# JSP i XML


Autorzy: Casey Kochmer, Erica Frandsen
Tłumaczenie: Bartosz Grabski
ISBN: 83-7197-802-2
Tytuł oryginału: JSP and XML. Integrating XML and Web Services in Your JSP Application
Format: B5, stron: 474
Przykłady na ftp: 53 kB

Książka „JSP i XML" ukazuje praktyczne aspekty wykorzystania języka XML do budowy aplikacji internetowych przy wykorzystaniu Java Server Pages (JSP). Po omówieniu podstaw JSP i XML-a oraz towarzyszących im standardów (XSL, XPath, DOM) autor opisuje bardziej zaawansowane aspekty użycia języka XML w projektach opartych o JSP i przechodzi do omówienia metod tworzenia usług opartych na WWW.

W książce omówiono:

- XML, XSL i XPath
- DOM i parsery DOM (JDOM, dom4j)
- Szybkie parsowanie za pomocą SAX
- Tworzenie własnych znaczników JSP
- Uruchamianie własnych usług WWW
- Opis usługi WWW za pomocą WSDL

Książce towarzyszą dodatki poświęcone konfiguracji środowiska programistycznego, wprowadzeniu do JSP, tworzeniu bibliotek znaczników JSP oraz standardom XML, XSLT i Xpath.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

# Spis treści

Rozdział 4.

# Szybkie wprowadzenie do JSP i XML-a

W tym rozdziale zajmujemy się zagadnieniem łączenia JSP i XML-a. Pokażemy w nim, jak łatwo umieścić XML i XSLT na stronie JSP. Omówimy również kilka API, które umożliwiają wykorzystanie XML-a w Javie.

Rozdział zaczyna się krótkimi rozważaniami na temat powiązań między XML-em i JSP. Następnie przedstawiamy trzy podstawowe API: JAXP, Xerces i Xalan. Wykorzystując te biblioteki Javy pogłębimy wiedzę zdobytą w rozdziale 2., „Wprowadzenie do XML-a i XSL-a", i przedstawimy praktyczne zastosowania w środowisku JSP. Po omówieniu prostych kodów pokażemy pełniejszy przykład budowy procesora biuletynu do tworzenia strony informacyjnej. Na zakończenie rozdziału skupimy się na opisaniu innych API, które są wykorzystywane przez programistów JSP i XML.

## Związek pomiędzy XML-em i JSP

Wspólne wykorzystywanie JSP i XML-a ma sporo sensu. JSP służy do tworzenia wyjścia znakowego, a XML do opisywania danych znakowych; razem tworzą naturalne połączenie. Poznanie samego XML-a nie wystarczy, ponieważ XML ma na celu znakowanie danych, a nie ich przenoszenie. JSP pozwala przenosić dane do i ze źródła danych XML. W celu skutecznego wykorzystania XML-a w projekcie musimy poznać metody pobierania i umieszczania danych znakowych za pomocą JSP. Poniżej wymieniamy kilka często stosowanych metod wspólnego wykorzystania JSP i XML:

- Dane są przechowywane w pliku XML. Za pomocą JSP dane są importowane i konwertowane na stronę HTML w celu wyświetlenia w przeglądarce.
- Dane są przechowywane w bazie danych. Tworzony jest proces Java w celu odczytania danych z bazy danych i zamianę ich na dokument XML. Następnie dane są albo przesyłane do użytkownika jako plik XML, bądź plik XML jest przekształcany na inny format, taki jak strona HTML, do przeglądania przez użytkownika.

- Dane są przechowywane w pliku XML. Plik jest przetwarzany za pomocą Javy, a dane są formatowane i umieszczane w bazie danych. Za pomocą JSP uzyskuje się dostęp do bazy danych i wyświetla dane.
- Dane inicjalizacyjne aplikacji WWW są przechowywane w pliku XML. Plik jest odczytywany za pomocą JSP, a zachowanie aplikacji jest zmieniane w zależności od zawartości pliku.
- Dane są przenoszone między dwoma systemami w pliku XML. Technologia JSP jest wykorzystywana jako interfejs użytkownika w celu inicjacji procesu przesyłania albo pobierania danych.

Te przykłady wskazują, że dane są pobierane, zapisywane i konwertowane do różnych formatów. Trzeba nauczyć się wykorzystywać JSP jako warstwę pośrednią w celu ułatwienia przenoszenia danych używających XML albo innych formatów.

# Ostrzeżenie

Nadszedł właściwy moment na ostrzeżenie związane z wykorzystywaniem XML-a. XML to wspaniała technologia i można za jej pomocą robić zdumiewające rzeczy. Jednakże chcemy zauważyć, że nie jest to jedyne i właściwe rozwiązanie dla wszystkich problemów. Odpowiednie wykorzystanie XML-a pozwoli zwiększyć wydajność i oszczędzić czas. Złe użycie XML spowoduje *spowolnienie* działania aplikacji WWW, zmarnowanie czasu i pieniędzy przeznaczonych na rozwój projektu.

Systemy raportowania są zwykle dobrym miejscem na wykorzystanie XML-a i XSL-a z powodu możliwości filtrowania i formatowania, które można zastosować na tych samych zbiorach danych do utworzenia różnych raportów. Przykłady tego można znaleźć w rozdziale 11., „Wykorzystanie XML-a w systemach raportowania”, oraz w rozdziale 12., „Zaawansowany XML w systemach raportowania”. Można także użyć danych zapisanych w postaci XML na stronie WWW, umożliwiając klientom reorganizację danych w raportach bez potrzeby ponownego łączenia się z serwerem. Przykłady tego można znaleźć w rozdziale 13., „Rozważania o przeglądarce współpracującej z XML-em”.

Ponadto XML dodaje warstwę abstrakcji pomiędzy algorytmami biznesowymi i algorytmami prezentacji. Oznacza to, że zmiana wyglądu raportu wymaga jedynie aktualizacji arkusza stylów XSL, podczas gdy poziom generacji danych pozostaje niezmieniony. Podobnie, jeżeli zmienią się obiekty biznesowe tworzące raport, mogą one wciąż generować ten sam kod XML, dzięki czemu warstwa prezentacji będzie wciąż działać poprawnie. Tak więc warstwa XML jest jak gumowa uszczelka, która zapewnia sprawne działanie aplikacji WWW.

Jak powiedziano wcześniej w tej książce, format XML nie został zaprojektowany jako alternatywa dla baz danych. Chociaż pliki XML mogą być używane do przechowywania danych, najlepiej je wykorzystywać jako pliki inicjalizacyjne albo małe zbiory danych. Wykorzystanie XML-a do przechowywania i odczytywania dużych ilości danych jest metodą wolniejszą niż używanie bazy danych. Bazy danych udostępniają dodatkowe możliwości, takie jak zarządzanie transakcjami, które nie są częścią specyfikacji XML. Podczas projektowania dużych systemów wykorzystujących XML, powinieneś

dopasować projekt do mocnych i słabych stron XML-a. XML nie zastępuje baz danych, a raczej je uzupełnia. Projekty systemów powinny zakładać użycie obu tych technologii w zależności od potrzeb. W dalszych rozdziałach zademonstrujemy połączenie obu technologii.

Zdarzają się przypadki, kiedy wykorzystanie XML-a do przechowywania danych może być dobrym wyborem. Mógłbyś przykładowo chcieć umożliwić wielu klientom przeglądanie niezmienionych danych bez potrzeby połączenia z bazą danych bądź z siecią. Można by względnie szybko stworzyć bardzo prosty system, który wykorzystywał będzie przeglądarkę, XML i kilka arkuszy stylów do osiągnięcia pożądanego celu.

Czasami w projektach dane są zamieniane na format XML tylko po to, aby używać XML-a. Jest to złe podejście, ponieważ proces tworzenia i używania dokumentów XML dodaje kolejną warstwę przetwarzania do projektu. To zarówno zwiększa koszt z powodu dłuższego czasu rozwoju systemu, jak i spowalnia przetwarzanie. Dane powinny być zamieniane na XML tylko wtedy, kiedy wykorzystanie XML-a jest najlepszym rozwiązaniem.

