowiedzi protokołu HTTP 1.1 oraz ich znaczenie", w części poświęconej nagłówkowi `Content-Type`.

```
String contentType = "application/x-java-serialized-object";
response.setContentType(contentType);
```

2. **Stwórz strumień `ObjectOutpuStream`**.

```
ObjectOutputStream out =
  new ObjectOutputStream(response.getOutputStream());
```

3. **Zapisz strukturę danych przy użyciu metody `writeObject`**. W ten sposób można przesłać większość wbudowanych struktur danych. Jednak abyś mógł przesyłać *swoje własne* klasy, muszą one implementować interfejs `Serializable`. Na szczęście wymóg ten jest bardzo łatwy do spełnienia, gdyż interfejs ten nie definiuje żadnych metod. A zatem, wystarczy jedynie zadeklarować, że klasa go implementuje.

```
JakasKlasa wartosc = new JakasKlasa(...);
out.writeObject(wartosc);
```

4. **Opróżnij strumień wyjściowy, aby mieć pewność, że informacje zostały przesłane do klienta**.

```
out.flush();
```

W kolejnym podrozdziale przedstawiłem przykład wymiany danych realizowanej w powyższy sposób.

17.4 Przeglądarka zapytań wykorzystująca serializację obiektów i tunelowanie

Wiele osób ciekawi jakie typy zapytań są przesyłane do głównych mechanizmów wyszukiwawczych. Czasami jest to jedynie czysta ciekawość („Czy to prawda, że 64 procent zapytań kierowanych od serwisu AltaVista pochodzi od pracodawców szukających programistów znających technologie związane z językiem Java?"), czasami jednak nie, gdyż zdarza się, że autorzy dokumentów HTML, mając nadzieję na poprawienie notowań swych witryn, tworzą strony w taki sposób, aby odpowiadały one typom najczęściej zadawanych zapytań.

Ta część rozdziału przedstawia aplet oraz współpracujący z nim serwlet, które „na bieżąco" prezentują informacje z fikcyjnej witryny *super-search-engine.com*; a konkretnie rzecz biorąc, na specjalnej stronie WWW, wyświetlają cyklicznie aktualizowaną listę zapytań. Na listingu 17.4 przedstawiłem główny aplet, który korzystając z kilku klas pomocniczych (patrz listing 17.5) pobiera zapytania używając do tego celu wątku działającego w tle. Gdy użytkownik zainicjalizuje cały proces, aplet co pół sekundy wyświetla na przewijanej liście przykładowe zapytanie (przykładowy wygląd ten listy przedstawiłem na rysunku 17.3). W końcu listing 17.6 przedstawia serwlet uruchamiany na serwerze i generujący zapytania. Serwlet ten generuje losowe przykłady pytań zadawanych ostatnio przez użytkowników i przesyła 50 takich pytań obsługując każde żądanie klienta.

Jeśli skopiujesz kody źródłowe serwletu i apletu z serwera FTP Wydawnictwa HELION (*ftp://ftp.helion.pl/przyklady/jsjsp.zip*) i będziesz chciał samodzielnie uruchomić tę aplikację, to musisz wiedzieć, że będzie ona działać poprawnie wyłącznie jeśli jej główną stronę WWW wyświetlisz przy użyciu protokołu HTTP (czyli musisz zażądać wyświetlenia tej strony, posługując się adresem URL o postaci *http://...*). Pobranie i wyświetlenie strony bezpośrednio z dysku — przy użyciu adresu URL rozpoczynającego się od *file:* — sprawi, że aplikacja nie będzie działać, gdyż aplet nawiązując połączenie z serwletem komunikuje się z komputerem z którego został pobrany. Poza tym, metody klasy `URLConnection` nie działają poprawnie, jeśli strona zawierająca aplet nie została pobrana przy użyciu protokołu HTTP.

Listing 17.4 ShowQueries.java

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.net.*;

/** Aplet odczytuje tablice łańcuchów znaków zapisane w
 *  obiektach QueryCollection i wyświetla je w obszarze
 *  tekstowym wyposażonym w pionowy pasek przewijania.
 *  QueryCollection pobiera łańcuchy znaków za pośrednictwem
 *  strumienia przesyłającego serializowane obiekty,
 *  podłączonego do serwletu QueryGenerator.
 */

public class ShowQueries extends Applet
                         implements ActionListener, Runnable {
  private TextArea queryArea;
  private Button startButton, stopButton, clearButton;
  private QueryCollection currentQueries;
  private QueryCollection nextQueries;
  private boolean isRunning = false;
  private String address =
    "/servlet/coreservlets.QueryGenerator";
  private URL currentPage;

  public void init() {
```

```
    setBackground(Color.white);
    setLayout(new BorderLayout());
    queryArea = new TextArea();
    queryArea.setFont(new Font("Serif", Font.PLAIN, 14));
    add(queryArea, BorderLayout.CENTER);
    Panel buttonPanel = new Panel();
    Font buttonFont = new Font("SansSerif", Font.BOLD, 16);
    startButton = new Button("Start");
    startButton.setFont(buttonFont);
    startButton.addActionListener(this);
    buttonPanel.add(startButton);
    stopButton = new Button("Stop");
    stopButton.setFont(buttonFont);
    stopButton.addActionListener(this);
    buttonPanel.add(stopButton);
    clearButton = new Button("Usuń zapytania");
    clearButton.setFont(buttonFont);
    clearButton.addActionListener(this);
    buttonPanel.add(clearButton);
    add(buttonPanel, BorderLayout.SOUTH);
    currentPage = getCodeBase();
    // Zażądaj zbioru przykładowych zapytań. Zostaną
    // one pobrane przy wykorzystaniu wątku działającego w tle,
    // a przed próbą pobrania łańcuchów znaków aplet sprawdzi
    // czy pobieranie danych zostało zakończone.
    currentQueries = new QueryCollection(address, currentPage);
    nextQueries = new QueryCollection(address, currentPage);
  }

  /** Jeśli kliknąłeś przycisk "Start", system
   *  uruchomi wątek działający w tle i wyświetlający zapytania
   *  w obszarze tekstowym. Kliknięcie przycisku "Stop"
   *  zatrzymuje ten proces, a kliknięcie przycisku
   *  "Usuń zapytania" powoduje usunięcie całej zawartości
   *  obszaru tekstowego.
   */

  public void actionPerformed(ActionEvent event) {
    if (event.getSource() == startButton) {
      if (!isRunning) {
        Thread queryDisplayer = new Thread(this);
        isRunning = true;
        queryArea.setText("");
        queryDisplayer.start();
        showStatus("Wątek prezentujący zapytania uruchomiony...");
      } else {
        showStatus("Wątek prezentujący zapytania już działa...");
      }
    } else if (event.getSource() == stopButton) {
      isRunning = false;
      showStatus("Wątek prezentujący zapytania został zatrzymany...");
    } else if (event.getSource() == clearButton) {
      queryArea.setText("");
    }
  }

  /** Wątek działający w tle pobiera obiekt currentQueries
   *  i co pół sekundy wyświetla u dołu obszaru tekstowego
   *  jedno z zapytań zapisanych w tym obiekcie. Po wyświetleniu
   *  wszystkich zapytań, wątek kopiuje do obiektu
   *  currentQueries zawartość obiektu nextQueries, przesyła
   *  na serwer nowe żądanie w celu określenia nowej wartości
   *  obiektu nextQueries i powtarza cały proces.
   */

  public void run() {
    while(isRunning) {
      showQueries(currentQueries);
      currentQueries = nextQueries;
      nextQueries = new QueryCollection(address, currentPage);
    }
  }

  private void showQueries(QueryCollection queryEntry) {
    // Jesli żądanie zostało przesłane na serwer, lecz
    // wyniki jeszcze nie zostały otrzymane, to sprawdzaj
    // czy są dostępne co sekundę. Nie powinno się to
```

```
    // zdarzać często, jednak może się zdarzyć w przypadku
    // stosowania wolnych połączeń sieciowych bądź w
    // w przypadku przeciążenia serwera.
    while(!queryEntry.isDone()) {
      showStatus("Oczekiwanie na dane z serwera...");
      pause(1);
    }
    showStatus("Pobieranie danych z serwera...");
    String[] queries = queryEntry.getQueries();
    String linefeed = "\n";
    // umieszczaj zapytania w obszarze tekstowym co pół sekundy.
    for(int i=0; i<queries.length; i++) {
      if (!isRunning) {
        return;
      }
      queryArea.append(queries[i]);
      queryArea.append(linefeed);
      pause(0.5);
    }
  }

  public void pause(double seconds) {
    try {
      Thread.sleep((long)(seconds*1000));
    } catch(InterruptedException ie) {}
  }
}
```

Listing 17.5 QueryCollection.java

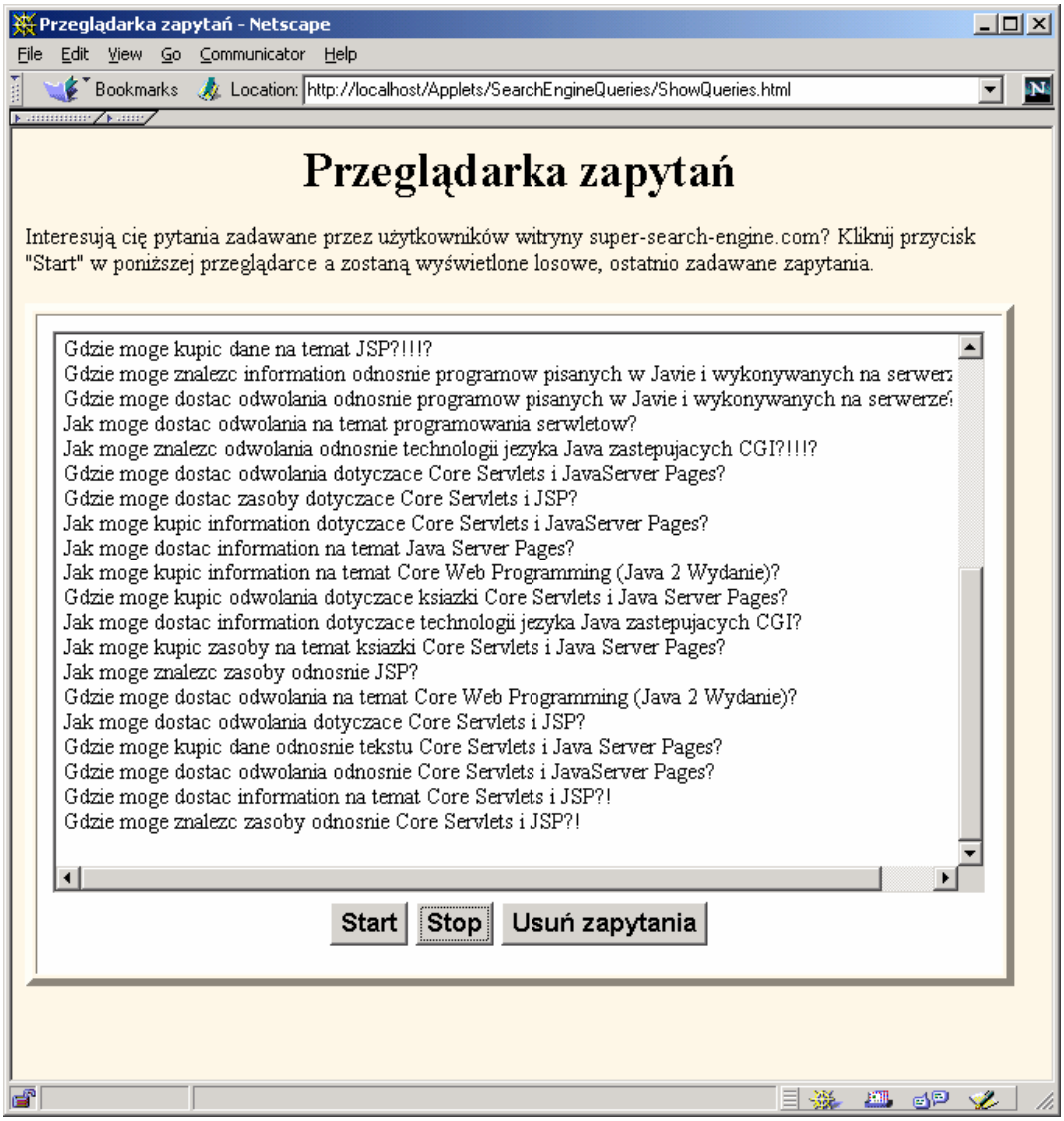
```
import java.net.*;
import java.io.*;

/** Gdy ta klasa zostanie stworzona, zwraca wartość od razu,
 *  jednak wartość ta zwraca false dla isDone
 *  oraz null dla getQueries. W międzyczasie, uruchamiany jest
 *  wątek (Thread) żądający pobrania z serwera tablicy łańcuchów
 *  znaków zawierającej zapytania i odczytujący je w jednym korku
 *  dzięki użyciu strumienia ObjectInputStream.
 *  Po odczytaniu wszystkich wyników, są one umieszczane w
 *  miejscu zwróconym przez getQueries, a fladze isDone
 *  przypisywana jest wartość true.
 *  Klasa używana przez aplet ShowQueries.
 */

public class QueryCollection implements Runnable {
  private String[] queries;
  private String[] tempQueries;
  private boolean isDone = false;
  private URL dataURL;

  public QueryCollection(String urlSuffix, URL currentPage) {
    try {
      // Trzeba podać wyłącznie końcówkę adresu URL,
      // gdyż jego pozostała część jest określana na
      // podstawie bieżącej strony.
      String protocol = currentPage.getProtocol();
      String host = currentPage.getHost();
      int port = currentPage.getPort();
      dataURL = new URL(protocol, host, port, urlSuffix);
      Thread queryRetriever = new Thread(this);
      queryRetriever.start();
    } catch(MalformedURLException mfe) {
      isDone = true;
    }
  }

  public void run() {
    try {
      tempQueries = retrieveQueries();
      queries = tempQueries;
    } catch(IOException ioe) {
      tempQueries = null;
      queries = null;
    }
    isDone = true;
```

```
  }

  public String[] getQueries() {
    return(queries);
  }

  public boolean isDone() {
    return(isDone);
  }

  private String[] retrieveQueries() throws IOException {
    URLConnection connection = dataURL.openConnection();
    // Upewnij się, że przeglądarka nie będzie przechowywać
    // tego żądania w pamięci podręcznej. To ważne, gdyż
    // chcemy za każdym razem pobierać różne zapytania.
    connection.setUseCaches(false);
    // Używam strumienia ObjectInputStream dzięki czemu,
    // za jednym zamachem można odczytać całą tablicę
    // łańcuchów znaków.
    ObjectInputStream in =
      new ObjectInputStream(connection.getInputStream());
    try {
      // Metoda readObject zwraca wartość typu Object,
      // a zatem konieczne jest przeprowadzenie odpowiedniego
      // rzutowania typów.
      String[] queryStrings = (String[])in.readObject();
      return(queryStrings);
    } catch(ClassNotFoundException cnfe) {
      return(null);
    }
  }
}
```

Listing 17.6 QueryGenerator.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Serwlet generujący tablicę łańcuchów znaków
 *  i przesyłający ją przy użyciu strumienia
 *  ObjectOutputStream do apletu lub innego klienta
 *  napisanego w języku Java.
 */

public class QueryGenerator extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    boolean useNumbering = true;
    String useNumberingFlag =
      request.getParameter("useNumbering");
    if ((useNumberingFlag == null) ||
        useNumberingFlag.equals("false")) {
      useNumbering = false;
    }
    String contentType =
      "application/x-java-serialized-object";
    response.setContentType(contentType);
    ObjectOutputStream out =
      new ObjectOutputStream(response.getOutputStream());
    String[] queries = getQueries(useNumbering);
    // Jeśli przesyłasz niestandardową strukturę danych, to
    // będziesz musiał zdefiniować ją jako "implements Serializable".
    out.writeObject(queries);
    out.flush();
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    doGet(request, response);
  }
```

```
  private String[] getQueries(boolean useNumbering) {
    String[] queries = new String[50];
    for(int i=0; i<queries.length; i++) {
      queries[i] = randomQuery();
      if (useNumbering) {
        queries[i] = "" + (i+1) + ": " + queries[i];
      }
    }
    return(queries);
  }

  // Prawdziwe pytania jakie kiedy¶ kto¶ zadał... :-)

  private String randomQuery() {
    String[] locations = { "Gdzie ", "Jak " };
    String[] actions =
      { "moge znalezc ", "moge dostac ", "moge kupic " };
    String[] sources =
      { "information ", "zasoby ", "dane ", "odwolania " };
    String[] prepositions = { "dotyczace ", "odnosnie ", "na temat " };
    String[] subjects =
      { "ksiazki Core Servlets i Java Server Pages",
        "tekstu Core Servlets i Java Server Pages",
        "Core Servlets i JavaServer Pages",
        "Core Servlets i JSP",
        "ksiazki Core Web Programming (Java 2 Wydanie)",
        "Core Web Programming (Java 2 Wydanie)",
        "programowania serwletow", "Java Server Pages", "JSP",
        "technologii jezyka Java zastepujacych CGI",
        "programow pisanych w Javie i wykonywanych na serwerze" };
    String[] endings = { "?", "?", "?", "?!", "?!!!?" };
    String[][] sentenceTemplates =
      { locations, actions, sources,
        prepositions, subjects, endings };
    String query = "";
    for(int i=0; i<sentenceTemplates.length; i++) {
      query = query + randomEntry(sentenceTemplates[i]);
    }
    return(query);
  }

  private String randomEntry(String[] strings) {
    int index = (int)(Math.random()*strings.length);
    return(strings[index]);
  }
}
```



Rysunek 17.3 Aplet ShowQueries w akcji

17.5 Przesyłanie danych metodą POST i bezpośrednie przetwarzanie danych (tunelowanie HTTP)

Przesyłając dane metodą `GET`, aplet może obsłużyć uzyskane wyniki na dwa sposoby — nakazać przeglądarce aby je wyświetliła (tworząc egzemplarz obiektu klasy `URL` i wywołując metodę `getAppletContext().showDocument`) lub obsłużyć je samodzielnie (tworząc egzemplarz obiektu klasy `URL`, pobierając egzemplarz obiektu klasy `URLConnection`, otwierając strumień wejściowy i odczytując wyniki).

Te dwie metody przedstawiłem odpowiednio w podrozdziałach 17.1 oraz 17.3. Jednak przesyłając dane metodą `POST` można zastosować wyłącznie tę drugą metodę, gdyż konstruktor klasy `URL` nie udostępnia żadnego sposobu na dołączenie danych do żądania. Przesyłanie danych metodą `POST` ma podobne wady i zalety, jakie występują przy przesyłaniu danych z apletów metodą `GET`. Pierwsza z dwóch podstawowych wad wynika z faktu, iż program obsługujący aplet musi działać na tym samym komputerze, z którego aplet został pobrany. Druga z wad polega na tym, że aplet musi samodzielnie prezentować otrzymane wyniki — nie istnieje żadne sposób przekazania

kodu HTML do przeglądarki. Do zalet należy natomiast zaliczyć fakt, iż program działający na serwerze może być prostszy (gdyż nie musi przedstawiać wyników w formie kodu HTML), a aplet jest w stanie aktualizować prezentowane wyniki bez konieczności ponownego ładowania strony. Co więcej, aplety przesyłające dane metodą `POST` mogą nie tylko odczytywać serializowane dane generowane przez serwlet, lecz także są w stanie użyć odpowiedniego strumienia wyjściowego, aby przesłać serializowane dane *do* serwletu. To całkiem ważna zaleta, gdyż przesyłanie serializowanych informacji upraszcza transmisję, a tunelowanie pozwala na wykorzystanie istniejących połączeń przekazywanych przez zapory ogniowe, nawet jeśli zestawienie bezpośredniego połączenia jest niemożliwe. Aplety używające żądań `GET` mogą odczytywać serializowane informacje (patrz podrozdział 17.4) lecz nie są w stanie ich przesyłać, gdyż do adresów URL nie można dodawać danych binarnych.

Poniżej przedstawiłem trzynaście czynności jakie należy wykonać, aby aplet mógł przesyłać do serwletu dane metodą `POST` i odczytywać wyniki zwrócone przez serwlet. Choć ilość etapów opisanego procesu jest całkiem duża, to na szczęście same czynności są stosunkowo proste. Przedstawiony poniżej kod został nieco uproszczony poprzez pominięcie boków `try/catch` w których powinien być zapisany.

1. **Utwórz kopię obiektu `URL` odwołujący się do komputera, z którego został pobrany aplet.** Ponieważ użyty adres URL musi się odwoływać do komputera z którego aplet został pobrany, a zatem, także tym razem, najbardziej sensownym rozwiązaniem jest podanie końcówki adresu i automatyczne określenie jego pozostałych elementów.
```
URL aktualnaStrona = getCodeBase();
String protokol = aktualnaStrona.getProtocol();
String host = aktualnaStrona.getHost();
int port = aktualnaStrona.getPort();
String urlSuffix = "/servlet/JakisSerwlet";
URL daneURL = new URL(protokol, host, port, urlSuffix);
```

2. **Utwórz kopię obiektu `URLConnection`.** Obiekt ten zostanie wykorzystany do pobrania strumienia wyjściowego i wejściowego, za pośrednictwem których będzie realizowana wymiana informacji z serwerem.
```
URLConnection polaczenie = daneURL.openConnection();
```

3. **Poinformuj przeglądarkę, iż nie należy przechowywać wyników w pamięci podręcznej.**
```
polaczenie.setUseCaches(false);
```

4. **Poproś system o możliwość przesyłania informacji, a nie samego ich odbierania.**
```
polaczenie.setDoOutput(true);
```

5. **Stwórz strumień `ByteArrayOutputStream`, który zostanie wykorzystany do buforowania danych przesyłanych na serwer.** Obiekt `ByteArrayOutputStream` jest używany w tym aplecie w takim samym celu, w jakim wykorzystaliśmy go podczas implementacji trwałych połączeń HTTP (patrz podrozdział 7.4) — czyli do określenia wielkości przesyłanych informacji, którą musimy podać w nagłówku żądania `Content-Length`. Konstruktor klasy `ByteArrayOutputStream` wymaga podania początkowej wielkości bufora, jednak nie ma ona dużego znaczenia, gdyż w razie potrzeby bufor zostanie automatycznie powiększony.
```
ByteArrayOutputStream strumienBajtowy = new ByteArrayOutputStream(512);
```

6. **Skojarz strumień wyjściowy z obiektem `ByteArrayOutputStream`.** Jeśli chcesz przesyłać zwyczajne informacje podane przez użytkownika w formularzu, posłuż się strumieniem `PrintWriter`, jeśli natomiast chcesz przesyłać serializowane struktury danych, użyj strumienia `ObjectOutputStream`.
```
PrintWriter out = new PrintWriter(strumienBajtowy, true);
```

7. **Zapisz dane w buforze.** W przypadku zwyczajnych informacji pochodzących z formularza, użyj metody `print`; aby zapisać serializowane dane wyższego poziomu, posłuż się metodą `writeObject`.
```
String wartosc1 = URLEncoder.encode(jakasWartosc1);
```

```
String wartosc2 = URLEncoder.encode(jakasWartosc2);
String dane = "param1=" + wartosc1 +
              "&param2=" + wartosc2;    // zwróć uwagę na "&"
out.print(dane); // zwróć uwagę, że używamy metody print, nie println
out.flush();     // konieczne gdyż nie używamy metody println
```

8. **Wygeneruj nagłówek `Content-Length`.** Chociaż przesyłając żądania `GET` nie trzeba używać tego nagłówka, to jednak przypadku użycia żądań `POST` musi on zostać podany.
```
polaczenie.setRequestProperty
  ("Content-Length", String.valueOf(strumienBajtowy.size()));
```

9. **Wygeneruj nagłówek `Content-Type`.** Przeglądarka Netscape Navigator używa domyślnie typu `multipart/form-data`, jednak przesłanie zwyczajnych informacji pochodzących z formularza wymaga użycia typu `application/x-www-form-urlencoded`, standardowo stosowanego w przeglądarce Internet Explorer. A zatem, ze względu na zachowanie przenaszalności kodu, przesyłając dane pochodzące z formularzy, powinieneś jawnie określić ich typ. W przypadku przesyłania serializowanych informacji, określanie typu nie ma znaczenia.
```
polaczenie.setRequestProperty
  ("Content-Type", "application/x-www-form-urlencoded" );
```

10. **Prześlij dane.**
```
strumienBajtowy.writeTo(polaczenie.getOutputStream());
```

11. **Otwórz strumień wejściowy.** W przypadku odczytywania zwyczajnych danych tekstowych bądź binarnych używany jest strumień `BufferedReader`, natomiast w razie odczytu serializowanych obiektów — strumień `ObjectInputStream`.
```
BufferedReader in =
  new BufferedReader(new InputStreamReader(polaczenie.getInputStream()));
```

12. **Odczytaj wyniki.** Konkretny sposób wykonania tej czynności zależy od rodzaju informacji przesyłanych z serwera. W przykładzie przedstawionym poniżej „robimy coś” z każdym wierszem odczytanych informacji.
```
String wiersz;
while((wiersz = in.readLine()) != null) {
  zrobCosZ(wiersz);
}
```

13. Pogratuluj sobie. O tak, procedura przesyłania danych metodą `POST` jest długa i męcząca. Na szczęście, jest także stosunkowo schematyczna. A poza tym, zawsze możesz skopiować odpowiedni przykład z serwera FTP Wydawnictwa HELION (*ftp://ftp.helion.pl/przyklady/jsjsp.zip*).

W kolejnym podrozdziale przedstawiłem przykład apletu przesyłającego dane w powyższy sposób.

17.6 Aplet przesyłający dane metodą POST

Na listingu 17.7 przedstawiłem aplet działający według metody opisanej w poprzednim podrozdziale. Aplet przesyła metodą `POST` informacje pod adres wskazany przez użytkownika, wykorzystując do tego celu obiekt `URLConnection` oraz skojarzony z nim strumień `ByteArrayOutputStream`. Aplet korzysta także z klasy `LabeledTextFiled` przedstawionej na początku książki na listingu 2.2, której kod źródłowy można pobrać z serwera FTP Wydawnictwa HELION (*ftp://ftp.helion.pl/przyklady/jsjsp.zip*).

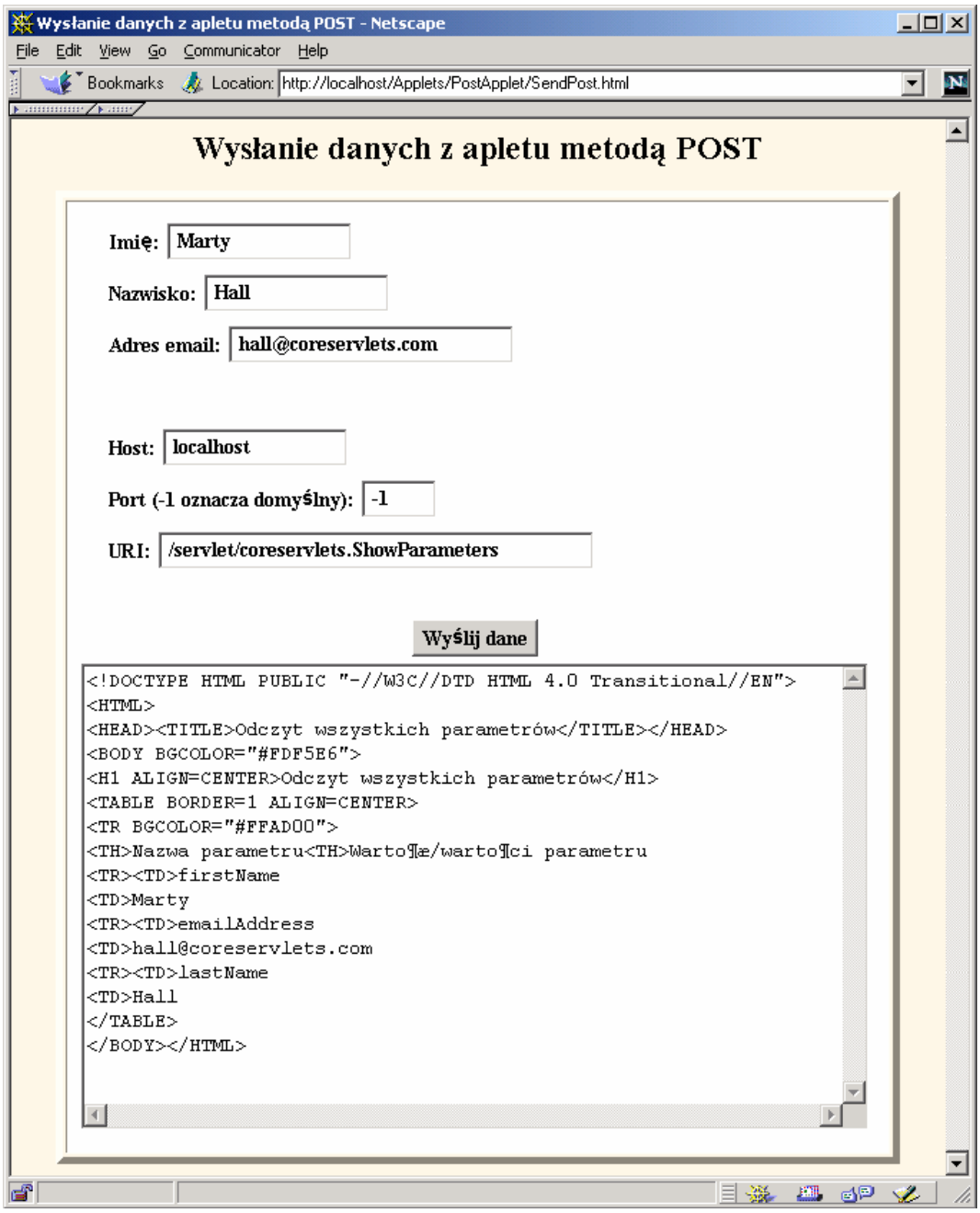
Na rysunku 17.4 przedstawiłem wyniki przesłania danych z tego apletu do serwletu `ShowParameters`, a na rysunku 17.5 do prostego serwera HTTP *EchoServer*.