Podstawowa zasada brzmi: nigdy nie przekształcaj danych do postaci XML, jeżeli nie ma to określonego celu. Nie używaj technologii XML dla niej samej.

# JAXP, Xerces i Xalan

Istnieje wiele narzędzi do przetwarzania i przekształcania XML-a. Każde z nich ma swoje zalety i wady. Jednakże trzeba od czegoś zacząć, omówimy więc API, które są najczęściej używane z JSP. Przełoży się to na napisanie przykładu używającego API JAXP. API JAXP używa parsera Javy Xerces i procesora arkuszy stylów Xalan. Jeżeli jeszcze nie umieściłeś plików *xerces.jar* i *xalan.jar* w katalogu *lib* serwera Tomcat, zrób to teraz. Lokalizacja plików do skopiowania podana jest poniżej.

## Xerces (parser XML)

Mówiąc prosto, Xerces jest parserem i generatorem XML. Jego nazwa nawiązuje do motyla Xerces Blue. Parser implementuje specyfikacje W3C XML i DOM (poziomy 1. i 2.), oraz standard SAX.

Xerces jest dołączany do Tomcata, dlatego nie musisz go instalować. Być może jednak za pewien czas będziesz potrzebował najnowszej wersji z powodu nowych możliwości. Użyj następującego URL-a, aby skopiować najnowszą wersję Xercesa: *http://xml.apache.org/xerces2-j/index.html*.

DOM i SAX to różne metody przetwarzania XML, które będą omówione w rozdziale 5., „Używanie DOM”, i rozdziale 6., „Programowanie SAX”. Mówiąc krótko, DOM tworzy w pamięci drzewiastą strukturę, którą można programowo manipulować. SAX jest sekwencyjnym, sterowanym zdarzeniami czytnikiem XML.

W tej książce używamy Xercesa w wersji 1.4.3.

## Xalan (procesor XSLT)

Xalan jest procesorem arkuszy stylów XSLT. Jego nazwa nawiązuje do rzadkiego instrumentu muzycznego. Xalan implementuje specyfikację W3C XSLT i instrukcje XPath. Niestety Xalan nie jest dołączany do Tomcata, dlatego trzeba go zainstalować. Skopiuj Xalana z *http://xml.apache.org/xalan-j/index.html*. Umieść plik *xalan.jar* w katalogu *lib* serwera Tomcat. Zatrzymaj i uruchom ponownie serwer w celu zarejestrowania nowych klas.

W tej książce używamy Xalana w wersji 2.2.d10.

## JAXP (podstawowe przetwarzanie XML)

Technicznie JAXP to API dla przetwarzania XML. Dokładniej mówiąc, jest to warstwa abstrakcji, ponieważ nie zawiera żadnej funkcjonalności przetwarzania. JAXP jest pojedynczym API, przez które można uzyskać dostęp do różnych obsługiwanych procesorów XML i XSL. Jest to podobne do JDBC dla parserów XML. JAXP nie dodaje żadnej nowej funkcjonalności do przetwarzania XML, umożliwia jedynie standardowy sposób wykorzystania różnych implementacji procesorów XML. Poprzez JAXP można używać różnych implementacji standardowych procesorów XML, takich jak SAX i DOM.

Aby skutecznie wykorzystywać JAXP ważne jest, aby używać tylko jego API i unikać bezpośredniego użycia jakichkolwiek podrzędnych API zależnych od implementacji. Przyczyną tego faktu jest to, że warstwa JAXP jest warstwą abstrakcji. Jakiekolwiek bezpośrednie wywołania podrzędnych API XML zaprzeczają idei posiadania warstwy podstawowej. Ta zasada jest szczególnie istotna, kiedy aplikacja ma być wdrożona w różnych sytuacjach, a w każdym wdrożeniu zostanie wykorzystany inny procesor XML. JAXP zapewnia aplikacji niezależność od implementacji procesora XML.

JAXP został stworzony w celu ułatwienia wykorzystania XML-a na platformie Javy. Osiągnięto to dzięki wspólnemu interfejsowi do dowolnie wybranego procesora XML.

JAXP jest zawarty w serwerze Tomcat i nie wymaga żadnej instalacji. Jeżeli chcesz przejrzeć źródło JAXP albo sprawdzić szczegóły dotyczące najnowszej wersji, odwiedź stronę *http://java.sun.com/xml/jaxp.html*.

W tej książce używamy JAXP w wersji 1.1.

### Przykład z wykorzystaniem JAXP, XSL-a i XML-a

Ten przykład symuluje losowe generowanie bannerów, które można spotkać w całej sieci WWW. Istotne jest, że ta wersja jest ograniczona do wyświetlania tylko tekstu. W rozdziale 1., „Integracja JSP i danych”, utworzyliśmy podobny przykład używający różnych metod. Dane dla tej wersji będą przechowane w statycznym pliku XML. Za pomocą JAXP zastosujemy arkusz stylów XSL do dokumentu XML i wyślemy wyniki na wyjście. Na początek przykład będzie służył do wyświetlenia wszystkich danych znajdujących się w pliku XML. Następnie zmodyfikujemy kod, aby wybierał losowo łącze do wyświetlenia.

Zacznijmy od pliku danych XML, pokazanego na wydruku 4.1. Ten i pozostałe pliki w tym rozdziale powinny zostać zapisane w katalogu *webapps* serwera Tomcat w ścieżce dostępu *webapps/xmlbook/chapter4/BannerAds.xml*.

**Wydruk 4.1.** *BannerAds.xml; plik danych XML dla przykładu JAXP*

```
<?xml version="1.0"?>
<BANNERS>
    <BANNERAD>
        <NAME>JSPInsider</NAME>
        <LINK>http://www.jspinsider.com</LINK>
        <LINKTEXT>JSP News</LINKTEXT>
    </BANNERAD>
    <BANNERAD>
        <NAME>Sun</NAME>
        <LINK>http://www.sun.com</LINK>
        <LINKTEXT>Witryna Java</LINKTEXT>
    </BANNERAD>
    <BANNERAD>
        <NAME>Helion</NAME>
        <LINK>http://www.helion.pl</LINK>
        <LINKTEXT>Wydawnictwo Helion</LINKTEXT>
    </BANNERAD>
    <BANNERAD>
        <NAME>Jakarta</NAME>
        <LINK>http://jakarta.apache.org</LINK>
        <LINKTEXT>Kewl Tools</LINKTEXT>
    </BANNERAD>
</BANNERS>
```

Zauważ, że w tym pliku znajdują się cztery grupy danych, z których każda zawiera informację o łączu bannera. Każda grupa danych jest właściwie zagnieżdżona w elemencie głównym `BANNERS`.

Następnie musimy utworzyć arkusz stylów XSL, pokazany na wydruku 4.2, w celu zastosowania formatowania do pliku XML. Mówiąc w skrócie, ten arkusz stylów utworzy wiersz tabeli dla każdego elementu dziecka `BANNERAD` znajdującego się wewnątrz elementu głównego `BANNERS`. Potem dane tekstowe znajdujące się w różnych węzłach zostaną wybrane i umieszczone w komórkach tabeli.

**Wydruk 4.2.** *BannerAds.xsl; arkusz stylów XSL dla przykładu JAXP*

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<HTML>
<HEAD><TITLE>Bannery z JAXP</TITLE></HEAD>
<BODY>
    <TABLE border="1">
    <TR>
        <TH>Nazwa</TH>
        <TH>Łącze</TH>
        <TH>Tekst łącza</TH>
    </TR>
```

```
    <xsl:for-each select="BANNERS/BANNERAD">
    <TR>
        <TD><xsl:value-of select="NAME" /></TD>
        <TD><xsl:value-of select="LINK" /></TD>
        <TD><xsl:value-of select="LINKTEXT" /></TD>
    </TR>
    </xsl:for-each>
    </TABLE>
</BODY>
</HTML>
</xsl:template>
</xsl:stylesheet>
```

Instrukcja XPath `select` wybiera wszystkie dzieci `BANNERAD` elementu głównego `BANNERS`.

```
<xsl:for-each select="BANNERS/BANNERAD">
```

Spowoduje to przetwarzanie ciała elementu `for-each` dla każdego elementu wybranego przez wyrażenie `select`.

Nadszedł czas, aby utworzyć stronę JSP, która używa JAXP w celu zastosowania przekształcania arkusza stylów do pliku XML. Kod jest pokazany na wydruku 4.3.

**Wydruk 4.3.** *BannerAds_JAXP.jsp; JSP dla przykładu JAXP*

```
<%@ page
  import="javax.xml.transform.*,
          javax.xml.transform.stream.*,
          java.io.*"
%>

<%
String  ls_path = request.getServletPath();
        ls_path = ls_path.substring(0,ls_path.indexOf("BannerAds_JAXP.jsp")) ;

String  ls_xml  = application.getRealPath(ls_path + "BannerAds.xml");
String  ls_xsl  = application.getRealPath(ls_path + "BannerAds.xsl");

StreamSource xml = new StreamSource(new File(ls_xml));
StreamSource xsl = new StreamSource(new File(ls_xsl));
StreamResult result = new StreamResult(out);

TransformerFactory tFactory = TransformerFactory.newInstance();
Transformer transformer = tFactory.newTransformer(xsl);
transformer.transform(xml, result);
%>
```

Na początku strony zadeklarowane są pakiety używane w kodzie. Pierwszy pakiet, `javax.xml.transform.*`, zawiera klasy `Transformer` i `TransformerFactory`. Te klasy przekształcają drzewo źródłowe, XML i XSL, w drzewo wynikowe. W następnym zadeklarowanym pakiecie, a mianowicie `javax.xml.tranform.stream.*`, znajdują się klasy `StreamResult` i `StreamSource`. `StreamSource` to obsługiwane źródło XML. Nie czyni żadnej różnicy, czy jest to plik czytany w strumieniu danych albo istniejąca reprezentacja danych XML. `StreamSource` zajmuje się za nas tymi szczegółami. `StreamResult`

określa, gdzie chcemy wysłać wynikowy dokument XML. Te klasy służą jako schowki dla plików XML i XSL przed przekształceniem i dla końcowego wyjścia po przetwarzaniu. Na końcu znajduje się pakiet `java.io.*`, który obsługuje zarządzanie plikami zewnętrznymi.

Następnie JSP odczytuje własną ścieżkę dostępu. Sprawdzamy ścieżkę dostępu w celu upewnienia się, że używamy właściwej ścieżki podczas pobierania plików XML i XSL. Kiedy już mamy określoną ścieżkę dostępu, używamy jej do umieszczenia plików XML i XSL w obiektach `StreamSource`, które służą jako schowki. Dalej tworzony jest obiekt `StreamResult` w celu przesłania wyników przekształcenia. W tym przykładzie wyniki są przesyłane bezpośrednio do wyjścia JSP, czyli obiektu niejawnego `out` typu `javax.servlet.jsp.JspWriter`.

Wszystko, co do tej pory zrobiliśmy, miało na celu przygotowanie do uruchomienia przekształcenia XML/XSLT. Następnie tworzony jest egzemplarz obiektu `TransformerFactory`. *Fabryki* (ang. *factories*) są używane do tworzenia określonych przekształceń. Fabryka ustawia wszystkie nadające się do ponownego wykorzystania informacje, których proces XML będzie później potrzebował podczas wykonywania rzeczywistego przekształcenia XSL danych XML. Używamy `TransformerFactory` do utworzenia określonego obiektu `Transformer`. Każdy utworzony `Transformer` obsługuje pojedynczy plik XSL. Gdybyśmy musieli przetworzyć pięć różnych plików XSL, utworzylibyśmy pięć egzemplarzy tego obiektu albo inicjalizowalibyśmy pojedynczy `Transformer` pięć razy. Tak więc podczas tworzenia obiektu `Transformer` określamy, którego pliku XSL ma on używać. Następnie wywołujemy metodę przekształcającą w celu przetworzenia dokumentu XML. Wymaga to wiedzy, który plik XML ma być przetwarzany (który obiekt `StreamSource`) oraz dokąd przesłać wyjście (`StreamResult`). Całe przekształcenie obejmuje tylko kilka linii kodu JAXP. Wyniki pokazano na rysunku 4.1.


**Rysunek 4.1.** *Wyniki przekształcenia JAXP*

Możliwość wyświetlenia wyników przekształcenia XSLT jest sama w sobie potężna, ale rozszerzymy ten przykład o dodatkową funkcjonalność. Rzeczywisty system obsługi bannerów wyświetlałby tylko jeden losowo wybrany banner. Zmodyfikujemy nasz przykład, aby działał podobnie.

## Przykład z parametrami XSL

Zaczniemy od dodania znacznika parametru do arkusza stylów XSL. Znacznik ten utworzy parametr, który poinformuje arkusz stylów, który banner ma zostać wyświetlony. Wartość tego parametru będzie ustawiana przez stronę JSP (a dokładnie przez JAXP). Na wydruku 4.4 pokazano nowy arkusz stylów ze zmianami zaznaczonymi drukiem wytłuszczonym.

**Wydruk 4.4.** *BannerAds_Param.xsl; parametr ustawiany zewnętrznie*

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:param name="Choose"/>
<xsl:template match="/">
<HTML>
<HEAD><TITLE>Bannery z parametrem</TITLE></HEAD>
<BODY>
    <TABLE border="1">
    <TR>
        <TH>Nazwa</TH>
        <TH>Łącze</TH>
        <TH>Tekst łącza</TH>
    </TR>
    <xsl:for-each select="BANNERS/BANNERAD[$Choose]">
    <TR>
        <TD><xsl:value-of select="NAME" /></TD>
        <TD><xsl:value-of select="LINK" /></TD>
        <TD><xsl:value-of select="LINKTEXT" /></TD>
    </TR>
    </xsl:for-each>
    </TABLE>
</BODY>
</HTML>
</xsl:template>
</xsl:stylesheet>
```

Wewnątrz elementu `stylesheet` dodaliśmy element parametru o nazwie `Choose`.

```
<xsl:param name="Choose"/>
```

Ten element parametru ma globalny zakres dla całego arkusza stylów, ponieważ został umieszczony na zewnątrz znacznika `xsl:template`. Zwykle podczas deklaracji parametru ustawiana jest również jego domyślna wartość za pomocą atrybutu `select`. Wybraliśmy możliwość nieustawiania wartości domyślnej; gdybyśmy jednak ją ustawili, wyglądałoby to w ten sposób:

```
<xsl:param name="Choose" select="2" />
```

Innym miejscem, w którym można zadeklarować parametry, jest początek szablonu. W takim przypadku zakres parametru jest ograniczony do szablonu, w którym parametr został zadeklarowany.

Następna zmieniona część oryginalnego arkusza stylów z wydruku 4.2 to instrukcja `select` pętli `for-each`.

```
<xsl:for-each select="BANNERS/BANNERAD[$Choose]">
```

Instrukcja XPath `select` znów wybierze wszystkie elementy dzieci `BANNERAD` elementu głównego `BANNERS`. Jednakże teraz tylko te elementy `BANNERAD`, których numer pozycji równa się parametrowi w nawiasach kwadratowych, zostaną wybrane do przetworzenia w pętli `for-each`. W rezultacie zostanie wybrana tylko jedna ze wszystkich możliwych opcji, która będzie umieszczona na wyjściu przez kod zawarty w tej pętli.

Aby odwołać się do parametru w dowolnym miejscu w jego zakresie, użyj znaku dolara () przed nazwą parametru.