Listing 17.7 SendPost.java

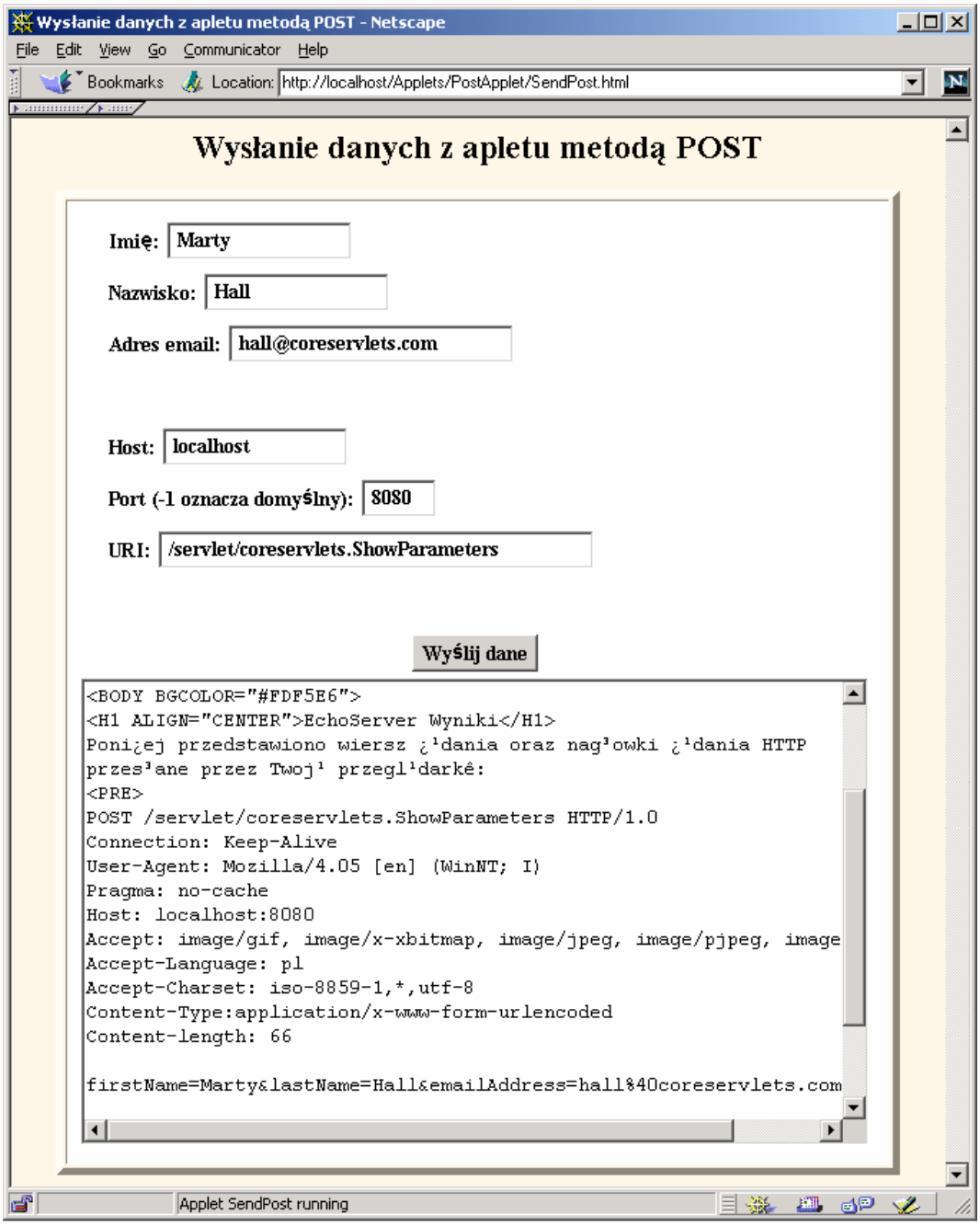
```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;

/** Aplet odczytujący wartości parametrów firstName,
 *  lastName oraz emailAddress i przesyłający je
 *  na serwer metodą POST (pod podany adres i przy
 *  użyciu portu o określonym numerze.
 */

public class SendPost extends Applet
                      implements ActionListener {
  private LabeledTextField firstNameField, lastNameField,
                           emailAddressField, hostField,
                           portField, uriField;
  private Button sendButton;
  private TextArea resultsArea;
  URL currentPage;

  public void init() {
    setBackground(Color.white);
    setLayout(new BorderLayout());
    Panel inputPanel = new Panel();
    inputPanel.setLayout(new GridLayout(9, 1));
    inputPanel.setFont(new Font("Serif", Font.BOLD, 14));
    firstNameField =
      new LabeledTextField("Imię:", 15);
    inputPanel.add(firstNameField);
    lastNameField =
      new LabeledTextField("Nazwisko:", 15);
    inputPanel.add(lastNameField);
    emailAddressField =
      new LabeledTextField("Adres email:", 25);
    inputPanel.add(emailAddressField);
    Canvas separator1 = new Canvas();
    inputPanel.add(separator1);
    hostField =
      new LabeledTextField("Host:", 15);

    // Aplety pobierane przez Internet mogą się
    // łączyć tylko z serwerem, z którego zostały pobrane.
    hostField.getTextField().setEditable(false);

    currentPage = getCodeBase();
    // metoda getHost zwraca pusty łańcuch znaków
    // jeśli aplet został pobrany z lokalnego dysku.
    String host = currentPage.getHost();
    String resultsMessage = "Tutaj zostaną wyświetlone wyniki...";
    if (host.length() == 0) {
      resultsMessage = "Błąd: musisz pobrać ten aplet z \n" +
                       "prawdziwego serwera WWW za pośrednictwem HTTP,\n" +
                       "a nie jako plik z lokalnego dysku.\n" +
                       "Nawet jeśli serwer działa na Twoim \n" +
                       "lokalnym komputerze.";
      setEnabled(false);
    }
    hostField.getTextField().setText(host);
    inputPanel.add(hostField);
    portField =
      new LabeledTextField("Port (-1 oznacza domyślny):", 4);
    String portString = String.valueOf(currentPage.getPort());
    portField.getTextField().setText(portString);
    inputPanel.add(portField);
    uriField =
      new LabeledTextField("URI:", 40);
    String defaultURI = "/servlet/coreservlets.ShowParameters";
    uriField.getTextField().setText(defaultURI);
    inputPanel.add(uriField);
    Canvas separator2 = new Canvas();
    inputPanel.add(separator2);
    sendButton = new Button("Wyślij dane");
    sendButton.addActionListener(this);
    Panel buttonPanel = new Panel();
    buttonPanel.add(sendButton);
    inputPanel.add(buttonPanel);
```

```
    add(inputPanel, BorderLayout.NORTH);
    resultsArea = new TextArea();
    resultsArea.setFont(new Font("Monospaced", Font.PLAIN, 14));
    resultsArea.setText(resultsMessage);
    add(resultsArea, BorderLayout.CENTER);
  }

  public void actionPerformed(ActionEvent event) {
    try {
      String protocol = currentPage.getProtocol();
      String host = hostField.getTextField().getText();
      String portString = portField.getTextField().getText();
      int port;
      try {
        port = Integer.parseInt(portString);
      } catch(NumberFormatException nfe) {
        port = -1; // na przykład, domyślnie port 80
      }
      String uri = uriField.getTextField().getText();
      URL dataURL = new URL(protocol, host, port, uri);
      URLConnection connection = dataURL.openConnection();

      // Upewnij się że przeglądarka nie będzie
      // przechowywać danych w pamięci podręcznej.
      connection.setUseCaches(false);

      // Poproś przeglądarkę o pozwolenie przesyłania danych
      // na serwer.
      connection.setDoOutput(true);

      ByteArrayOutputStream byteStream =
        new ByteArrayOutputStream(512); // Powiększ jeśli konieczne
      // Strumień zapisuje dane do buforu
      PrintWriter out = new PrintWriter(byteStream, true);
      String postData =
        "firstName=" + encodedValue(firstNameField) +
        "&lastName=" + encodedValue(lastNameField) +
        "&emailAddress=" + encodedValue(emailAddressField);

      // Zapisz dane do lokalnego buforu
      out.print(postData);
      out.flush(); // opróżnij bufor gdyż używamy metody print
                   // a nie println.

      // żądania POST muszą zawierać nagłówek Content-Length
      String lengthString =
        String.valueOf(byteStream.size());
      connection.setRequestProperty
        ("Content-Length", lengthString);

      // Netscape domyślnie zapisuje w nagłówku
      // Content-Type wartość multipart/form-data.
      // A zatem, jeśli chcesz wysyłać zwyczajne
      // informacje wprowadzone przez użytkownika w
      // formularzu, musisz samemu przypisać temu
      // nagłówkowi wartość
      // application/x-www-form-urlencoded, która
      // jest domyślnie stosowana w przeglądarce
      // Internet Explorer. Jeśli metodą POST przesyłasz
      // serializowane dane (posługując się przy tym
      // strumieniem ObjectOutputStream, to zawartość
      // nagłówka Content-Type nie ma znaczenia, a zatem
      // będziesz mógł pominąć ten krok.
      connection.setRequestProperty
        ("Content-Type", "application/x-www-form-urlencoded");

      // Zapisz dane do faktycznego strumienia wyjściowego
      byteStream.writeTo(connection.getOutputStream());

      BufferedReader in =
        new BufferedReader(new InputStreamReader
                           (connection.getInputStream()));
      String line;
      String linefeed = "\n";
      resultsArea.setText("");
      while((line = in.readLine()) != null) {
        resultsArea.append(line);
```

```
          resultsArea.append(linefeed);
        }
      } catch(IOException ioe) {
        // Wyświetl komunikat na konsoli Javy.
        System.out.println("IOException: " + ioe);
      }
    }

    // LabeledTextField to w rzeczywistości Panel zawierający
    // etykietę (Label) oraz pole tekstowe (TextField).
    // Poniższy kod pobiera zawartość pola tekstowego,
    // zapisuje ją w formacie URL i zwraca.

    private String encodedValue(LabeledTextField field) {
      String rawValue = field.getTextField().getText();
      return(URLEncoder.encode(rawValue));
    }

}
```



Rysunek 17.4 Wyniki użycia serwletu SendPost do przesłania danych metodą POST do serwletu ShowParameters (przedstawionego w podrozdziale 3.4. — „Przykład: Odczyt wszystkich parametrów”)



Rysunek 17.5 Wyniki użycia serwletu SendPost do przesłania danych metodą POST do serwera HTTP EchoServer (przedstawionego w podrozdziale 16.12. — „Testowy serwer WWW”)

17.7 Pomijanie serwera HTTP

Choć aplety mogą nawiązywać połączenia sieciowe wyłącznie z komputerem z którego zostały pobrane, to jednak nie muszą używać w tym celu tego samego portu (na przykład, portu 80). Oznacza to, że aplety mogą komunikować się z programami uruchamianymi na serwerze wykorzystując w tym celu gniazda, JDBC bądź RMI.

Aplety wykonują te czynności w identyczny sposób jak zwyczajne aplikacje pisane w języku Java. Dzięki temu tworząc je możesz wykorzystać dowolne znane Ci techniki obsługi gniazd, JDBC lub RMI. Jedynym warunkiem jest to, iż serwer sieciowy musi działać na tym samym serwerze WWW z którego został pobrany aplet.

Rozdział 18.
JDBC oraz zarządzanie pulami połączeń

JDBC udostępnia standardową bibliotekę zapewniającą możliwość korzystania z relacyjnych baz danych. Korzystając z JDBC API można uzyskać dostęp do bardzo wielu różnych baz danych SQL posługując się dokładnie tym samym kodem napisanym w języku Java. Koniecznie należy zauważyć, iż choć JDBC standaryzuje sposób nawiązywania połączenia z bazami danych, składnię poleceń używanych do przesyłania zapytań i zatwierdzania transakcji oraz struktury danych reprezentujące zwracane wyniki, to jednak *nie* podejmuje prób standaryzacji składni języka SQL. Oznacza to, że można korzystać z wszelkich rozszerzeń udostępnianych przez używaną bazę danych. Niemniej jednak, większość stosowanych zapytań jest zgodna ze standardową składnią języka SQL, dzięki temu, korzystając z JDBC można zmieniać komputery na których działają serwery baz danych, porty, a nawet rodzaj używanych baz danych wprowadzając jedynie minimalne zmiany w kodzie.

Oficjalnie „JDBC" nie jest akronimem, jednak nieoficjalnie jest to skrót od słów „Java Database Connectivity".

Notatka
JDBC to nie jest skrót.

Choć podanie szczegółowych i wyczerpujących informacji na temat programowania baz danych wykracza poza ramy niniejszego rozdziału, to omówię w nim podstawowe zagadnienia związane z wykorzystaniem JDBC, zakładając jednocześnie że już znasz język SQL. Więcej informacji na temat JDBC znajdziesz pod adresem *http://java.sun.com/products/jdbc/* oraz w internetowej dokumentacji pakietu `java.sql`; a jeśli szukasz jakiegoś podręcznika poświęconego JDBC, znajdziesz go pod adresem *http://java.sun.com/docs/books/tutorial/jdbc/*. Jeśli jeszcze nie dysponujesz żadną bazą danych, to do nauki możesz wykorzystać MySQL. Bazy tej można używać bezpłatnie we wszystkich systemach operacyjnych oprócz systemów Windows, a w systemach Windows można jej używać bezpłatnie w celach edukacyjnych i badawczych. Więcej informacji na temat tej bazy danych znajdziesz pod adresem *http://www.mysql.com/*.

18.1 Podstawowe etapy wykorzystania JDBC

Proces pobierania informacji z baz danych składa się z siedmiu etapów:

1. Załadowania sterownika JDBC.
2. Zdefiniowania adresu URL połączenia.
3. Nawiązania połączenia.
4. Stworzenia obiektu polecenia.
5. Wykonania zapytania lub aktualizacji danych.

6. Przetworzenia wyników.
7. Zamknięcia połączenia.

Poniżej nieco bardziej szczegółowo omówiłem każdy z tych etapów.

Załadowanie sterownika

Sterownik to niewielki program, który „wie" jak należy komunikować się z serwerem bazy danych. Aby załadować sterownik wystarczy jedynie załadować odpowiednią klasę — statyczny blok kodu wewnątrz niej automatycznie stworzy kopię sterownika i zarejestruje go w narzędziu zarządzającym sterownikami JDBC — tak zwanym menadżerze sterowników JDBC. Aby zapewnić jak największą elastyczność tworzonego kodu, należy unikać podawania „na stałe" nazwy używanej klasy.

Powyższe wymagania skłaniają do zadania dwóch interesujących pytań. Po pierwsze, w jaki sposób można załadować klasę bez tworzenia jej kopii? A po drugie, jak odwołać się do klasy, której nazwa nie jest znana podczas kompilacji kodu? Odpowiedź na oba te pytania jest identyczna — tajemnica tkwi w użyciu metody `Class.forName`. Metoda ta pobiera jeden argument — w pełni kwalifikowaną nazwę klasy (czyli nazwę klasy wraz z nazwami wszystkich pakietów do których ona należy), i ładuję tę klasę. Wywołanie tej metody może spowodować zgłoszenie wyjątku `ClassNotFoundException`, a zatem należy je umieścić wewnątrz bloku `try/catch`. Oto przykład użycia tej metody:

```
try {
  Class.forName("connect.microsoft.MicrosoftDriver");
  Class.forName("oracle.jdbc.driver.OracleDriver");
  Class.forName("com.sybase.jdbc.SybDriver");
} catch (ClassNotFoundException cnfe) {
  System.out.println("Błąd podczas ładowania sterownika: " + cnfe);
}
```

Jedną z najwspanialszych cech JDBC jest to, iż zmiana używanego serwera WWW nie wymaga wprowadzania jakichkolwiek zmian w kodzie. To sterownik JDBC (działający po stronie klienta) tłumaczy wywołania napisane w języku Java do formatu wymaganego przez serwer bazy danych. Taki sposób działania oznacza, że trzeba dysponować odpowiednim sterownikiem, przeznaczonym do obsługi używanej bazy danych. Informacje na temat w pełni kwalifikowanej nazwy klasy sterownika, której będziesz mógł używać w swoich programach, powinieneś znaleźć w jego dokumentacji. Większość firm tworzących bazy danych, udostępnia bezpłatne wersje sterowników JDBC przeznaczonych do obsługi tych baz; istnieje jednak wiele innych firm, które także udostępniają sterowniki dla starszych typów baz danych. Aktualną listę wszystkich dostępnych sterowników można znaleźć na stronie WWW pod adresem *http://java.sun.com/products/jdbc/ drivers.html*. Wiele firm podanych na tej liście udostępnia demonstracyjne wersje sterowników, które zazwyczaj mają ograniczony czas działania lub narzucają ograniczenia na ilość jednocześnie obsługiwanych połączeń. A zatem, można się nauczyć JDBC bez konieczności płacenia za sterowniki.

Ogólnie rzecz biorąc, metody `Class.forName` można używać do załadowania każdej klasy znajdującej się w katalogach określonych w zmiennej środowiskowej `CLASSPATH`. Jednak w praktyce, sterowniki JDBC są zazwyczaj dostarczane w formie plików *JAR*. A zatem, nie zapomnij dodać ścieżki dostępu do tych plików, do zmiennej środowiskowej `CLASSPATH` na swoim komputerze.

Określenie adresu URL połączenia

Po załadowaniu sterownika JDBC należy określić położenie serwera bazy danych. Adresy URL odwołujące się do baz danych używają protokołu `jdbc:` i zawierają informacje o nazwie komputera na którym działa serwer bazy danych, numerze używanego portu oraz nazwie bazy

danych (lub o odwołaniu do niej). Konkretny format zapisu takiego adresu URL będzie podany w dokumentacji dostarczonej wraz z konkretnym sterownikiem JDBC. Poniżej podałem dwa typowe przykłady:

```
String host = "dbhost.firma.com.pl";
String dbNazwa = "nazwaBazy";
int port = 1234;
String oracleURL = "jdbc:oracle:thin:@" + host +
                   ":" + port + ":" + dbNazwa;
String sybaseURL = "jdbc:sybase:Tds:" + host +
                   ":" + port + ":" + "?SERVICENAME=" + dbNazwa;
```

JDBC jest najczęściej wykorzystywane w serwletach oraz zwyczajnych aplikacjach, lecz czasami używa się go także w apletach. Jeśli używasz JDBC a aplecie, to musisz pamiętać, iż przeglądarki pozwalają apletom na nawiązywanie połączeń sieciowych wyłącznie z komputerami, z których aplety te zostały pobrane; zabezpieczenie to ma na celu uniemożliwienie apletom zbierania informacji o sieciach chronionych zaporami ogniowymi. W konsekwencji, aby używać JDBC w apletach, serwer bazy danych musi działać na tym samym komputerze co serwer WWW, lub konieczne jest wykorzystanie serwera pośredniczącego, który będzie kierował żądania odwołujące się do bazy danych, na odpowiedni komputer.

Nawiązanie połączenia

Aby nawiązać połączenie sieciowe, w wywołaniu metody `getConnection` klasy `DriverManager` należy podać adres URL, nazwę użytkownika bazy danych oraz hasło; tak jak pokazałem na poniższym przykładzie. Zwróć uwagę, iż wywołanie metody `getConnection` może zgłosić wyjątek `SQLException`, a zatem należy je zapisać wewnątrz bloku `try/catch`. W poniższym przykładzie pominąłem instrukcje `try` oraz `catch`, gdyż metody przedstawione w kolejnych częściach rozdziału także mogą zgłaszać te same wyjątki i dlatego wszystkie są zazwyczaj umieszczane w jednym bloku `try/catch`.

```
String nazwaUzytkownika = "jarek_debesciak";
String haslo = "tajemne";
Connection polaczenie =
   DriverManager.getConnection(oracleURL, nazwaUzytkownika, haslo);
```

Opcjonalną czynnością jaką można wykonać na tym etapie, jest odszukanie informacji dotyczących baz danych, przy użyciu metody `getMetaData` klasy `Connection`. Metoda ta zwraca obiekt `DatabaseMetaData`. Udostępniane przez niego metody pozwalają na określenie nazwy oraz numeru wersji samej bazy danych (służą do tego odpowiednio metody: `getDatabaseProductName` oraz `getDatabaseProductVersion`) lub sterownika JDBC (służą do tego odpowiednio metody `getDriverName` oraz `getDriverVersion`). Poniżej przedstawiłem stosowny przykład:

```
DatabaseMetaData dbMetadane = polaczenie.getMetaData();
String nazwaBazy = dbMetadane.getDatabaseProductName();
System.out.println("Baza danych: " + nazwaBazy);
String numerWersji = dbMetadane.getDatabaseProductVersion();
System.out.println("Numer wersji bazy: " + numerWersji;
```

Inne przydatne metody klasy `Connection` to: `prepareStatement` (metoda ta, omówiona w podrozdziale 18.6, tworzy kopię obiektu `PreparedStatement`), `prepareCall` (ta metoda tworzy kopię obiektu klasy `CallableStatement`), `rollback` (metoda odtwarza wszystkie czynności wykonane od momentu ostatniego wywołania metody `commit`), `commit` (metoda zatwierdza wszystkie czynności wykonane od czasu poprzedniego wywołania tej metody), `close` (metoda zamyka połączenie) oraz `isClosed` (która sprawdza czy połączenie zostało zakończone bądź czy upłynął czas jego ważności).

Stworzenie polecenia

Do przesyłania zapytań i poleceń do bazy danych używane są obiekty `Statement`. Kopię obiektu `Statement` można uzyskać wywołując metodę `createStatement` klasy `Connection`:

```
Statement polecenie = polaczenie.createStatement();
```

Wykonanie zapytania

Dysponując obiektem `Statement`, można już przesłać zapytanie SQL. Do tego celu wykorzystywana jest metoda `executeQuery`, zwracająca obiekt `ResultSet`. Poniżej przedstawiłem stosowny przykład:

```
String zapytanie = "SELECT kol1, kol2, kol3 FROM tabela";
ResultSet zbiorWynikow = polecenie.executeQuery(zapytanie);
```

Aby zmodyfikować bazę danych zamiast metody `executeQuery`, należy użyć metody `executeUpdate` i podać w jej wywołaniu łańcuch znaków zawierający polecenie SQL `UPDATE`, `INSERT` lub `DELETE`. Inne przydatne metody klasy `Statement` to: `execute` (która wykonuje dowolne polecenie) oraz `setQueryTimeout` (która określa maksymalny czas oczekiwania na wyniki). Można także tworzyć zapytania z parametrami — w takim przypadku wartości są przekazywane do prekompilowanych zapytań, o ściśle określonej postaci. Więcej informacji na ten temat znajdziesz w podrozdziale 18.6.

Przetworzenie wyników

Najprostszym sposobem obsługi wyników jest indywidualne przetworzenie każdego zwróconego wiersza. Do tego celu można wykorzystać metodę `next` klasy `RecordSet`, której wywołanie powoduje przejście do następnego wiersza tabeli wyników. Podczas przetwarzania wierszy można posługiwać się metodami `get`*`Xxx`*, które pobierają jako argument indeks kolumny lub jej nazwę, a zwracają zawartość wskazanej kolumny w formie wartości różnych typów. Na przykład, jeśli pobrana wartość ma być liczbą całkowitą, należy się posłużyć metodą `getInt`, jeśli łańcuchem znaków — metodą `getString`, i tak dalej. Dostępne są metody `get`*`Xxx`* zwracające wartości niemal wszystkich podstawowych typów danych dostępnych w języku Java. Jeśli chcesz po prostu wyświetlić wyniki, to możesz użyć metody `getString` niezależnie od faktycznego typu danych przechowywanych w danej kolumnie. W przypadku posługiwania się wersjami metod wykorzystującymi indeksy kolumn, należy zwrócić uwagę, iż kolumny są indeksowane od wartości 1 (zgodnie z konwencją przyjętą w języku SQL), a nie od 0 jak jest w przypadku tablic, wektorów oraz wielu innych struktur danych stosowanych w języku Java.

Ostrzeżenie
Pierwsza kolumna zbioru wyników (w obiekcie ResultSet) ma indeks 1 a nie 0.

Poniższy przykład wyświetla wartości trzech pierwszych kolumn wszystkich wierszy zbioru wyników.

```
while(zbiorWynikow.next()) {
  System.out.println(zbiorWynikow.getString(1) + " " +
                     zbiorWynikow.getString(2) + " " +
                     zbiorWynikow.getString(3));
}
```

Oprócz metod `get`*`Xxx`* i `next`, klasa `ResultSet` udostępnia jeszcze inne, przydatne metody. Można do nich zaliczyć metodę `findColumn` (która zwraca indeks kolumny o podanej nazwie), `wasNull` (która sprawdza czy ostatnie wywołanie dowolnej metody `get`*`Xxx`* zwróciło wartość `null`; w przypadku łańcuchów znaków można to sprawdzić porównując uzyskany wynik z wartością `null`)

oraz `getMetaData` (która pobiera informacje o zbiorze wyników i zwraca je w formie obiektu klasy `ResultSetMetaData`).

Niezwykle przydatna jest metoda `getMetaData`. Dysponując jedynie obiektem klasy `ResultSet`, aby poprawnie przetworzyć uzyskane wyniki, konieczna jest znajomość nazw, ilości oraz typów kolumn. W przypadku zapytań o znanym i ustalonym formacie, można założyć że będziemy dysponowali tymi informacjami. Jednak w przypadku pytań tworzonych „na bieżąco", warto jest mieć możliwość dynamicznego określenia informacji wysokiego poziomu dotyczących uzyskanych wyników. I w tym momencie uwidacznia się znaczenie klasy `ResultSetMetaData`. Pozawala ona określać liczbę, nazwy oraz typy kolumn dostępnych w obiektach klasy `ResultSet`. Przydatne metody klasy `ResultSetMetaData` to: `getColumnCount` (która zwraca ilość kolumn), `getColumnName(numerKolumny)` (która zwraca nazwę kolumny o podanym indeksie, przy czym numeracja kolumn rozpoczyna się od wartości 1), `getColumnType` (metoda ta zwraca wartość typu int, którą należy porównywać ze stałymi zdefiniowanymi w klasie `java.sql.Types`), `isSearchable` (która określa czy danej kolumny można użyć w klauzuli `WHERE`), `isNullable` (która określa czy dana kolumna może zawierać wartości `null`) oraz kilka innych metod, które udostępniają szczegółowe informacje na temat typu oraz precyzji wybranej kolumny. Klasa `ResultSetMetaData` nie zawiera jednak żadnych informacji na temat ilości pobranych wierszy. Oznacza to, że jedynym sposobem pobrania wszystkich wierszy jest cykliczne wywoływanie metody next klasy `ResultSet`, aż do momentu gdy zwróci ona wartość `false`.

Zamknięcie połączenia

Aby jawnie zamknąć połączenie należy posłużyć się wywołaniem o następującej postaci:

```
polaczenie.close();
```

Jeśli planujesz wykonywać jeszcze inne operacje na bazie danych, to powinieneś odłożyć zamknięcie połączenia na później, gdyż koszt otworzenia nowego połączenia jest zazwyczaj dosyć duży. W rzeczywistości wielokrotne wykorzystywanie istniejących połączeń jest tak ważną metodą optymalizacji, iż w podrozdziale 18.7 przedstawiłem bibliotekę służącą właśnie do tego celu, a w podrozdziale 18.8 pokazałem typowe oszczędności czasowe jakie można dzięki niej uzyskać.

18.2 Prosty przykład wykorzystania JDBC

Listing 18.3 przedstawia prostą klasę o nazwie `FuritTest`, która wykonuje cały proces opisany w poprzednim podrozdziale, aby wyświetlić zawartość prostej tabeli o nazwie `fruits`. Klasa ta używa argumentów przekazywanych w wierszu poleceń w celu określenia komputera, portu, nazwy bazy danych oraz typu używanego sterownika. Przykładowe wyniki jej wykonania przedstawiłem na listingach 18.1 oraz 18.2. Zamiast umieszczać nazwę sterownika oraz implementację czynności związanych z generacją poprawnie sformatowanego adresu URL, w głównej klasie naszej przykładowej aplikacji, przeniosłem odpowiedni kod do osobnej klasy o nazwie `DriverUtilities`, przedstawionej na listingu 18.4. W ten sposób minimalizuję ilość miejsc w kodzie, które należy zmodyfikować w razie wykorzystania różnych sterowników.