Na koniec mamy nowy dokument JSP, pokazany na wydruku 4.5. Ta strona zawiera oryginalny kod z wydruku 4.3 z pewnym dodatkowym przetwarzaniem. Musimy teraz dowiedzieć się, ile bannerów znajduje się w pliku XML, a następnie losowo wybrać liczbę z tego zakresu. Po wybraniu tej liczby umieszczamy parametr w arkuszu stylów XSL, tak więc tylko te dane zostaną wyświetlone.

**Wydruk 4.5.** *BannerAds_Param.jsp; elementy i parametry XSL*

```
<%@ page
  import="javax.xml.transform.*,
          javax.xml.transform.stream.*,
          java.io.*,
          javax.xml.parsers.*,
          org.w3c.dom.*"
%>

<%
String  ls_path = request.getServletPath();
        ls_path = ls_path.substring(0,ls_path.indexOf("BannerAds_Param.jsp")) ;

String  ls_xml  = application.getRealPath(ls_path + "BannerAds.xml");
String  ls_xsl  = application.getRealPath(ls_path + "BannerAds_Param.xsl");

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setValidating(false);

DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse(new File(ls_xml));

NodeList nodes = doc.getElementsByTagName("BANNERAD");
int intNodeLength = nodes.getLength();

int intRandom = 1 + (int) (Math.random() * intNodeLength);

StreamSource xml = new StreamSource(new File(ls_xml));
StreamSource xsl = new StreamSource(new File(ls_xsl));

StreamResult result = new StreamResult(out);

TransformerFactory tFactory = TransformerFactory.newInstance();
Transformer transformer = tFactory.newTransformer(xsl);

transformer.setParameter("Choose", new Integer(intRandom));

transformer.transform(xml, result);
%>
```

Na nowej stronie JSP dodaliśmy dwa pakiety. Pierwszy dodany pakiet `javax.xml.parsers.*` daje nam dostęp do implementacji JAXP DOM i pozwala na utworzenie w pamięci drzewiastej struktury dokumentu XML za pomocą klas `DocumentBuilder`

i `Document`. Kolejny pakiet to `org.w3c.dom.*`. Za pomocą tego pakietu mamy dostęp do definicji DOM W3C. Dzięki temu możemy utworzyć zbiór wybranych węzłów, używając obiektu `NodesList`.

Następnie tworzymy egzemplarz obiektu `DocumentBuilderFactory`. Po utworzeniu obiekt ten zostanie wykorzystany do utworzenia obiektu `Document`, za pocą którego przetworzymy plik XML. Popatrz na linię:

```
factory.setValidating(false);
```

Ta linia jest ważna, ponieważ określa, czy prawidłowość strukturalna pliku XML jest sprawdzana podczas ładowania, czy też nie. Ponieważ ten plik nie ma DTD albo schematu, wyłączamy sprawdzanie dokumentu.

Krótko mówiąc, sprawdzanie prawidłowości struktury oznacza, że porównujemy strukturę dokumentu XML z definicją tej struktury. Jeżeli się zgadzają, dokument jest poprawny pod względem strukturalnym.

Po załadowaniu pliku XML do obiektu `Document` jesteśmy gotowi do obliczenia, ile danych o bannerach znajduje się w pliku XML:

```
NodeList nodes = doc.getElementsByTagName("BANNERAD");
int intNodeLength = nodes.getLength();
```

Patrząc na plik XML pokazany na wydruku 4.1, widzimy, że każdy element `BANNERAD` zawiera informację o jednym bannerze. Tworzymy `NodeList` i wybieramy tylko elementy o nazwie `BANNERAD`. `NodeList` zawiera teraz pojedynczy węzeł dla każdego bannera. Użycie `getLength()` na tej liście zwraca całkowitą liczbę bannerów.

W tej chwili chcielibyśmy wspomnieć, że liczenie elementów-dzieci jest złym pomysłem przy przetwarzaniu wielkich ilości danych. Działa to dobrze dla używanych przez nas dokumentów XML, ponieważ są one małe. Jednak podczas pracy z obszernymi dokumentami XML załadowanie całego dokumentu do pamięci jedynie w celu policzenia elementów zakończyłoby się ogromnym spadkiem wydajności przetwarzania.

Teraz możemy użyć metody `random` oraz liczby bannerów jako czynnika skalującego w celu otrzymania liczby losowej:

```
int intRandom = 1 + (int) (Math.random() * intNodeLength);
```

Następnie ustawiany jest parametr w arkuszu stylów:

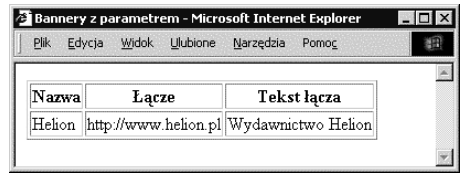
```
transformer.setParameter("Choose", new Integer(intRandom));
```

Ustawiamy parametr o nazwie `Choose` w arkuszu stylów XSL o wartości równej otrzymanej liczbie losowej. To ta liczba określi, który banner ma być wyświetlony.

To jest dopiero początek. Za pomocą podobnych technik moglibyśmy określić, które dane użytkownik ogląda, wykorzystując login użytkownika. Można też przekazywać parametry w celu zmiany sposobu formatowania raportu przez arkusz stylów XSL albo dynamicznej zmiany kolejności sortowania. Później, kiedy wkroczymy do królestwa tworzenia dokumentów XML opartych na danych pobieranych z bazy danych, nauczysz się, jak implementować te opcje.

Na rysunku 4.2 pokazano dane wyjściowe utworzone przez JSP. Są one oczywiście losowe, więc nie zobaczysz tej samej strony przy każdym wykonaniu kodu JSP.

**Rysunek 4.2.** *Losowo wybrany banner*



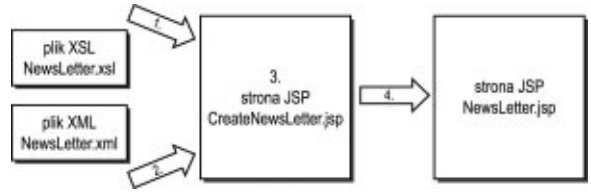

## Przesyłanie przekształcenia do pliku JSP

Czasami przetwarzanie plików XML i XSL jest nadmiarowe, ponieważ te same dane wyjściowe są tworzone wielokrotnie. Przykładem tego jest biuletyn. Załóżmy, że jest strona WWW, która wyświetla biuletyn. Zawartość tego biuletynu, znajdująca się w pliku XML, zmienia się raz w tygodniu, ale przez resztę czasu pozostaje ona niezmieniona. W tym przypadku byłoby marnotrawieniem mocy przetwarzania przekształcanie dokumentu XML za pomocą arkusza stylów XSL za każdym razem, kiedy użytkownik zażąda strony. Zamiast tego powinniśmy przesłać wyniki przekształcenia do innego pliku JSP i nakazać użytkownikowi pobranie uprzednio przetworzonej strony. W ten sposób możemy utworzyć przekształcenie, które będzie wykorzystywane tylko przy zmianie źródłowego pliku XML przez wywołanie strony przetwarzającej.

Popatrzmy teraz na stronę JSP, która przetwarza pliki XML i XSL i umieszcza wyniki na innej stronie JSP. Strona przetwarzająca nazywa się *CreateNewsLetter.jsp*, a wyniki są umieszczane na *NewsLetter.jsp*. Ten przykład jest luźno oparty na biuletynie JSPBuzz, który jest opublikowany na witrynie WWW JSP Insider.

Schemat przetwarzania jest pokazany na rysunku 4.3.

**Rysunek 4.3.** *Tworzenie nowego pliku z danych wyjściowych*




Kroki pokazane na rysunku 4.3 są następujące:

1. *CreateNewsLetter.jsp* wczytuje dokument XSL.
2. *CreateNewsLetter.jsp* wczytuje dokument XML.
3. XML i XSL są przekształcane do postaci wynikowej.
4. Wynik przekształcenia jest umieszczany w *NewsLetter.jsp*.