Listing 18.1 Wyniki wykonania aplikacji FruitTest (połączenie z bazą Oracle działającą w systemie Solaris)

```
Prompt> java coreservlets.FruitTest dbhost1.apl.jhu.edu PTE
        hall xxxx oracle
Database: Oracle
Version: Oracle7 Server Release 7.2.3.0.0 - Production Release
PL/SQL Release 2.2.3.0.0 - Production

Porównanie jabłek  i pomarańczy
============================
```

```
QUARTER  APPLES  APPLESALES  ORANGES  ORANGESALES  TOPSELLER
    1     32248   $3547.28    18459    $3138.03      Maria
    2     35009   $3850.99    18722    $3182.74      Bob
    3     39393   $4333.23    18999    $3229.83      Joe
    4     42001   $4620.11    19333    $3286.61      Maria
```

Listing 18.2 Wyniki wykonania aplikacji FruitTest (połączenie z bazą Sybase działającą w systemie Windows NT)

```
Prompt> java coreservlets.FruitTest dbhost2.apl.jhu.edu 605741
        hall xxxx sybase
Baza danych: Adaptive Server Anywhere
Wersja: 6.0.2.2188

Porównanie jabłek  i pomarańczy
============================
quarter  apples  applesales  oranges  orangesales  topseller
    1     32248   $3547.28    18459    $3138.03      Maria
    2     35009   $3850.99    18722    $3182.74      Bob
    3     39393   $4333.23    18999    $3229.83      Joe
    4     42001   $4620.11    19333    $3286.61      Maria
```

Listing 18.3 FruitTest.java

```
package coreservlets;

import java.sql.*;

/** Przykład użycia JDBC, który nawiązuje połączenie
 *  z bazą Oracle lub Sybase i wyświetla wartości
 *  z góry określonych kolumn tabeli "fruits".
 */

public class FruitTest {

  /** Odczytuje nazwę komputera, bazy danych, użytkownika,
   *  hasło dostępu oraz dostawcy z wiersza poleceń.
   *  Identyfikator dostawcy jest używany w celu określenia
   *  jaki sterownik należy załadować oraz jak sformatować
   *  adres URL. Sterownik, adres URL, nazwa komputera,
   *  nazwa użytkownika oraz hasło są następnie
   *  przekazywane do metody showFruitTable.
   */

  public static void main(String[] args) {
    if (args.length < 5) {
      printUsage();
      return;
    }
    String vendorName = args[4];
    int vendor = DriverUtilities.getVendor(vendorName);
    if (vendor == DriverUtilities.UNKNOWN) {
      printUsage();
      return;
    }
    String driver = DriverUtilities.getDriver(vendor);
    String host = args[0];
    String dbName = args[1];
    String url = DriverUtilities.makeURL(host, dbName, vendor);
    String username = args[2];
    String password = args[3];
    showFruitTable(driver, url, username, password);
  }

  /** Pobiera tabelę i wyświetla całą jej zawartość. */

  public static void showFruitTable(String driver,
                                    String url,
                                    String username,
                                    String password) {
    try {
      // Załaduj sterownik bazy danych, jeśli jeszcze nie jest to zrobione.
      Class.forName(driver);
      // Nawiąż połączenie sieciowe z bazą danych.
      Connection connection =
```

```
        DriverManager.getConnection(url, username, password);
      // Pobierz informacje dotyczące samej bazy danych.
      DatabaseMetaData dbMetaData = connection.getMetaData();
      String productName =
        dbMetaData.getDatabaseProductName();
      System.out.println("Baza danych: " + productName);
      String productVersion =
        dbMetaData.getDatabaseProductVersion();
      System.out.println("Wersja: " + productVersion + "\n");
      System.out.println("Porównanie jabłek i pomarańczy\n" +
                         "============================");
      Statement statement = connection.createStatement();
      String query = "SELECT * FROM fruits";
      // Wyślij zapytanie do bazy danych i zapisz wyniki.
      ResultSet resultSet = statement.executeQuery(query);
      // Pobierz informacje dotyczące konkretnej tabeli.
      ResultSetMetaData resultsMetaData =
        resultSet.getMetaData();
      int columnCount = resultsMetaData.getColumnCount();
      // Indeksy kolumn są liczone od 1 (jak w SQLu) a nie
      // od 0 (jak w języku Java).
      for(int i=1; i<columnCount+1; i++) {
        System.out.print(resultsMetaData.getColumnName(i) +
                         "  ");
      }
      System.out.println();
      // Wyświetl wyniki.
      while(resultSet.next()) {
        // Kwartał
        System.out.print("    " + resultSet.getInt(1));
        // Ilość jabłek
        System.out.print("     " + resultSet.getInt(2));
        // Sprzedaż jabłek
        System.out.print("   $" + resultSet.getFloat(3));
        // Ilość pomarańczy
        System.out.print("    " + resultSet.getInt(4));
        // Sprzedaż pomarańczy
        System.out.print("    $" + resultSet.getFloat(5));
        // Najlepszy sprzedawca
        System.out.println("      " + resultSet.getString(6));
      }
    } catch(ClassNotFoundException cnfe) {
      System.err.println("Błąd ładowania sterownika: " + cnfe);
    } catch(SQLException sqle) {
      System.err.println("Błąd przy nawiązywaniu połączenia: " + sqle);
    }
  }

  private static void printUsage() {
    System.out.println("Użycie: FruitTest komputer nazwaBazyDanych " +
                       "nazwaUżytkownika hasło oracle|sybase.");
  }
}
```

Listing 18.4 DriverUtilities.java

```
package coreservlets;

/** Proste narzędzia służące do tworzenia połączeń JDBC
 *  z bazami danych Oracle oraz Sybase. To <I>nie </I> jest
 *  kod ogólnego przeznaczenia - został od dostosowany do
 *  konfiguracji <I>mojego</I> komputera.
 */

public class DriverUtilities {
  public static final int ORACLE = 1;
  public static final int SYBASE = 2;
  public static final int UNKNOWN = -1;

  /** Tworzy URL zapisany w formacie odpowiednim dla używanych
   *  przeze mnie sterowników baz danych Oracle i Sybase.
   */

  public static String makeURL(String host, String dbName,
                               int vendor) {
    if (vendor == ORACLE) {
```

```
      return("jdbc:oracle:thin:@" + host + ":1521:" + dbName);
    } else if (vendor == SYBASE) {
      return("jdbc:sybase:Tds:" + host  + ":1521" +
             "?SERVICENAME=" + dbName);
    } else {
      return(null);
    }
  }

  /** Zwraca w pełni kwalifikowaną nazwę sterownika. */

  public static String getDriver(int vendor) {
    if (vendor == ORACLE) {
      return("oracle.jdbc.driver.OracleDriver");
    } else if (vendor == SYBASE) {
      return("com.sybase.jdbc.SybDriver");
    } else {
      return(null);
    }
  }

  /** Kojarzy nazwę z wartością. */

  public static int getVendor(String vendorName) {
    if (vendorName.equalsIgnoreCase("oracle")) {
      return(ORACLE);
    } else if (vendorName.equalsIgnoreCase("sybase")) {
      return(SYBASE);
    } else {
      return(UNKNOWN);
    }
  }
}
```

Działanie przedstawionego przykładu nie zależy do sposobu utworzenia tablic bazy danych, a jedynie od ich ostatecznego formatu. A zatem, przykładowo, można by użyć interaktywnego narzędzia służącego do obsługi i zarządzania bazami danych. Jednak w rzeczywistości, do stworzenia przykładowej tabeli, także wykorzystałem JDBC — aplikację użytą do tego celu przedstawiłem na listingu 18.5. Jak na razie możesz pobieżnie przejrzeć kod tego programu, gdyż użyłem w nim narzędzi, które przedstawię dopiero w dalszej części rozdziału.

Listing 18.5 FruitCreation.java

```
package coreservlets;

import java.sql.*;

/** Tworzy prostą tabelę o nazwie "fruits" w bazie
 *  danych Oracle lub Sybase.
 */

public class FruitCreation {
  public static void main(String[] args) {
    if (args.length < 5) {
      printUsage();
      return;
    }
    String vendorName = args[4];
    int vendor = DriverUtilities.getVendor(vendorName);
    if (vendor == DriverUtilities.UNKNOWN) {
      printUsage();
      return;
    }
    String driver = DriverUtilities.getDriver(vendor);
    String host = args[0];
    String dbName = args[1];
    String url =
      DriverUtilities.makeURL(host, dbName, vendor);
    String username = args[2];
    String password = args[3];
    String format =
      "(quarter int, " +
      "apples int, applesales float, " +
```

```
        "oranges int, orangesales float, " +
        "topseller varchar(16))";
      String[] rows =
      { "(1, 32248, 3547.28, 18459, 3138.03, 'Maria')",
        "(2, 35009, 3850.99, 18722, 3182.74, 'Bob')",
        "(3, 39393, 4333.23, 18999, 3229.83, 'Joe')",
        "(4, 42001, 4620.11, 19333, 3286.61, 'Maria')" };
      Connection connection =
        DatabaseUtilities.createTable(driver, url,
                                      username, password,
                                      "fruits", format, rows,
                                      false);
      // Sprawdzenie czy tabela została poprawnie utworzona.
      // W celu zwiększenia efektywności wykorzystywane
      // jest to samo połączenie.
      DatabaseUtilities.printTable(connection, "fruits",
                                   11, true);
    }

    private static void printUsage() {
       System.out.println("Użycie: FruitCreation komputer nazwaBazyDanych " +
                          "nazwaUżytkownika hasło oracle|sybase.");
    }
  }
```

Chciałem podać jeszcze jedną informację przeznaczoną dla osób, które do tej pory nie zetknęły się jeszcze z pakietami. Otóż, klasa `FruitTest` należy do pakietu `coreservlets`, a zatem jej plik klasowy będzie przechowywany w katalogu o nazwie *coreservlets*. Przed kompilacją tej klasy, do zmiennej środowiskowej `CLASSPATH` dodałem nazwę katalogu zawierającego katalog *coreservlets* (pliki JAR zawierające sterowniki JDBC także powinne się znajdować w katalogu, którego nazwa została podana w zmiennej środowiskowej `CLASSPATH`). Dzięki temu mogę skompilować klasę `FruitTest` wydając polecenie `javac FruitTest.java` z poziomu podkatalogu *coreservlets*. Jednak aby uruchomić przykład, muszę się posłużyć pełną nazwą pakietu — `java` **`coreservlets`**`.FuitTest` . . . .

18.3 Narzędzia ułatwiające korzystanie z JDBC

W wielu aplikacjach, uzyskanych wyników nie trzeba przetwarzać wiersz po wierszu. Na przykład, w serwletach oraz dokumentach JSP bardzo często stosuje się rozwiązanie polegające na sformatowaniu wszystkich wyników (traktując je jako łańcuchy znaków) i wyświetleniu ich w postaci tabeli HTML (patrz podrozdziały 18.4 oraz 18.5), arkusza kalkulacyjnego programu Excel (patrz podrozdział 11.2) lub informacji rozmieszczonych w różnych miejscach dokumentu HTML. W takich sytuacjach można sobie ułatwić przetwarzanie wyników, tworząc metody które pobierają i zapamiętują całą zawartość obiektu `RecordSet` umożliwiając jej późniejsze wykorzystanie.

W tej części rozdziału przedstawiłem dwie klasy, które udostępniają wspomniane wcześniej możliwości funkcjonalne, jak również dodatkowe metody służące do formatowania, wyświetlania oraz tworzenia tabel. Podstawowa klasa, o nazwie `DatabaseUtilities`, implementuje cztery, przedstawione poniższej metody statyczne ułatwiające wykonywanie często spotykanych zadań:

1. **`getQueryResults`**

   Ta metoda nawiązuje połączenie z bazą danych, wykonuje zapytanie, pobiera wszystkie zwrócone wiersze w postaci tablic łańcuchów znaków i zapisuje je wewnątrz obiektu klasy `DBResult` (patrz listing 18.7). W obiekcie `DBResult` metoda ta zapisuje także nazwę oprogramowania serwera bazy danych, numer wersji serwera, nazwy wszystkich kolumn oraz kopię obiektu `Connection`. Dostępne są dwie wersje metody `getQueryResults` — pierwsza z nich tworzy nowe połączenie z bazą danych, a druga wykorzystuje istniejące połączenie.
2. **`createTable`**

Ta metoda, na podstawie nazwy tabeli, łańcucha znaków określającego formaty kolumn oraz tablicy łańcuchów znaków określających wartości poszczególnych wierszy, nawiązuje połączenie z bazą danych, usuwa istniejącą wersję wskazanej tabeli, wykonuje polecenie `CREATE TABLE` o określonej postaci, a następnie serię poleceń `INSERT INTO`, które wstawiają do nowej tabeli podane wiersze danych. Także ta metoda jest dostępna w dwóch wersjach, z których pierwsza tworzy nowe połączenie, a druga wykorzystuje połączenie już istniejące.

3. **`printTable`**

   Ta metoda dysponując nazwą tabeli nawiązuje połączenie ze wskazaną bazą danych, pobiera wszystkie wiersze określonej tabeli i wyświetla je przy użyciu standardowego strumienia wyjściowego. Metoda pobiera wyniki, zamieniając podaną nazwę tabeli na polecenie SQL o postaci `SELECT * FROM` *`nazwaTabeli`* i przekazując je jako argument wywołania metody `getQueryResults`.

4. **`printTableData`**

   Ta metoda dysponując obiektem klasy `DBResults` zwróconym w wyniku wykonania zapytania SQL, wyświetla uzyskane wyniki przy użyciu standardowego strumienia wyjściowego. Metoda ta jest stosowana w metodzie `printTable`, lecz można jej używać w celach testowych do wyświetlania dowolnych wyników.

Główny kod klays został przedstawiony na listingu 18.6, natomiast listing 18.7 zawiera kod pomocniczej klasy `DBResults`, która przechowuje wszystkie uzyskane wyniki i zwraca je w formie tablic łańcuchów znaków (metoda `getRow`) lub w postaci tabel HTML (metoda `toHTMLTable`). Przedstawione poniżej, dwa przykładowe polecenia wykonują zapytanie SQL, pobierają jego wyniki i wyświetlają w postaci tabeli HTML, której nagłówki zawierają nazwy kolumn tabeli i są wyświetlone na jasnoniebieskim tle.

```
DBResults wyniki =
  DatabaseUtilities.getQueryResults(sterownik, url,
                                    nazwaUzytkownika, haslo,
                                    zapytanie, true);
out.println(wyniki.toHTMLTable("CYAN"));
```

Tabele HTML mogą spełniać podwójną rolę i reprezentować arkusze kalkulacyjne programu Microsoft Excel (patrz podrozdział 11.2), a zatem metoda `toHTMLTable` udostępnia niezwykle prostą metodę prezentowania wyników zapytań zarówno w postaci tabel HTML jak i arkuszy kalkulacyjnych.

Pamiętaj, że kody źródłowe obu prezentowanych tu klas (`DatabaseUtilities` oraz `DBResult`), podobnie zresztą jak kod źródłowe wszystkich pozostałych przykładów prezentowanych w niniejszej książce, można skopiować z serwera FTP Wydawnictwa HELION — *ftp://ftp.helion.pl/przyklady/jsjsp.zip* i stosować lub modyfikować bez żadnych ograniczeń.

Listing 18.6 DatabaseUtilities.java

```
package coreservlets;

import java.sql.*;

public class DatabaseUtilities {

  /** Nawiązuje połączenie z bazą danych, wykonuje podane
   *  zapytanie i zapisuje uzyskane wyniki w obiekcie DBRresults.
   *  Jeśli połączenie z bazą danych zostanie otworzone (określa
   *  to argument "close"), to będzie można je pobrać przy użyciu
   *  metody DBResults.getConnection.
   */

  public static DBResults getQueryResults(String driver,
                                          String url,
                                          String username,
```

```
                                        String password,
                                        String query,
                                        boolean close) {
  try {
    Class.forName(driver);
    Connection connection =
      DriverManager.getConnection(url, username, password);
    return(getQueryResults(connection, query, close));
  } catch(ClassNotFoundException cnfe) {
    System.err.println("Błąd ładowania sterownika: " + cnfe);
    return(null);
  } catch(SQLException sqle) {
    System.err.println("Błąd przy nawiązywaniu połączenia: " + sqle);
    return(null);
  }
}

/** Pobiera wyniki podobnie jak w poprzedniej metodzie, jednak
 *  nie tworzy nowego połączenia a wykorzystuje połączenie już
 *  istniejące.
 */

public static DBResults getQueryResults(Connection connection,
                                        String query,
                                        boolean close) {
  try {
    DatabaseMetaData dbMetaData = connection.getMetaData();
    String productName =
      dbMetaData.getDatabaseProductName();
    String productVersion =
      dbMetaData.getDatabaseProductVersion();
    Statement statement = connection.createStatement();
    ResultSet resultSet = statement.executeQuery(query);
    ResultSetMetaData resultsMetaData =
      resultSet.getMetaData();
    int columnCount = resultsMetaData.getColumnCount();
    String[] columnNames = new String[columnCount];
    // Indeksy kolumn rozpoczynają się od 1 (jak w SQL-u)
    // a nie od 0 (jak w języku Java).
    for(int i=1; i<columnCount+1; i++) {
      columnNames[i-1] =
        resultsMetaData.getColumnName(i).trim();
    }
    DBResults dbResults =
      new DBResults(connection, productName, productVersion,
                    columnCount, columnNames);
    while(resultSet.next()) {
      String[] row = new String[columnCount];
      // Indeksy z obiekcie ResultSet rozpoczynają się od 0.
      for(int i=1; i<columnCount+1; i++) {
        String entry = resultSet.getString(i);
        if (entry != null) {
          entry = entry.trim();
        }
        row[i-1] = entry;
      }
      dbResults.addRow(row);
    }
    if (close) {
      connection.close();
    }
    return(dbResults);
  } catch(SQLException sqle) {
    System.err.println("Błąd przy nawiązywaniu połączenia: " + sqle);
    return(null);
  }
}

/** Tworzy tabelę o określonym formacie i zapisuje w niej
 *  podane wiersze danych
 */

public static Connection createTable(String driver,
                                     String url,
                                     String username,
                                     String password,
                                     String tableName,
```

```
                                    String tableFormat,
                                    String[] tableRows,
                                    boolean close) {
  try {
    Class.forName(driver);
    Connection connection =
      DriverManager.getConnection(url, username, password);
    return(createTable(connection, username, password,
                       tableName, tableFormat,
                       tableRows, close));
  } catch(ClassNotFoundException cnfe) {
    System.err.println("Błąd ładowania sterownika: " + cnfe);
    return(null);
  } catch(SQLException sqle) {
    System.err.println("Błąd przy nawiązywaniu połączenia: " + sqle);
    return(null);
  }
}

/** Podobna do poprzedniej metody, lecz używa istniejącego połączenia. */

public static Connection createTable(Connection connection,
                                    String username,
                                    String password,
                                    String tableName,
                                    String tableFormat,
                                    String[] tableRows,
                                    boolean close) {
  try {

    Statement statement = connection.createStatement();
    // Usuwa aktualną tabelę jeśli taka istnieje, lecz nie zgłasza błędów
    // jeśli tabeli nie ma. Do tego celu służy osobny blok try/catch.
    try {
      statement.execute("DROP TABLE " + tableName);
    } catch(SQLException sqle) {}
    String createCommand =
      "CREATE TABLE " + tableName + " " + tableFormat;
    statement.execute(createCommand);
    String insertPrefix =
      "INSERT INTO " + tableName + " VALUES";
    for(int i=0; i<tableRows.length; i++) {
      statement.execute(insertPrefix + tableRows[i]);
    }
    if (close) {
      connection.close();
      return(null);
    } else {
      return(connection);
    }
  } catch(SQLException sqle) {
    System.err.println("Błąd przy tworzeniu tabeli: " + sqle);
    return(null);
  }
}

public static void printTable(String driver,
                              String url,
                              String username,
                              String password,
                              String tableName,
                              int entryWidth,
                              boolean close) {
  String query = "SELECT * FROM " + tableName;
  DBResults results =
    getQueryResults(driver, url, username,
                    password, query, close);
  printTableData(tableName, results, entryWidth, true);
}

/** Wyświetla całą zawartość tabeli. Każdy element zostanie
 *  wyświetlony w kolumnie o szerokości "entryWidth" znaków,
 *  a zatem należy podać wartość, która będzie co najmniej równa
 *  długości najdłuższego łańcucha znaków.
 */

public static void printTable(Connection connection,
```

```
                              String tableName,
                              int entryWidth,
                              boolean close) {
  String query = "SELECT * FROM " + tableName;
  DBResults results =
    getQueryResults(connection, query, close);
  printTableData(tableName, results, entryWidth, true);
}

public static void printTableData(String tableName,
                                  DBResults results,
                                  int entryWidth,
                                  boolean printMetaData) {
  if (results == null) {
    return;
  }
  if (printMetaData) {
    System.out.println("Baza danych: " +
                       results.getProductName());
    System.out.println("Wersja: " +
                       results.getProductVersion());
    System.out.println();
  }
  System.out.println(tableName + ":");
  String underline =
    padString("", tableName.length()+1, "=");
  System.out.println(underline);
  int columnCount = results.getColumnCount();
  String separator =
    makeSeparator(entryWidth, columnCount);
  System.out.println(separator);
  String row = makeRow(results.getColumnNames(), entryWidth);
  System.out.println(row);
  System.out.println(separator);
  int rowCount = results.getRowCount();
  for(int i=0; i<rowCount; i++) {
    row = makeRow(results.getRow(i), entryWidth);
    System.out.println(row);
  }
  System.out.println(separator);
}

// Łańcuch znaków postaci "|  xxx |  xxx |  xxx |"

private static String makeRow(String[] entries,
                              int entryWidth) {
  String row = "|";
  for(int i=0; i<entries.length; i++) {
    row = row + padString(entries[i], entryWidth, " ");
    row = row + " |";
  }
  return(row);
}

// Łańcuch znaków postaci "+------+------+------+"

private static String makeSeparator(int entryWidth,
                                    int columnCount) {
  String entry = padString("", entryWidth+1, "-");
  String separator = "+";
  for(int i=0; i<columnCount; i++) {
    separator = separator + entry + "+";
  }
  return(separator);
}

private static String padString(String orig, int size,
                                String padChar) {
  if (orig == null) {
    orig = "<null>";
  }
  // Używa obiektu StringBuffer, a nie wielokrotnej konkatenacji
  // łańcuchów znaków, aby uniknąć tworzenia zbyt wielu tymczasowych
  // obiektów klasy String..
  StringBuffer buffer = new StringBuffer("");
  int extraChars = size - orig.length();
  for(int i=0; i<extraChars; i++) {
```

```
      buffer.append(padChar);
    }
    buffer.append(orig);
    return(buffer.toString());
  }
}
```

Listing 18.7 DBResults.java

```
package coreservlets;

import java.sql.*;
import java.util.*;

/** Klasa służąca do przechowywania pełnych wyników zwróconych
 *  zapytanie SQL. Klasa ta różni się od klasy ResultSet pod
 *  kilkoma względami:
 *  <UL>
 *    <LI>ResultSet nie koniecznie zawiera wszystkie informacje;
 *        w przypadku próby pobrania dalszych wierszy wyników,
 *        następuje ponowne połączenie i pobranie wyników z bazy
 *        danych.
 *    <LI>Ta klasa przechowuje wyniki w tablicach, w formie
 *        łańcuchów znaków.
 *    <LI>Ta klasa zawiera informacje przechowywany w DatabaseMetaData
 *        (nazwę oprogramowania serwera bazy danych i numer wersji)
 *        oraz ResultSetMetaData (nazwy kolumn).
 *    <LI>Ta klasa udostępnia metodę toHTMLTable która zwraca
 *        wszystkie informacje przedstawione w formie jednego
 *        długiego łańcucha znaków zawierającego tabelę HTML.
 *  </UL>
 */

public class DBResults {
  private Connection connection;
  private String productName;
  private String productVersion;
  private int columnCount;
  private String[] columnNames;
  private Vector queryResults;
  String[] rowData;

  public DBResults(Connection connection,
                   String productName,
                   String productVersion,
                   int columnCount,
                   String[] columnNames) {
    this.connection = connection;
    this.productName = productName;
    this.productVersion = productVersion;
    this.columnCount = columnCount;
    this.columnNames = columnNames;
    rowData = new String[columnCount];
    queryResults = new Vector();
  }

  public Connection getConnection() {
    return(connection);
  }

  public String getProductName() {
    return(productName);
  }

  public String getProductVersion() {
    return(productVersion);
  }

  public int getColumnCount() {
    return(columnCount);
  }

  public String[] getColumnNames() {
    return(columnNames);
  }
```

```
  public int getRowCount() {
    return(queryResults.size());
  }

  public String[] getRow(int index) {
    return((String[])queryResults.elementAt(index));
  }

  public void addRow(String[] row) {
    queryResults.addElement(row);
  }

  /** Wyświetl wyniki w formie tabeli HTML, w której
   *  nagłówkami są nazwy kolumn, a pozostałe dane
   *  są zapisane w zwyczajnych komórkach tabeli.
   */

  public String toHTMLTable(String headingColor) {
    StringBuffer buffer =
      new StringBuffer("<TABLE BORDER=1>\n");
    if (headingColor != null) {
      buffer.append("  <TR BGCOLOR=\"" + headingColor +
                    "\">\n    ");
    } else {
      buffer.append("  <TR>\n    ");
    }
    for(int col=0; col<getColumnCount(); col++) {
      buffer.append("<TH>" + columnNames[col]);
    }
    for(int row=0; row<getRowCount(); row++) {
      buffer.append("\n  <TR>\n    ");
      String[] rowData = getRow(row);
      for(int col=0; col<getColumnCount(); col++) {
        buffer.append("<TD>" + rowData[col]);
      }
    }
    buffer.append("\n</TABLE>");
    return(buffer.toString());
  }
}
```

18.4 Wykorzystanie narzędzi ułatwiających obsługę JDBC

Teraz przekonajmy się w jaki sposób narzędzie przedstawione w poprzedniej części rozdziału mogą ułatwić pobieranie i prezentację informacji z baz danych. Listing 18.8 przedstawia klasę, która nawiązuje połączenie z bazą danych określoną w wierszu poleceń systemu i wyświetla całą zawartość tabeli *employee*. Listingi 18.9 oraz 18.10 przedstawiają odpowiednio wyniki uzyskane w przypadku wykorzystania bazy danych Oracle oraz Sybase. Listing 18.11 przedstawia podobną klasę, która wykonuje to samo zapytanie lecz prezentuje wyniki w postaci tabeli HTML. Listing 18.12 przedstawia natomiast wynikowy, źródłowy kod HTML. W podrozdziale 18.8. — „Zarządzanie pulami połączeń: Studium zagadnienia", umieszczę taką tabelę HTML na stronie WWW.

Kod JDBC służący do stworzenia tabeli *employees* przedstawiłem na listingu 18.3.

Listing 18.8 EmployeeTest.java

```
package coreservlets;

import java.sql.*;

/** Nawiązuje połączenie z bazą danych Oracle lub Sybase
 *  i wyświetla zawartość tabeli "employees".
 */
```

```
public class EmployeeTest {
  public static void main(String[] args) {
    if (args.length < 5) {
      printUsage();
      return;
    }
    String vendorName = args[4];
    int vendor = DriverUtilities.getVendor(vendorName);
    if (vendor == DriverUtilities.UNKNOWN) {
      printUsage();
      return;
    }
    String driver = DriverUtilities.getDriver(vendor);
    String host = args[0];
    String dbName = args[1];
    String url =
      DriverUtilities.makeURL(host, dbName, vendor);
    String username = args[2];
    String password = args[3];
    DatabaseUtilities.printTable(driver, url,
                                 username, password,
                                 "employees", 12, true);
  }

  private static void printUsage() {
    System.out.println("Użycie: EmployeeTest komputer nazwaBazyDanych " +
                       "nazwaUżytkownika hasło oracle|sybase.");
  }
}
```

Listing 18.9 Wyniki wykonania aplikacji EmployeeTest (połączenie z bazą Oracle działającą w systemie Solaris)

```
Prompt> java coreservlets.EmployeeTest dbhost1.apl.jhu.edu PTE
        hall xxxx oracle
Baza danych: Oracle
Wersja: Oracle7 Server Release 7.2.3.0.0 - Production Release
PL/SQL Release 2.2.3.0.0 - Production

employees:
==========
+-------------+-------------+-------------+-------------+-------------+
|          ID |   FIRSTNAME |    LASTNAME |    LANGUAGE |      SALARY |
+-------------+-------------+-------------+-------------+-------------+
|           1 |         Wye |       Tukay |       COBOL |       42500 |
|           2 |       Britt |        Tell |         C++ |       62000 |
|           3 |         Max |     Manager |        none |       15500 |
|           4 |       Polly |     Morphic |   Smalltalk |       51500 |
|           5 |       Frank |    Function | Common Lisp |       51500 |
|           6 |      Justin |Timecompiler |        Java |       98000 |
|           7 |         Sir |        Vlet |        Java |      114750 |
|           8 |         Jay |        Espy |        Java |      128500 |
+-------------+-------------+-------------+-------------+-------------+
```

Listing 18.10 Wyniki wykonania aplikacji EmployeeTest (połączenie z baza Sybase działającą w systemie Windows NT)

```
Prompt> java coreservlets.EmployeeTest dbhost2.apl.jhu.edu 605741
        hall xxxx sybase
Baza danych: Adaptive Server Anywhere
Wersja: 6.0.2.2188

employees:
==========
+-------------+-------------+-------------+-------------+-------------+
|          id |   firstname |    lastname |    language |      salary |
+-------------+-------------+-------------+-------------+-------------+
|           1 |         Wye |       Tukay |       COBOL |     42500.0 |
|           2 |       Britt |        Tell |         C++ |     62000.0 |
|           3 |         Max |     Manager |        none |     15500.0 |
|           4 |       Polly |     Morphic |   Smalltalk |     51500.0 |
|           5 |       Frank |    Function | Common Lisp |     51500.0 |
|           6 |      Justin |Timecompiler |        Java |     98000.0 |
```

```
|           7 |         Sir |        Vlet |        Java |    114750.0 |
|           8 |         Jay |        Espy |        Java |    128500.0 |
+-------------+-------------+-------------+-------------+-------------+
```

Listing 18.11 EmployeeTest2.java

```
package coreservlets;

import java.sql.*;

/** Nawiązuje połączenie z bazą danych Oracle lub Sybase
 *  i wyświetla całą zawartość tabeli "employees" w formie
 *  tabeli HTML.
 */

public class EmployeeTest2 {
  public static void main(String[] args) {
    if (args.length < 5) {
      printUsage();
      return;
    }
    String vendorName = args[4];
    int vendor = DriverUtilities.getVendor(vendorName);
    if (vendor == DriverUtilities.UNKNOWN) {
      printUsage();
      return;
    }
    String driver = DriverUtilities.getDriver(vendor);
    String host = args[0];
    String dbName = args[1];
    String url =
      DriverUtilities.makeURL(host, dbName, vendor);
    String username = args[2];
    String password = args[3];
    String query = "SELECT * FROM employees";
    DBResults results =
      DatabaseUtilities.getQueryResults(driver, url,
                                        username, password,
                                        query, true);
    System.out.println(results.toHTMLTable("CYAN"));
  }

  private static void printUsage() {
    System.out.println("Użycie: EmployeeTest2 komputer nazwaBazyDanych " +
                       "nazwaUżytkownika hasło oracle|sybase.");
  }
}
```

Listing 18.12 Wyniki wykonania programu EmployeeTest2 (połączenie z bazą Sybase działającą w systemie Windows NT)

```
Prompt> java coreservlets.EmployeeTest2 dbhost2 605741
        hall xxxx sybase
<TABLE BORDER=1>
  <TR BGCOLOR="CYAN">
    <TH>id<TH>firstname<TH>lastname<TH>language<TH>salary
  <TR>
    <TD>1<TD>Wye<TD>Tukay<TD>COBOL<TD>42500.0
  <TR>
    <TD>2<TD>Britt<TD>Tell<TD>C++<TD>62000.0
  <TR>
    <TD>3<TD>Max<TD>Manager<TD>none<TD>15500.0
  <TR>
    <TD>4<TD>Polly<TD>Morphic<TD>Smalltalk<TD>51500.0
  <TR>
    <TD>5<TD>Frank<TD>Function<TD>Common Lisp<TD>51500.0
  <TR>
    <TD>6<TD>Justin<TD>Timecompiler<TD>Java<TD>98000.0
  <TR>
    <TD>7<TD>Sir<TD>Vlet<TD>Java<TD>114750.0
  <TR>
    <TD>8<TD>Jay<TD>Espy<TD>Java<TD>128500.0
</TABLE>
```

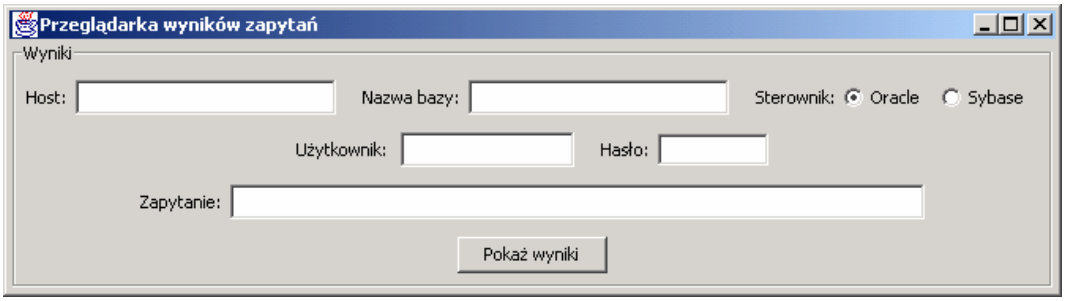
Listing 18.13 EmployeeCreation.java

```
package coreservlets;

import java.sql.*;

/** Tworzy prostą tabelę "employees" przy użyciu klasy
 *  DatabaseUtilities.
 */

public class EmployeeCreation {
  public static Connection createEmployees(String driver,
                                           String url,
                                           String username,
                                           String password,
                                           boolean close) {
    String format =
      "(id int, firstname varchar(32), lastname varchar(32), " +
      "language varchar(16), salary float)";
    String[] employees =
      {"(1, 'Wye', 'Tukay', 'COBOL', 42500)",
       "(2, 'Britt', 'Tell',   'C++',   62000)",
       "(3, 'Max',  'Manager', 'none',  15500)",
       "(4, 'Polly', 'Morphic', 'Smalltalk', 51500)",
       "(5, 'Frank', 'Function', 'Common Lisp', 51500)",
       "(6, 'Justin', 'Timecompiler', 'Java', 98000)",
       "(7, 'Sir', 'Vlet', 'Java', 114750)",
       "(8, 'Jay', 'Espy', 'Java', 128500)" };
    return(DatabaseUtilities.createTable(driver, url,
                                         username, password,
                                         "employees",
                                         format, employees,
                                         close));
  }

  public static void main(String[] args) {
    if (args.length < 5) {
      printUsage();
      return;
    }
    String vendorName = args[4];
    int vendor = DriverUtilities.getVendor(vendorName);
    if (vendor == DriverUtilities.UNKNOWN) {
      printUsage();
      return;
    }
    String driver = DriverUtilities.getDriver(vendor);
    String host = args[0];
    String dbName = args[1];
    String url =
      DriverUtilities.makeURL(host, dbName, vendor);
    String username = args[2];
    String password = args[3];
    createEmployees(driver, url, username, password, true);
  }

  private static void printUsage() {
    System.out.println("Użycie: EmployeeCreation komputer nazwaBazyDanych " +
                       "nazwaUżytkownika hasło oracle|sybase.");
  }
}
```

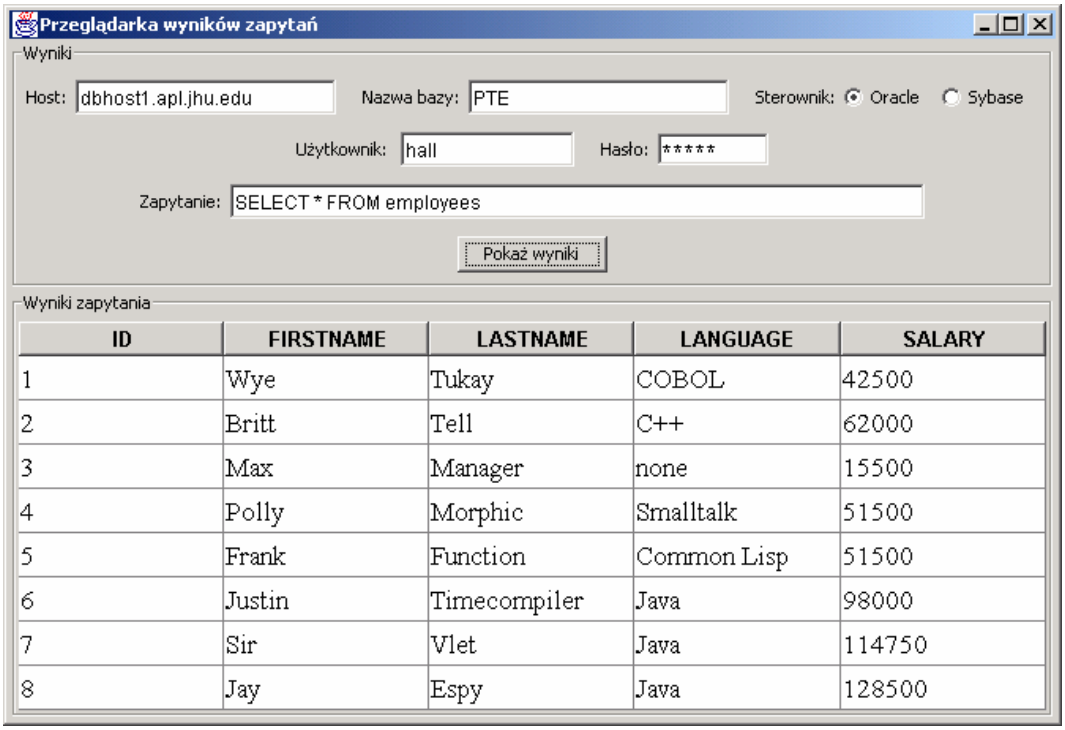
18.5 Interaktywna przeglądarka zapytań

Jak na razie wszystkie informacje pobierane z baz danych były uzyskiwane w wyniku wykonania zapytań, których postać była już znana podczas pisania programu. Jednak w wielu praktycznie wykorzystywanych aplikacjach, zapytania są tworzone na podstawie informacji podawanych przez użytkowników podczas działania programu.

Czasami zapytanie ma z góry określony format, choć niektóre jego wartości się zmieniają. W takich przypadkach należy stosować wstępnie przygotowywane zapytania, które opiszę szczegółowo w podrozdziale 18.6. Jednak w innych przypadkach zmienia się nawet format całego zapytania. Na szczęście nie stanowi to żadnego problemu, gdyż dzięki klasie `ResultSetMetaData` można określić ilość, nazwy oraz typy kolumn przechowywanych w obiekcie `RecordSet` (co opisałem w podrozdziale 18.1. — „Podstawowe etapy wykorzystania JDBC"). W rzeczywistości, pomocnicze narzędzia przedstawione na listingu 18.6 przechowują „meta dane" w obiekcie klasy `DBResults` zwracanym przez metodę `showQueryData`. Dzięki tym „meta danym" implementacja interaktywnej, graficznej przeglądarki zapytań staje się bardzo prostym zadaniem (patrz rysunki 10 18.1 do 18.5). Kod źródłowy przeglądarki prezentowanej na tych rysunkach przedstawiłem w następnym podrozdziale.

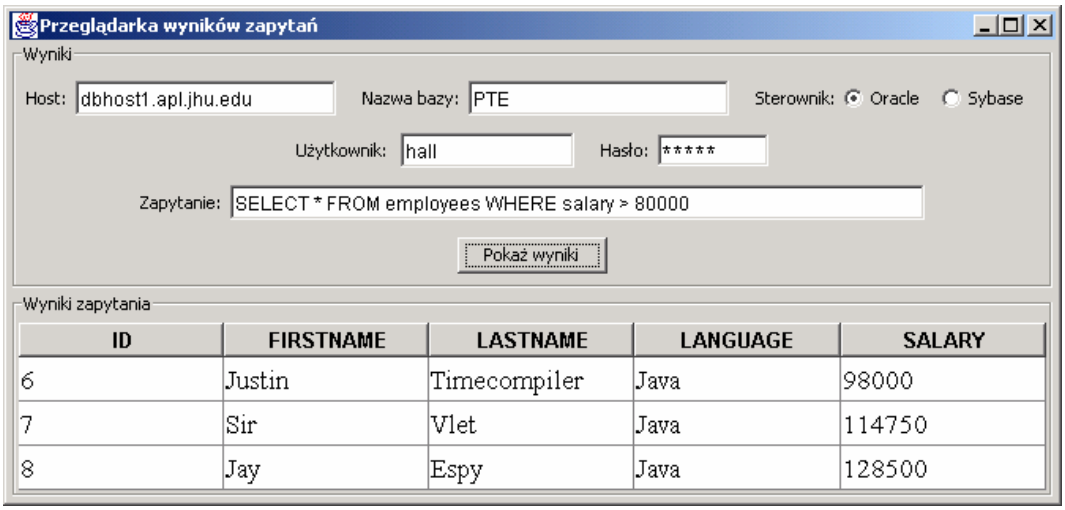


Rysunek 18.1 Początkowy wygląd przeglądarki zapytań

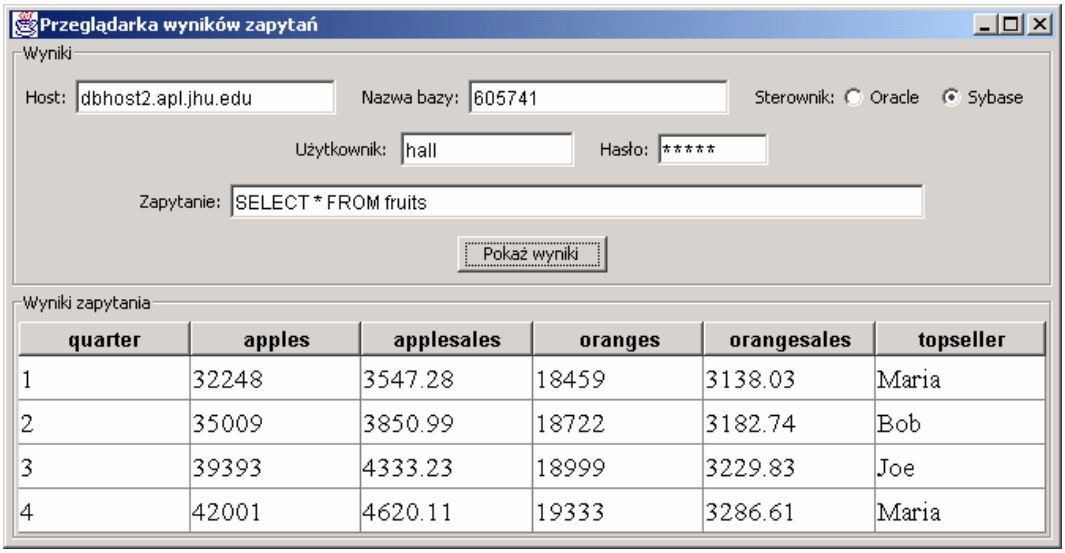


ID	FIRSTNAME	LASTNAME	LANGUAGE	SALARY
1	Wye	Tukay	COBOL	42500
2	Britt	Tell	C++	62000
3	Max	Manager	none	15500
4	Polly	Morphic	Smalltalk	51500
5	Frank	Function	Common Lisp	51500
6	Justin	Timecompiler	Java	98000
7	Sir	Vlet	Java	114750
8	Jay	Espy	Java	128500

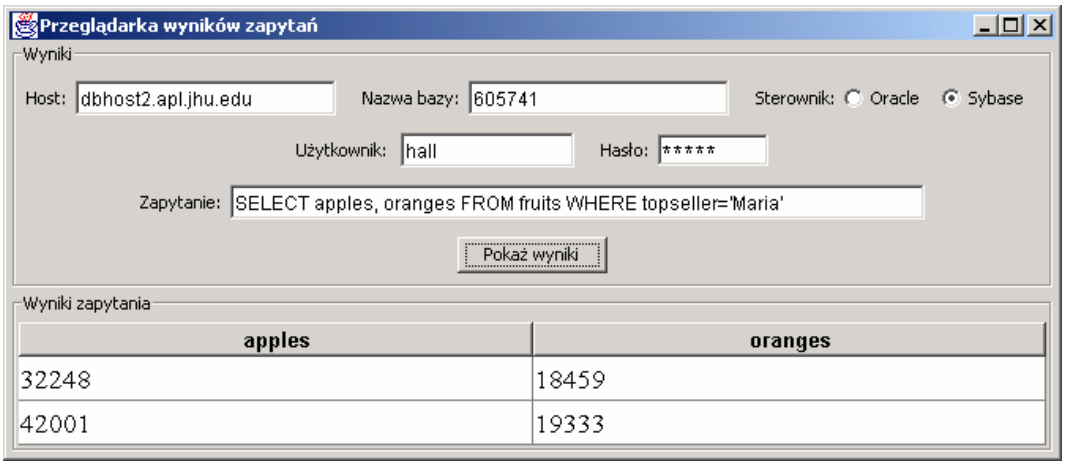
Rysunek 18.2 Wygląd przeglądarki po wyświetleniu całej zawartości tabeli employees pobranej z bazy danych Oracle



Rysunek 18.3 Wygląd przeglądarki po wyświetleniu całej części zawartości tabeli employees pobranej z bazy danych Oracle



Rysunek 18.4 Wygląd przeglądarki po wyświetleniu całej zawartości tabeli fruits pobranej z bazy danych Sybase



Rysunek 18.5 Przeglądarka zapytań, po przesłaniu zapytania dotyczącego tylko dwóch kolumn tabeli fruits z bazy danych Sybase

Kod przeglądarki zapytań

Stworzenie aplikacji o wyglądzie przedstawionym na rysunkach od 18.1 do 18.5 jest stosunkowo proste. W rzeczywistości, dysponując narzędziami ułatwiającymi korzystanie z baz danych, przedstawionymi w poprzednim podrozdziale, znacznie więcej wysiłku należy poświęcić na stworzenie odpowiedniego interfejsu użytkownika, niż na zaimplementowanie komunikacji z bazą danych. Pełny kod aplikacji przedstawiłem na listingu 18.14, poniżej jednak pokrótce omówię cały proces wykonywany w momencie gdy użytkownik kliknie przycisk *Pokaż wyniki*.

W pierwszej kolejności program odczytuje nazwę komputera, numer portu, nazwę bazy danych, użytkownika, hasło oraz typ używanego sterownika. Informacje te są pobierane z odpowiednich pól graficznego interfejsu użytkownika. Następnie program wykonuje zapytanie i zapamiętuje uzyskane wyniki. W tym celu używana jest metoda `getQueryResults`:

```
DBResults results =
  DatabaseUtilities.getQueryResults(driver, url,
                                    username, password,
                                    query, true);
```

Następnie program przekazuje uzyskane wyniki do *modelu tabeli* (przedstawionego na listingu 18.15). Jeśli nie spotkałeś się jeszcze z biblioteką Swing, to przyda Ci się wyjaśnienie, że *model tabeli* spełnia rolę spoiwa łączącego tabele `JTable` z wyświetlanymi w nich informacjami.

```
DBResultsTableModel model = new DBResultsTableModel(results);
JTable table = new JTable(model);
```

I w końcu, program wyświetla tabelę w dolnym regionie obiektu `JFrame` i wywołuje metodę `pack`, informując w ten sposób obiekt `JFrame`, że powinien uaktualnić swoje wymiary dopasowując je do wymiarów tabeli.

Listing 18.14 QueryViewer.java

```
package coreservlets;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.*;

/** Interaktywna przeglądarka wyników zapytań. Nawiązuje połączenie
 *  z podaną bazą danych Oracle lub Sybase, wykonuje zapytanie,
 *  i przedstawia uzyskane wyniki w formie tabeli JTable.
 */

public class QueryViewer extends JFrame
                         implements ActionListener{
  public static void main(String[] args) {
    new QueryViewer();
  }

  private JTextField hostField, dbNameField,
                     queryField, usernameField;
  private JRadioButton oracleButton, sybaseButton;
  private JPasswordField passwordField;
  private JButton showResultsButton;
  private Container contentPane;
  private JPanel tablePanel;

  public QueryViewer () {
    super("Przeglądarka wyników zapytań");
    WindowUtilities.setNativeLookAndFeel();
    addWindowListener(new ExitListener());
    contentPane = getContentPane();
    contentPane.add(makeControlPanel(), BorderLayout.NORTH);
    pack();
    setVisible(true);
  }

  /** Po naciśnięciu przycisku "Pokaż wyniki" lub klawisza
   *  RETURN gdy miejsce wprowadzania znajduje się w polu
   *  tekstowym służącym do podawania zapytań, zapytanie jest
```

```
   *  wykonywane, a jego wyniki zostają zapisane w tabeli JTable.
   *  Następnie wielkość okna programu jest modyfikowana, tak
   *  aby można w nim było wyświetlić tabelę z wynikami.
   */

  public void actionPerformed(ActionEvent event) {
    String host = hostField.getText();
    String dbName = dbNameField.getText();
    String username = usernameField.getText();
    String password =
      String.valueOf(passwordField.getPassword());
    String query = queryField.getText();
    int vendor;
    if (oracleButton.isSelected()) {
      vendor = DriverUtilities.ORACLE;
    } else {
      vendor = DriverUtilities.SYBASE;
    }
    if (tablePanel != null) {
      contentPane.remove(tablePanel);
    }
    tablePanel = makeTablePanel(host, dbName, vendor,
                                username, password,
                                query);
    contentPane.add(tablePanel, BorderLayout.CENTER);
    pack();
  }

  // Wykonuje zapytanie i umieszcza wyniki w tabeli
  // JTable, która z kolei jest wyświetlana w panelu JPanel.

  private JPanel makeTablePanel(String host,
                                String dbName,
                                int vendor,
                                String username,
                                String password,
                                String query) {
    String driver = DriverUtilities.getDriver(vendor);
    String url = DriverUtilities.makeURL(host, dbName, vendor);
    DBResults results =
      DatabaseUtilities.getQueryResults(driver, url,
                                        username, password,
                                        query, true);
    JPanel panel = new JPanel(new BorderLayout());
    if (results == null) {
      panel.add(makeErrorLabel());
      return(panel);
    }
    DBResultsTableModel model =
      new DBResultsTableModel(results);
    JTable table = new JTable(model);
    table.setFont(new Font("Serif", Font.PLAIN, 17));
    table.setRowHeight(28);
    JTableHeader header = table.getTableHeader();
    header.setFont(new Font("SansSerif", Font.BOLD, 13));
    panel.add(table, BorderLayout.CENTER);
    panel.add(header, BorderLayout.NORTH);
    panel.setBorder
      (BorderFactory.createTitledBorder("Wyniki zapytania"));
    return(panel);
  }

  // Panel zawierający pola tekstowe, pola wyboru oraz przycisk

  private JPanel makeControlPanel() {
    JPanel panel = new JPanel(new GridLayout(0, 1));
    panel.add(makeHostPanel());
    panel.add(makeUsernamePanel());
    panel.add(makeQueryPanel());
    panel.add(makeButtonPanel());
    panel.setBorder
      (BorderFactory.createTitledBorder("Wyniki"));
    return(panel);
  }

  // Panel zawierający pola tekstowe określające komputer oraz nazwę
  // bazy danych, oraz przycisk opcji służący określenia używanego
```

```
  // sterownika. Umieszczany w panelu kontrolnym.

  private JPanel makeHostPanel() {
    JPanel panel = new JPanel();
    panel.add(new JLabel("Host:"));
    hostField = new JTextField(15);
    panel.add(hostField);
    panel.add(new JLabel("    Nazwa bazy:"));
    dbNameField = new JTextField(15);
    panel.add(dbNameField);
    panel.add(new JLabel("    Sterownik:"));
    ButtonGroup vendorGroup = new ButtonGroup();
    oracleButton = new JRadioButton("Oracle", true);
    vendorGroup.add(oracleButton);
    panel.add(oracleButton);
    sybaseButton = new JRadioButton("Sybase");
    vendorGroup.add(sybaseButton);
    panel.add(sybaseButton);
    return(panel);
  }

  // Panel zawierający pola tekstowe służące do określenia nazwy
  // użytkownika oraz hasła. Wyświetlany na panelu kontrolnym.

  private JPanel makeUsernamePanel() {
    JPanel panel = new JPanel();
    usernameField = new JTextField(10);
    passwordField = new JPasswordField(10);
    panel.add(new JLabel("Użytkownik: "));
    panel.add(usernameField);
    panel.add(new JLabel("    Hasło:"));
    panel.add(passwordField);
    return(panel);
  }

  // Panel zawierający pole tekstowe służące do podawania zapytań.
  // Wyświetlany na panelu sterującym.

  private JPanel makeQueryPanel() {
    JPanel panel = new JPanel();
    queryField = new JTextField(40);
    queryField.addActionListener(this);
    panel.add(new JLabel("Zapytanie:"));
    panel.add(queryField);
    return(panel);
  }

  // Panel zawierający przycisk "Pokaż wyniki".
  // Wyświetlany na panelu sterującym.

  private JPanel makeButtonPanel() {
    JPanel panel = new JPanel();
    showResultsButton = new JButton("Pokaż wyniki");
    showResultsButton.addActionListener(this);
    panel.add(showResultsButton);
    return(panel);
  }

  // Wyświetla ostrzeżenie w przypadku podania błędnego zapytania.

  private JLabel makeErrorLabel() {
    JLabel label = new JLabel("Brak wyników", JLabel.CENTER);
    label.setFont(new Font("Serif", Font.BOLD, 36));
    return(label);
  }
}
```

Listing 18.15 DBResultsTableModel.java

```
package coreservlets;

import javax.swing.table.*;

/** Prosta klasa dostarczająca tabeli JTable informacji
 *  o tym jak należy pobierać odpowiednie dane z obiektów
 *  DBResults (które są używane do przechowywania wyników
```

```
 *  zapytań).
 */

public class DBResultsTableModel extends AbstractTableModel {
  private DBResults results;

  public DBResultsTableModel(DBResults results) {
    this.results = results;
  }

  public int getRowCount() {
    return(results.getRowCount());
  }

  public int getColumnCount() {
    return(results.getColumnCount());
  }

  public String getColumnName(int column) {
    return(results.getColumnNames()[column]);
  }

  public Object getValueAt(int row, int column) {
    return(results.getRow(row)[column]);
  }
}
```

Listing 18.16 WindowUtilities.java

```
package coreservlets;

import javax.swing.*;
import java.awt.*;

/** Kilka narzędzi ułatwiających stosowanie okien
 *  w pakiecie Swing.
 */

public class WindowUtilities {

  /** Informuje system że należy używać rodzimego wyglądu
   *  (ang. "look and feel"), podobnie jak w wersjach wcześniejszych
   *  W pozostałych przypadkach używany jest wygląd "Metal" (Java)
   */

  public static void setNativeLookAndFeel() {
    try {
      UIManager.setLookAndFeel
        (UIManager.getSystemLookAndFeelClassName());
    } catch(Exception e) {
      System.out.println("Błąd przy wyborze wyglądu: " + e);
    }
  }

  public static void setJavaLookAndFeel() {
    try {
      UIManager.setLookAndFeel
        (UIManager.getCrossPlatformLookAndFeelClassName());
    } catch(Exception e) {
      System.out.println("Błąd przy wyborze wyglądu: " + e);
    }
  }

  public static void setMotifLookAndFeel() {
    try {
      UIManager.setLookAndFeel
        ("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
    } catch(Exception e) {
      System.out.println("Błąd przy wyborze wyglądu Motif: " + e);
    }
  }
}
```

Listing 18.17 ExitListener.java

```
package coreservlets;

import java.awt.*;
import java.awt.event.*;

/** Klasa dołączana do obiektu Frame lub JFrame
 *  najwyższego poziomu w aplikacji; umożliwia
 *  zamykanie okna aplikacji.
 */

public class ExitListener extends WindowAdapter {
  public void windowClosing(WindowEvent event) {
    System.exit(0);
  }
}
```

18.6 Przygotowane polecenia (prekompilowane zapytania)

Jeśli masz zamiar wykonywać podobne polecenia SQL wiele razy, to wykorzystanie „przygotowanych" poleceń może być bardziej efektywne, niż posługiwanie się poleceniami „nieprzetworzonymi". Cały pomysł polega na przygotowaniu sparametryzowanego polecenia zapisanego w standardowym formacie, które przed wykonaniem zostanie przesłane do bazy danych i skompilowane. Miejsca polecenia, w których będą umieszczone wartości oznaczane są znakami zapytania. Przy każdym użyciu wstępnie przygotowanego polecenia, określa się wartości oznaczonych wcześniej parametrów. Służy do tego metoda `setXxx` określająca parametr, którego wartość chcesz podać (parametry są określane za pomocą indeksów, o wartościach liczonych poczynając od 1) oraz jego typ (na przykład: `setInt`, `setString`, itp.). Następnie można wywołać metodę `executeQuery` (aby wykonać zapytanie i pobrać wyniki zwrócone w formie obiektu `RecordSet`) lub posłużyć się metodami `execute`/`executeUpdate` (aby zmodyfikować zawartość tabeli), podobnie jak w przypadku standardowych poleceń SQL. Na przykład, gdybyś chciał dać podwyżki wszystkim pracownikom, których dane są zapisane w tabeli *employees*, mógłbyś posłużyć się poniższym fragmentem kodu:

```
Connection polaczenie =
  DriverManager.getConnection(url, uzytkownik, haslo);
String szablon =
  "UPDATE employees SET salary = ? WHERE id = ?";
PreparedStatement polecenie =
  polaczenie.prepareStatement(szablon);
float[] nowePlace = okreslNowePlace();
int[] idPracownikow = pobierzIdPracownikow();
for (int i=0; i<idPracownikow.length; i++) {
  polecenie.setFloat(1, nowePlace[i]);
  polecenie.setInt(2, idPracownikow[i]);
  polecenie.execute();
}
```

Poprawa efektywności jaką można uzyskać dzięki wykorzystaniu przygotowanych poleceń może być bardzo różna — zależy ona od tego jak dobrze serwer obsługuje takie zapytania i jak efektywnie sterownik wykonuje zwyczajne — „nieprzetwarzane" — polecenia. Na przykład, na listingu 18.18 przedstawiłem klasę, która przesyła na serwer 40 różnych przygotowanych zapytań, a następnie wykonuje te same 40 zapytań w „standardowy" sposób. Na komputerze PC łączącym się z Internetem przy użyciu połączenia modemowego o szybkości 28,8 kbps i korzystającego z bazy danych Oracle, wykonanie zapytań przygotowanych zajęło tylko *połowę* czasu koniecznego do wykonania tych samych zapytań w formie „nieprzetworzonej". W tym przypadku wykonanie poleceń przygotowany zajmowało średnio 17.5 sekundy, a „nieprzetworzonych" — 35 sekund. W przypadku wykorzystania tej samej bazy danych Oracle oraz szybkiego połączenia poprzez sieć lokalną, wykonanie poleceń przygotowanych zajęło 70 procent czasu koniecznego do wykonania

poleceń „nieprzetworzonych” (odpowiednio, 0.22 sekundy dla poleceń przygotowanych i 0.31 sekundy dla poleceń „nieprzetworzonych”). W przypadku wykorzystania bazy danych Sybase, czas konieczny do wykonania obu rodzajów poleceń był niemal identyczny i to bez względu na szybkość używanego połączenia. Jeśli chcesz sprawdzić efektywność działania swojego systemu komputerowego, skopiuj plik *DriverUtilities.java* z serwera FTP Wydawnictwa HELION (*ftp://ftp.helion.pl/przyklady/jsjsp.zip*), dodaj do niego informacje o używanych sterownikach i wykonaj program *PreparedStatements*.