Kod tworzy biuletyn ze źródła danych XML i arkusza stylów XSL. Biuletyn zawiera artykuły i łącza na temat Javy i XML-a. Umieścimy wyniki przetwarzania JSP w innym pliku JSP, tak więc przekształcanie nie będzie wielokrotnie wywoływane i niepotrzebnie wykonywane. Dzięki temu łącza na stronie WWW mogą wskazywać plik wyjściowy zamiast pliku przetwarzającego.

Zaczniemy od nowego pliku XML znajdującego się na wydruku 4.6. Ten plik XML zawiera informacje o nagłówku i zawartości biuletynu. Zapisz ten plik jako *webapps/xmlbook/chapter4/NewsLetter.xml*.

**Wydruk 4.6.** *NewsLetter.xml; dane dla przykładu biuletynu*

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<newsletter volume="II" issue="15">
    <header>
        <title>Przykładowy biuletyn</title>
        <description>Demonstracja przesłania wyjścia przekształcenia XML/XSL
        ↳do pliku JSP</description>
        <date>15.06.2002</date>
    </header>
    <article position="1">
        <author>Dennis M. Sosnoski </author>
        <title>XML w Java : Modele dokumentu, część 1: Wydajność</title>
        <description>Niezbędna lektura dla każdego, kto używa Java i XML. Jest to
        ↳zarówno przegląd bieżących parserów XML, jak i porównanie wyników
        ↳wydajności. Świetny artykuł.</description>
        <link>http://www-106.ibm.com/developerworks/xml/library/
        ↳x-injava/index.html</link>
        <date>Wrzesień 2001</date>
    </article>
    <article position="2">
        <author>Marshall Lamb</author>
        <title>Generacja dynamicznego XML za pomocą technologii JavaServer Pages
        ↳</title>
        <description>Dobry artykuł opisujący wykorzystanie JSP do tworzenia
        ↳dynamicznej zawartości z użyciem XML. </description>
        <link>http://www-106.ibm.com/developerworks/library/j-dynxml.html</link>
        <date>Grudzień 2001</date>
    </article>
    <links position="1">
        <title>Witryna Tomcat</title>
        <link>http://jakarta.apache.org/tomcat/index.html</link>
    </links>
    <links position="2">
        <title>Witryna W3C XML</title>
        <link>http://www.w3.org/XML/</link>
    </links>
    <links position="3">
        <title>Podręcznik Java</title>
        <link>http://java.sun.com/docs/books/tutorial/</link>
    </links>
</newsletter>
```

Następnie mamy arkusz stylów XSL. Ten plik, pokazany na wydruku 4.7, powinien zostać zapisany jako *webapps/xmlbook/chapter4/NewsLetter.xsl*.

**Wydruk 4.7.** *NewsLetter.xsl; arkusz stylów biuletynu*

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<xsl:stylesheet  version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
```

```
<xsl:template match="/">
    <html><head><title>Przykładowy biuletyn</title></head>
    <body bgcolor="#ECFCEA">

    <center><font size="4pt"><b>
        <xsl:value-of select="newsletter/header/title" /><br/>
        Rok <xsl:value-of select="newsletter/@volume" /> #
        <xsl:value-of select="newsletter/@issue" /><br/>
        <xsl:value-of select="newsletter/header/date" /><br/>
        <xsl:value-of select="newsletter/header/description" />
    </b></font></center>
    <table border="0" width="100%">
        <tr>
            <td bgcolor="#9EC49A">
                <b><font color="#000000">Spis treści</font></b>
            </td>
        </tr>
        <tr>
            <td><br/><a href="#article">Interesujące artykuły</a>
                <ul>
                <xsl:for-each select="newsletter/article" >
                    <li><a href="#article{@position}">
                        <xsl:value-of select="title"/></a>
                    </li>
                </xsl:for-each>
                </ul>
            </td>
        </tr>
        <tr>
            <td><a href="#article">Użyteczne łącza</a>
                <ul>
                <xsl:for-each select="newsletter/links" >
                    <li><a href="#links{@position}">
                        <xsl:value-of select="title"/></a>
                    </li>
                </xsl:for-each>
                </ul>
            </td>
        </tr>
    </table>
    <table width="100%"><tr>
        <td bgcolor="#9EC49A"><a name="article">
            <font color="#000000"><b>Interesujące artykuły</b></font></a>
        </td></tr>
    </table>
    <xsl:apply-templates select="newsletter/article" />
    <table width="100%"><tr>
        <td bgcolor="#9EC49A"><a name="links">
            <font color="#000000"><b>Użyteczne łącza</b></font></a>
        </td></tr>
    </table>
    <xsl:apply-templates select="newsletter/links" />
    </body>
    </html>
</xsl:template>
<xsl:template match="newsletter/article">
    <table style="border-style:groove;border-width:2px;" width="100%">
```

```
        <tr><td>
            <table border="0" width="100%">
                <tr><td colspan="2">
                    <a href="{link}" name="article{@position}">
                    <xsl:value-of select="title"/></a>
                    </td>
                </tr>
                <tr>
                    <td width="50%"><xsl:value-of select="author"/>
                    </td>
                    <td><xsl:value-of select="date"/></td>
                </tr>
                <tr>
                    <td colspan="2"><xsl:value-of select="description"/></td>
                </tr>
            </table>
        </td></tr>
    </table><br/>
</xsl:template>

<xsl:template match="newsletter/links">
    <table border="0" width="100%">
        <tr>
            <td><a href="{link}" name="link{@position}">
                <xsl:value-of select="title"/></a>
            </td>
        </tr>
    </table>
</xsl:template>
</xsl:stylesheet>
```

## Więcej o szablonach XSL

W tym arkuszu stylów używane są głównie elementy omówione w rozdziale 2., „Wprowadzenie do XML-a i XSL-a". Znajduje się w nim jednak kilka nowych rzeczy, które musimy omówić. Pierwszą z nich jest użycie szablonów. W rozdziale 2. wprowadziliśmy definicję szablonów. Przypominamy, że szablony udostępniają metody, za pomocą których dopasowujemy i pobieramy dane z dokumentu XML. Atrybut `match` znacznika `template` określa, które części dokumentu XML zostaną dopasowane przez element szablonu. Oto przykład:

```
<xsl:template match="/">
```

Ten znacznik szablonu, znajdujący się na początku wydruku 4.7, wybiera element główny dokumentu XML przez użycie `"/"`. Cały dokument XML jest przetwarzany przez ten szablon.

Jednak szablony mogą być tak zdefiniowane, że formatują albo przekształcają tylko wybraną część dokumentu XML. Zauważ, że instrukcja XPath jest używana w atrybucie szablonu `match` w celu określenia, która część dokumentu XML zostanie wybrana do szablonu. Oznacza to, że możemy być tutaj bardziej dokładni lub przeciwnie — ogólni, w zależności od tego na ile wyrażenia XPath pozwalają nam na wybranie, które elementy XML zostaną dopasowane do szablonu.

W arkuszu stylów z wydruku 4.7 utworzyliśmy trzy szablony. Pierwszy, który został omówiony wcześniej, wybiera cały dokument XML. Pozostałe dwa formatują tylko elementy dzieci `article` i `links` elementu głównego. Te szablony są zdefiniowane na końcu wydruku. Szablon artykułu wygląda następująco:

```
<xsl:template match="newsletter/links">
    <table border="0" width="100%">
        <tr>
            <td><a href="{link}" name="link{@position}">
                <xsl:value-of select="title"/></a>
            </td>
        </tr>
    </table>
</xsl:template>
```

Zauważ, że atrybut `select` znacznika szablonu wybiera tylko te elementy `links`, które są dziećmi elementu głównego `newsletter`. Szablony są bardzo przydatne podczas organizacji dużych arkuszy stylów. Używanie różnych szablonów do formatowania określonych elementów bardziej ułatwia wyszukiwanie i szybką aktualizację albo zmianę formatowania niż przeszukiwanie pojedynczego wielkiego szablonu.