Listing 18.18 PreparedStatements.java

```
package coreservlets;

import java.sql.*;

/** Przykład umożliwiający przetestowanie różnic w czasie
 *  wykonania zapytania SQL pomiędzy zastosowaniem
 *  nieprzetworzonych zapytań i zapytań przygotowanych.
 *  Uzyskane wyniki w bardzo dużym stopniu zależą od
 *  używanego sterownika i serwera bazy danych, a ich
 *  wartości mogą się znacząco różnić. Przy mojej
 *  konfiguracji systemu, wykonanie przygotowanego zapytania
 *  w bazie danych Oracle, przy użyciu wolnego połączenia
 *  modemowego zajmowało połowę czasu koniecznego do
 *  wykonania tego samego zapytania w wersji nieprzetworzonej;
 *  W przypadku wykorzystania szybkiego połączenia LAN,
 *  wykonanie zapytania przygotowanego zajmowało 70% czasu
 *  koniecznego do wykonania tego samego zapytania w formie
 *  nieprzetworzonej. W przypadku wykorzystania bazy
 *  danych Sybase, uzyskane wyniki nie zależały od
 *  szybkości używanego połączenia.
 */

public class PreparedStatements {
  public static void main(String[] args) {
    if (args.length < 5) {
      printUsage();
      return;
    }
    String vendorName = args[4];
    int vendor = DriverUtilities.getVendor(vendorName);
    if (vendor == DriverUtilities.UNKNOWN) {
      printUsage();
      return;
    }
    String driver = DriverUtilities.getDriver(vendor);
    String host = args[0];
    String dbName = args[1];
    String url =
      DriverUtilities.makeURL(host, dbName, vendor);
    String username = args[2];
    String password = args[3];
    // Użyj "print" tylko po to by potwierdzić, że wszystko jest OK
    // nie używaj przy określaniu czasów wykonywania zapytań.
    boolean print = false;
    if ((args.length > 5) && (args[5].equals("print"))) {
      print = true;
    }
    Connection connection =
      getConnection(driver, url, username, password);
    if (connection != null) {
      doPreparedStatements(connection, print);
      doRawQueries(connection, print);
    }
  }

  private static void doPreparedStatements(Connection conn,
                                           boolean print) {
    try {
      String queryFormat =
        "SELECT lastname FROM employees WHERE salary > ?";
      PreparedStatement statement =
        conn.prepareStatement(queryFormat);
      long startTime = System.currentTimeMillis();
      for(int i=0; i<40; i++) {
```

```
        statement.setFloat(1, i*5000);
        ResultSet results = statement.executeQuery();
        if (print) {
          showResults(results);
        }
      }
      long stopTime = System.currentTimeMillis();
      double elapsedTime = (stopTime - startTime)/1000.0;
      System.out.println("40-krotne wykonanie przygotowanego " +
                         "zapytania zajęło " +
                         elapsedTime + " sekund.");
    } catch(SQLException sqle) {
      System.out.println("Błąd przy wykonywaniu zapytania: " + sqle);
    }
  }

  public static void doRawQueries(Connection conn,
                                  boolean print) {
    try {
      String queryFormat =
        "SELECT lastname FROM employees WHERE salary > ";
      Statement statement = conn.createStatement();
      long startTime = System.currentTimeMillis();
      for(int i=0; i<40; i++) {
        ResultSet results =
          statement.executeQuery(queryFormat + (i*5000));
        if (print) {
          showResults(results);
        }
      }
      long stopTime = System.currentTimeMillis();
      double elapsedTime = (stopTime - startTime)/1000.0;
      System.out.println("40-krotne wykonanie zapytania zajęło " +
                         elapsedTime + " sekund.");
    } catch(SQLException sqle) {
      System.out.println("Błąd przy wykonywaniu zapytania: " + sqle);
    }
  }

  private static void showResults(ResultSet results)
      throws SQLException {
    while(results.next()) {
      System.out.print(results.getString(1) + " ");
    }
    System.out.println();
  }

  private static Connection getConnection(String driver,
                                          String url,
                                          String username,
                                          String password) {
    try {
      Class.forName(driver);
      Connection connection =
        DriverManager.getConnection(url, username, password);
      return(connection);
    } catch(ClassNotFoundException cnfe) {
      System.err.println("Błąd ładowania sterownika: " + cnfe);
      return(null);
    } catch(SQLException sqle) {
      System.err.println("Błąd przy nawiązywaniu połączenia: " + sqle);
      return(null);
    }
  }

  private static void printUsage() {
    System.out.println("Użycie: PreparedStatements komputer " +
                       "nazwaBazyDanych nazwaUżytkownika hasło " +
                       "oracle|sybase [print].");
  }
}
```

18.7 Zarządzanie pulami połączeń

Otwieranie połączeń z bazą danych jest procesem czasochłonnym. W przypadku krótkich zapytań, znacznie więcej czasu może zabrać nawiązanie połączenia z bazą niż samo wykonanie zapytani i pobranie wyników. Właśnie z tego względu, w aplikacjach które wielokrotnie nawiązują połączenia z tą samą bazą danych, sensownym rozwiązaniem jest wielokrotne wykorzystywanie tych samych obiektów `Connection`. W tej części rozdziału przedstawię klasę służącą do *zarządzania pulą połączeń* — czyli nawiązania pewnej liczby połączeń z bazą danych i wielokrotnego wykorzystywania ich do łączenia klientów z bazą. Użycie tej klasy w serwletach i dokumentach JSP może dać duże korzyści, gdyż zazwyczaj, z góry wiadomo jaka baza danych będzie używana (na przykład, zostaje ona określona w metodzie `init`). Na przykład, serwlet przedstawiony w podrozdziale 18.8 wykazuje siedmiokrotny wzrost efektywności działania po użyciu w nim klasy zarządzającej pulą połączeń.

Klasa zarządzająca pulą połączeń powinna móc wykonywać następujące czynności:

1) nawiązać połączenia przed ich wykorzystaniem;
2) zarządzać dostępnymi połączeniami;
3) nawiązywać nowe połączenia;
4) oczekiwać aż połączenie stanie się dostępne;
5) zamknąć połączenie gdy będzie to konieczne.

Poniżej opiszę sposób realizacji każdego z tych zadań. Pełny kod źródłowy klasy `ConnectionPool` przedstawiłem na listingu 18.19, można go skopiować z serwera FTP Wydawnictwa HELION (*ftp://ftp.helion.pl/przyklady/jsjsp.zip*), podobnie jak kody źródłowe wszystkich pozostałych przykładów przedstawionych w niniejszej książce.

1. **Nawiązanie połączeń zanim będą używane.**

   To zadanie będzie wykonywane w konstruktorze klasy. Nawiązanie kilku połączeń „zawczasu”, czyli zanim będą wykorzystywane, przyspiesza obsługę wielu jednocześnie nadsyłanych żądań, lecz wymaga dodatkowego czasu na samym początku działania programu. W efekcie, serwlet, który na samym początku tworzy bardzo wiele połączeń powinien stworzyć pulę połączeń w metodzie `init`, a Ty powinieneś mieć pewność, że serwlet zostanie zainicjalizowany zanim użytkownicy zaczną nadsyłać żądania. W przykładzie przedstawionym poniżej, do przechowywania połączeń dostępnych, niedostępnych oraz aktualnie wykorzystywanych, używane są wektory. Załóżmy, że metoda `makeNewConnection` używa zapisanego już wcześniej adresu URL, nazwy użytkownika oraz hasła, po czym, po prostu, wywołuje metodę `getConnection` obiektu klasy `DriverManager`.

```
availableConnections = new Vector(initialConnections);
busyConnections = new Vector();
for (int i=0; i<initialConnections; i++) {
  availableConnections.addElement(makeNewConnection());
}
```

2. **Zarządzanie dostępnymi połączeniami.**

   Jeśli będzie potrzebne połączenie i jednocześnie w puli będzie dostępne połączenie, które w danej chwili nie jest wykorzystywane, to należy je przenieść na listę połączeń używanych i zwrócić. Lista aktualnie używanych połączeń jest także wykorzystywana do sprawdzania ograniczeń dotyczących ilości połączeń oraz w sytuacjach gdy zostało wydane jawne polecenie zamknięcia wszystkich połączeń. Jedno ostrzeżenie — połączenia mogą być zamykane po upłynięciu pewnego okresu czasu, a zatem, przed zwróceniem połączenia należy upewnić się, że wciąż jest ono otwarte. Jeśli okaże się że połączenie zostało zamknięte, to należy je usunąć i powtórzyć cały proces. Usunięcie połączenia udostępnia wolny element, który może zostać wykorzystany przez procesy, które potrzebowały połączenia w chwili, gdy limit dostępnych połączeń został wyczerpany. A zatem, w takiej sytuacji należy uaktywnić wszystkie oczekujące wątki (przy użyciu metody `notifyAll`) i

sprawdzić czy można kontynuować ich realizację (na przykład, poprzez nawiązanie nowego połączenia).

```
public synchronized Connection getConnection()
    throws SQLException {
  if (!availableConnections.isEmpty()) {
    Connection existingConnection =
      (Connection)availableConnections.lastElement();
    int lastIndex = availableConnections.size() - 1;
    availableConnections.removeElementAt(lastIndex);
    if (existingConnection.isClosed()) {
      notifyAll();    // wolne miejsce dla oczekujących wątków
      return(getConnection()); // powtórz proces
    } else {
      busyConnections.addElement(existingConnection);
      return(existingConnection);
    }
  }
}
```

3. **Nawiązywanie nowych połączeń.**

Jeśli jest potrzebne połączenie, a w danej chwili nie ma żadnego dostępnego połączenia i limit ilości połączeń został wyczerpany, to należy uruchomić wątek działający w tle, którego zadaniem będzie nawiązanie nowego połączenia. Następnie trzeba zaczekać na pojawienie się pierwszego dostępnego połączenia, niezależnie od tego czy będzie to już istniejące, czy też nowoutworzone połączenie.

```
if ((totalConnections() < maxConnections) &&
    !connectionPending) { // pending - trwa nawiązanie połączenia w tle
  makeBackgroundConnection();
}
  try {
    wait(); // zablokuj i zawieś ten wątek.
  } catch(InterruptedException ie) {}
  return(getConnection()); // spróbuj jeszcze raz
```

4. **Oczekiwanie aż połączenie będzie dostępne.**

Ta sytuacja zachodzi gdy nie ma żadnych dostępnych połączeń, a jednocześnie został wyczerpany limit ilości połączeń. Oczekiwanie na połączenie powinno się zakończyć bez konieczności ciągłego sprawdzania czy jakieś połączenie jest już dostępne. Naturalnym rozwiązaniem tego problemu jest wykorzystanie metody `wait`, która usuwa blokadę synchronizującą wątek i zawiesza jego wykonywanie aż do czasu wywołania metody `notify` lub `notifyAll`. Ponieważ wywołanie metody `notifyAll` może pochodzić z kilku różnych źródeł, wątki, których działanie zostało wznowione powinne sprawdzić czy ich realizacja może być kontynuowana. W tym przypadku najprostszym sposobem wykonania tego zadania jest powtórzenie próby uzyskania połączenia.

```
try {
  wait();
} catch(InterruptedException ie) {}
return(getConnection());
```

Może się zdarzyć, że będziesz chciał aby klienci nie czekali i, w sytuacji gdy nie będzie żadnego dostępnego połączenia a limit ilości połączeń został już wyczerpany, zdecydujesz się zgłosić wyjątek. W takim przypadku możesz posłużyć się następującym fragmentem kodu:

```
throw new SQLException("Przekroczono limit ilości połączeń");
```

5. **Zamknięcie połączenia w razie konieczności.**

Zwróć uwagę, iż połączenia są zamykane podczas automatycznego usuwania ich z pamięci, a zatem nie zawsze będziesz musiał jawnie je zamykać. Jednak czasami będziesz chciał dysponować większą i bardziej jawną kontrolną nad przebiegiem całego procesu.

```
public synchronized void closeAllConnections() {
  // metoda closeConnection pobiera wszystkie połączenia
  // przechowywane w podanym wektorze, zamyka każde z nich wywołując
  // metodę close i ignoruje wszelkie zgłaszane wyjątki
```

```
    closeConnections(availableConnections);
    availableConnections = new Vector();
    closeConnections(busyConnections);
    busyConnections = new Vector();
  }
```

Poniżej podałem pełny kod klasy `ConnectionPool`.

Listing 18.19 ConnectionPool.java

```
package coreservlets;

import java.sql.*;
import java.util.*;

/** Klasa do stworzenia puli połączeń JDBC, ich wielkrotnego
 *  wykorzystywania i zarządzania nimi.
 */

public class ConnectionPool implements Runnable {
  private String driver, url, username, password;
  private int maxConnections;
  private boolean waitIfBusy;
  private Vector availableConnections, busyConnections;
  private boolean connectionPending = false;

  public ConnectionPool(String driver, String url,
                        String username, String password,
                        int initialConnections,
                        int maxConnections,
                        boolean waitIfBusy)
      throws SQLException {
    this.driver = driver;
    this.url = url;
    this.username = username;
    this.password = password;
    this.maxConnections = maxConnections;
    this.waitIfBusy = waitIfBusy;
    if (initialConnections > maxConnections) {
      initialConnections = maxConnections;
    }
    availableConnections = new Vector(initialConnections);
    busyConnections = new Vector();
    for(int i=0; i<initialConnections; i++) {
      availableConnections.addElement(makeNewConnection());
    }
  }

  public synchronized Connection getConnection()
      throws SQLException {
    if (!availableConnections.isEmpty()) {
      Connection existingConnection =
        (Connection)availableConnections.lastElement();
      int lastIndex = availableConnections.size() - 1;
      availableConnections.removeElementAt(lastIndex);
      // Jeśli połączenie z listy dostępnych połączeń będzie
      // zamknięte (np.: gdy upłyną limit czasu istnienia
      // połączenia), to należy usunąć je z listy dostępnych
      // połączeń i powtórzyć cały proces pobierania połączenia.
      // Należy także uaktywnić wszystkie wątki oczekujące na
      // połączenie ze względu na przekroczenie limitu ilości
      // połączeń (maxConnection).
      if (existingConnection.isClosed()) {
        notifyAll(); // wolne miejsce dla oczekujących wątków
        return(getConnection()); // powtórz proces
      } else {
        busyConnections.addElement(existingConnection);
        return(existingConnection);
      }
    } else {

      // Trzy możliwe przypadki:
      // 1) Nie został osiągnięty limit ilości połączeń
      //    (maxConnections). A zatem nawiązujemy połączenie
      //    w tle jeśli aktualnie jakieś połączenie nie jest
      //    nawiązywane; a następnie czekamy na następne dostępne
      //    połączenie (niezależnie od tego czy było to nowe połączenie,
```

```
    //    czy też połączenie utworzone już wcześniej).
    // 2) Został osiągnięty limit ilości połączeń (maxConnections)
    //    oraz flaga waitIfBusy ma wartość false. W takim przypadku
    //    należy zgłosić wyjątek SQLException.
    // 3) Został osiągnięty limit ilości połączeń (maxConnections)
    //    oraz flaga waitIfBusy ma wartość true. W takim przypadku
    //    należy wykonać te same czynności co w drugiej części
    //    punktu 1) - poczekać aż jakieś połączenie będzie dostępne.

    if ((totalConnections() < maxConnections) &&
        !connectionPending) {
      makeBackgroundConnection();
    } else if (!waitIfBusy) {
      throw new SQLException("Przekroczono limit ilości połączeń");
    }
    // Poczekaj na nawiązanie nowego połączenia
    // (jeśli została wywołana metoda makeBackgroundConnection)
    // lub na udostępnienie jakiegoś już istniejącego połączenia.
    try {
      wait();
    } catch(InterruptedException ie) {}
    // ktoś zwolnił połączenie - spróbuj może będzie dostępne.
    return(getConnection());
  }
}

// Nie można stworzyć nowego połączenia jeśli żadne połączenia
// nie są dostępne, gdyż w przypadku korzystania z wolnego
// połączenia sieciowego może to zabrać nawet kilka sekund.
// Zamiast tego uruchom nowy wątek, który nawiąże nowe
// połączenie z bazą danych, a następnie poczekaj.
// Działanie wątku zostanie wznowione jeśli zostanie
// udostępnione nowe połączenie, lub jeśli będzie można
// użyć jakiegoś już istniejącego połączenia.

private void makeBackgroundConnection() {
  connectionPending = true;
  try {
    Thread connectThread = new Thread(this);
    connectThread.start();
  } catch(OutOfMemoryError oome) {
    // Przerwij tworzenie nowego połączenia
  }
}

public void run() {
  try {
    Connection connection = makeNewConnection();
    synchronized(this) {
      availableConnections.addElement(connection);
      connectionPending = false;
      notifyAll();
    }
  } catch(Exception e) { // Wyjątek SQLException lub
    // OutOfMemory. Przerwij tworzenie nowego połączenia
    // i poczekaj aż zostanie udostępnione któreś z już
    // istniejących połączeń.
  }
}

// Ta metoda jawnie tworzy nowe połączenie. W czasie
// inicjalizacji obiektu ConnectionPool wywoływana jest
// normalnie, a podczas korzystania z puli - jest
// wywoływana w tle.

private Connection makeNewConnection()
    throws SQLException {
  try {
    // Załaduj sterownik bazy danych, jeśli jeszcze
    // nie został załadowany.
    Class.forName(driver);
    // Nawiąż połączenie sieciowe z bazą danych.
    Connection connection =
      DriverManager.getConnection(url, username, password);
    return(connection);
  } catch(ClassNotFoundException cnfe) {
    // Upraszczam blok try/catch dla osób korzystających z
```

```
        // tego kodu i obsługuję tylko jeden typ wyjątków.
        throw new SQLException("Nie można znaleźć klasy dla sterownika: " +
                               driver);
      }
    }

    public synchronized void free(Connection connection) {
      busyConnections.removeElement(connection);
      availableConnections.addElement(connection);
      // Uaktywnij wszystkie wątki oczekujące na połączenie.
      notifyAll();
    }

    public synchronized int totalConnections() {
      return(availableConnections.size() +
             busyConnections.size());
    }

    /** Metoda zamyka wszystkie połączenia. Używaj jej
     *  z dużą ostrożnością: przed jej wywołaniem należy
     *  upewnić się, że żadne połączenia nie są w danej
     *  chwili używane. Zauważ, że <I>nie musisz</I> używać
     *  tej metody gdy już zakończysz używanie obiektu
     *  ConnectionPool, gdyż system gwarantuje, że wszystkie
     *  połączenia zostaną zamknięte podczas automatycznego
     *  czyszczenia pamięci. Jednak metoda ta daje większą
     *  kontrolę nad tym, kiedy połączenia będą zamykane.
     */

    public synchronized void closeAllConnections() {
      closeConnections(availableConnections);
      availableConnections = new Vector();
      closeConnections(busyConnections);
      busyConnections = new Vector();
    }

    private void closeConnections(Vector connections) {
      try {
        for(int i=0; i<connections.size(); i++) {
          Connection connection =
            (Connection)connections.elementAt(i);
          if (!connection.isClosed()) {
            connection.close();
          }
        }
      } catch(SQLException sqle) {
        // Zignoruj błędy, i tak wszystko będzie
        // usunięte z pamięci
      }
    }

    public synchronized String toString() {
      String info =
        "ConnectionPool(" + url + "," + username + ")" +
        ", dostępnych=" + availableConnections.size() +
        ", używanych=" + busyConnections.size() +
        ", limit=" + maxConnections;
      return(info);
    }
  }
```

18.8 Zarządzanie pulami połączeń: Studium zagadnienia

W porządku, dysponujemy już klasą `ConnectionPool`, ale jak możemy ją wykorzystać? Przekonajmy się. Na listingu 18.20 przedstawiłem prosty serwlet, który w metodzie `init` tworzy kopię obiektu `ConnectionPool`, a następnie, dla każdego otrzymanego żądania pobiera informacje z bazy danych i zwraca je w formie tabeli HTML. Na listingu 18.21 przedstawiłem z kolei dokument HTML, który 25 razy odwołuje się do naszego przykładowego serwletu i wyświetla zwrócone przez

niego wyniki w 25 ramkach. Wygląd wynikowej strony WWW można zobaczyć na rysunku 18.6. Nasz serwlet wymusza aby wyniki nie by przechowywane w pamięci podręcznej przeglądarki, a zatem wyświetlenie strony przedstawionej na listingu 18.21 powoduje wygenerowanie 25 niemal jednoczesnych żądań HTTP i 25 odwołań do bazy danych, realizowanych przy użyciu zarządzania pulą połączeń. Powyższy sposób zgłaszania żądań przypomina to, co mogłoby się zdarzyć na bardzo popularnej witrynie WWW, nawet gdyby do obsługi każdej ze stron byłyby używany osobny serwlet.

Listing 18.20 ConnectionPoolServlet.java

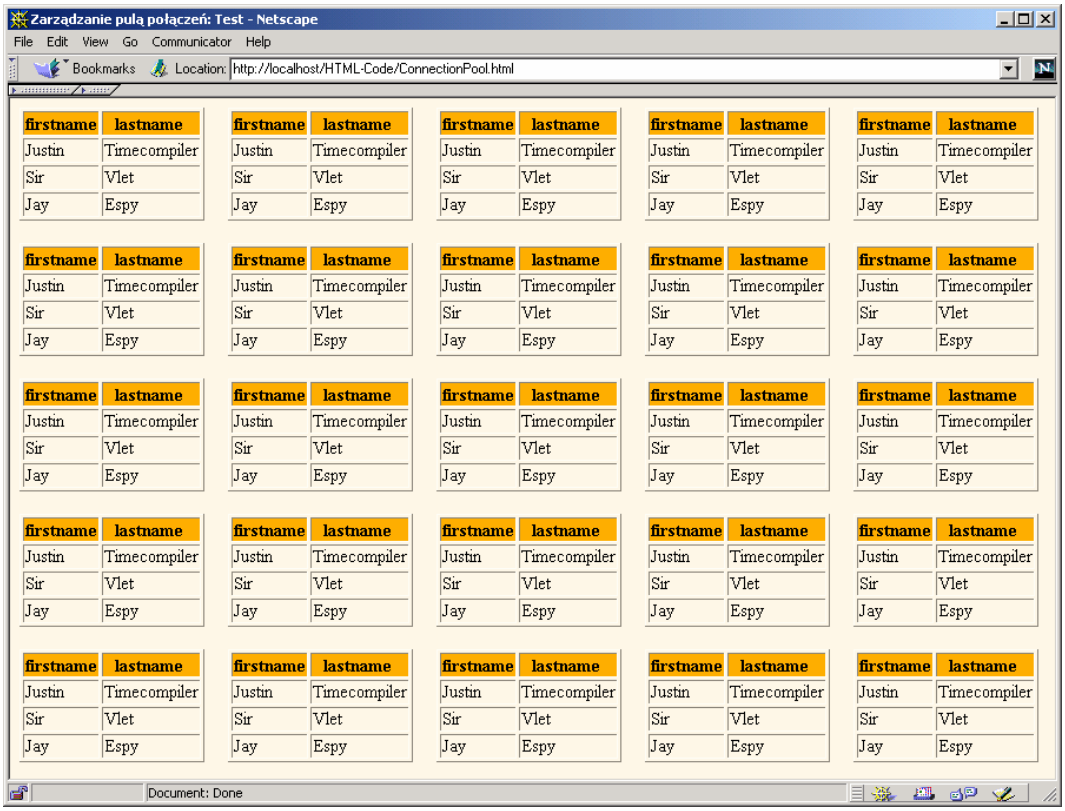
```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;

/** Serwlet odczytuje informacje z bazy danych i wyświetla
 *  je w postaci tabeli HTML. Serwlet wykorzystuje pulę
 *  połączeń w celu optymalizacji pobierania danych z bazy.
 *  Doskonałym testem jest strona ConnectionPool.html,
 *  która wyświetla wyniki wykonania tego serwletu w 25
 *  ramkach.
 */

public class ConnectionPoolServlet extends HttpServlet {
  private ConnectionPool connectionPool;

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    String table;
    try {
      String query =
        "SELECT firstname, lastname " +
        " FROM employees WHERE salary > 70000";
      Connection connection = connectionPool.getConnection();
      DBResults results =
          DatabaseUtilities.getQueryResults(connection,
                                            query, false);
      connectionPool.free(connection);
      table = results.toHTMLTable("#FFAD00");
    } catch(Exception e) {
      table = "Błąd: " + e;
    }
    response.setContentType("text/html");
    // Zażądaj aby przeglądarka nie przechowywała wyników
    // w pamięci podręcznej. Więcej informacji na ten temat
    // znajdziesz w podrozdziale 7.2 książki Java Servlet i
    // Java Server Pages.
    response.setHeader("Pragma", "no-cache"); // HTTP 1.0
    response.setHeader("Cache-Control", "no-cache"); // HTTP 1.1
    PrintWriter out = response.getWriter();
    String title = "Test wykorzystania puli połączeń";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<CENTER>\n" +
                table + "\n" +
                "</CENTER>\n</BODY></HTML>");
  }

  /** Inicjalizuje pulę połączeń w momencie inicjowania
   *  serwletu. Aby uniknąć opóźnień czasowych w momencie
   *  otrzymania pierwszego żądania skierowanego do tego
   *  serwletu, należy go zawczasu załadować samemu, lub
   *  tak skonfigurować serwera, aby serwlet był
   *  automatycznie ładowany po ponownym uruchomieniu
   *  serwera.
   */

  public void init() {
    int vendor = DriverUtilities.SYBASE;
    String driver = DriverUtilities.getDriver(vendor);
```

```
    String host = "128.220.101.65";
    String dbName = "605741";
    String url = DriverUtilities.makeURL(host, dbName, vendor);
    String username = "hall";
    String password = "hall";
    try {
      connectionPool =
        new ConnectionPool(driver, url, username, password,
                           initialConnections(),
                           maxConnections(),
                           true);
    } catch(SQLException sqle) {
      System.err.println("Błąd tworzenia puli połączeń: " + sqle);
      getServletContext().log("Błąd tworzenia puli połączeń: " + sqle);
      connectionPool = null;
    }
  }

  public void destroy() {
    connectionPool.closeAllConnections();
  }

  /** Przesłoń tę metodę w klasie potomnej aby zmienić
   *  początkową ilość połączeń.
   */

  protected int initialConnections() {
    return(10);
  }

  /** Przesłoń tę metodę w klasie potomnej aby zmienić
   *  maksymalną ilość tworzonych połączeń.
   */

  protected int maxConnections() {
    return(50);
  }
}
```

Listing 18.21 ConnectionPool.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Frameset//EN">
<HTML>
<HEAD><TITLE>Zarządzanie pulą połączeń: Test</TITLE></HEAD>

<!-- Powoduje wygenerowanie 25 niemal jednoczesnych żądań
     skierowanych do tego samego serwletu. -->

<FRAMESET ROWS="*,*,*,*,*" BORDER=0 FRAMEBORDER=0 FRAMESPACING=0>
  <FRAMESET COLS="*,*,*,*,*">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
  </FRAMESET>
  <FRAMESET COLS="*,*,*,*,*">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
  </FRAMESET>
  <FRAMESET COLS="*,*,*,*,*">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
  </FRAMESET>
  <FRAMESET COLS="*,*,*,*,*">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
```

```
  </FRAMESET>
  <FRAMESET COLS="*,*,*,*,*">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
    <FRAME SRC="/servlet/coreservlets.ConnectionPoolServlet">
  </FRAMESET>
</FRAMESET>

</HTML>
```



Rysunek 18.6 Dokument z ramkami, którego wyświetlenie w przeglądarce powoduje wygenerowanie 25 niemal jednoczesnych żądań odwołujących się do tego samego serwletu

Listing 18.22 przedstawia nową wersję serwletu z listingu 18.20, który wykorzystuje pulę zawierającą wyłącznie jedno połączenie, a kolejny listing — 18.23 — prezentuje kolejną wersję która w ogóle nie korzysta z puli połączeń. Dwa powyższe serwlety są wykorzystywane przy wyświetlaniu stron WWW, niemal takich samych jak strona przestawiona na listingu 18.21. W tabeli 18.1 przedstawiłem czasy konieczne do wyświetlenia stron odwołujących się do powyższych serwletów.

Listing 18.22 ConnectionPoolServlet2.java

```
package coreservlets;

/** Zmodyfikowana wersja serwletu ConnectionPoolServlet
 *  która wykorzystuje tylko jedno połączenie z bazą danych
 *  kolejkując wszystkie żądania jej użycia. Serwlet
 *  ten jest wykorzystywany przy
 *  porównywaniu zysków czasowych jakie daje
 *  stosowanie pul połączeń.
 */

public class ConnectionPoolServlet2
      extends ConnectionPoolServlet {
```

```
  protected int initialConnections() {
    return(1);
  }

  protected int maxConnections() {
    return(1);
  }
}
```

Listing 18.23 ConnectionPoolServlet3.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;

/** Zmodyfikowana wersja serwletu
 *  ConnectionPoolServlet która NIE KORZYSTA z puli
 *  połączeń. Serwlet ten jest wykorzystywany przy
 *  porównywaniu zysków czasowych jakie daje
 *  stosowanie pul połączeń.
 */

public class ConnectionPoolServlet3 extends HttpServlet {
  private String url, username, password;

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    String table;
    String query =
      "SELECT firstname, lastname " +
      " FROM employees WHERE salary > 70000";
    try {
      Connection connection =
        DriverManager.getConnection(url, username, password);
      DBResults results =
        DatabaseUtilities.getQueryResults(connection,
                                          query, true);
      table = results.toHTMLTable("#FFAD00");
    } catch(Exception e) {
      table = "Exception: " + e;
    }
    response.setContentType("text/html");
    // Zażądaj aby przeglądarka nie przechowywała wyników
    // w pamięci podręcznej. Więcej informacji na ten temat
    // znajdziesz w podrozdziale 7.2 książki Java Servlet i
    // Java Server Pages.
    response.setHeader("Pragma", "no-cache"); // HTTP 1.0
    response.setHeader("Cache-Control", "no-cache"); // HTTP 1.1
    PrintWriter out = response.getWriter();
    String title = "Test wykorzystania pul połączeń (*Bez* użycia puli)";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<CENTER>\n" +
                table + "\n" +
                "</CENTER>\n</BODY></HTML>");
  }

  public void init() {
    try {
      int vendor = DriverUtilities.SYBASE;
      String driver = DriverUtilities.getDriver(vendor);
      Class.forName(driver);
      String host = "128.220.101.65";
      String dbName = "605741";
      url = DriverUtilities.makeURL(host, dbName, vendor);
      username = "hall";
      password = "hall";
    } catch(ClassNotFoundException e) {
      System.err.println("Błąd podczas inicjalizacji: " + e);
      getServletContext().log("Błąd podczas inicjalizacji: " + e);
    }
```

```
    }
  }
```

Tabela 18.1 Czas wyświetlania stron odwołujących się do serwletów, które w różnym stopniu korzystały z puli połączeń

Warunki	Średni czas
Wolne połączenie modemowe z bazą danych, 10 początkowo dostępnych połączeń, dopuszczalny limit 50 połączeń (`ConnectionPoolServlet`).	11 sekund
Wolne połączenie modemowe z bazą danych, wielokrotnie wykorzystywane pojedyncze połączenie (`ConnectionPoolServlet2`).	22 sekundy
Wolne połączenie modemowe z bazą danych, pula połączeń nie używana (`ConnectionPoolServlet3`).	82 sekundy
Szybkie połączenie z bazą danych poprzez sieć lokalną, 10 początkowo dostępnych połączeń, dopuszczalny limit 50 połączeń (`ConnectionPoolServlet`).	1,8 sekundy
Szybkie połączenie z bazą danych poprzez sieć lokalną, wielokrotnie wykorzystywane pojedyncze połączenie (`ConnectionPoolServlet2`).	2,0 sekundy
Szybkie połączenie z bazą danych poprzez sieć lokalną, pula połączeń nie używana (`ConnectionPoolServlet3`).	2,8 sekundy

I jeszcze słowo przypomnienia — serwlety ładują sterownik JDBC, a zatem, serwer WWW musi mieć do niego dostęp. W przypadku przeważającej większości serwerów, wystarczy w tym celu umieścić plik JAR zawierający sterownik w katalogu *lib* serwera lub rozpakowując zawartość tego pliku do katalogu *classes*. Wszelkie informacje na ten temat powinieneś znaleźć w dokumentacji serwera.