Jeżeli nie ma żadnych szablonów, które dopasowują części dokumentu XML, niedopasowane części dokumentu będą wysyłane na wyjście jako dane tekstowe bez znakowania. Dzieje się tak, ponieważ szablon domyślny dla elementów odrzuca znaczniki i wysyła na wyjście tekst. Na przykład wyobraź sobie arkusz stylów z wydruku 4.7 bez pierwszego szablonu, który dopasowuje element główny. Jeżeli chciałbyś zobaczyć efekt, to po uruchomieniu strony JSP (dalej w rozdziale) usuń pierwszy szablon i zobacz, co się wydarzy.

Pierwszą częścią danych wyjściowych dokumentu XML będą dane tekstowe z elementów niedopasowanych w jakichkolwiek szablonach. W tym przykładzie jest to element `header` i jego dzieci (ten tekst jest pierwszym, ponieważ przetwarzanie wykonywane jest od początku w kolejności źródłowego pliku XML, kiedy nie ma żadnych szablonów, które dopasowują element główny). Następnie elementy `links` i `article`, które dopasowują pozostałe szablony, są odpowiednio formatowane.

Na końcu popatrzmy na kolejny nowy znacznik związany z szablonem:

```
<xsl:apply-templates select="newsletter/article" />
```

Ten element pozwala na wskazanie w określonym punkcie wewnątrz innego szablonu, który szablon ma zostać zastosowany do sformatowania elementów wskazywanych przez wartość atrybutu `select` tego znacznika. W tym przypadku zostanie wywołany szablon, który formatuje dzieci `article` elementu głównego `newsletter`. Na wydruku 4.6 są one używane do wysyłania na wyjście wyniku wywoływanego szablonu we właściwym miejscu.

## Więcej o dostępie do danych elementu XML

Musimy teraz odejść od szablonów, aby omówić kolejną nową składnię, która jest używana w kodzie pokazanym na wydruku 4.7. Załóżmy, że tworzona jest strona HTML

przez przekształcenie XML/XSL. Jak możemy wybrać wartości tekstowe XML ze znacznika, aby umieścić je wewnątrz atrybutu HTML `href`? Możemy użyć znacznika `value-of`, czyż nie? Utworzylibyśmy coś wyglądającego w ten sposób:

```
<a href="<value-of select="elementname" />">
```

Czy widzisz jakiś błąd w tej linii? Ta linia nie jest poprawna, ponieważ znaczniki XML nie są właściwie zagnieżdżone, albo dokładniej, znacznik jest otwierany wewnątrz innego znacznika. Potrzebujemy sposobu na obejście tego problemu. Okazuje się, że istnieje skrócona notacja, która używa nawiasów klamrowych w celu określenia węzłów, które muszą zostać wybrane. Ta notacja jest używana w arkuszu stylów z wydruku 4.7 i wygląda następująco:

```
<td><href="{link}" name="link{@position}">
```

Skrót jest używany dwa razy w powyższym fragmencie kodu. Wartość atrybutu `href` znacznika zakotwiczenia staje się danymi tekstowymi aktualnie wybranego elementu `link` z dokumentu XML. Nazwa znacznika zakotwiczenia jest kombinacją ciągu `link` i wartości atrybutu `position` znajdującego się w dokumencie XML. Czyż nie jest to sprytne?

Kiedy już rozumiemy budowę plików XML i XSL, przejdźmy do strony JSP. Ta strona jest bardzo podobna do tej pokazanej na wydrukach 4.3 i 4.5. Jedyną różnica jest, że zamiast umieszczania wyniku przekształcenia w buforze wyjściowym przez obiekt `StreamResult`, mamy umieścić je w innym pliku JSP.

Na wydruku 4.8 pokazano kod źródłowy dla pliku JSP, który przekształci XML za pomocą arkusza stylów XSL. Różnice między tym i wydrukiem 4.3 są zaznaczone drukiem wytłuszczonym. Zapisz ten plik jako *webapps/xmlbook/chapter4/CreateNewsLetter.jsp*.

**Wydruk 4.8.** *CreateNewsLetter.jsp; przesyłanie wyjścia do pliku*

```
<%@ page
  import="javax.xml.transform.*,
          javax.xml.transform.stream.*,
          java.io.*"
%>

<%
    String  ls_path = request.getServletPath();
            ls_path = ls_path.substring(0,ls_path.indexOf("CreateNewsLetter.jsp")) ;

    String  ls_JSPResult   = application.getRealPath(ls_path + "NewsLetter.jsp");
    String  ls_xml  = application.getRealPath(ls_path + "NewsLetter.xml");
    String  ls_xsl  = application.getRealPath(ls_path + "NewsLetter.xsl");

    FileWriter l_write_file = new FileWriter(ls_JSPResult);

    StreamSource xml = new StreamSource(new File(ls_xml));
    StreamSource xsl = new StreamSource(new File(ls_xsl));

    StreamResult result = new StreamResult(l_write_file);

    TransformerFactory tfactory = TransformerFactory.newInstance();
```

```
    Transformer transformer = tfactory.newTransformer(xsl);
    transformer.transform(xml, result);

    l_write_file.close();
%>
<jsp:forward page="NewsLetter.jsp"/>
```

Ten plik zaczyna się tak samo jak pozostałe JSP utworzone w tym rozdziale. Pobieramy ścieżkę dostępu do samego pliku JSP i następnie używamy tej ścieżki do utworzenia ciągu określającego, gdzie umieszczone są pliki XML, XSL i wynikowa strona JSP. Przyjmujemy, że wszystkie pliki będą umieszczone w tym samym katalogu. Różnicą tym razem jest, że tworzymy dodatkowy ciąg dla pliku wynikowego JSP:

```
String  ls_JSPResult   = application.getRealPath(ls_path + "NewsLetter.jsp");
```

Następnie tworzymy obiekt `FileWriter`, który zostanie wykorzystany do zapisu pliku znakowego, mianowicie naszego pliku wynikowego o nazwie *NewsLetter.jsp*:

```
FileWriter l_write_file = new FileWriter(ls_JSPResult);
```

Dalej obiekty `StreamSource` są ponownie używane do przechowywania XML i XSL. Obiekt `TransformerFactory` jest używany do utworzenia transformatora, który zastosuje arkusz stylów do pliku XML i umieści wynik w obiekcie `StreamResult`, który wysyła dane wyjściowe do pliku JSP.

Po zakończeniu przekształcenia musimy zamknąć wynikowy plik JSP:

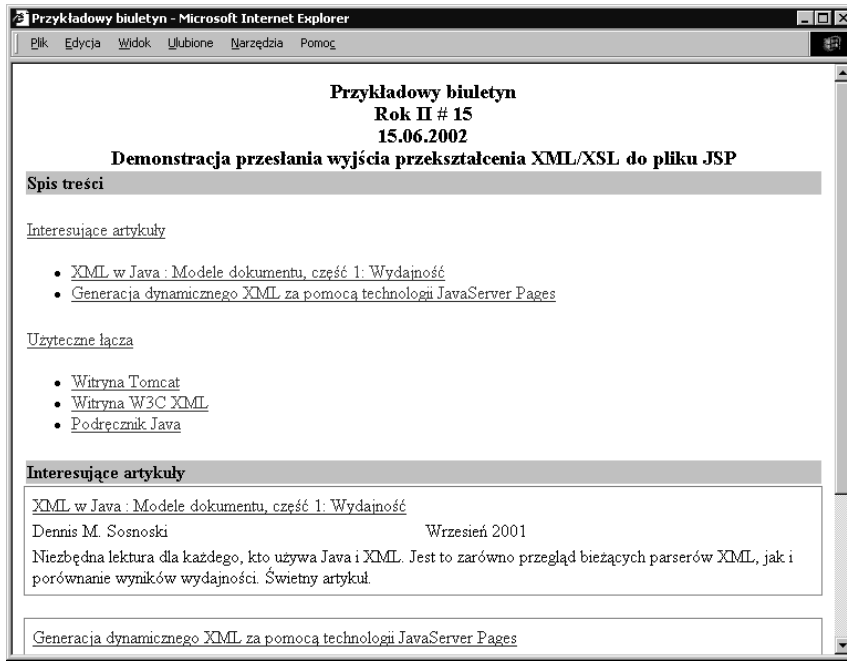
```
l_write_file.close();
```

Na końcu wykonamy przekierowanie strony, abyśmy mogli zobaczyć wyniki. Użycie następującego elementu akcji JSP przekieruje serwer do innego pliku, który zostanie przesłany do przeglądarki WWW:

```
<jsp:forward page="Newsletter.jsp"/>
```

Zwykle, aby oglądać nasz biuletyn, będziemy używać pliku JSP o nazwie *NewsLetter.jsp*. Ten plik nie wykonuje żadnych przekształceń, jest to raczej wynik przekształcenia. Po uruchomieniu pliku *CreateNewsLetter.jsp* popatrz na źródło *NewsLetter.jsp*. Jest to zwykły plik HTML, który zawiera wyniki przekształcenia i wygląda w przeglądarce tak, jak na rysunku 4.4.