18.9 Współużytkowanie pul połączeń

Każdy z serwletów przedstawionych w poprzedniej części rozdziału używał własnej puli połączeń. Taki sposób działania ma sens, jeśli poszczególne serwlety wykonują znacząco odmienne czynności i korzystają z różnych baz danych. Jednak równie często zdarza się, że niektóre lub nawet wszystkie serwlety działające na serwerze używają tej samej bazy danych, a co za tym idzie, mogą korzystać z jednej puli połączeń. Istnieją dwie podstawowe metody współużytkowania pul połączeń. Pierwszą z nich jest wykorzystanie kontekstu serwletu (metoda ta jest charakterystyczna dla serwletów), a drugą — użycie metod statycznych lub specjalnych klas umożliwiających stworzenie wyłącznie jednego obiektu tej klasy (jest to jedna z technik programowania w języku Java, a klasy takie określane są jako „singleton”).

Współużytkowanie pul połączeń przy wykorzystaniu kontekstu serwletu

Przy użyciu metody `getServletContext` można pobrać obiekt `ServletContext` wykorzystywany przez wszystkie serwlety działające na serwerze (lub w danej aplikacji WWW, jeśli serwer jest w stanie je obsługiwać). Obiekt ten udostępnia metodę `setAttribute`, której argumentami są — łańcuch znaków oraz obiekt klasy `Object`. Metoda ta powoduje zapisanie podanego obiektu w tablicy, w elemencie którego kluczem jest podany łańcuch znaków. Obiekt ten

można następnie pobrać w dowolnej chwili przy użyciu metody `getAttribute`. Metoda `getAttribute` wymaga podania jednego argumentu — łańcucha znaków. Jeśli podany klucz nie istnieje, metoda zwraca wartość `null`.

A zatem, serwlety, które korzystają z tej samej bazy danych książek, mogłyby współużytkować pulę połączeń, gdyby każdy z nich wykonywał następujące czynności:

```
ServletContext contekst = getServletContext();
ConnectionPool pulaKsiazek =
  (ConnectionPool) contekst.getAttribute("pula-ksiazek");
if (pulaKsiazek == null) {
  pulaKsiazek = new ConnectionPool(...);
  contekst.setAttribute("pula-ksiazek", pulaKsiazek);
}
```

Współużytkowanie pul połączeń przy wykorzystaniu klas „singleton”

Zamiast współużytkować pule połączeń przy użyciu kontekstu serwletu (`ServletContext`) można wykorzystać w tym celu zwyczajne metody statyczne. Na przykład, można by stworzyć klasę `BookPool` zawierającą metody `setPool` oraz `getPool`, a każdy serwlet wywoływałby metodę `BookPool.getPool` i porównywał uzyskany wynik z wartością `null`, a w razie konieczności tworzył nową kopię puli połączeń. Jednak w takim przypadku, każdy serwlet musi wykonywać ten sam kod, a poza tym, serwlet mógłby przez przypadek usunąć wspólnie używana pulę zwrócona przez metodę `BookPool.getPool` i zastąpić ją nową.

Lepszym rozwiązaniem jest zaimplementowanie koniecznych możliwości funkcjonalnych w, tak zwanej, klasie „singletn”. Jest to zwyczajna klasą, jednak dzięki użyciu prywatnego konstruktora istnieje możliwość utworzenia wyłącznie jednej kopii obiektu tej klasy. Kopia ta jest pobierana przy użyciu metody statycznej, która sprawdza czy jedyna kopia obiektu już istnieje i jeśli istnieje to ją zwraca, a jeśli nie — to tworzy. Poniżej przedstawiłem schemat takiej klasy o nazwie `BookPool`. Każdy serwlet, który chce z niej korzystać, może pobrać pulę połączeń wywołując metodę `BookPool.getInstance()`.

```
public class BookPool extends ConnectionPool {
  private BookPool pool = null;

  private BookPool(...) {
    super(...); // wywołaj konstruktor klasy bazowej
    ...
  }

  public static synchronized BookPool getInstance() {
    if (pool == null) {
      pool = new BookPool(...);
    }
    return pool;
  }
}
```

Dodatek A.
Krótki przewodnik po serwletach i JSP

A.1 Prezentacja serwletów i JSP

Zalety serwletów

- **wydajność** — wątki zamiast procesów, jedna kopia serwletu, trwałość,
- **wygoda** — bardzo wiele narzędzi wysokiego poziomu,
- **moc** — komunikacja z serwerami, współużytkowanie danych, zarządzanie pulami, trwałość,
- **przenośność** — działają niemal we wszystkich systemach operacyjnych i na wszystkich serwerach,
- **bezpieczeństwo** — żadnych przepełnień buforów, brak możliwości wykorzystania powłoki systemowej w sposób niezamierzony przez autorów aplikacji,
- **niskie koszty** — niedrogie rozszerzenia serwera jeśli obsługa serwletów nie jest wbudowana.

Zalety JSP

- **w porównaniu z ASP** — lepszy język do generacji dynamicznych informacji, przenośność,
- **w porównaniu z PHP** — lepszy język do generacji dynamicznych informacji,
- **w porównaniu z samymi serwletami** — wygodniejsza generacja kodu HTML,
- **w porównaniu z SSI** — większa elastyczność i możliwości,
- **w porównaniu z JavaScript** — wykonywanie programów na serwerze, bogatszy język,
- **w porównaniu ze statycznym kodem HTML** — możliwość dynamicznej generacji informacji.

Bezpłatnie dostępne oprogramowanie do obsługi serwletów i JSP

- Tomcat: *http://jakarta.apache.org/*,
- JSWDK: *http://java.sun.com/products/jsp/archive.html*,
- JRun: *http://www.allaire.com/products/jrun/*,
- ServletExec: *http://newatlanta.com/*,
- LiteWebServer: *http://www.gefionsoftware.com/*,
- Java Web Server: *http://www.sun.com/software/jwebserver/try/*.

Dokumentacja

- *http://java.sun.com/products/jsp/download.html*,
- *http://java.sun.com/products/servlet/2.2/javadoc/index.html*,
- *http://www.java.sun.com/j2ee/j2sdkee/techdocs/api/*.

Kompilacja serwletów: informacje podawane w zmiennej środowiskowej CLASSPATH

- pliki klasowe serwletów (zazwyczaj przechowywane w *katalog_instalacyjny/lib/servlet.jar*),
- pliki klasowe JSP (zazwyczaj przechowywane w *katalog_instalacyjny/lib/jsp.jar*, *.../jspengine.jar* lub *.../jasper.jar*),
- główny katalog, w którym są przechowywane pliki klasowe serwletów (na przykład, *katalog_instalacyjny/webpages/WEB-INF/classes*).

Standardowe katalogi serwera Tomcat 3.0

- *katalog_instalacyjny/webpages/WEB-INF/classes* — standardowe położenie plików klasowych serwletów,
- *katalog_instalacyjny/classes* — alternatywne położenie plików klasowych serwletów,
- *katalog_instalacyjny/lib* — położenie plików JAR zawierających pliki klasowe.

Standardowe katalogi serwera Tomcat 3.1

- *katalog_instalacyjny/webpages/ROOT/WEB-INF/classes* — standardowe położenie plików klasowych serwletów,
- *katalog_instalacyjny/classes* — alternatywne położenie plików klasowych serwletów,
- *katalog_instalacyjny/lib* — położenie plików JAR zawierających pliki klasowe.

Standardowe katalogi serwera JSWDK 1.0.1

- *katalog_instalacyjny/webpages/WEB-INF/servlets* — standardowe położenie plików klasowych serwletów,
- *katalog_instalacyjny/classes* — alternatywne położenie plików klasowych serwletów,
- *katalog_instalacyjny/lib* — położenie plików JAR zawierających pliki klasowe.

Standardowe katalogi serwera Java Web Server 2.0

- *katalog_instalacyjny/servlets* — położenie często modyfikowanych plików klasowych serwletów (automatyczne przeładowywanie),
- *katalog_instalacyjny/classes* — położenie plików klasowych serwletów które nie są często modyfikowane,
- *katalog_instalacyjny/lib* — położenie plików JAR zawierających pliki klasowe.

A.2 Pierwsze serwlety

Prosty serwlet

HelloWWW.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWWW extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {

    response.setContentType("text/html; encoding=ISO-8859-2");
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>Witaj WWW</TITLE></HEAD>\n" +
                "<BODY>\n" +
                "<H1>Witaj WWW</H1>\n" +
                "</BODY></HTML>");
  }
}
```

Instalacja serwletów

- umieść plik klasowy serwletu w kartotekach podanych w punkcie A.1,
- umieść plik klasowy serwletu w podkartotekach odpowiadających pakietowi do jakiego serwlet należy.

Uruchamianie serwletów

- *http://host/servlet/NazwaSerwletu*,
- *http://host/servlet/pakiet.NazwaSerwletu*,
- dowolne położenie zdefiniowane poprzez odpowiednie skonfigurowanie serwera.

Cykl życiowy serwletów

- **public void init() throws ServletException**, **public void init(ServletConfig config) throws ServletException**

  Obie te metody są wykonywane bezpośrednio po załadowaniu serwletu do pamięci. *Nie* są one natomiast wywoływane podczas obsługi żądań. Parametry inicjalizacyjne serwletu można odczytywać przy użyciu metody `getInitParameter`.
- **public void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException**

  Metoda jest wywoływana przez serwer w nowym wątku, podczas obsługi każdego nadesłanego żądania. Powoduje ona przekazanie sterowania do jednej z metod `doGet`, `doPost`, lub innej. Nie należy przesłaniać tej metody.
- **public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException**

  Metoda obsługuje żądania `GET`. Należy ją przesłonić, aby podać sposób działania serwletu.

- **public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException**

  Metoda obsługuje żądania POST. Należy ją przesłonić, aby podać sposób działania serwletu. Jeśli chcesz, aby żądania GET i POST były obsługiwane w ten sam sposób, to w tej metodzie wywołaj metodę doGet.
- **doPut**, **doTrace**, **doDelete**, itd.

  Te metody obsługują rzadziej spotykane żądania protokołu HTTP, takie jak PUT, TRACE, itd.
- **public void destroy()**

  Metoda jest wywoływana w momencie gdy serwer usuwa egzemplarz serwletu z pamięci. *Nie* jest ona wywoływana po zakończeniu obsługi każdego z nadsyłanych żądań.
- **public long getLastModified(HttpServletRequest request)**

  Metoda jest wywoływana gdy klient, ze względu na wykorzystywanie możliwości przechowywania stron w pamięci podręcznej, prześle warunkowe żądanie GET.
- **SingleThreadModel**

  Zaimplementowanie tego interfejsu sprawi, że serwer nie będzie wykonywać jednoczesnych odwołań do serwletu.

A.3 Obsługa żądań: Dane przesyłane z formularzy

Odczyt parametrów

- **request.getParameter**: zwraca pierwszą wartość parametru,
- **request.getParameterValues**: zwraca tablicę wszystkich wartości parametru.

Przykład serwletu

ThreeParams.java

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Prosty servlet odczytujący wartości trzech parametrów
 *  przesłanych z formularza.
 */

public class ThreeParams extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html; charset=ISO-8859-2");
    PrintWriter out = response.getWriter();
    String title = "Odczyt trzech parametrów";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<UL>\n" +
                "  <LI><B>param1</B>: "
                + request.getParameter("param1") + "\n" +
                "  <LI><B>param2</B>: "
                + request.getParameter("param2") + "\n" +
                "  <LI><B>param3</B>: "
                + request.getParameter("param3") + "\n" +
                "</UL>\n" +
                "</BODY></HTML>");
  }
}
```

Przykład formularza

ThreeParamsForm.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Pobieranie wartości trzech parametrów</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">Pobieranie wartości trzech parametrów</H1>

<FORM ACTION="/servlet/coreservlets.ThreeParams">
  Parameter pierwszy:  <INPUT TYPE="TEXT" NAME="param1"><BR>
  Parameter drugi: <INPUT TYPE="TEXT" NAME="param2"><BR>
  Parameter trzeci:  <INPUT TYPE="TEXT" NAME="param3"><BR>
  <CENTER>
    <INPUT TYPE="SUBMIT" VALUE="Prześlij">
  </CENTER>
</FORM>

</BODY>
</HTML>
```

Filtrowanie znaków specjalnych HTML

- Znaki `<`, `>`, `"` oraz `&` należy zastępować odpowiednimi symbolami HTML — `<`, `>`, `"` oraz `&`. Do wykonania takiej zamiany można użyć metody `ServletUtilities.filter(`*`stringZKodemHTML`*`)`. Więcej informacji na ten temat znajdziesz w podrozdziale 6.3.

A.4 Obsługa żądań: Nagłówki żądań HTTP

Metody odczytujące nagłówki żądania

Wszystkie poniższe metody zostały zdefiniowane w interfejsie `HttpServletRequest`:

- **public String getHeader(String nazwaNaglowka)**

  Metoda zwraca wartość dowolnego nagłówka żądania. Jeśli nagłówka o podanej nazwie nie ma w żądaniu, metoda zwraca wartość `null`.
- **public Enumeration getHeaders(String nazwaNaglowka)**

  Zwraca wartości wszystkich wystąpień nagłówka o podanej nazwie. Dostępna w mechanizmach obsługi serwletów zgodnych ze specyfikacją Java Servlet 2.2.
- **public Enumeration getHeaderNames()**

  Zwraca nazwy wszystkich nagłówków przesłanych w aktualnie obsługiwanym żądaniu.
- **public long getDateHeader(String nazwaNaglowka)**

  Odczytuje wartość nagłówka reprezentującego datę i konwertuje ją do postaci dat używanych w języku Java (czyli do ilości milisekund jakie upłynęły od początku 1970 roku).
- **public int getIntHeader(String nazwaNaglowka)**

  Odczytuje wartość nagłówka reprezentującego liczbę całkowitą i konwertuje ją do liczby typu `int`. Jeśli nagłówka o podanej nazwie nie ma w żądaniu, metoda zwraca wartość

`-1`. Jeśli wartość nagłówka nie jest poprawnie zapisaną liczbą całkowitą, to metoda zgłasza wyjątek `NumberFormatException`.

- **public Cookie[] getCookies()**

  Zwraca tablicę obiektów `Cookie`. Jeśli w żądaniu nie zostały przesłane żadne cookies, to zwracana jest pusta tablica (o zerowej długości). Więcej informacji na temat cookies znajdziesz w rozdziale 8.
- **public int getContentLength()**

  Zwraca liczbę typu `int` reprezentującą wartość nagłówka `Content-Type`. W przypadku gdy wartość ta nie jest znana, zwracana jest liczba `-1`.
- **public String getContentType()**

  Zwraca zawartość nagłówka `Content-Type`, jeśli został on umieszczony w żądaniu (na przykład dla dołączanych plików). Jeśli nagłówek nie został podany, zwracana jest wartość `null`.
- **public String getAuthType()**

  Zwraca jeden z łańcuchów znaków: `"BASIC"`, `"DIGEST"` lub `"SSL"`, bądź wartość `null`.
- **public String getRemoteUser()**

  Jeśli jest wykorzystywane uwierzytelnianie, to wywołanie tej metody zwraca nazwę użytkownika; w przeciwnym przypadku zwracana jest wartość `null`.

Inne informacje o żądaniu

- **public String getMethod()**

  Zwraca rodzaj (metodę) żądania HTTP (`"GET"`, `"POST"`, `"HEAD"`, itp.)
- **public String getRequestURI()**

  Zwraca fragment adresu URL zapisany po nazwie komputera i numerze portu.
- **public String getProtocol()**

  Zwraca łańcuch znaków określający używaną wersję protokołu HTTP (zazwyczaj jest to `"HTTP/1.0"` lub `"HTTP/1.1"`).

Najczęściej używane nagłówki żądań protokołu HTTP 1.1

Wszelkie informacje na temat protokołu HTTP 1.1 można znaleźć w pliku RFC 2616. Pliki RFC można znaleźć, na przykład, na witrynie *http://www.rfc-editor.org/*.

- **Accept** — typy MIME, które przeglądarka jest w stanie obsługiwać.
- **Accept-Encoding** — rodzaje kodowania (np.: gzip lub compress) jakie przeglądarka jest w stanie obsługiwać. Przykład wykorzystania kompresji przedstawiłem w podrozdziale 4.4.
- **Authorization** — identyfikacja użytkownika wykorzystywana przez zasoby, do których dostęp jest chroniony hasłem. Stosowny przykład przedstawiłem w podrozdziale 4.5. Zazwyczaj stosowana metoda przesyłania informacji o nazwie użytkownika i haśle polega nie na wykorzystaniu mechanizmów protokołu HTTP lecz zwykłych formularzy HTML; informacje te po przesłaniu do serwletu są następnie zapisywane w obiekcie sesji.
- **Connection** — w przypadku protokołu HTTP 1.0 wartość `keep-alive` tego nagłówka oznacza, że przeglądarka jest w stanie obsługiwać trwałe połączenia. W protokole HTTP 1.1 trwałe połączenia są wykorzystywane domyślnie. Aby umożliwić wykorzystanie trwałych połączeń HTTP, serwlet powinien podać wartość nagłówka `Content-Length` wywołując metodę `setContentLength` (w celu określenia wielkości generowanych informacji można posłużyć się strumieniem `ByteArrayOutputStream`). Przykład wykorzystania trwałych połączeń przedstawiłem w podrozdziale 7.4.

- **Cookie** — cookies przesyłane wcześniej z serwera do klienta. Aby określić ich wartości należy posłużyć się metodą `getCookies`, a nie `getHeader`. Więcej informacji na temat cookies podałem w rozdziale 8.
- **Host** — nazwa komputera podana w oryginalnym adresie URL. W protokole HTTP 1.1 nagłówek ten jest wymagany.
- **If-Modified-Since** — określa, że klient chce pobrać stronę wyłącznie jeśli została ona zmodyfikowana po określonej dacie. Nagłówków `If-Modified-Since` nie należy obsługiwać bezpośrednio, zamiast tego należy zaimplementować metodę `getLastModified`. Stosowny przykład przedstawiłem w podrozdziale 2.8.
- **Referer** — adres URL strony, która była wyświetlona w przeglądarce w chwili, gdy wysyłano żądanie.
- **User-Agent** — łańcuch znaków identyfikujący przeglądarkę, która przesłała żądanie.

A.5 Dostęp do standardowych zmiennych CGI

Zazwyczaj nie powinno się myśleć o zmiennych CGI, lecz raczej o informacjach przesyłanych w żądaniu, generowanych w odpowiedzi, bądź też o informacjach związanych z serwerem WWW.

Możliwości, które nie zostały opisane gdzie indziej

- **getServletContext().getRealPath("uri")** — kojarzy URI z faktyczną ścieżką,
- **request.getRemoteHost()** — nazwa komputera, z którego zostało przesłane żądanie,
- **request.getRemoteAddr()** — adres IP komputera, z którego zostało przesłane żądanie.

Odpowiedniki zmiennych CGI dostępne w serwletach

- **AUTH_TYPE**: `request.getAuthType()`,
- **CONTENT_LENGTH**: `request.getContentLength()`,
- **CONTNET_TYPE**: `request.getContentType()`,
- **DOCUMENT_ROOT**: `getServletContext().getRealPath("/")`,
- **HTTP_XXX_YYY**: `request.getHeader("Xxx-Yyy")`,
- **PATH_INFO**: `request.getPathInfo()`,
- **PATH_TRANSLATED**: `request.getPathTranslated()`,
- **QUERY_STRING**: `request.getQueryString()`,
- **REMOTE_ADDR**: `request.getRemoteAddr()`,
- **REMOTE_HOST**: `request.getRemoteHost()`,
- **REMOTE_USER**: `request.getRemoteUser()`,
- **REQUEST_METHOD**: `request.getMethod()`,
- **SCRIPT_NAME**: `request.ServletPath()`,
- **SERVER_NAME**: `request.getServerName()`,
- **SERVER_PORT**: `request.getServerPort()`,
- **SERVER_PROTOCOL**: `request.getProtocol()`,
- **SERVER_SOFTWARE**: `getServletContext().getServerInfo()`.

A.6 Generacja odpowiedzi: Kody statusu HTTP

Format odpowiedzi HTTP

Wiersz statusu (wersja protokołu HTTP, kod statusu, komunikat), nagłówki odpowiedzi, pusty wiersz, dokument — dokładnie w podanej kolejności. Oto przykład:

```
HTTP/1.1 200 OK
Content-Type: text/plain

Witaj Świecie
```

Metody określające kod statusu

Wszystkie podane poniżej metody zostały zdefiniowane w interfejsie `HttpServletResponse`. Kod statusu należy określać przed przesłaniem do klienta jakiejkolwiek zawartości generowanego dokumentu.

- **public void setStatus(int kodStatusu)**
  Określając kod statusu należy posługiwać się odpowiednimi stałymi, a nie jawnie podawanymi liczbami całkowitymi.
- **public void sendError(int kod, String komunikat)**
  Zapisuje komunikat w niewielkim dokumencie HTML.
- **public void sendRedirect(String url)**
  Specyfikacja Java Servlet 2.2 zezwala na podawanie względnych adresów URL.

Kategorie kodów statusu

- **100 – 199** — kody informacyjne, klient powinien odpowiedzieć na nie wykonując jakąś czynność,
- **200 – 299** — żądanie zostało poprawnie obsłużone,
- **300 – 399** — plik został przeniesiony; w takim przypadku odpowiedź zazwyczaj zawiera nagłówek Location określający nowe położenie pliku,
- **400 – 499** — błąd klienta,
- **500 – 599** — błąd serwera.

Najczęściej wykorzystywane kody statusu protokołu HTTP 1.1

- **200** (**OK**) — wszystko w porządku, po nagłówku odpowiedz zostanie przesłany dokument. Domyślna odpowiedź serwera.
- **204** (**No Content**) — przeglądarka nie powinna zmieniać wyświetlanej strony.
- **301** (**Moved Permanently**) — żądany dokument został na stałe przeniesiony w inne miejsce (określone przy użyciu nagłówka odpowiedzi `Location`). Przeglądarki automatycznie pobierają dokument ze wskazanego, nowego położenia.
- **302** (**Found**) — żądany dokument został tymczasowo przeniesiony w inne miejsce (określone przy użyciu nagłówka odpowiedzi `Location`). Przeglądarki automatycznie pobierają dokument ze wskazanego, nowego położenia. Aby wygenerować ten nagłówek w serwlecie należy posłużyć się metodą `sendRedirect`, a nie `setHeader`. Stosowny przykład przestawiłem w podrozdziale 6.3.

- **401** (**Unauthorized**) — przeglądarka zażądała dostępu do strony chronionej hasłem, bez przesłania poprawnego nagłówka `Authorization`. Stosowny przykład przedstawiłem w podrozdziale 4.5.
- **404** (**Not Found**) — Nie ma takiej strony. Serwlet powinien generować ten nagłówek przy użyciu metody `sendError`. Stosowny przykład przedstawiłem w podrozdziale 6.3

A.7 Generacja odpowiedzi: Nagłówki odpowiedzi protokołu HTTP

Generacja dowolnych nagłówków

Wszystkie poniższe metody zostały zdefiniowane w interfejsie `HttpServletResponse`. Nagłówki odpowiedzi należy generować przed przesłaniem do przeglądarki jakiejkolwiek treści dokumentu.

- **public void setHeader(String nazwaNaglowka, String wartoscNaglowka)**
  Podaje wartość dowolnego nagłówka.
- **public void setDateHeader(String nazwaNaglowka, long milisekundy)**
  Konwertuje liczbę typu `long` określającą ilość sekund jakie upłynęły od początku 1970 roku, do postaci daty zapisanej w formacie GMT.
- **public void setIntHeader(String nazwaNaglowka, int wartoscNaglowka)**
  Użycie tej metody zapobiega konieczności dokonania konwersji liczby typu `int` do postaci łańcucha znaków (`String`), którą trzeba by wykonać przed wywołaniem metody `setHeader`.
- **addHeader**, **addDateHeader**, **addIntHeader**
  Dodaje kolejny nagłówek o podanej nazwie, nie usuwając poprzednich nagłówków, które zostały podane wcześniej. Metody dostępne wyłącznie w mechanizmach obsługi serwletów zgodnych ze specyfikacją Java Servlet 2.2.

Generacja najczęściej używanych nagłówków

- **setContentType** — podaje wartość nagłówka `Content-Type`. Serwlety niemal zawsze używają tej metody. Najczęściej używane typy MIME przedstawiłem w tabeli 7.1.
- **setContentLength** — podaje wartość nagłówka `Content-Length`, wykorzystywanego przy stosowaniu trwałych połączeń HTTP. Do buforowania generowanego dokumentu wynikowego i określenia jego długości można się posłużyć strumieniem `ByteArrayOutputStream`. Stosowny przykład przedstawiłem w podrozdziale 7.4.
- **addCookie** — dodaje wartość do nagłówka `Set-Cookie`. Więcej informacji na temat cookies znajdziesz w rozdziale 8.
- **sendRedirect** — podaje wartość nagłówka `Location` (a dodatkowo zmienia także kod statusu). Stosowny przykład przedstawiłem w podrozdziale 6.3.

Najczęściej używane nagłówki odpowiedzi protokołu HTTP 1.1

- **Allow** — metody żądań obsługiwane przez serwer. Te nagłówki są automatycznie generowane przez domyślną metodę `service`, gdy serwlet otrzymuje żądania `OPTIONS`.
- **Cache-Control** — wartość `no-cache` tego nagłówka zapobiega przechowywaniu dokumentu w pamięci podręcznej przeglądarki. Na wypadek, gdyby przeglądarka korzystała wyłącznie

z protokołu HTTP 1.0, warto wraz z tym nagłówkiem wygenerować także nagłówek `Pragma` o wartości `no-cache`.

- **Content-Encoding** — określa sposób kodowania dokumentu. Przeglądarki odtwarzają kodowanie, przywracając oryginalną postać dokumentu przed jego przetworzeniem. Przed użyciem kodowania serwlet musi upewnić się, że przeglądarka je obsługuje (sprawdzając wartość nagłówka żądania `Accept-Encoding`). Przykład kompresji przesyłanych dokumentów przedstawiłem w podrozdziale 4.4
- **Content-Length** — ilość bajtów przesyłanych w odpowiedzi. Patrz metoda `setContentLength` opisana we wcześniejszej części podrozdziału.
- **Content-Type** — typ MIME zwracanego dokumentu. Patrz metoda `setContentType` opisana we wcześniejszej części podrozdziału.
- **Expires** — czas, po którym dokument należy uznać za nieaktualny i usunąć z pamięci podręcznej przeglądarki. Nagłówek ten należy generować przy wykorzystaniu metody `setDateHeader`.
- **Last-Modified** — czas ostatniej modyfikacji dokumentu. Wartości tego nagłówka nie powinno się podawać wprost. Zamiast tego należy zaimplementować metodę `getLastModified`. Stosowny przykład przedstawiłem w podrozdziale 2.8.
- **Location** — adres URL pod który przeglądarka powinna przesłać kolejne żądanie. Nagłówka tego nie należy podawać wprost, lecz wygenerować przy użyciu metody `sendRedirect`. Stosowny przykład przedstawiłem w podrozdziale 6.3.
- **Pragma** — wartość `no-cache` tego nagłówka powoduje, że przeglądarki wykorzystujące protokół HTTP 1.0 nie będą przechowywać zawartości otrzymanego dokumentu w pamięci podręcznej. Patrz także nagłówek odpowiedzi `Cache-Control` opisany w podrozdziale 7.2.
- **Refresh** — ilość sekund, po upłynięciu których przeglądarka powinna ponownie odświeżyć stronę. Nagłówek może także zawierać adres URL strony, którą przeglądarka ma pobrać. Stosowny przykład przedstawiłem w podrozdziale 7.3.
- **Set-Cookie** — cookie, które przeglądarka powinna zapamiętać. Nagłówka tego nie należy generować wprost, lecz przy użyciu metody `addCookie`. Wszelkie informacje na temat wykorzystania cookies podałem w rozdziale 8.
- **WWW-Authenticate** — typ oraz obszar autoryzacji jaki przeglądarka powinna podać w nagłówku `Authorization` przesłanym w kolejnym żądaniu. Stosowny przykład przedstawiłem w podrozdziale 4.5

Generacja obrazów GIF przez serwlety

- **Stwórz obraz GIF.**
  W tym celu należy wywołać metodę `createImage` klasy `Component`.
- **Narysuj coś na stworzonym obrazie.**
  Wywołaj metodę `getGraphics` obiektu klasy `Image`, a następnie narysuj coś przy użyciu standardowych operacji graficznych.
- **Wygeneruj odpowiedni nagłówek Content-Type.**
  W tym celu posłuż się wywołaniem o postaci `response.setConentType("image/gif")`.
- **Pobierz strumień wyjściowy.**
  W tym celu użyj wywołania `response.getOutputStream()`.
- **Przekaż obraz zapisany w formacie GIF do strumienia wyjściowego.**
  Do tego celu można wykorzystać klasę `GifEncoder` Jefa Poskanzera (patrz *http://www.acme.com/java/*).

A.8 Obsługa cookies

Typowe zastosowania cookies

- identyfikacja użytkowników podczas sesji na witrynach zajmujących się handlem elektronicznym,
- unikanie konieczności przesyłania nazwy i hasła użytkownika,
- dostosowywanie witryn do preferencji użytkowników,
- wyświetlanie reklam dostosowanych do zainteresowań użytkowników.

Problemy napotykane przy stosowaniu cookies

- Cookies stwarzają zagrożenie dla prywatności lecz nie dla bezpieczeństwa osób korzystających z Internetu.
- Oto przyczyny zagrożenia prywatności użytkowników — serwery mogą pamiętać czynności wykonywane przez użytkowników w poprzednich sesjach, jeśli użytkownik poda jakieś informacje personalne, to mogą one zostać połączone z jego poprzednimi czynnościami; serwery mogą przekazywać pomiędzy sobą cookies (wykorzystując jakiś wspólny serwer, taki jak *doubleclick.net*, z którego wszystkie będą pobierać, na przykład, obrazki); niepoprawnie zaprojektowane witryny mogą przechowywać w cookies ważne informacje, takie jak numery kart kredytowych.

Ogólny sposób użycia cookies

- Przesłanie cookie do przeglądarki (standardowy sposób):