Teraz możesz zapytać, dlaczego tworzymy końcowe wyjście w postaci strony JSP, chociaż zawiera ona tylko kod HTML? Zwykle potrzebne będzie inne przetwarzanie, które zdarza się na docelowej stronie JSP. Na przykład w biuletynie JSPBuzz, gdzie wykonywany jest podobny proces, końcowa strona wyjścia dołącza specjalne pliki JSP nagłówka i stopki, aby obsłużyć standardowe nagłówki i stopki naszej witryny. Zaletą takiego systemu jest to, że biuletyn wykonuje tylko przetwarzanie, które musi się zdarzyć przy każdym wywołaniu, a rzeczywiste przekształcenie XML/XSL zdarza się tylko raz na dwa tygodnie, w momencie publikacji biuletynu.




**Rysunek 4.4.** *Wynik uruchomienia CreateNewsLetter.jsp*

## Dużo XML-a i XSL-a, niewiele JSP

Przedstawimy teraz szersze spojrzenie na zagadnienia, które właśnie omówiliśmy. Możesz zastanawiać się, dlaczego tyle czasu zajmujemy się tym, co dzieje się wewnątrz XML i XSL. Faktycznie część omawiająca JSP w tym rozdziale wydaje się dość niewielka. Faktem jest, że można spojrzeć na technologię JSP jak na klej, który utrzymuje wszystko razem.

Typowa strona JSP/XML wymaga tylko trochę kodu JSP, aby utrzymać wszystko razem, ale wymaga dużo XML-a i XSL-a w celu wygenerowania wyników. Tak więc omówienie XML-a i XSL-a jest tak ważne jak JSP, ponieważ w tych przykładach XML i XSL rzeczywiście wykonują większość pracy, która musi zostać zrobiona. Prawdziwa tajemnica projektu JSP leży w wiedzy, jak połączyć razem wiele elementów, zanim zostaną one przesłane do klienta. Kod JSP w większości aplikacji JSP jest faktycznie najmniejszą częścią całego kodu.

Większość zadań na jakiejkolwiek stronie JSP jest rozdzielana na inne procesy, takie jak baza danych, przekształcenia XSLT, JavaBeans, usługi WWW i inne procesy zewnętrzne. Strony JSP są używane właśnie w celu połączenia wszystkiego razem. Faktem jest, że JSP stanowi najprostszą część w zbiorze tych zagadnień. Pierwszym dużym krokiem w kierunku stania się wielkim programistą JSP jest zrozumienie, że twoje umiejętności zależą bardziej od wiedzy o wszystkich wspólnie wykorzystywanych technologiach i umiejętności ich połączenia w celu utworzenia danych wyjściowych dla klienta. Można powiedzieć, że najlepsze strony JSP to takie, które zawierają jak najmniej kodu JSP i mają jak największą modularność.

# JSP i XML: Przegląd

Gdzie teraz jesteśmy? Wcześniej w tym rozdziale omawialiśmy JAXP i JSP. JAXP umożliwia nam łatwy dostęp do dokumentów XML i zastosowanie arkuszy XSL do plików XML. JAXP umożliwia też sterowane zdarzeniami przetwarzania XML. Jednak musimy zacząć od omówienia po kolei warstw przetwarzania, które było wykonywane w przykładach.

Najpierw mamy XML, który jest językiem używanym do tworzenia dokumentów. Te dokumenty są połączeniem danych i znakowania używanego do opisywania przechowywanych danych. Następnie strony JSP są używane jako szablon do wykonywania skryptów w celu uruchomienia kodu po stronie serwera.

Problemem jest, że strona JSP i dokument XML nie znajdują się w tym samym miejscu o tym samym czasie. W kodzie wygląda to tak, jak gdyby wszystko znajdowało się w jednym miejscu. Jednakże JSP jest procesem wykonywanym w pamięci, podczas gdy plik XML jest przechowywany w innej lokalizacji.

Oznacza to, że JSP nigdy bezpośrednio nie używa pliku XML. Musimy utworzyć *reprezentację* danych XML, do której kod ma dostęp. Reprezentacja dokumentu jest logiczną konstrukcją dokumentu XML, która znajduje się w pamięci. Mamy kilka możliwości odnośnie rodzajów reprezentacji, których możemy używać, ale ostatecznie wszystkie są oparte na pewnym modelu reprezentacji danych.

JAXP jest wspaniały, ponieważ pozwala nam łatwo przekształcać plik XML za pomocą XSLT. We wszystkich naszych wcześniejszych przykładach główny wysiłek był ukierunkowany na obsługę przekształceń XSLT na pliku XML. Jednak co możemy zrobić, kiedy potrzebujemy zmodyfikować plik XML? Albo co wydarzy się, kiedy chcemy utworzyć wyjście przez użycie kodu Javy zamiast przekształcenia XSLT? W tych przypadkach potrzebujemy zajrzeć pod powierzchnię JAXP.

W jednym z przykładów kod otrzymał kilka określonych danych z pliku XML i następnie dołączył je do dokumentu XSL. Jednak kod nie objaśniał, co zdarzało się za kulisami dostępu do danych. W tym przypadku wykorzystywaliśmy bardziej bezpośrednio inne API w celu badania plików XML. To prowadzi nas do następnego odkrycia: kiedy zagłębiamy się dalej, znajdujemy wiele API, których możemy używać w celu wykonania różnych działań na plikach XML. Możesz zastanawiać się, czy jest to rzeczywiście potrzebne, aby znać wszystkie API. Powinniśmy znać tylko najważniejsze API, z których każde umożliwia rozwiązanie innego problemu. Oznacza to, że powinniśmy omówić podstawowe API, które możemy wykorzystać.

# API XML i XSL w Javie

W tym dziale omawiamy API Javy, których jeszcze nie przedstawialiśmy. W późniejszych rozdziałach użyjemy kilku z tych API, aby ułatwić życie użytkownikom XML-a.

# DOM (obiektowy model dokumentu XML)

*DOM* (ang. *Document Object Model — obiektowy model dokumentu*) będzie omówiony szczegółowo w rozdziale 5., „Używanie DOM". DOM przedstawia standardową reprezentację W3C dokumentu XML. Zaletą DOM jest fakt, że jest standardem i te same metody DOM, które działają w kodzie Javy, będą działać w przeglądarce WWW używającej JavaScriptu, albo w aplikacji napisanej w języku C. DOM jest szeroko znany we wspólnocie użytkowników.

Największym problemem DOM jest to, że tworzy on wielkie reprezentacje dokumentów XML w pamięci. Czyni to DOM wolnym i często niepraktycznym podczas obsługi wielkich dokumentów XML, ponieważ cały dokument XML musi zostać zanalizowany przez DOM przed rozpoczęciem przetwarzania. Problem ten był siłą napędową stworzenia wielu innych API XML, ponieważ istnieje duże zapotrzebowanie na obsługę większych dokumentów.

Najpopularniejszym parserem Java DOM jest Xerces. Kolejny często używany parser DOM to Crimson. Jednakże Xerces 2.0 jest uważany za substytut parsera Crimson. Jak wspomniano wcześniej, Xerces 1.4.3 daje najlepszą podstawę do użycia go jako parsera DOM w tej książce. Jest tak z powodu faktu, że Tomcat 4.0 używa Xercesa w wersji 1.4.3. Xerces 2.0 jest wciąż w fazie w rozwoju.

Witryna główna specyfikacji DOM to *http://www.w3.org/DOM/*.

# SAX (parser XML)

*SAX* (ang. *Simple API for XML — proste API dla XML*) jest narzędziem do sterowanego zdarzeniami przetwarzania XML. SAX jest szeroko używany i będzie szczegółowo omówiony w rozdziale 6., „Programowanie SAX".

Krótko mówiąc, SAX umożliwia programistom przetwarzanie dokumentu XML poprzez definiowanie *zdarzeń* (ang. *events*). Zdarzenie jest wywołaniem zwrotnym od parsera SAX, kiedy napotykane są pewne warunki podczas sekwencyjnego odczytu dokumentu XML przez parser SAX. Użytkownicy mogą podjąć dowolne działanie podczas wywołania zwrotnego, od czytania danych z pliku XML do działania na danych. Jest to ostatecznie decyzja programisty, czy tworzyć wywołania zwrotne. Od programisty zależy też określenie, jakie działanie podjąć przy otrzymaniu wywołania zwrotnego od SAX. SAX szybko staje się API niższego poziomu. Oznacza to, że chociaż wielu programistów używa SAX, dzieje się to pośrednio przez inne narzędzia albo API, takie jak JAXP. Używanie SAX bezpośrednio wymaga dużo pracy, ale jest szybkie i skuteczne. Wiele innych API używa SAX jako szybki sposób na wczytanie pliku XML. Faktycznie wiele narzędzi XML używa SAX jako domyślnego parsera XML ze względu na szybkość jego działania. Jednak powinieneś zawsze sprawdzić dokumentację używanego przez Ciebie narzędzia, aby być pewnym, jaki jest domyślny parser, gdyż może się to różnić w zależności od narzędzia.

Witrynę główną SAX można znaleźć pod adresem *http://www.saxproject.org/*.

# JDOM (reprezentacja dokumentu XML)

JDOM to metoda tworzenia reprezentacji Javy dokumentu XML.

JDOM obejmuje kilka klas, które pozwalają na użycie SAX albo DOM do przeczytania danych XML do jego własnej reprezentacji Javy. JDOM obsługuje wiele często używanych opcji, które obsługuje DOM. W przeciwieństwie do wielu innych API omówionych w tym dziale, które używają interfejsów Javy do zdefiniowania reprezentacji Javy, JDOM używa konkretnych klas w celu zdefiniowania reprezentacji Javy. To podejście czyni JDOM łatwiejszym do bezpośredniego użycia niż SAX albo DOM. Jego wadą jest ograniczona elastyczność w implementacji. Dodaje to JDOM innego smaku niż większość innych API. Jednym z efektów jest to, że nie zobaczysz obiektów-fabryk wewnątrz JDOM.

JDOM wykorzystuje sporo z API Javy `Collection`, a szczególnie interfejsów Javy `List` i `Map`, w celu zdefiniowania reprezentacji. Oznacza to, że programista może używać standardowych struktur Javy w celu posługiwania się reprezentacją JDOM.

JDOM nie umożliwia przetwarzania XML; zamiast tego trzeba użyć SAX albo wczytać reprezentację DOM. JDOM udostępnia kilka klas, które ułatwiają użycie SAX albo DOM jako źródła danych.

JDOM jest wymieniony jako JSR-102 na witrynie Java Community Process.

Projekt JDOM można znaleźć na witrynie *http://www.jdom.org/*.

# dom4j (reprezentacja dokumentu XML)

dom4j to projekt o jawnym kodzie źródłowym, który zawiera zbiór API Javy używanych do manipulacji plikami XML, XSLT i XPath.

To API jest podobne do JDOM w tym sensie, że tworzy opartą na Javie reprezentację dokumentu XML. Jednak różni się też użyciem interfejsów Javy. Używanie interfejsów ma swoje zalety (elastyczny i rozszerzalny projekt) i wady (trzeba poznać kilka dodatkowych warstw). To API pozwala na użycie obiektów Javy `Collection`, które upraszczają obsługę struktury XML.

Z dom4j łatwo jest używać standardu DOM i mieć jednocześnie dostęp do wygodnej w użyciu reprezentacji dokumentu w Javie. Ponadto dom4j można łatwo zintegrować z JAXP, tak jak SAX i DOM. Nadzwyczaj przyjemną cechą dom4j jest fakt, że zawiera oparty na zdarzeniach model przetwarzania dokumentów XML. To pozwala parserowi dom4j na obsługę wielkich plików XML w częściach, a nie jako pojedynczych dokumentów. Oznacza to, że cały dokument XML nie musi znajdować się w pamięci (duży plus kiedy ma się do czynienia z obszernym plikiem XML). Użyjemy tej cechy w rozdziale 14., „Tworzenie usługi WWW”.

Projekt dom4j można znaleźć na witrynie *http://dom4j.org/*.

## JAXB (parser i reprezentacja dokumentu XML)

*JAXB* (ang. *Java Architecture for XML Binding — architektura JAVA dla łączenia XML*) jest używany w celu automatyzacji odwzorowania pomiędzy dokumentami XML i Javą. Unikalnym podejściem JAXB jest użycie schematu XML (obecnie DTD, a w przyszłych wersjach schematu XML) do stworzenia zoptymalizowanej klasy Javy w celu wykonywania przetwarzania reprezentacji dokumentu XML. To pozwala JAXB na automatyczne użycie tych dostosowanych klas Javy do przetwarzania dokumentu XML szybciej niż SAX, udostępniając jednocześnie reprezentację struktury danych dokumentu XML, która jest podobna do DOM. Jego wadą jest fakt, że wynikowa reprezentacja dokumentu XML jest funkcjonalna tylko względem schematu XML, do którego została wygenerowana. JAXB używa języka wiązań, aby pozwolić programiście na optymalizację wynikowej reprezentacji dokumentu XML względem schematu i potrzeb projektu.

JAXB jest rozwiązaniem, które działa najlepiej, kiedy potrzebna jest reprezentacja dokumentu wymagająca szybkiego sprawdzania prawidłowości struktury zawartości, a dokument XML ma stały format.

W czasie pisania tej książki był to wciąż nowy projekt. Jednak najprawdopodobniej jego wynikiem będzie jedno ze standardowych API do obsługi dokumentów XML.

Projekt JAXB można znaleźć pod adresem *http://java.sun.com/xml/jaxb/*.

JAXB jest wymieniony jako JSR-031 na witrynie Java Community Process.

# Podsumowanie

Ten rozdział dał ci pierwszy posmak wspólnego wykorzystania XML-a i JSP. JAXP czyni łatwym połączenie plików XML i arkuszy stylów XSL w celu tworzenia nowych stron. Jednak XSLT jest tylko częścią większego obrazu w używaniu XML-a i JAXP. Podczas badania co jeszcze można zrobić szybko, zobaczymy, że dostępnych jest wiele API. Każde API udostępnia unikalne możliwości, zalety oraz wady. Chociaż moglibyśmy omówić zalety każdego z nich na papierze, to podczas kodowania odkryjemy zalety różnych narzędzi. W następnych rozdziałach nauczysz się szczegółów, jak programowo badać, modyfikować i wykonywać operacje na dokumencie XML.

Wykorzystując JAXP, odkryliśmy kilka zadziwiających faktów dotyczących JSP. Pierwszym faktem jest, że tajemnica wykorzystania JSP leży raczej w połączeniu różnych technologii, aniżeli w samym używaniu JSP. Drugim jest, że siła projektu JSP leży w jego modularności, a JSP jest w rzeczywistości tylko klejem albo szablonem do łączenia modułów.