```
Cookie c = new Cookie("nazwa", "wartosc");
c.setMaxAge(...);
// określenie innych atrybutów cookie
response.addCookie(c);
```

- Przesłanie cookie do przeglądarki (sposób uproszczony):

  Wykorzystaj klasę `LongLivedCookie` przedstawioną w podrozdziale 8.5.

- Odczytanie wartości cookie przesłanego przez przeglądarkę (sposób standardowy):

```
Cookie[] cookies = request.getCookies();
for (int i=0; i<cookies.length; i++) {
  Cookie c = cookies[i];
  if (c.getName().equals("jakasNazwa")) {
    zrobSocZCookie(c);
    break;
  }
}
```

- Odczytanie wartości cookie przesłanego przez przeglądarkę (sposób uproszczony):

  Pobierz cookie lub wartość cookie z tablicy wszystkich przesłanych cookies przy użyciu jednej z metod `ServletUtilities.getCookie` lub `ServletUtilities.getCookieValue`.

Metod do obsługi cookies

- **getComment**, **setComment** — pierwsza metoda pobiera a druga określa tekst komentarza. Metoda ta nie jest dostępna w cookies zgodnych z pierwszą wersją protokołu cookies (o numerze 0; czyli z protokołem wykorzystywanym przez przeważającą większość aktualnie używanych przeglądarek).
- **getDomain**, **setDomain** — pierwsza metoda pobiera a druga określa domenę, której dotyczy cookie. Częścią tej domeny musi być nazwa komputera, na którym działa serwlet.

- **getMaxAge**, **setMaxAge** — pierwsza metoda pobiera a druga określa czas wygaśnięcia ważności cookie (wyrażony w sekundach). Jeśli wartość ta nie zostanie podana, to cookie będzie istnieć tylko do końca bieżącej sesji przeglądarki. Patrz klasa pomocnicza `LongLivedCookie` przedstawiona w podrozdziale 8.5.
- **getName**, **setName** — pierwsza metoda pobiera a druga określa nazwę cookie. W przypadku tworzenia nowego cookie jego nazwy nie określa się przy użyciu metody `setName`, lecz podaje w wywołaniu konstruktora. W przypadku wykorzystania tablicy cookies przesłanych z przeglądarki metoda `getName` służy do odszukania wybranego cookie.
- **getPath**, **setPath** — pierwsza metoda pobiera a druga określa ścieżkę, której dotyczy cookie. Jeśli ścieżka nie zostanie jawnie określona, to cookie będzie dotyczyć wszystkich stron znajdujących się w katalogu w którym znajdowała się strona która wygenerowała dane cookie oraz w jego podkatalogach.
- **getSecure**, **setSecure** — pierwsza metoda pobiera a druga określa flagę określającą czy cookie ma być przesyłane wyłącznie po bezpiecznych połączeniach (wykorzystujących SSL) czy też można je przesyłać niezależnie od używanego połączenia.
- **getValue**, **setValue** — pierwsza metoda pobiera a druga określa wartość cookie. W przypadku tworzenia nowego cookie jego wartości nie określa się przy użyciu metody `setValue`, lecz podaje w wywołaniu konstruktora. W razie wykorzystania tablicy cookies przesłanych z przeglądarki, wybrane cookie odszukuje się przy użyciu metody `getName`, a jego wartość jest następnie pobierana za pomocą metody `getValue`.
- **getVersion**, **setVersion** — pierwsza metoda pobiera a druga określa wersję protokołu cookies. Domyślnie stosowana jest wersja 0; należy jej używać aż do momentu gdy przeglądarki zaczną obsługiwać wersję 1.

A.9 Śledzenie sesji

Pobieranie informacji o sesji — getValue

```
HttpSession sesja = request.getSession(true);
ShoppingCart koszyk =
  (ShoppingCart) sesja.getValue("shoppingCart");
if (koszyk == null) { // w sesji nie ma jeszcze koszyka
  koszyk = new ShoppingCart();
  sesja.putValue("shoppingCart", koszyk);
}
zrobCosZ(koszyk);
```

Kojarzenie informacji z sesją — putValue

```
HttpSession sesja = request.getSession(true);
sesja.putValue("referringPage", request.getHeader("Referer"));
ShoppingCart koszyk =
  (ShoppingCart) sesja.getValue("previousItems");
if (koszyk == null) { // brak koszyka w sesji
  koszyk = new ShoppingCart();
  sesja.putValue("previousItems", koszyk);
}
String idElem = request.getParameter("itemID");
if (idElem != null) {
  koszyk.addItem(Catalog.getItem(idElem));
}
```

Metody interfejsu HttpSession

- **public Object getValue(String nazwa) [2.1]**
  public Object getAttribute(String nazwa) [2.2]
  Pobiera wartość zapisaną wcześniej w obiekcie sesji. Jeśli z podaną nazwą nie jest skojarzona żadna wartość, zwracana jest wartość `null`.
- **public void putValue(String nazwa, Object wartosc) [2.1]**
  public void setAttribute(String nazwa, Object wartosc) [2.2]
  Kojarzy wartość z nazwą. Jeśli wartość jest obiektem implementującym interfejs `HttpSessionBindingListener`, to wywoływana jest metoda `valueBound` tego obiektu. Jeśli uprzednia wartość skojarzona z podaną nazwą jest obiektem implementującym interfejs `HttpSessionBindingListener` to wywoływana jest metoda `valueUnbound`.
- **public void removeValue(String nazwa) [2.1]**
  public void removeAttribute(String nazwa) [2.2]
  Usuwa wszelkie wartości skojarzone z podaną nazwą. Jeśli usuwana wartość jest obiektem implementującym interfejs `HttpSessionBindingListener` to wywoływana jest metoda `valueUnbound` tego obiektu.
- **public String[] getValueNames() [2.1]**
  public Enumeration getAttributeNames() [2.2]
  Zwraca nazwy wszystkich atrybutów zapisanych w sesji.
- **public String getId()**
  Zwraca unikalny identyfikator generowany dla każdej sesji.
- **public boolean isNew()**
  Zwraca wartość `true` jeśli klient (przeglądarka) jeszcze nic nie wie o sesji; w przeciwnym przypadku zwracana jest wartość `false`.
- **public long getCreationTime()**
  Zwraca czas utworzenie sesji (wyrażony jako ilość milisekund jaka upłynęła od początku 1970 roku). Aby uzyskać wartość którą można wydrukować, należy użyć liczby zwróconej przez tę metodę jako argumentu konstruktora klasy `Date` lub metody `setTimeInMillis` klasy `GregorianCalendar`.
- **public long getLastAccessedTime()**
  Zwraca czas określający kiedy po raz ostatni sesja została przesłana z klienta (przeglądarki) na serwer.
- **public int getMaxInactiveInterval()**
  public void setMaxInactiveInterval(int sekundy)
  Pierwsza z tych metod pobiera, a druga podaje czas (wyrażony w sekundach) określający jak długo sesja może być nieużywana nim zostanie automatycznie unieważniona. Przekazanie wartości ujemnej w wywołaniu metody `setMaxInactiveTime` informuje, że sesja nie ma być automatycznie unieważniana. Ta wartość to nie to samo co czas wygaśnięcia ważności cookie.
- **public void invalidate()**
  Unieważnia sesję i usuwa przechowywane w niej obiekty.

Kodowanie adresów URL

W przypadkach gdy serwlety implementują śledzenie sesji przy wykorzystaniu metody przepisywania adresów URL, należy dać systemowi możliwość ich zakodowania.

- Zwyczajne adresy URL

```
String oryginalnyURL = wzglednyLubBezwzglednyAdresURL;
String zakodowanyURL = response.encodeURL(oryginalnyURL);
out.println("<A HREF=\"" + zakodowanyURL + "\">...</A>");
```

- Adresy URL używane do przekierowań

```
String oryginalnyURL = jakisURL; //specyfikacja 2.2 zezwala na używanie
                                 //adresów względnych.
String zakodowanyURL = response.endoceRedirectURL(oryginalnyURL);
response.sendRedirect(zakodowanyURL);
```

A.10 Elementy skryptowe JSP

Typy elementów skryptowych

- Wyrażenia: `<%= wyrażenie %>`

  Przetwarzane i wstawiane do wyników generowanych przez serwlet. Można także wykorzystać alternatywny sposób zapisu:

```
<jsp:expression>
  wyrażenie
</jsp:expression>
```

- Skryptlety: `<% kod %>`

  Wstawiane do metody `_service` serwletu (wywoływanej przez metodę `service`). Skryptlety można także zapisywać w alternatywny sposób:

```
<jsp:scriptlet>
  kod
</jsp:scriptlet>
```

- Deklaracje: `<%! kod %>`

  Umieszczane wewnątrz klasy serwletu, poza jakimikolwiek metodami. Można je także zapisywać w alternatywny sposób:

```
<jsp:declaration>
  kod
</jsp:declaration>
```

Tekst szablonu

- aby został wygenerowany łańcuch znaków `<%` należy użyć łańcucha `<\%`,
- `<%-- komentarz JSP --%>`,
- `<!-- komentarz HTML -->`,
- pozostały tekst, który nie zawiera żadnych elementów specjalnych JSP, jest przekazywany do strony wynikowej.

Predefiniowane zmienne

Poniżej przedstawiłem niejawne obiekty, które zawsze są dostępne w wyrażeniach i skryptletach JSP (lecz nie w deklaracjach).

- **request** — obiekt `HttpServletRequest` skojarzony z nadesłanym żądaniem.
- **response** — obiekt `HttpServletResponse` skojarzony z odpowiedzią przesyłaną do klienta.
- **out** — obiekt `JspWriter` (jest to zmodyfikowana wersja klasy `PrintWriter`) używany do przesyłania wyników do klienta.
- **session** — obiekt `HttpSession` skojarzony z danym żądaniem; więcej informacji na temat sesji znajdziesz w rozdziale 9.
- **application** — obiekt `ServletContext`, który można uzyskać przy użyciu wywołania `getServletConfig().getContext()`. Obiekt ten jest wspólnie wykorzystywany przez

wszystkie serwlety oraz dokumenty JSP uruchamiane na danym serwerze lub tworzące daną aplikację WWW. Więcej informacji na ten temat podałem w rozdziale 15.

- **config** — obiekt `ServletConfig` dla danej strony JSP.
- **pageContext** — obiekt `PageContext` skojarzony z bieżącą stroną JSP. Informacje na temat użycia obiektów tego typu przedstawiłem w podrozdziale 13.4.
- **page** — odpowiednik słowa kluczowego `this` (reprezentuje egzemplarz serwletu); aktualnie niezbyt przydatny; zarezerwowany do użycia w przyszłości.

A.11 Dyrektywa page: określanie postaci generowanych serwletów

Atrybut import

- `<%@ page import="`*`pakiet.klasa`*`" %>`,
- `<%@ page import="`*`pakiet.klasa1, ..., pakiet.klasaN`*`" %>`.

Atrybut contentType

- `<%@ page contentType="`*`typMIME`*`" %>`,
- `<%@ page contentType="`*`typMIME`*`; charset=`*`zbiór_znaków`*`" %>`,
- przy użyciu dyrektywy `page` używanego zbioru znaków nie można określać warunkowo; jeśli jednak konieczne będzie warunkowe określenie wartości tego atrybutu, to należy się posłużyć skryptletem o postaci `<% response.setContentType("..."); %>`.

Przykład użycia atrybutu contentType

Excel.jsp

```
<%@ page contentType="application/vnd.ms-excel" %>
<%-- Zwróć uwagę, iż pomiędzy kolumnami zostały zapisane znaki
     tabulacji a nie odstępy --%>
1997   1998   1999   2000   2001 (przewidywany)
12.3   13.4   14.5   15.6   16.7
```

Przykład wykorzystania metody setContentType

AppleAndOranges.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Porównanie sprzedaży jabłek i pomarańczy</TITLE>
<LINK REL=STYLESHEET
      HREF="JSP-Styles.css"
      TYPE="text/css">
</HEAD>

<BODY>
<CENTER>
<H2>Porównanie sprzedaży jabłek i pomarańczy</H2>

<%
String format = request.getParameter("format");
if ((format != null) && (format.equals("excel"))) {
  response.setContentType("application/vnd.ms-excel");
}
%>
```

```
<TABLE BORDER=1>
  <TR><TH></TH><TH>Jabłka<TH>Pomarańcze
  <TR><TH>Pierwszy kwartał<TD>2307<TD>4706
  <TR><TH>Drugi kwartał<TD>2982<TD>5104
  <TR><TH>Trzeci kwartał<TD>3011<TD>5220
  <TR><TH>Czwarty kwartał<TD>3055<TD>5287
</TABLE>

</CENTER>
</BODY>
</HTML>
```

Atrybut isThreadSafe

- `<%@ page isThreadSafe="true" %> <%-- wartość domyślna --%>`,
- `<%@ page isThreadSafe="false" %>`,
- przypisanie wartości `true` temu atrybutowi oznacza że kod został przystosowany do działania wielowątkowego i system może używać go do jednoczesnej obsługi wielu żądań; natomiast przypisanie temu atrybutowi wartości `false` oznacza, że serwlet wygenerowany na podstawie strony JSP ma implementować interfejs `SingleThreadModel` (patrz podrozdział 2.6),
- kod nieprzystosowany do wykonywania wielowątkowego:

```
<%! private int idNum = 0; %>
<% String userID = "userID" + idNum;
   out.println("Twój identyfikator użytkownika to: " + userID + "." );
   idNum = idNum + 1; %>
```

- kod przystosowany do działania wielowątkowego:

```
<%! private int idNum = 0; %>
<% synchronized(this) {
     String userID = "userID" + idNum;
     out.println("Twój identyfikator użytkownika to: " + userID + "." );
     idNum = idNum + 1;
   } %>
```

Atrybut session

- `<%@ page session="true" %> <%-- wartość domyślna --%>`,
- `<%@ page session="false" %>`.

Atrybut buffer

- `<%@ page buffer="rozmiar_w_kb" %>`,
- `<%@ page buffer="none" %>`,
- serwery mogą korzystać z bufora większego niż podany, jednak gwarantuje się, że używany bufor nie będzie mniejszy od podanego. Na przykład, dyrektywa `<%@ page buffer="32kb" %>` oznacza, że wynikowy dokument ma być buforowany i nie powinien być przesyłany do klienta aż do momentu nagromadzenia 32 kilobajtów informacji lub zakończenia jego generacji.

Atrybut autoflush

- `<%@ page autoflush="true" %> <%-- wartość domyślna --%>`,
- `<%@ page autoflush="false" %>`,
- w przypadku przypisania wartości `none` atrybutowi `buffer`, przypisanie wartości `false` atrybutowi `autoflush` nie jest dozwolone.

Atrybut extends

- `<%@ page extends="pakiet.klasa" %>`.

Atrybut info

- `<%@ page info="Jakiś dowolny komunikat" %>`.

Atrybut errorPage

- `<%@ page errorPage="względnyAdresURL" %>`,
- zgłoszony wyjątek będzie automatycznie dostępny na wskazanej stronie błędu, jako zmienna `exception`. Przykłady wykorzystania stron błędów przedstawiłem na listingach 11.5 oraz 11.6.

Atrybut isErrorPage

- `<%@ page isErrorPage="true" %>`,
- `<%@ page isErrorPage="false" %> <%-- wartość domyślna --%>`,
- stosowne przykłady przedstawiłem na listingach 11.5 oraz 11.6.

Atrybut language

- `<%@ page language="cobol" %>`,
- jak na razie nie warto zawracać sobie głowy tym atrybutem, gdyż jedyną jego dozwoloną wartością jest `java`.

Zapis XML-owy

- **Zapis zwyczajny:**

```
<%@ page atrybut="wartość" %>
<%@ page import="java.util.*" %>
```

- **Zapis XML-owy:**

```
<jsp:directive.page atrybut="wartość" />
<jsp:directive.page import="java.util.*" />
```

A.12 Dołączanie plików i apletów do dokumentów JSP

Dołączanie plików w czasie przekształcania strony

- `<%@ include file="względnyAdresURL" %>`,
- zmiana dołączanego pliku nie musi wcale powodować ponownego przekształcenia dokumentu JSP do postaci serwletu. Dlatego należy samemu zmienić dokument JSP lub zaktualizować jego datę modyfikacji. Oto prosty i wygodny sposób:

```
<%-- Navbar.jsp data modyfikacji 3/1/2000 --%>
<%@ include file="Navbar.jsp" %>
```

Dołączanie plików w czasie obsługi żądania

- `<jsp:include page="`*`względnyAdresURL`*`" flush="true" />`,
- w serwletach ten sam efekt można uzyskać dzięki wykorzystaniu metody `include` interfejsu `RequestDispatcher` (stosowny przykład podałem w podrozdziale 15.3),
- ze względu na błąd występujący w Java Web Serverze dołączane pliki muszą mieć rozszerzenie *.html* lub *.htm*.

Aplety obsługiwane przy użyciu Java Plug-In: Prosty przypadek

- **Zwyczajna postać:**

```
<APPLET CODE="MojAplet.class"
        WIDTH="475" HEIGHT="350">
</APPLET>
```

- **Postać stosowana w JSP umożliwiająca wykorzystanie Java Plug-Ina:**

```
<jsp:plugin type="applet"
            code="MojAplet.class"
            width="475" height="350">
</jsp:plugin>
```

Atrybuty znacznika jsp:plugin

Przy podawaniu nazw atrybutów uwzględniana jest wielkość liter; wszystkie wartości atrybutów muszą być umieszczone pomiędzy znakami cudzysłowu lub apostrofu.

- **type** — w przypadku apletów, atrybut ten powinien mieć wartość `applet`,
- **code** — używany identycznie jak atrybut `CODE` elementu `APPLET`,
- **width** — używany identycznie jak atrybut `WIDTH` elementu `APPLET`,
- **height** — używany identycznie jak atrybut `HEIGHT` elementu `APPLET`,
- **codebase** — używany identycznie jak atrybut `CODEBASE` elementu `APPLET`,
- **align** — używany identycznie jak atrybut `ALIGN` elementu `APPLET`,
- **hspace** — używany identycznie jak atrybut `HSPACE` elementu `APPLET`,
- **vspace** — używany identycznie jak atrybut `VSPACE` elementu `APPLET`,
- **archive** — używany identycznie jak atrybut `ARCHIVE` elementu `APPLET`,
- **name** — używany identycznie jak atrybut `NAME` elementu `APPLET`,
- **title** — używany identycznie jak atrybut `CODE` elementu `APPLET` (oraz niemal wszystkich innych elementów HTML dostępnych w języku HTML 4.0), określa on tytuł elementu jaki należy wyświetlić w etykiecie ekranowej lub użyć przy indeksowaniu,
- **jreversion** — określa wymaganą wersję Java Runtime Environment (JRE — środowiska wykonawczego Javy); domyślna wersja to 1.1,
- **iepluginurl** — określa adres URL spod którego można pobrać plug-in przeznaczony do użytku w Internet Explorerze,
- **nspluginurl** — określa adres URL spod którego można pobrać plug-in przeznaczony do użytku w Netscape Navigatorze.

Parametry określane w kodzie HTML: jsp:param

- **Zwyczajna postać:**

```
<APPLET CODE="MojAplet.class"
        WIDTH="475" HEIGHT="350">
  <PARAM NAME="PARAMETR1" VALUE="WARTOŚĆ1">
  <PARAM NAME="PARAMETR2" VALUE="WARTOŚĆ2">
```

```
</APPLET>
```

- **Zapis stosowany w JSP w przypadku wykorzystania Java Plug-In:**

```
<jsp:plugin type="applet"
            code="MojAplet.class"
            width="475" height="350">
  <jsp:params>
    <jsp:param name="PARAMETR1" value="WARTOŚĆ1" />
    <jsp:param name="PARAMETR2" value="WARTOŚĆ2" />
  </jsp:params>
</jsp:plugin>
```

Tekst alternatywny

- **Postać zwyczajna:**

```
<APPLET CODE="MojAplet.class"
        WIDTH="470" HEIGHT="350">
  <B>Błąd: ten przykład wymaga przeglądarki obsługującej język Java.</B>
</APPLET>
```

- **Zapis stosowany w JSP w przypadku wykorzystania Java Plug-In:**

```
<jsp:plugin type="applet"
            code="MojAplet.class"
            width="475" height="350">
  <jsp:fallback>
    <B>Błąd: ten przykład wymaga przeglądarki obsługującej
       język Java.</B>
  </jsp:fallback>
</jsp:plugin>
```

- Java Web Server nie obsługuje prawidłowo elementu `jsp:fallback`.

A.13 Wykorzystanie komponentów JavaBeans w dokumentach JSP

Podstawowe wymagania jakie należy spełnić by klasa była komponentem

1. Klasa musi implementować pusty konstruktor (konstruktor który nie pobiera żadnych argumentów).
2. Klasa nie może posiadać żadnych publicznych zmiennych instancyjnych (pól).
3. Musi zapewniać dostęp do trwale przechowywanych wartości za pośrednictwem metod `get`*`Xxx`* (ewentualnie `is`*`Xxx`*) oraz `set`*`Xxx`*.

Podstawowe sposoby użycia komponentów

- `<jsp:useBean id="`*`nazwa`*`" class="`*`pakiet.Klasa`*`" />`,
- `<jsp:getProperty name="`*`nazwa`*`" property="`*`właściwość`*`" />`,
- `<jsp:setProperty nazwa="`*`nazwa`*`" property="`*`właściwość`*`" value="wartość" />`.

Kojarzenie właściwości z parametrami przesłanymi w żądaniu

- **Konkretne właściwości:**

```
<jsp:setProperty
```

```
name="licznik"
property="ilosc"
param="ilosc" />
```

- **Automatyczna konwersja typów** — dla typów podstawowych konwersja wykonywana jest analogicznie do działania metody `valueOf` klasy reprezentującej dany typ podstawowy.
- **Wszystkie właściwości:**
```
<jsp:setProperty name="licznik" property="*" />
```

Wspólne wykorzystywanie komponentów: Atrybut scope znacznika akcji jsp:useBean

Przykłady wspólnego wykorzystywania komponentów podane w rozdziale 15.

- **page** — Wartość domyślna. Określa ona, iż oprócz skojarzenia komponentu ze zmienną lokalną, należy go także umieścić na okres obsługi bieżącego żądania w obiekcie klasy `PageContext`.
- **application** — Oznacza ona, iż oprócz skojarzenia komponentu ze zmienną lokalną, należy go także umieścić we wspólnie wykorzystywanym w obiekcie `ServletContext`. Obiekt ten jest dostępny jako predefiniowana zmienna `application`; można go także pobrać przy użyciu metody `getServletContext()`.
- **session** — Oznacza ona, że oprócz skojarzenia komponentu ze zmienną lokalną, należy go także zapisać w obiekcie `HttpSession` skojarzonym z aktualnie obsługiwanym żądaniem. Komponent można pobrać przy użyciu metody `getValue`.
- **request** — Oznacza ona, że oprócz skojarzenia komponentu ze zmienną lokalną, na czas obsługi bieżącego żądania należy go także umieścić w obiekcie `ServletRequest`. Komponent taki można pobrać przy użyciu metody `getAttribute`.

Warunkowe tworzenie komponentów

- Użycie znacznika akcji `jsp:useBean` powoduje utworzenie nowej kopii komponentu wyłącznie w sytuacji gdy nie istnieje jeszcze komponent o tym samym identyfikatorze (`id`) i zasięgu (`scope`). Jeśli jednak uda się odnaleźć komponent o tym samym identyfikatorze i zasięgu, to zostanie on skojarzony ze zmienną określoną przy użyciu atrybutu `id`.
- Znaczniki akcji `jsp:setProperty` można wykonać warunkowo, w zależności od utworzenia nowego komponentu:
```
<jsp:useBean ...>
  znaczniki
</jsp:useBean>
```

A.14 Tworzenie bibliotek znaczników

Klasa obsługi znacznika

- Klasa implementuje interfejs `Tag` rozszerzając klasę `TagSupport` (jeśli definiowany znacznik nie ma zawartości lub jeśli jest ona generowana bez żadnych modyfikacji) bądź klasę `BodyTagSupport` (jeśli trzeba w jakiś sposób przetworzyć zawartość definiowanego znacznika.
- `doStartTag` — metoda zawiera kod jaki należy wykonać po odnalezieniu znacznika otwierającego.

- `doEndTag` — metoda zawiera kod jaki należy wykonać po odnalezieniu znacznika zamykającego.
- `doAfterBody` — metoda zawiera kod używany do przetworzenia zawartości znacznika.

Plik deskryptora biblioteki znaczników

- Wewnątrz elementu `taglib` umieszczane są elementy `tag` opisujące każdy z definiowanych znaczników. Na przykład:

```
<tag>
  <name>prime</name>
  <tagclass>coreservlets.tags.PrimeTag</tagclass>
  <info>Wyświetla losową liczbę pierwszą o długości N cyfr.</info>
  <bodycontent>EMPTY</bodycontent>
  <attribute>
    <name>length</name>
    <required>false</required>
  </attribute>
</tag>
```

Plik JSP

- `<%@ taglib uri="jakisPlikDeskryptora.tld" prefix="prefiks" %>`,
- `<prefiks:nazwaZnacznika />`,
- `<prefiks:nazwaZnacznika>zawartość</prefiks:nazwaZnacznika>`.

Przypisywanie atrybutów znacznikom

- **Klasa obsługi znacznika:**
  Implementuje metodę `set`*`Xxx`* dla każdego atrybutu *`xxx`*.
- **Deskryptor biblioteki znaczników:**

```
<tag>
  ...
  <attribute>
    <name>length</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue> <%-- czasami --%>
  </attribute>
</tag>
```

Dołączanie zawartości znacznika

- **Klasa obsługi znacznika:**
  Metoda `doStartTag` powinna zwracać wartość `EVAL_BODY_INCLUDE` a nie wartość `SKIP_BODY`.
- **Deskryptor biblioteki znaczników:**

```
<tag>
  ...
  <bodycontent>JSP</bodycontent>
</tag>
```

Opcjonalne dołączanie zawartości znacznika

- **Klasa obsługi znacznika:**
  Powinna zwracać bądź to wartość `EVAL_BODY_INCLUDE`, bądź `SKIP_BODY`, w zależności od wartości parametru przekazanego w żądaniu.

Przetwarzanie zawartości znacznika

- **Klasa obsługi znacznika:**
  Powinna stanowić klasę potomną klasy `BodyTagSupport` i implementować metodę `doAfterBody`. Aby pobrać obiekt klasy `BodyContent` opisujący zawartość znacznika, należy wywołać metodę `getBodyContent`. Klasa `BodyContent` posiada trzy kluczowe metody — `getEnclosingWriter`, `getReader` oraz `getString`. Metoda `doAfterBody` powinna zwracać wartość `SKIP_BODY`.

Wielokrotne dołączanie lub przetwarzanie zawartości znacznika

- **Klasa obsługi znacznika:**
  Aby ponownie przetworzyć zawartość znacznika metoda `doAfterBody` powinna zwrócić wartość `EVAL_BODY_TAG`, a aby zakończyć przetwarzanie — wartość `SKIP_BODY`.

Stosowanie zagnieżdżonych znaczników

- **Klasa obsługi znacznika:**
  Aby określić w jakim znaczniku został umieszczony aktualnie przetwarzany znacznik, klasa która go implementuje może posłużyć się metodą `findAncestorWithClass`. Wszelkie informacje można umieszczać w polach zewnętrznego znacznika.
- **Deskryptor biblioteki znaczników:**
  Wszystkie znaczniki należy deklarować niezależnie od siebie, bez względu na strukturę w jakiej będą zapisywane w dokumencie JSP.

A.15 Integracja serwletów i dokumentów JSP

Opis ogólny

- Serwlet realizuje początkowy etap obsługi żądania, odczytuje parametry, cookies, informacje o sesji, itd.
- Następnie serwlet wykonuje wszelkie konieczne obliczenia oraz pobiera potrzebne informacje z bazy danych.
- Następnie serwlet zapisuje uzyskane informacje w komponentach JavaBeans.
- Serwlet przekazuje żądanie dalej, do jednej z wielu stron JSP odpowiedzialnych za prezentację ostatecznych wyników.
- Dokument JSP pobiera potrzebne informacje z komponentów JavaBeans.

Składnia służąca do przekazania żądania

```
String url = "/sciezka/prezentacja1.jsp";
RequestDispatcher dispatcher = getServletContext().getRequestDipatcher(url);
dispatcher.forward();
```

Przekazywanie żądań do zwyczajnych dokumentów HTML

- Jeśli serwlet obsługuje wyłącznie żądania `GET`, to nie są konieczne żadne modyfikacje.

- Jeśli jednak serwlet obsługuje żądania `POST`, to także strona docelowa musi być w stanie obsługiwać te żądania — w tym celu należy zmienić jej nazwę ze *strona.html* na *strona.jsp*.

Tworzenie globalnie dostępnych komponentów JavaBeans

- **Serwlet wstępnie obsługujący żądanie:**
```
Typ1 wartosc1 = obliczWartoscNaPodstawieZadania(request);
getServletContext().setAttribute("klucz1", wartosc1);
```

- **Dokument JSP kończący obsługę żądania:**
```
<jsp:useBean id="klucz1" class="Typ1" scope="application" />
```

Tworzenie komponentów JavaBeans dostępnych w sesji

- **Serwlet wstępnie obsługujący żądanie:**
```
Typ1 wartosc1 = obliczWartoscNaPodstawieZadania(request);
HttpSession sesja = request.getSession(true);
session.putValue("klucz1", wartosc1);
```

- **Dokument JSP kończący obsługę żądania:**
```
<jsp:useBean id="klucz1" class="Typ1" scope="session" />
```

Interpretacja względnych adresów URL na stronie docelowej

- Adres URL oryginalnego serwletu jest używany przy przetwarzaniu przekierowywanych żądań. Przeglądarka nie zna prawdziwego adresu URL, a zatem będzie określać względne adresu URL na podstawie adresu, do którego żądanie było skierowane.

Alternatywne sposoby pobierania obiektu RequestDispatcher (wyłącznie Java Servlet 2.2)

- **przy użyciu nazwy** — użyj metody `getNamedDispatcher` interfejsu `ServletContext`,
- **przy użyciu ścieżki określonej względem położenia serwletu** — użyj metody `getRequestDispatcher` interfejsu `HttpServletRequest` a nie jednej z metod interfejsu `ServletContext`.

Dołączenie danych statycznych lub dynamicznych

- **Podstawowy sposób użycia:**
```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.print("...");
RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher("/sciezka/zasob");
dispatcher.include(request, response);
out.print("...");
```

- Odpowiednikiem powyższego fragmentu kodu w technologii JSP jest znacznik akcji `jsp:include` (lecz nie dyrektywa `include`).

Przekazywanie żądań ze stron JSP

- `<jsp:forward page="względnyAdresURL" />`.

A.16 Stosowanie formularzy HTML

Element FORM

- Standardowa postać:
```
<FORM ACTION="adresURL" ...> ... </FORM>
```
- Atrybuty: ACTION (wymagany), METHOD, ENCTYPE, TARGET, ONSUBMIT, ONRESET, ACCEPT, ACCEPT-CHARSET.

Pola tekstowe

- Standardowa postać:
```
<INPUT TYPE="TEXT" NAME="..." ...>
```
(brak znacznika zamykającego)
- Atrybuty: NAME (wymagany), VALUE, SIZE, MAXLENGTH, ONCHANGE, ONSELECT, ONFOCUS, ONBLUR, ONKEYDOWN, ONKEYPRESS, ONKEYUP.
- Różne przeglądarki w różny sposób obsługują przesyłanie formularza po naciśnięciu klawisza *Enter* w polu tekstowym. Z tego względu warto umieszczać w formularzach specjalne przyciski (typu SUBMIT) lub mapy odnośników, które jawnie przesyłają formularz.

Pola hasła

- Standardowa postać:
```
<INPUT TYPE="PASSWORD" NAME="..." ...>
```
(brak znacznika zamykającego)
- Atrybuty: NAME (wymagany), VALUE, SIZE, MAXLENGTH, ONCHANGE, ONSELECT, ONFOCUS, ONBLUR, ONKEYDOWN, ONKEYPRESS, ONKEYUP.
- Jeśli formularz zawiera pola hasła, to zawsze należy go przesyłać metodą POST.

Obszary tekstowe

- Standardowa postać:
```
<TEXTAREA NAME="..." ROWS=xxx COLS=yyy ...>
  Jakiś tekst
</TEXTAREA>
```
- Atrybuty: NAME (wymagany), ROWS (wymagany), COLS (wymagany), WRAP (niestandardowy), ONCHANGE, ONSELECT, ONFOCUS, ONBLUR, ONKEYDOWN, ONKEYPRESS, ONKEYUP.
- Odstępy w tekście zapisanym pomiędzy otwierającym i zamykającym znacznikiem elementu TEXTAREA są zachowywane, umieszczony wewnątrz kod HTML jest traktowany dosłownie, przetwarzane są wyłącznie symbole HTML takie jak <, >, itp.

Przyciski SUBMIT

- Standardowa postać:
```
<INPUT TYPE="SUBMIT" ...>
```
(brak znacznika zamykającego)
- Atrybuty: NAME, VALUE, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR.
- Po kliknięciu przycisku tego typu formularz jest przesyłany do serwletu bądź innego programu wykonywanego na serwerze, określonego poprzez atrybut ACTION elementu FORM.

Alternatywna postać przycisków SUBMIT

- Standardowa postać:
```
<BUTTON TYPE="SUBMIT" ...>
  kod HTML
</BUTTON>
```
- Atrybuty: NAME, VALUE, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR.
- Ten sposobu tworzenia przycisków SUBMIT może być używany wyłącznie w Internet Explorerze.

Przyciski RESET

- Standardowa postać:
  `<INPUT TYPE="RESET"...>` (brak znacznika zamykającego)
- Atrybuty: VALUE, NAME, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR . (Za wyjątkiem atrybutu VALUE, wszystkie pozostałe atrybuty są przeznaczone wyłącznie do użytku w skryptach JavaScript.)

Alternatywna postać przycisków RESET

- Standardowa postać:
```
<BUTTON TYPE="RESET" ...>
  kod HTML
</BUTTON>
```
- Atrybuty: VALUE, NAME, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR.
- Ten sposobu tworzenia przycisków RESET może być używany wyłącznie w Internet Explorerze.

Przyciski JavaScript

- Standardowa postać:
  `<INPUT TYPE="BUTTON" ...>` (brak znacznika zamykającego)
- Atrybuty: NAME, VALUE, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR.

Alternatywna postać przycisków JavaScript

- Standardowa postać:
```
<BUTTON TYPE="BUTTON" ...>
  kod HTML
</BUTTON>
```
- Atrybuty: NAME, VALUE, ONCLICK, ONDBLCLICK, ONFOCUS, ONBLUR.
- Ten sposób tworzenia przycisków JavaScript może być używany wyłącznie w Internet Explorerze.

Pola wyboru

- Standardowa postać:
  `<INPUT TYPE="CHECKBOX" NAME="..." ...>` (brak znacznika zamykającego)
- Atrybuty: NAME (wymagany), VALUE, CHECKED, ONCLICK, ONFOCUS, ONBLUR.
- Para nazwa-wartość jest przesyłana na serwer wyłącznie w przypadku gdy pole wyboru zostało zaznaczone.

Przyciski opcji

- Standardowa postać:
  ```
  <INPUT TYPE="RADIO" NAME="..." VALUE="..." ...>
  ```
  (brak znacznika zamykającego)
- Atrybuty: NAME (wymagany), VALUE (wymagany), CHECKED, ONCLICK, ONFOCUS, ONBLUR.
- Grupę przycisków opcji tworzy się nadając tą samą nazwę wszystkim należącym do niej przyciskom.

Listy rozwijane

- Standardowa postać:
  ```
  <SELECT NAME="nazwa" ...>
    <OPTION VALUE="wartosc1">Tekst opcji 1
    <OPTION VALUE="wartosc2">Tekst opcji 2
    ...
    <OPTION VALUE="wartoscN">Tekst opcji N
  </SELECT>
  ```
- Atrybuty elementu SELECT: NAME (wymagany), SIZE, MULTIPLE, ONCLICK, ONFOCUS, ONBLUR, ONCHANGE.
- Atrybuty elementu OPTION: SELECTED, VALUE.

Elementy kontrolne umożliwiające przesyłanie plików na serwer

- Standardowa postać:
  ```
  <INPUT TYPE="FILE" ...>
  ```
  (brak znacznika zamykającego)
- Atrybuty: NAME (wymagany), VALUE (wymagany), SIZE, MAXLENGTH, ACCEPT, ONCHANGE, ONSELECT, ONFOCUS, ONBLUR (niestandardowy).
- W deklaracji elementu FORM należy przypisać atrybutowi ENCTYPE wartość multipart/form-data.

Mapy odnośników obsługiwane na serwerze

- Standardowa postać:
  ```
  <INPUT TYPE="IMAGE" ...>
  ```
  (brak znacznika zamykającego)
- Atrybuty: NAME (wymagany), SRC, ALIGN.
- Można także dodać atrybut ISMAP do elementu IMG zapisanego wewnątrz elementu A.

Pola ukryte

- Standardowa postać:
  ```
  <INPUT TYPE="HIDDEN" NAME="..." VALUE="...">
  ```
  (brak znacznika zamykającego)
- Atrybuty: NAME (wymagany), VALUE.

Możliwości dostępne w Internet Explorerze

- element FIELDSET (używany wraz z LEGEND) — służy do grupowania elementów kontrolnych,
- atrybut TABINDEX — umożliwia określenie kolejności przechodzenia pomiędzy poszczególnymi elementami kontrolnymi formularza przy użyciu klawisza *Tab*,

- obie powyższe możliwości są elementami specyfikacji języka HTML 4.0, lecz żadna z nich nie jest dostępna w przeglądarce Netscape Navigator 4.

A.17 Wykorzystanie apletów jako interfejsu użytkownika dla serwletów

Przesyłanie danych metodą GET i wyświetlanie strony wynikowej

```
String jakiesDane =
  nazwa1 + "=" + URLEncoder.encode(wartosc1) + "&" +
  nazwa2 + "=" + URLEncoder.encode(wartosc2) + "&" +
  ...
  nazwaN + "=" + URLEncoder.encode(wartoscN);
try {
  URL urlProgramu = new URL( bazowyURL + "?" + jakiesDane );
  getAppletContext().showDocument(urlProgramu);
} catch(MalformedURLException mue) { ... }
```

Przesyłanie danych metodą GET i bezpośrednie przetwarzanie wyników (tunelowanie HTTP)

1. **Stwórz obiekt `URL` odwołujący się do komputera, z którego aplet został pobrany.** Zazwyczaj adres URL będzie tworzony na podstawie nazwy komputera, z którego aplet został pobrany.
   ```
   URL biezacaStrona = getCodeBase();
   String protokol = biezacaStrona.getProtocol();
   String host = biezacaStrona.getHost();
   int port = biezacaStrona.getPort();
   String konURL = "/servlet/jakisSerwlet";
   URL daneURL = new URL( protokol, host, port, konURL);
   ```

2. **Stwórz obiekt `URLConnection`.** Metoda `openConnection` klasy `URL` zwraca obiekt `URLConnection`, który umożliwi nam uzyskanie potrzebnych strumieni komunikacyjnych.
   ```
   URLConnection polaczenie = daneURL.openConnection();
   ```

3. **Poinformuj przeglądarkę, że nie należy przechowywać danych w pamięci podręcznej.**
   ```
   polaczenie.setCaches(false);
   ```

4. **Podaj wszystkie nagłówki HTTP, których chcesz użyć.** Jeśli chcesz podać jakieś nagłówki żądania HTTP (patrz rozdział 4) to będziesz mógł to zrobić przy wykorzystaniu metody `setRequestProperty`.
   ```
   polaczenie.setRequestProperty("naglowek", "wartosc");
   ```

5. **Stwórz strumień wejściowy.** Jest kilka rodzajów strumieni którymi możesz się posłużyć, jednak najczęściej stosowanym jest `BufferedReader`. To właśnie w momencie tworzenia strumieni, w niewidoczny sposób jest nawiązywane połączenie sieciowe z serwerem.
   ```
   BufferedReader in =
     new BufferedReader( new InputStreamReader(
                           polaczenie.getInputStream()));
   ```

6. **Odczytaj wszystkie wiersze dokumentu.** Po prostu odczytuj kolejne wiersze, aż do momentu gdy strumień wejściowy zwróci wartość `null`.
   ```
   String wiersz;
   while ((wiersz = in.readLine()) != null) {
     zrobCosZ(wiersz);
   }
   ```

7. **Zamknij strumień wejściowy.**
```
in.close();
```

Przesyłanie serializowanych danych: Kod apletu

1. **Stwórz obiekt `URL` odwołujący się do komputera, z którego został pobrany aplet.** Najlepszym sposobem jest podanie końcowej części adresu i automatyczne skonstruowanie całego adresu URL.
```
URL biezacaStrona = getCodeBase();
String protokol = biezacaStrona.getProtocol();
String host = biezacaStrona.getHost();
int port = biezacaStrona.getPort();
String konURL = "/servlet/jakisSerwlet";
URL daneURL = new URL(protokol, host, port, konURL);
```

2. **Stwórz obiekt `URLConnection`.** Obiekt `URLConnection` zwraca metoda `openConnection` klasy `URL`. Obiekt ten posłuży nam do uzyskania strumieni komunikacyjnych.
```
URLConnection polaczenie = daneURL.openConnection();
```

3. **Poinformuj przeglądarkę, że nie należy przechowywać danych w pamięci podręcznej.**
```
polaczenie.setUseCaches(false);
```

4. **Podaj wszystkie nagłówki HTTP, których chcesz użyć.**
```
polaczenie.setRequestProperty("naglowek", "wartosc");
```

5. **Stwórz strumień `ObjectInputStream`.** Konstruktor tej klasy wymaga podania zwyczajnego, nieprzetworzonego strumienia wejściowego, pobranego z obiektu klasy `URLConnection`.
```
ObjectInputStream in =
  new ObjectInputStream( polaczenie.getInputStream());
```

6. **Odczytaj dane przy użyciu metody `readObject`.** Metoda `readObject` zwraca dane typu `Object`; a zatem otrzymane wyniki trzeba rzutować do konkretnego typu, odpowiadającego typowi danych przesłanych przez serwlet.
```
JakasKlasa wartosc = (JakasKlasa) in.readObject();
```

7. **Zamknij strumień wejściowy.**
```
in.close();
```

Przesyłanie serializowanych danych: Kod serwletu

1. **Określ, że przesyłane będą dane binarne.** W tym celu należy podać, że typem MIME odpowiedzi będzie `application/x-java-serialized-object`. Jest to standardowy typ MIME obiektów zakodowanych przy użyciu strumienia `ObjectOutputStream`; w praktyce jednak, ze względu na fakt iż to aplet (a nie przeglądarka) odczytuje przesyłane informacje, określanie typu MIME nie ma większego znaczenia. Więcej informacji na temat typów MIME znajdziesz w podrozdziale 7.2, w części poświęconej nagłówkowi odpowiedzi `Content-Type`.
```
String typ = "application/x-java-serialized-object";
response.setContentType(typ);
```

2. **Stwórz strumień `ObjectOutputStream`.**
```
ObjectOutputStream out =
  new ObjectOutputStream(response.getOutputStream());
```

3. **Zapisz dane przy użyciu metody `writeObject`.** W ten sposób można przesyłać znaczą większość wbudowanych typów danych. Jednak abyś mógł przesyłać w ten sposób klasy, które samemu zdefiniowałeś, będziesz musiał zaimplementować w nich interfejs `Serializable`.
```
JakasKlasa wartosc = new JakasKlasa(...);
out.writeObject(wartosc);
```

4. **Opróżnij strumień, aby mieć pewność, że wszystkie zapisane w nim informacje zostały przesłane do klienta.**
```
out.flush();
```

Przesyłanie danych metodą POST i bezpośrednie przetwarzanie wyników (tunelowanie HTTP)

1. **Stwórz obiekt `URL` odwołujący się do komputera, z którego aplet został pobrany.** Najlepszym sposobem jest podanie końcowej części adresu i automatyczne skonstruowanie całego adresu URL.
```
URL biezacaStrona = getCodeBase();
String protokol = biezacaStrona.getProtocol();
String host = biezacaStrona.getHost();
int port = biezacaStrona.getPort();
String konURL = "/servlet/jakisSerwlet";
URL daneURL = new URL(protokol, host, port, konURL);
```

2. **Stwórz obiekt `URLConnection`.**
```
URLConnection polaczenie = daneURL.openConnection();
```

3. **Poinformuj przeglądarkę, iż nie należy przechowywać danych w pamięci podręcznej.**
```
polaczenie.setUseCaches(false);
```

4. **Poproś system o możliwość przesyłania danych, a nie samego ich odczytywania.**
```
polaczenie.setDoOutput(true);
```

5. **Stwórz strumień `ByteArrayOutputStream`, który zostanie użyty do buforowania danych przesyłanych na serwer.** Strumień `ByteArrayOutputStream` jest wykorzystywany w tym samym celu co podczas obsługi trwałych połączeń HTTP, opisanych w podrozdziale 7.4. Służy on do określenie wielkości danych, która będzie nam potrzebna do wygenerowania nagłówka `Content-Length`.
```
ByteArrayOutputStream strumBajtowy =
  new ByteArrayOutpuStream(521);
```

6. **Skojarz strumień wyjściowy ze strumieniem `ByteArrayOutputStream`.** Jeśli chcesz przesłać zwyczajne informacje podane przez użytkownika w formularzu, to możesz do tego celu wykorzystać strumień `PrintWriter`; jeśli jednak chcesz przekazywać serializowane dane, to powinieneś posłużyć się strumieniem `ObjectOutputStream`.
```
PrintWriter out = new PrintWriter(strumBajtowy, true);
```

7. **Zapisz dane w buforze.** W przypadku zapisu informacji podanych w formularzy, posłuż się metodą `print`, natomiast serializowane obiekty wysokiego poziomu zapisuj przy użyciu metody `writeObject`.
```
String wartosc1 = URLEncoder.encode(jakasWartosc1);
String wartosc2 = URLEncoder.encode(jakasWartosc2);
String dane = "param1=" + wartosc1 +
              "&param2=" + wartosc2; // Zwróć uwagę na "&"!
out.print(dane); // Zwróć uwagę na użycie metody print, a nie println!
out.flush();     // Konieczne gdyż nie używamy metody println
```

8. **Wygeneruj nagłówek `Content-Length`.** W przypadku użycia metody `POST` nagłówek ten jest konieczny, choć w żądaniach `GET` nie jest on używany.
```
polaczenie.setRequestProperty
  ("Content-Length", String.valueOf(strumBajtowy.size()));
```

9. **Wygeneruj nagłówek `Content-Type`.** Przeglądarka Netscape Navigator domyślnie używa typu `multipart/form-data`, jednak przesyłanie zwyczajnych danych z formularza wymaga użycia typu `application/x-www-form-urlencoded` domyślnie stosowanego w Internet Explorerze. A zatem, w przypadku przesyłania zwyczajnych informacji podanych w

formularzu, aby zapewnić przenaszalność aplikacji, powinieneś jawnie określić typ MIME przesyłanych danych. W przypadku przesyłania serializowanych danych, określanie wartości tego nagłówka nie ma znaczenia.

```
polaczenie.setRequestProperty
  ("Content-Type", "application/x-www-form-urlencoded");
```

10. **Wyślij dane.**

```
strumBajtowy.writeTo(polaczenie.getOutputStream());
```

11. **Otwórz strumień wejściowy.** Do odczytu danych ASCII lub danych binarnych jest zazwyczaj używany strumień BufferedReader, natomiast do odczytu serializowanych obiektów Javy — strumień `ObjectInputStream`.

```
BufferedReader in =
  new BufferedReader(new InputStreamReader(
                       polaczenie.getInputStream()));
```

12. **Odczytaj wyniki.** Konkretny sposób odczytywania danych będzie zależny od rodzaju informacji przesyłanych przez serwer. Przedstawiony poniżej przykład *robi coś* z każdym wierszem odczytanych wyników:

```
String wiersz;
while ((wiersz = in.readLine()) != null) {
  zrobCosZ(wiersz);
}
```

Pomijanie serwera HTTP

Aplety mogą komunikować się bezpośrednio z serwerem działającym na komputerze z którego zostały pobrane, przy użyciu:

- zwyczajnych, nieprzetworzonych gniazd,
- gniazd oraz strumieni służących do transmisji obiektów,
- JDBC,
- RMI,
- innych protokołów sieciowych.

A.18 JDBC i zarządzanie pulami połączeń z bazami danych

Podstawowe etapy wykorzystania JDBC

1. **Załadowanie sterownika JDBC.** Listę wszystkich dostępnych sterowników można znaleźć pod adresem *http://industry.java.sun.com/products/jdbc/drivers*. Oto przykład:

```
Class.forName("pakiet.KlasaSterownika");
Class.forName("oracle.jdbc.driver.OracleDriver");
```

2. **Zdefiniowanie adresu URL połączenia.** Konkretny format adresu jaki należy zastosować, będzie podany w dokumentacji używanego sterownika.

```
String host = "dbhost.firma.com.pl";
String nazwaBazy = "jakasBaza";
int port = 1234;
String oracleURL = "jdbc:oracle:thin:@" + host +
                   ":" + port + ":" + "?SERVICENAME=" + nazwaBazy;
```

3. **Nawiązanie połączenia.**

```
String nazwaUzytkownika = "janek_kowalski";
String haslo = "supertajne";
```

```
Connection polaczenie =
  DriverManager.getConnection(oracleURL,nazwaUzytkownika,haslo);
```

Opcjonalnie, podczas wykonywania tego etapu można także pobrać informacje dotyczące bazy danych. Służy do tego metoda `getMetaData` klasy `Connection`. Metoda ta zwraca obiekt `DatabaseMetaData`, który udostępnia metody umożliwiające pobranie informacji o nazwie i wersji samej bazy danych (`getDatabaseProductName`, `getDatabaseProductVersion`) oraz używanym sterowniku (`getDriverName` oraz `getDriverVersion`).

4. **Stworzenie obiektu polecenia.**
```
Statement polecenie = polaczenie.createStatement();
```

5. **Wykonanie zapytania lub aktualizacji bazy.**
```
String zapytanie = "SELECT ko1, kol2, kol3 FROM tabela";
ResultSet zbiorWynikow = polecenie.executeQuery(zapytanie);
```

6. **Przetworzenie wyników.** Aby przejść do kolejnego wiersza wyników, należy wywołać metodę `next`; wartości poszczególnych kolumn z bieżącego wiersza pobierane są przy użyciu metod `get`*`Xxx`*`(`*`indeks`*`)` lub `get`*`Xxx`*`(`*`nazwaKolumny`*`)` (przy czym indeks pierwszej kolumny ma wartość 1, a nie 0).
```
while(zbiorWynikow.next()) {
  System.out.println(zbiorWynikow.getString(1) + " " +
                     zbiorWynikow.getString(2) + " " +
                     zbiorWynikow.getString(3));
}
```

7. **Zamknięcie połączenia.**
```
polaczenie.close();
```

Narzędzia obsługi baz danych

Poniżej opisałem statyczne metody zdefiniowane w klasie `DatabaseUtilities` (patrz listing 18.6).

- **getQueryResults**

  Metoda ta nawiązuje połączenie z bazą danych, wykonuje podane zapytanie, pobiera wszystkie wiersze wyników jako tablice łańcuchów znaków i zapisuje je w obiekcie `DBResults` (patrz listing 18.7). W obiekcie `DBResults` umieszczane są także — nazwa oprogramowania serwera bazy danych, numer jego wersji, nazwy wszystkich pobranych kolumn oraz obiekt `Connection`. Dostępne są dwie wersje tej metody — pierwsza z nich nawiązuje nowe połączenie z bazą danych, a druga wykorzystuje już istniejące połączenie. Klasa `DBResults` dysponuje prostą metodą `toHTMLTable`, która przedstawia wszystkie wyniki w formie tabeli HTML, której można użyć jako zwyczajną tabelę bądź jako arkusz kalkulacyjny programu Microsoft Excel.
- **createTable**

  Metoda ta pobiera następujące informacje: nazwę tabeli, łańcuch znaków określający formaty poszczególnych kolumn oraz tablicę łańcuchów znaków zawierającą poszczególne wiersze tabeli. Na podstawie tych informacji metoda nawiązuje połączenie z bazą danych, usuwa istniejącą wersję podanej tabeli, wykonuje polecenie `CREATE TABLE` o podanym formacie, a następnie serię poleceń `INSERT INTO` wstawiając do nowo utworzonej tabeli podane wiersze danych. Także ta metoda jest dostępna w dwóch wersjach, pierwsza z nich nawiązuje nowe połączenie z bazą danych, a druga używa połączenia już istniejącego.
- **printTable**

  Dysponując nazwą tabeli, metoda `printTable` nawiązuje połączenie ze wskazaną bazą danych, pobiera wszystkie wiersze tabeli i wyświetla je przy wykorzystaniu standardowego strumienia wyjściowego. Metoda pobiera wyniki, zamieniając nazwę tabeli

na zapytanie o postaci "`SELECT * FROM nazwaTabeli`" i przekazując je w wywołaniu metody `getQueryResults`.

- **printTableData**

  Dysponując obiektem `DBResults` uzyskanym w wyniku wykonania zapytania, metoda ta wyświetla wyniki wykorzystując do tego standardowy strumień wyjściowy. Metoda ta jest wykorzystywana przez metodę `printTable`, jednak można jej także używać do testowania i prezentacji dowolnych wyników pobieranych z baz danych.

Przygotowane polecenia (prekompilowane zapytania)

- **Użyj metody `polaczenie.prepareStatement` aby skompilować podane polecenie SQL; parametry tego polecenia oznacz znakami zapytania.**

```
String szablonPolecenia =
            "UPDATE pracownik SET pensja = ? WHERE id = ? ";
PreparedStatement polecenie =
            polaczenie.prepareStatement(szablonPolecenia);
```

- **Użyj metod `polecenie.setXxx`, aby określić wartości parametrów przygotowanego polecenia.**

```
polecenie.setFloat(1, 1.234);
polecenie.setInt(2, 33);
```

- **Wykonaj polecenie przy użyciu metody `execute`.**

```
polecenie.execute();
```

Etapy implementacji puli połączeń

Jeśli nie chcesz zawracać sobie głowy szczegółami implementacji puli połączeń, po prostu posłuż się klasą `ConnectionPool` stworzoną i opisaną w 18 rozdziale niniejszej książki. W przeciwnym przypadku zaimplementuj opisane poniżej możliwości funkcjonalne puli połączeń:

1. **Nawiązanie połączeń z bazą danych.**

   To zadanie należy wykonać w konstruktorze klasy. Konstruktor ten, należy natomiast wywoływać w metodzie `init` serwletu. W przedstawionym poniżej fragmencie kodu do przechowywania dostępnych, oczekujących połączeń, oraz aktualnie wykorzystywanych, niedostępnych połączeń z bazą danych używane są wektory (obiekty `Vector`):

```
availableConnections = new Vector(initialConnections);
busyConnections = new Vector();
for(int i=0; i<initialConnections; i++) {
  availableConnections.addElement(makeNewConnection());
}
```

2. **Zarządzanie dostępnymi połączeniami.**

   Jeśli jest potrzebne połączenie i jest dostępne nieużywane połączenie, to należy je przenieść na listę połączeń wykorzystywanych i zwrócić. Lista połączeń wykorzystywanych jest stosowana do sprawdzania limitu ilości wszystkich połączeń, oraz w sytuacji gdy pula otrzyma jawne żądanie zamknięcia wszystkich połączeń z bazą danych. Usunięcie połączenia udostępnia wolny element, który może zostać wykorzystany przez procesy, które potrzebowały połączenia w chwili, gdy limit dostępnych połączeń został wyczerpany. A zatem, w takiej sytuacji należy uaktywnić wszystkie oczekujące wątki (przy użyciu metody `notifyAll`) i sprawdzić czy można kontynuować ich realizację.

```
public synchronized Connection getConnection()
    throws SQLException {
  if (!availableConnections.isEmpty()) {
    Connection existingConnection =
      (Connection)availableConnections.lastElement();
    int lastIndex = availableConnections.size() - 1;
```

```
      availableConnections.removeElementAt(lastIndex);
      if (existingConnection.isClosed()) {
        notifyAll();    // wolne miejsce dla oczekujących wątków
        return(getConnection()); // powtórz proces
      } else {
        busyConnections.addElement(existingConnection);
        return(existingConnection);
      }
    }
  }
```

3. **Nawiązywanie nowych połączeń z bazą danych.**

   Jeśli jest potrzebne połączenie, a w danej chwili nie ma żadnego dostępnego połączenia i limit ilości połączeń został wyczerpany, to należy uruchomić wątek działający w tle, którego zadaniem będzie nawiązanie nowego połączenia. Następnie trzeba zaczekać na pojawienie się pierwszego dostępnego połączenia, niezależnie od tego czy będzie to już istniejące, czy też nowoutworzone połączenie.

```
if ((totalConnections() < maxConnections) &&
    !connectionPending) { // pending - trwa nawiązanie połączenia w tle
  makeBackgroundConnection();
}
try {
  wait(); // zablokuj i zawieś ten wątek.
} catch(InterruptedException ie) {}
return(getConnection()); // spróbuj jeszcze raz
```

4. **Oczekiwanie na pojawienie się dostępnego połączenia.**

   Ta sytuacja zachodzi gdy nie ma żadnych dostępnych połączeń, a jednocześnie został wyczerpany limit ilości połączeń. Oczekiwanie na połączenie powinno się zakończyć bez konieczności ciągłego sprawdzania czy jakieś połączenie jest już dostępne. Naturalnym rozwiązaniem tego problemu jest wykorzystanie metody `wait`, która usuwa blokadę synchronizującą wątek i zawiesza jego wykonywanie aż do czasu wywołania metody `notify` lub `notifyAll`.

```
try {
  wait();
} catch(InterruptedException ie) {}
return(getConnection());
```

5. **Zamknięcie połączeń gdy będzie to konieczne.**

   Zwróć uwagę, iż połączenia są zamykane podczas automatycznego usuwania ich z pamięci, a zatem nie zawsze będziesz musiał jawnie je zamykać. Jednak czasami będziesz chciał dysponować większą i bardziej jawną kontrolną nad przebiegiem całego procesu.

```
public synchronized void closeAllConnections() {
  // metoda closeConnection pobiera wszystkie połączenia
  // przechowywane w podanym wektorze, zamyka każde z nich wywołując
  // metodę close i ignoruje wszelkie zgłaszane wyjątki
  closeConnections(availableConnections);
  availableConnections = new Vector();
  closeConnections(busyConnections);
  busyConnections = new Vector();
}
```

Rezultaty czasowe wykorzystania puli połączeń	
Warunki	Średni czas
Wolne połączenie modemowe z bazą danych, 10 początkowo dostępnych połączeń, dopuszczalny limit 50 połączeń (`ConnectionPoolServlet`).	11 sekund
Wolne połączenie modemowe z bazą danych, wielokrotnie wykorzystywane pojedyncze połączenie (`ConnectionPoolServlet2`).	22 sekundy

Wolne połączenie modemowe z bazą danych, pula połączeń nie używana (`ConnectionPoolServlet3`).	82 sekundy
Szybkie połączenie z bazą danych poprzez sieć lokalną, 10 początkowo dostępnych połączeń, dopuszczalny limit 50 połączeń (`ConnectionPoolServlet`).	1,8 sekundy
Szybkie połączenie z bazą danych poprzez sieć lokalną, wielokrotnie wykorzystywane pojedyncze połączenie (`ConnectionPoolServlet2`).	2,0 sekundy
Szybkie połączenie z bazą danych poprzez sieć lokalną, pula połączeń nie używana (`ConnectionPoolServlet3`).	2,8 sekundy